KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
Celestijnenlaan 200A – B-3001 Heverlee

ABDUCTIVE CONSTRAINT LOGIC PROGRAMMING: IMPLEMENTATION AND APPLICATIONS

Promotoren:
Prof. dr. Danny De Schreye
Prof. dr. Marc Denecker

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de toegepaste wetenschappen

door

Bert VAN NUFFELEN

june 2004

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
Celestijnenlaan 200A – B-3001 Heverlee

ABDUCTIVE CONSTRAINT LOGIC PROGRAMMING: IMPLEMENTATION AND APPLICATIONS

Jury:
Prof. dr. ir. Jean Berlamont, voorzitter
Prof. dr. Danny De Schreye, promotor
Prof. dr. Marc Denecker, promotor
Prof. dr. Antonis C. Kakas (University of Cyprus)
Prof. dr. Ofer Arieli (Academic College of Tel Aviv)
Prof. dr. ir. Eric Steegmans
Prof. dr. ir. Maurice Bruynooghe

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de toegepaste wetenschappen

door

Bert VAN NUFFELEN

U.D.C. 681.3*I2

june 2004

D/D/2004/7515/5

ISBN 90-5682-512-7

Abductive Constraint Logic Programming: Implementation and Applications

Bert Van Nuffelen
Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium

Abstract

Abductive reasoning is a form of reasoning that naturally arises in the context of Declarative Problem Solving. In Declarative Problem Solving, the expert formulates his domain knowledge of the problem as a logic theory. A solution of the problem is then the outcome of reasoning process that uses this logic theory as input. For many problems, the reasoning task requires to compute an interpretation of a relation so that the query is entailed by the logic theory augmented with this interpretation. This inference is called *abduction*.

This dissertation consists of two parts. The first part discusses the development of an abductive constraint logic solver, called the $\mathcal{A}$system, for the knowledge representation language ID-Logic. We show how to transform ID-Logic theories in normal abductive logic programs which form the input of the $\mathcal{A}$system. Procedurally, the $\mathcal{A}$system is a mixture of existing abductive logic procedures, i.e. SLDNFA, IFF and ACLP. Typical for the procedure is the use of (existing) subsolvers to perform a part of the reasoning. The procedure is sound and complete w.r.t. the three-valued completion semantics. Our main contributions are at the implementation level. In order to obtain an efficient system good design decisions must be made. In particular, we discuss specialized data structures, the evaluation of the inference rules, the organization of the search tree, efficient equality reasoning and the integration of a finite domain constraint solver. The system is validated experimentally using classical AI problems.

In the second part of the dissertation, we present our contributions to the integration of multiple independent databases. We address two issues in this domain. The first concerns the differences in the ontologies between the databases. Each database stores information in a language which is often different from other databases. Merging the information in the databases requires that these languages are related in a formal way with each other. Our solution presents an ID-Logic mediator-based system that integrates the information of the databases by relating the database's languages with a common global language. The second issue addressed is contradictory information. When data from independent databases are merged, contradictions with respect to a set of integrity constraints may arise. We present two approaches to restore in a coherent way the consistency of an inconsistent database based on repairs, i.e. a special kind of database updates. One approach considers represent the problem as an abductive logic theory and it uses the $\mathcal{A}$system as computational engine to compute the (most preferred) repairs. In the other approach the repairs are the models of propositional theory, that is derived from the data and the integrity constraints by an elegant encoding.

Preface

Imagine . . . you are eight years old and you say "I want to be just like daddy.".

Imagine . . . you got your first home computer (MSX) and your father uses it to grow collyflowers, trees and other plants on the screen.

Imagine . . . you got curious and you want to know what *real* Artificial Intelligence (AI) is.

Imagine . . . you go to the university for studying Computer Science and starts afterwards a PhD at the Artificial Intelligence lab.

Imagine . . . you all finishes it and you still do not know what AI is exactly all about.

It seems that you have failed your mission of life, but . . .

you do know that you have implemented a system that solves the *Einstein Riddle*[a] in a few seconds, stunning your colleague computer experts since according to Einstein only the smartest people can solve the riddle. And you feel happy.

[a] Variants of this riddle appear frequently as puzzles in newspapers and journals. With a bit of patience anyone can solve the riddle.

A PhD typically leads to an in-depth specialization in a rather small domain. In my case, it started just like that. I started working on the first prototypes of the abductive system, the $\mathcal{A}$system, which is central in the thesis. However, this was not the start of further specialization, but merely an eye-opener to many other problem domains in Computer Science. I encountered so many challenging problems and interesting questions that another PhD can be filled. I will leave that for someone else.

I am grateful to my promoters Danny De Schreye and Marc Denecker to get me in contact with such a challenging domain, allowing me to lift a small piece of my child's dream: "What is Artificial Intelligence?". Danny De Schreye has been the stimulating and enthusiastic alpha and omega of the PhD work. Those moments are always the hardest. Marc Denecker has been my scientific supervisor for about seven years. His questions, during our numerous coffee breaks, were helpful guidelines for polishing and directing the work. The seeds of this PhD are found in his PhD. It has been a pleasure to grow out of these seeds new work.

I thank the members of my jury, Professors Eric Steegmans, Antonis Kakas, Ofer Arieli and Maurice Bruynooghe, Marc Denecker and Danny De Schreye for reading my thesis en providing me with many helpful comments. I am happy they accepted to be a member of the jury, since the intensive collaboration I had over the years. Many thanks go to Antonis Kakas for the stimulating discussions on the implementation of the $\mathcal{A}$system and for always providing a warm welcome in Cyprus and London, Ofer Arieli for pulling at me at the end of his stay in Leuven to start the work on database integration and for the intense long distance collaboration (maybe one day we can visit you), and Maurice Bruynooghe for the many incentives to new research directions and the carefully proofreading of the 'integration' papers. Being part of this team has been a real pleasure.

Of course a PhD is only possible when there is the right atmosphere exists. Many thanks go to Nikolay Pelov as long time office mate and PhD fellow. I want to thank him for the many suggestions for improvement of the work, especially for the work on equality reasoning. I also want to thank Emmanuel De Mot, David Gilles, Maarten Mariën and Álvaro Cortés-Calabuig, Joost Vennekes, Pieter Bekaert, not only for the many discussions on research topics, but also for the many (mind) relaxing coffee breaks. In that respect, Gerda, Henk, Alexander and many other Alma and departmental colleagues, thank you for the joyful lunch entertainment. The $\mathcal{A}$system implementation would not have been succesful without the kind hosts George Fakas and Neophythos in Cyprus. To enlighten the darkest days, there was the wonderful music from the whole world and the many, many friends of Caminhando, the solidarity choir. The every week rehearsals have been a constant input of enthusiasm and creativity. I wish those who are now in their final stage of the PhD much success. Finally, all my friends who have supported me and my wife Sief during the last year, I own you much.

Mijn dank verder gaat ook uit naar mijn ouders, moemoe, schoonouders en familie die veel geduld hebben gehad tijdens de afgelopen maanden waarin ik me zo weinig liet zien. Mijn hulpvaardige correctoren Peter Van Nuffelen, Pieter Claeys, Jeroen VanHoutte, Alfons Van Nuffelen en Sylvie De Ruyck, dankjewel voor de vele aanwijzingen: het maakte dit werk heel wat beter.

Tenslotte wil ik bijzondere dank uitdrukken voor Sief, mijn geliefde en vrouw, die me de ruimte gaf om deze tekst te maken. Je voordurende aanmoediging heeft veel voor mij betekend. Dit werk is ook een beetje jouw werk.

The research in this thesis has been supported by the following projects and grants: the GOA project "LP+ : a second generation Logic Programming Language", the European project "KIT: Computational Logic for Flexible Solutions to Applications" and the FWO project "Theory of nonmonotone Inductive Definitions and its use for building Knowledge-based systems".

Bert
28 juni 2004

Aan Wim, Bert en Piet,
die dit moment bijwonen vanuit ons hart.

Contents

Chapter 1

Introduction

1.1 Knowledge Engineering and Declarative Problem Solving

The last decades, our (Western) society has changed incredibly. In less than 30 years the computer has infiltrated in almost all aspects of our daily life. Computers regulate and control traffic lights, airplanes, our mobile phone and many more systems. It is man's main tool in solving all kinds of problems. The computer allows us to solve complex problems correctly, more accurately and faster than ever. Despite these benefits, we experience a large gap between our human problem description and the final computer based solution. One of the main reasons for this is that the used "computer/programming" languages are more oriented towards the fine grained logic of a computer than towards the high level human view.

This gap has been and still is a driving force in computer science. Consider for example the developments in software engineering. A modern software engineering process consists of several phases in which the natural language problem specification is gradually formalized and refined. At the end of this process a computer program is obtained which encodes a solution to the problem. During the last decade, under the stimulation of object oriented software design, a number of important methods and tools that support and guide this software engineering process, have been developed. To name a few: problem analyzing methods such as *use cases*, modeling languages such as UML and methodological guidelines such as *design patterns*. By using them, the produced software satisfies the imposed quality requirements. But, software engineering is a process that is tailored towards algorithm based solutions[1].

For a large group of problems, an algorithm based solution is not satisfactory. The situation is well illustrated by the class of combinatorial problems including

[1]By an 'algorithm', we understand a prescription on how to solve a particular problem.

scheduling and planning. Applying the common software engineering techniques in order to obtain an algorithm based solution is often not very successful. Due to the inherent complexity of these problems, algorithm based solutions will be complex and difficult to maintain. This approach becomes even less appropriate, if we take into account that problems of this type tend to vary regularly. The changes often have the effect that the assumptions on which the algorithms are based, are violated; hence forcing the design of a new algorithm. For this reason and many more, one investigates methods that are more flexible. Computationally, the performance of these methods is often comparable to (and sometimes better than) algorithm based solutions.

Such problems are usually tackled by *Declarative Problem Solving* (DPS). This method covers the idea where a human expert formulates its domain knowledge as a formal (logic) theory which will be interpreted by a computer program to solve the specified tasks. To be successful, a DPS method must have the following three components:

1. A *logic* consisting of a formal language and precise and clear formal semantics.
2. A *declarative reading* for the formulas specified in that language.
3. A *solver* that reasons with a specification formulated in the language.

Although the formulation of these aspects hints towards general DPS methods, like the one we study in this dissertation, many small, more domain and task specific, DPS methods exist. For instance, one can view SQL for databases, constraint solvers and strips planning systems as domain specific DPS methods. All these approaches provide a logic which the human expert uses to formulate his specific domain knowledge. The task (e.g. querying the database, finding a consistent variable assignment or a plan) is solved by an intelligent solver. The 'solving related information' required from the expert is limited to either reformulating its domain knowledge, or to tuning the parameters of the solver. It relieves the expert from developing an algorithm to solve the task, and allows him or her to tackle the task from a more abstract level.

An important aspect of DPS is the ability to intuitively understand the formal expressions as statements in the real world domain of discourse. For many existing DPS methods, including the ones we have mentioned, this is somehow troublesome. For example, the following finite domain constraints[2]

$$x \neq y, x \neq z, z \neq y, z \in \{1,2,3\}, x \in \{1,2,3\}, y \in \{1,2,3\}$$

do not give any clue about the problem which is actually solved. A good DPS method will make this information explicit in the specification. One can argue

[2] x, y and z are variables which possible values are given by their domain, i.e. a finite set of values. Here $x \in \{1,2,3\}$ denotes that the domain of the variable x is the set $\{1,2,3\}$.

that giving appropriate names to the above variables will solve the issue. This is, however, insufficient.

$$\begin{array}{l} vertex_1 \neq vertex_2, vertex_1 \neq vertex_3, vertex_3 \neq vertex_2, \\ vertex_3 \in \{red, blue, yellow\}, \\ vertex_1 \in \{red, blue, yellow\}, \\ vertex_2 \in \{red, blue, yellow\} \end{array}$$

These names tell us that the problem (domain) concerns a graph with (at least) three vertices that must have a different color. More cannot be deduced. E.g. it is still unknown that the actual problem concerns the coloring of a graph which has the form of a triangle.

Since DPS focuses on producing correct formal representations of a problem domain for a particular task, the classical software engineering approach is often inadequate. The alternative process for DPS, what we call *knowledge engineering*, is sketched below[3]. Basically, it consists of three steps which are more or less executed in the presented order. Similar to the classical software production process the formal specification will be revised several times in order to reach the final specification. For a DPS-method having a logic with formal semantics, the major decisions in knowledge engineering are

1. The selection of an appropriate alphabet.

   The alphabet, i.e. the symbols used for denoting the objects and relations in the problem domain, must be well-chosen: the abstraction level should be appropriate, the names should be intuitive and allow easy understanding, and for each relevant notion in the problem domain there should be a symbol. This design is of great importance, since it determines many properties in the next phases.

   For example, suppose we want to model a graph coloring problem. This problem domain has two kinds of objects: the vertices of the graph and the colors. Let $vertex(\cdot)$ denote that an object is a vertex; $color(\cdot)$ denotes that an object is a color. Furthermore, a graph connects vertices via its edges. $edge(\cdot,\cdot)$ denotes this relation. The color of a vertex is denoted by $colorOf(\cdot,\cdot)$. These four relations form an appropriate alphabet for the problem.

2. The formulation of the (general) domain knowledge as logical statements using the designed language.

   In this phase, the declarative reading of the formal statements becomes important. For a well-designed alphabet and well-designed logic, the intuitive interpretation by reading the logical formulas as if it were natural languages expressions over the domain under consideration, should exactly match the

[3] In the literature, one finds other meanings for the term knowledge engineering.

reality of the problem domain. Analogously, the formal semantics assigned to the formulas should correspond with reality. By this, the expert has a tool for verifying if the modeled logic theory, i.e. the set of all formal statements that are written down, expresses his domain knowledge.

The actual formulation is determined by on the one hand the used logic, and on the other hand the tasks for which this domain knowledge is formulated. It happens that some relevant knowledge of the problem domain is inexpressible. This is either due to the limitations of the used logic, or it points to a bad design of the alphabet. In the latter case, the alphabet must be revised; the former situation is somehow more troublesome. When no alphabet that allows to express properly the knowledge can be found, a new and more expressive logic has to be selected.

Applied to our graph coloring example, the domain knowledge is specified in logic statements as follows. For now, it suffices to understand the next first order logic statements on an intuitive level as the formalizations of the real world domain knowledge. Below, the logic we will use in the thesis is elaborated in more detail. The graph coloring problem specifies that two connected vertices have a different color.

$$\forall x, y, cx, cy.\ edge(x, y) \land colorOf(x, cx) \land colorOf(y, cy) \rightarrow cx \neq cy.$$

Also each vertex has a color (the first assertion), which is unique (the second):

$$\begin{array}{l}\forall\ x.\ vertex(x) \rightarrow \exists\ cx.colorOf(x, cx).\\ \forall\ x, cx, cy.\ colorOf(x, cx) \land colorOf(x, cy) \rightarrow cx = cy.\end{array}$$

There is additional knowledge that ensures that the relations are well-typed.

$$\begin{array}{l}\forall x, y.\ edge(x, y) \rightarrow vertex(x) \land vertex(y).\\ \forall x, c.\ colorOf(x, c) \rightarrow vertex(x) \land colorOf(c).\end{array}$$

3. The formulation of the task that has to be solved, i.e. the problem instance and the query, together with the selection and application of an appropriate reasoning tool.

   In this phase, the actual problem solving happens. A concrete task specification merges the general domain knowledge (sometimes called the background knowledge) that has been specified previously, and the specific knowledge about the concrete problem instance. The latter usually contains an initial state description and a description of the goal (or query) that must be reached (answered). Typically, the background knowledge is formulated in such a way that the information of the instance is easily added. For example, in Logic Programming, the initial state is often given as a set of facts.

   Having a final task description, an appropriate solver must be chosen. Some solvers are dedicated to one task (e.g. strips planners), while others have a large application area (e.g. abductive solvers and model generators). Often the dedication of a solver to specific tasks reflects itself in the logic the solver accepts. In general, an expressive logic is needed to formulate the domain knowledge such that the declarative reading of the specification matches the problem domain. Solving a task in a declarative way requires, therefore, solvers for this expressive logic. Another approach is the transformation of the specification in the expressive logic into a formulation of a task specific solver. In this case, a reverse transformation must be designed in order to represent the answers constructed by the specific solver in terms of the original specification (of course only when it is necessary).

If we want to solve the graph coloring problem for a triangular graph with three colors, the following instance specific information must be added:

$$\begin{array}{lll} vertex(v_1). & edge(v_1, v_2). & color(red). \\ vertex(v_2). & edge(v_1, v_2). & color(blue). \\ vertex(v_3). & edge(v_2, v_3). & color(yellow). \end{array}$$

Observe now that a solution to the problem is given by a set of atoms of $colorOf(\cdot, \cdot)$ that satisfy the domain knowledge. Constructing such set requires a different kind of reasoning than the search for a variable assignment. Hence, a constraint solver that can be used for the latter computational task, is unsuited for this task. For this task, one can use an abductive solver.

Although DPS is applicable to a wide range of problems, it is not the ultimate solution. For example, consider the development of a Graphical User Interface (GUI). One can model the interaction between the human and the interface as a logic theory. However, in the end, this logical representation shall not be used as the implementation that effectively controls the GUI. For a GUI, the type of data input (e.g. String and Int) is relevant, and such details are normally left out in the high level view of the logical representation. Hence, the logic theory forms here an abstract model of the software that must be constructed. Knowledge engineering forms in this context a first and important subprocess in the software engineering process. In general, the knowledge engineering process results for some problems in (a part of) the final product, while for other problems the formal representation must be further elaborated into a working software product.

1.2 Declarative Problem Solving in the thesis

Above we have mentioned that the logic determines the success of the declarative modeling of a problem domain. The logic used in this thesis is a very expressive knowledge representation language, called ID-Logic [92, 94] which stands for *Inductive Definition Logic*. ID-Logic has been designed for the above presented view on Declarative Problem Solving.

The foundation of this logic, as the name indicates, is the believe that definitions are frequently occurring and natural form of (human's) knowledge. Intuitively, a definition can be understood as a constructive description of a relation. ID-Logic provides a corresponding logical concept to capture this definitional knowledge. (These are also called *definitions*.) Another kind of knowledge are assertions, i.e. universally true statements about the problem domain. This kind also is supported by ID-Logic. Formally, ID-Logic is an extension of classical logic with non-monotone inductive definitions. Its formal semantics is based on the well-founded semantics for logic programs.

Other important properties of ID-Logic are the ability to represent incomplete knowledge, its modularity and elaboration tolerance. The first one is closely related to the theme of our thesis: abductive reasoning. Incomplete knowledge manifests itself in the existence of multiple models for the ID-Logic theory. Intuitively, this is related to the absence of a definition for some relations.

Recall the graph coloring example from above: for the problem formulation, the predicate $colorOf(\cdot,\cdot)$ has been introduced to denote the color of a vertex. Consider now a colored graph. This means that we have knowledge about the color of each vertex. This knowledge may take the form of a table, which forms the definition of the $colorOf(\cdot,\cdot)$ predicate. For the graph coloring problem, this table is absent. We have no knowledge about the exact color of a vertex in the graph. The only knowledge we have is that each vertex has a color and that the color of one vertex is different from the color of a connected vertex. This information does not include the knowledge about the concrete color of a vertex. Hence the predicate $colorOf(\cdot,\cdot)$ has no definition in the ID-Logic theory that formalizes the graph coloring problem.

The ID-Logic formalization of a problem domain is a result in its own right. It makes the expert's knowledge of the problem domain explicit and thus accessible for others. However our interest is to solve the (graph coloring) problem. In order to find a solution, the right form of reasoning must be applied: the needed reasoning must construct a table for the $colorOf(\cdot,\cdot)$ predicate that satisfies the assertions. This form of reasoning is called *abduction*.

Abductive reasoning or, more specifically, the construction of an efficient abductive constraint solver is the main theme of this dissertation. The development of an abductive solver is presented in the first part of the thesis. In a second part, we present a study of the problem of the integration of multiple independent databases. This problem domain forms an important research topic in its own right. In this dissertation we will tackle this domain from our Declarative Problem Solving perspective, using ID-Logic and our developed abductive solver as tools. In the last section of that part, we will present a domain specific method that is not based on these tools.

1.2.1 Abductive inference

Like deduction and induction, abduction is a form of reasoning: it is the construction of plausible hypotheses to explain an observation with respect to given background knowledge. Already the Greek philosopher Aristotle had studied this form of reasoning, but it was the modern philosopher C.S. Pierce (1839–1924) who gave it the name abduction [118]. In the late eighties, Eshghi and Kowalski started to explore abductive inference in the context of Logic Programming [116, 117, 170], founding in this way the field of Abductive Logic Programming (ALP) [153, 97, 98]. One line of research used abduction to give a formal account to non-monotonic semantics for logic programs. This research has contributed to the foundations of ID-Logic. One can argue that ID-Logic provides epistemological foundations to (Abductive) Logic Programming. For the other line of research, ALP was a computational paradigm for solving a variety of problems. These problems include AI planning problems [116, 202], scheduling problems [159, 160, 238], diagnosis problems [216, 84], natural language processing [237, 30, 71], multi agent reasoning [249, 169], and many more. (See [98] for a survey of applications.) Our work fits in this view on abduction.

Formally, Abductive Logic Programming is based on these notions. An Abductive Logic theory is a triple ($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$) where $\mathcal{P}$ is a logic program, $\mathcal{A}$ a set of abducible atoms and $\mathcal{IC}$ a set of integrity constraints. The formal semantics of an Abductive Logic theory the S-models of $\mathcal{P}$ which satisfy the integrity constraints $\mathcal{IC}$, where S denotes a formal semantics as defined in Logic Programming (e.g. completion/stable/well-founded). An abductive procedure computes for a query $\mathcal{Q}$ and an Abductive Logic theory ($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$) a set of abducible atoms $\Delta \subseteq \mathcal{A}$ and a substitution σ such that the logic program augmented with Δ is a S-model of the Abductive Logic theory, and the augmented program entails the query: $\mathcal{P} \cup \Delta \models_S \mathcal{Q}\sigma$, and the integrity constraints are satisfied by the augmented logic program: $\mathcal{P} \cup \Delta \models_S \mathcal{IC}$.

Over the years many abductive inference procedures have been proposed. The most influential ones are Eshghi-Kowalski [116, 117, 170], Kakas-Mancarella [156, 157, 154], SLDNFA [95, 96], IFF [129] and ACLP [152, 161]. Despite the large application area for abduction and the many abductive procedures that have been proposed, none of these abductive procedures have got a robust implementation which could serve as a general solver for abductive logic programs. The prototypes that had been implemented, did suffice for solving a limited number of problems in order to demonstrate that it is feasible to solve the problem with an abductive procedure. They lacked, however, robustness and sufficient overall efficiency in order to be usable for the wide range of problems in which abduction can be applied. A first step towards such a system has been the ACLP-system. Its basic implementation was robust and sufficient efficient so that it could be applied for solving a real world problem, i.e. the scheduling of the crews of an aircraft company [162, 200], with minor domain specific adaptations.

Our contribution is exactly at this level. We have developed a new abductive constraint logic system, called the $\mathcal{A}$system. The ultimate goal of the system is to be an efficient abductive reasoning system that can be used to solve real world problems. The work presented in the first part of this thesis is a first step towards this goal. Since there was little known about the technical needs for such a system, our research, presented here, explores various implementation techniques that improve the efficiency of the system. This work has resulted in a powerful abductive reasoning system that is a clear improvement over the earlier implementations and that can compete in performance with related reasoning systems (i.e. ASP solvers).

The proof procedure of the $\mathcal{A}$system is a combination of the abductive proof procedures SLDNFA, IFF and ACLP. The basis forms SLDNFA[4], but it has been reformulated as a set of state rewrite rules similar to the IFF-procedure. An $\mathcal{A}$system state consists of a goal stack and a store. In an $\mathcal{A}$system derivation, a goal formula is selected from the goal stack and the appropriate inference rule is applied. This process reduces the goal formulas to basic formulas which are collected in the store. Depending the basic formula, either they are just stored, e.g. the abducible atoms are stored in a set Δ, or further evaluated by a constraint solver. Hence, the $\mathcal{A}$system depends for a part of its reasoning on external solvers. This multi-solver idea is inherited from ACLP: the $\mathcal{A}$system takes advantage of (existing) constraint solvers to optimize and improve its reasoning. We have integrated a finite domain constraint solver and a (dis)equality constraint solver. The integration of the constraint solvers is done in a *modular* way, which facilitates the extension of the $\mathcal{A}$system with new solvers.

For a given query $\mathcal{Q}$ and an Abductive Logic theory ($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$), an $\mathcal{A}$system derivation starts from the initial state composed of an empty store and a goal stack containing $\mathcal{Q}$ and the integrity constraints $\mathcal{IC}$. A solution is found when the goal stack is empty and the store, i.e. all substores, is consistent. The substore Δ will then contain the abducible atoms which we were looking for. The $\mathcal{A}$system will backtrack when its current state is known to be inconsistent, i.e. either a formula is reduced to false or one of the constraint stores is inconsistent. It may happen that no solution can be found; in that case, the query is not entailed by the abductive logic program.

The $\mathcal{A}$system inherits from SLDNFA its soundness and completeness results w.r.t. the three-valued completion semantics. Because of these results the $\mathcal{A}$system is not a sound and complete abductive solver for ID-Logic theories, since ID-Logic is based on the stronger well-founded semantics. Fortunately, for a large class of theories which are very common in practice both semantics coincide.

Our main concern in the implementation of the $\mathcal{A}$system is *efficiency*. Other qualities that are taken into account are *robustness*, *scalability* and *modularity*. Some important factors that influence these qualities are the following four:

[4]The original representation SLDNFA was formulated as an extension of SLDNF.

1. The *data structures* must be designed so that they support the evaluation of the inference rules. A good design of the internal data structures may save computation time by reducing lookup and storage times. Moreover the scalability of the system is often determined by the data structures.

2. The evaluation cost of one *inference step* is another important factor. When the evaluation of one or more inference rules is very costly then the overall performance of the system will decrease. Moreover since the majority of the computation time is consumed by the application of the inference rules, a tiny reduction of the evaluation time may result in a substantial reduction of the overall computation time.

3. The efficiency of the *search process* is determined by the application order of the inference rules. If this is badly chosen, the system may need a lot of backtracking to find a solution. In the worst case the system thrashes. Of all factors this is the most important one since a badly designed search process will often result in a unsatisfactory behavior.

   The search process influences the robustness of the system. For example, due to the inadequate design of the search process, many prototypes of abductive systems and also the first prototypes of the $\mathcal{A}$system were very sensitive to the ordering of the literals (in the query). Such effects are not desirable. Our aim is to build a system that can be used as much as possible as a blackbox. The user should (first) focus on the correct modeling of the problem, instead of tuning the representation so that the system solves the problem efficiently. For those cases that efficiency is important, the system must provide the necessary means to tune the system for a given problem. An important way of tuning is the applied search strategy. For this reason, the search process should be parametric in the applied strategy.

4. The efficiency of the *subsolvers* is the last factor. The subsolvers are constraint solvers which are very efficient reasoners for predefined languages. Since the constraint solvers perform a (large) part of the $\mathcal{A}$system's reasoning, their reasoning efficiency is an important factor in the overall efficiency of the $\mathcal{A}$system. As mentioned, their efficiency will be exploited by handing selected formulas of a constraint domain during the evaluation process of the $\mathcal{A}$system to the corresponding constraint solver. Also since backtracking at the level of the $\mathcal{A}$system often is more costly than at the level of the constraint solver, choice points also are moved to the constraint solver. This requires that the constraint solver can handle disjunctions.

In this thesis each factor is elaborated. In what follows next, we summarize the contributions of our work w.r.t. the above factors in some more detail.

The $\mathcal{A}$system has been implemented as a meta-program on top of Prolog. (The actual used system is Sicstus Prolog [239].) A meta-program is a program that treats another program as input. Meta-programming is commonly used to develop new interpreters, debuggers, etc. The classical meta-programming approach is the non-ground version: the variables of the object program, i.e. our abductive logic program, are represented by the variables of the interpreting meta-program, i.e. as a Prolog variable. Because the formulas of the abductive logic program contain universally quantified variables and free variables and Prolog programs can only have universally quantified variables, there is a mismatch between the variables of the Prolog meta-program and those of the abductive logic program. To deal with the differences in quantification, the object-formulas are copied during the derivation. This copying forms a substantial overhead in the system. Therefore we have studied an alternative implementation using ground meta-programming. This implementation avoids the copying, but introduces another costly operation: the maintenance of the bindings of the variables. By the variable sharing, the Prolog system will maintain the bindings in the classical meta-programming. In the ground approach, this is the responsibility of the meta-program itself, which is much more costly than the low level support of the Prolog system. When all (side) effects are taken into account, the ground meta-programming yields a faster execution.

With respect to the search process, much effort went into the design of a least-commitment strategy: when a choice point is inferred its evaluation is delayed until the system has no other means to continue the search process. In the latter case one of the suspended choice points is selected and a backtrack point is created. The creation of backtrack points should be avoided as much as possible: in general it holds that the more backtrack points are needed, the less efficient the system is. In order to reduce the number of backtrack points and to improve the selection of the best branch in a backtrack point, we have designed a forward propagation mechanism. This mechanism can determine if a branch in a choice point is disentailed by the current state of the constraint stores. Based on this information, suspended choice points can become deterministic: either one branch is left to explore, or all branches are disentailed meaning that the current state of the system is inconsistent and backtracking should happen.

The basic technique behind this forward propagation is *reification*, which is the association of a boolean variable to a formula. When the formula is (dis)entailed w.r.t. the constraint store the boolean variable is 1 (0), and vice versa. For efficiency purposes, we need to be able to remove reified formulas from the constraint stores. Since this operation was not available in existing constraint solvers, we have extended the reification inference.

We have integrated two constraint solvers in the system: a (dis)equality constraint solver and a finite domain constraint solver. The first is a necessary part of the system. Without doubt, one can state that 'equality reasoning' is the most applied inference in the system. We have implemented a new constraint solver, called $\mathcal{E}$, for equalities and disequalities between terms of a first order language. That had to be done since no existing equality constraint solver had the right properties. In the first prototypes, we applied an extended unification algorithm. That resulted in a 'generate and test' behavior of the system because the disequalities were completely passive in this implementation. An additional motivation for the implementation was the need for reification of disjunctions or conjunctions of (dis)equalities for being able to implement the forward propagation in the search process. With this solver some obvious inefficiencies from the early abductive reasoning systems, such as the 'generate and test' behavior, are avoided.

The integration of the finite domain solver is a different story. Although its integration in the $\mathcal{A}$system has important advantages, e.g. the ability to perform arithmetic reasoning on integers and the ability to solve problems with a finite domain efficiently, the $\mathcal{A}$system hits often the limitations of the solver. In large problem instances, it can happen that the finite domain constraint store is inconsistent and that the constraint solver is unable to detect this. To resolve this, we have added intermediate consistency checks. Since such checks may take too long, we have explored several approximations. These do not guarantee that a store is consistent, but inconsistent stores are earlier detected than without a check. Many of the problems can be brought back to that the constraint store is generated automatically. That is in contrast with the basic assumption of the finite domain solver, that the constraint store is set up by a well designed human program.

Although our work did not result in large computationally advances of the $\mathcal{A}$system, this study was important. Our first experiments with reification used finite domain expressions. Also, the encountered problems are now clearly identified and placed within their context. Finally, our experiences are valuable for later integrations with new constraint solvers, in the sense that one can anticipate possible problems.

These and many other design decisions make the $\mathcal{A}$system a powerful abductive constraint reasoning system. The computational performance of the $\mathcal{A}$system is validated by a number of experiments. We have focussed on artificial problems such as classical Constraint Satisfaction Problems and AI-planning problems since these problems have well-described properties which allow us to analyze more easily the behavior of the $\mathcal{A}$system. Moreover it allows us to compare the $\mathcal{A}$system with related systems.

Of course, the presented implementation is only a first important step towards a general purpose abductive solver. There are many open issues for which intelligent solutions are needed. These range from user support, via improvements on the logical inference to optimizations of the implementation. We will highlight the most important ones.

The system is implemented on top of Sicstus Prolog (use at least version 3.11) and can be downloaded from `http://www.cs.kuleuven.ac.be/~dtai/kt/`. It can be freely used for academic and research purposes.

1.2.2 Integration of databases

One of the main applications of computers is the storage of information. Systems that provide information about a domain are called knowledge bases. It is common that for one domain several knowledge bases are available. Each one has (a part of the) information of the domain. In general, when we want to get information about the domain, the information of several knowledge bases must be combined. This merging of information is also needed for other goals: e.g. when we want to build a new knowledge base that stores the information of a group of knowledge bases.

The problems that arise in this task of integration of knowledge are non-trivial: there may be differences in the alphabets, the merged information may be incomplete or (worser) inconsistent, etc... With regard to any of these problems, there is a vast amount of literature available, describing a large number of different methods. In the second part of the thesis, we will consider the integration problem from a knowledge representation perspective using ID-Logic. In this study, we will identify several tasks that require abductive reasoning to be solved. For several of these we have elaborated small experiments for demonstrating that the $\mathcal{A}$system is able the handle these tasks. Although we will stress our 'knowledge representation' contributions to this problem domain, this extensive study also acts as yet another demonstration of the presence of abductive reasoning in declarative problem solving.

We will restrict ourselves to the most common information source: a database. A database is a structure consisting of an *alphabet*, a *database instance* and a set of *integrity constraints*. The alphabet is used to formalize the information of the domain that is stored by the database. The database instance stores the concrete knowledge of the database. This is sometimes called the extensional knowledge. It is represented as a set of atoms (tuples) of the predicates (relations) in the alphabet that are known to be true. All atoms not in this set are false. The integrity constraints formalize the intentional knowledge of the database. The database instance must satisfy these, otherwise the database is inconsistent.

Our information integration problem can now be made concrete: given a set of databases (called the sources), the goal is to construct a single consistent database whose information is as close as possible to the 'union' of the databases. This integration problem contains the following subproblems:

1. the integration of the alphabets
2. the integration of the integrity constraints
3. the integration of the data instances

Each subproblem requires its own approach. Since the second and third problem focus on the issue of consistency, the same methods can be applied to both. In our thesis, we will consider the first and the last problem. These also occur most frequently in practice.

The first subproblem is tackled in the context of the construction of a virtual integrated database. Instead of materializing the database that contains the integrated data of all sources, the goal is to build a structure such that the information of the sources is derivable on request. The virtual database provides a common view on the information stored in the sources. (Therefore it is commonly called the global database.) This releases the user from the need for reformulation of its queries in each of the alphabets of the sources. Without a global alphabet the user is obliged to query each source individually, which is a tedious and time consuming task. The setup of a virtual database can be done centrally so that many people benefit from it. Another advantage is that the information that is derivable is up to date. When a source updates its information this is immediately reflected in the virtual database. A requirement which is commonly imposed is that the global database has the ability to deal with the addition and deletion of sources in a natural way. This is important since the sources often will be connected with the global database via a communication network.

The main problem in such virtual integration is the formalization of the relationships between the alphabets of the sources and the global alphabet. Each alphabet (that of the sources and the global database) offers a different view on (a part of) the common information domain. This common domain guarantees that there are semantical relationships between relations in one alphabet and another. These are called the *ontological relationships*. Sometimes these are so strong that one relation of one alphabet can be defined in terms of the other alphabet. However, such strong relationship does not always exist. In that case, the relation is only partially definable by the other alphabet. It may happen, however, that several alphabets (i.e. sources) together can define precisely the relation (of the global database). An independent issue is the actual knowledge the sources have about their own relations. A source may have complete or incomplete knowledge about its relations. This (meta-)information is valuable since it may be exploited in the to-be-constructed logic theory. Note that even if all sources have complete knowledge about their relations, the global database may have incomplete knowledge about its relations.

The ultimate goal is the design of a query answering procedure for the global database. In general, there will be incomplete knowledge about some relations in the global database. That makes that we can distinguish between two types of answers for a query. Those entailed by the global database are *certain* answers, while those satisfied by the global database are *possible* answers.

In the literature one distinguishes basically between two approaches. The Global-As-View defines the global database relations in terms of the relations of the sources. The Local-As-View can be regarded as the inverse of Global-As-View, in which the source relations are views on the global database. For both approaches query answering procedures have been defined and implemented in several, some even commercial, systems. The existing work is amalgam of variants of both approaches, each with its own assumptions on the problem context. One reason for this variety is the concernment with the computability of the query answering for a specific problem context. Over-stressing the computability often reduces the conciseness and generality of the approach.

Our work tackles the problem from a knowledge representation perspective. We have designed an ID-Logic based framework in which the knowledge of the global database is the composition of a number of (auxiliary) ID-Logic theories. Our ID-Logic framework covers both Global-As-View and Local-As-View modeling. It turns out that ID-Logic is a very suited logic for this application. All fundamental properties of ID-Logic are exploited: definitions, incomplete knowledge and modularity. The main result of this framework is that it is a step towards a better understanding of the problem in its generality.

At the level of query answering, we will highlight the possible answer semantics. In the literature, very few handle this semantics. In our opinion, the natural answer for a query is in exceptional situations, e.g. when a source is temporary unavailable, given by the possible answer semantics. Procedurally this semantics is related to abductive reasoning or model generation. Abductive reasoning has an advantage over model generation because it computes an explanation formula for the query. By a smart design of the ID-Logic theories in the framework, this explanation is understandable by the user of the global database.

The second problem that we have considered concerns the coherent integration of the databases. For this problem, it is supposed that the databases share the same alphabet and that there is a consistent set of integrity constraints that must be satisfied by the integrated data. When the database instances are merged together (e.g. by taking the union), the new database instance may violate the integrity constraints. Since inconsistency makes all classical logical reasoning trivial, inconsistent databases should not be constructed. In contrast to database updates which handle inconsistencies via the rollback mechanism of transactions, it is not an option to restore the consistency by not applying the integration. It requires either consistency restoring methods, i.e. *coherent integration* for database instances, or the application of logics in which inconsistency does not lead to trivialization of the inference (*para-consistent logics*).

Our approach is a coherent integration method: it defines a *repair* of the inconsistent database. A repair consists of a set of tuples that must be inserted and a set of tuples that must be retracted from the database instance so that the updated database is consistent. Obviously there are many repairs possible. For example the retraction of all tuples from an inconsistent database is a valid repair, but it is not really helpful. Therefore, preference criteria are defined in order to select only the most appropriate ones. The most frequent occurring criteria are the minimal cardinality and the set inclusion criterion. In our work, we formulate the repaired database by means of a meta-theory in ID-Logic. For this theory, the computation of a repair requires abductive reasoning. Hence we can apply the $\mathcal{A}$system. In order to find the most preferred repairs, the $\mathcal{A}$system is extended with an optimization module. The advantages of this formalization are its flexibility (which is shown by some extensions of the problems setting) and its ability to deal with non-ground insertions. Additionally we show that the $\mathcal{A}$system terminates for the computation of the repairs with respect to key and foreign key integrity constraints.

We also present a second approach for the same problem. It is an elegant method in which the repairs are represented as the models of a special constructed propositional theory, called the *signed theory* of an inconsistent database. We show that the preferred repairs according to the set inclusion and minimal cardinality criterion correspond to minimal models (for the corresponding preference criterion) of the signed theories. This encoding allows the use of any (propositional) model generator or satisfiability solver for computing repairs of inconsistent databases. We present results for experiments with several systems from four different computational paradigms.

1.3 Overview of the thesis

The introductory chapter gives an overview of the different logic knowledge representation approaches related to our approach in this thesis. We focus on ID-Logic and Abductive Logic Programming, which form the foundations of this dissertation, and we show how they are related. We also provide an introduction to earlier formalisms on which the work is based, i.e. first order logic, logic programming and constraint logic programming and related formalisms, i.e. Answer Set Programming and Description Logics.

Our contributions are organized in the next two chapters. Chapter 3 presents the implementation of the $\mathcal{A}$system. First, the proof procedure of the $\mathcal{A}$system is presented. This includes the inference rules and soundness, completeness and termination results. Next we discuss the basic framework of the $\mathcal{A}$system (Section 3.4). This section elaborates the first three factors that determine the efficiency of the system. The search process is discussed, followed by the presentation of the main data structures used in the $\mathcal{A}$system. Then, we study the encoding of the inference rules as a meta-program. The next two sections discuss the integration of the constraint solvers in the $\mathcal{A}$system. Since there was no efficient equality solver with the right properties available, we have implemented ourselves a new equality solver, the $\mathcal{E}$-solver. This constraint solver is presented in Section 3.5. The integration of the finite domain constraint solver is the subject of Section 3.6. Here we mainly discuss our solutions to deal with the incompleteness of the constraint solver. Section 3.7 discusses the improvement of the reification reasoning. Section 3.8 presents a small optimization for a special class of abducibles: open functions.

Finally, we demonstrate the current computational efficiency of the $\mathcal{A}$system and compare the system with related systems (Section 3.9). Possible future work is presented in Section 3.10.

Chapter 4 presents our contributions to the integration of databases. After the preliminaries, Section 4.2 presents our solution for the construction of a virtual global database. The work on coherent integration of databases is presented in Section 4.3. The approach based upon the signed theories is presented in Section 4.4.

The thesis is concluded with a reflection on our contributions in Chapter 5.

Publications and co-authors

The work on the $\mathcal{A}$system is joint work with M. Denecker and A. Kakas. Only part of the presented advances in the implementation of the $\mathcal{A}$system has been published. An early prototype of the $\mathcal{A}$system, called SLDNFAC, has been published at the ERCIM'99 workshop [103]. There we considered the extension of SLDNFA with finite domain constraints. The $\mathcal{A}$system itself has been described at the IJCAI'01 conference [163]. At the NMR'00 workshop we presented an extension with aggregates [263]. (This extension is not integrated in the thesis.) Additionally, we have been presented our system at several locations: NMR'00 [258], LPNMR'01 [264] and BNAIC'01 [164].

The work on the integration of databases is joint work with O. Arieli, M. Bruynooghe and M. Denecker. The work on coherent integration of databases has been the subject of several publications: at the LPAR'01 conference [26], at the PCL'02 workshop [23] and the FOIKS'04 conference [25]. The work of [26] and [23] has been integrated in an article in the Journal of Artificial Intelligence Research [24]. The most recent work is the work on the ID-Logic framework for mediator based systems. This is presented at the CAISE'04 conference [262]. To the latter work A. Cortés-Calabuig also has contributed.

Other publications, also not integrated in the thesis, include the work on the transformation of object oriented models to ID-Logic, which was joint work with P. Bekaert, D. Gilis, M. Bruynooghe and M. Denecker. It has been presented at the ER2002 conference [42, 41]. Some other publications are a study of preferential reasoning [260], and two poster presentations [259, 261].

Chapter 2

Abductive Constraint Logic Programming

Fifty years of research in Declarative Programming Languages and Knowledge Representation has produced a lot of insight and a large amount of formalisms. One of these is Abductive Logic Programming, a paradigm that is central in this dissertation.

This introductory chapter's main purpose is to provide the background for the work presented in the thesis. It starts with an overview of basic notions of First Order Logic, Logic Programming and Constraint Logic Programming.

In the main part of the chapter, we introduce the knowledge representational and computational context of the thesis. First we present the knowledge representation logic ID-Logic, an extension of classical logic with non-monotone inductive definitions. This logic has emerged from research on epistemological and knowledge representational aspects of Abductive Logic Programming.

The presentation of ID-Logic is followed by an introduction to Abductive Logic Programming. Abduction is a computational paradigm that is suited for reasoning with incomplete knowledge. More specifically, abduction is the process of deriving plausible hypothesizes to explain an observation with respect to given background knowledge. The derived hypothesizes 'complete' the gaps in the knowledge in a sensible way.

The relation between ID-Logic and Abductive Logic Programming is elaborated next. We present a transformation of ID-Logic theories into Abductive Logic theories, which forms the technical contribution in this chapter. By this transformation, abductive solvers such as the $\mathcal{A}$system, the abductive solver which we have implemented, can be used to solve abductive problems specified in ID-Logic (provided that the formal semantics coincide).

The section on Abductive Logic Programming is concluded with an overview of the abductive systems that have been proposed in the literature and an overview of the different integrations with Constraint Logic Programming.

The chapter is concluded with a short overview of two related approaches: Description Logics and Answer Set Programming.

2.1 Preliminaries

This section summarizes the basic notions of First Order Logic and (Constraint) Logic Programming. We discuss first order languages, the basic notions of interpretations and models of First Order theories and finally Logic Programming. For a detailed overview on these subjects, the reader is referred to [187]. These preliminaries are then concluded with an introduction to an extension of Logic Programming: Constraint Logic Programming. For Constraint Logic Programming, we refer to [149, 141, 234, 191, 13].

2.1.1 First Order Languages

The basis of any first order language is its alphabet.

DEFINITION 2.1 An *alphabet* is formed by these classes of symbols:

1. variables e.g. $\{X, Y, Z, \ldots\}$
2. constants e.g. $\{a, b, c, \ldots\}$
3. function symbols e.g. $\{f/n, g/m, \ldots\}$
4. predicate symbols e.g. $\{p/n, q/m, \ldots\}$
5. connectives e.g. $\{\neg, \wedge, ',', \vee, ';', \leftarrow, \rightarrow, \leftrightarrow, \ldots\}$
6. quantifiers e.g. $\{\forall\ , \exists\ \}$
7. punctuation symbols e.g. $\{'(', ')', '.', \ldots\}$

Function and predicate symbols (also called the functors and predicates) always have a corresponding arity. Above we used a common notation: f/n represent the function name f with arity n (and similar for predicates). We will use sometimes an alternative notation: $f(\cdot,\cdot)$, denoting $f/2$. Since usually only the first four classes are problem dependent and designed by the user (the other normally have a fixed meaning), an alphabet is used to refer to only those four symbols sets. A synonym for an alphabet is a *signature*.

Terms are expressions built from variables, constants and function symbols.

DEFINITION 2.2 A *term* is

- a variable X, or
- a constant c, or
- a compound term $f(t_1, \ldots, t_n)$ where f is a function symbol with arity n and t_i are terms

A tuple of variables $X_1, X_2, X_3, \ldots$ (resp. terms $t_1, t_2, \ldots$) is denoted as $\overline{X}$ (resp. $\overline{t}$). For example $f(\overline{t})$ is a shorthand for $f(t_1, \ldots, t_n)$.

DEFINITION 2.3 An *atom* is $p(t_1, \ldots, t_n)$ with p a predicate symbol of arity n and $t_1, \ldots, t_n$ terms.

A term (or atom) is called *ground* when it contains no variables. When it contains at least one variable, it is often called *nonground*.

Atoms can be combined in expressions.

DEFINITION 2.4 A basic *well-formed formula* is an atom. If F and G are well-formed formulas and X is a variable, then

- (connectives) $\neg F, F \wedge G, F \vee G, F \leftarrow G, F \rightarrow G, F \leftrightarrow G$
- (quantifiers) $\forall\ X.F$ and $\exists\ X.F$

are well-formed formulas.

DEFINITION 2.5 A *First Order Language* $\mathcal{L}$ based on an alphabet A is the collection of all well-formed formulas that can be constructed from that alphabet A.

The following terminology is adopted concerning variables and quantifiers.

DEFINITION 2.6 A quantifier $\forall$, $\exists$ *binds* the variable X in the formula $\forall\ X.F$ (resp. $\exists\ X.F$). The *scope* of ∀ X (∃ X) in these formulas is F. If a variable is not bound by a quantifier in a formula it is called *free*.

DEFINITION 2.7 A formula F is *closed* when none of the variables in F occurs free. ∀(F), resp. ∃(F), denote the *universal* (resp. *existential*) *closure* of the formula F. This formula is obtained by adding a universal (existential) quantifier for every free variable of F in front of F.

A closed formula is also called a *sentence*.

DEFINITION 2.8 A first order logic *theory* based on a first order language $\mathcal{L}$ is a set of closed formulas in the language $\mathcal{L}$.

2.1.2 Models and Interpretations

The previous section defined the syntax of a first order language. Now we are giving the formulas a meaning.

DEFINITION 2.9 A *pre-interpretation* J of a first order language $\mathcal{L}$ is defined by

- a *domain* D: a non-empty set of objects
- a set J_c containing an assignment to an element of D for each constant of $\mathcal{L}$.
- a set J_f containing a mapping from D^n to D for each function symbol f/n of $\mathcal{L}$.

DEFINITION 2.10 An *interpretation* of a first order language $\mathcal{L}$ is a pre-interpretation J (having a domain D) augmented with an assignment from D^n to $\{\textbf{true, false}\}$ for each predicate p/n of $\mathcal{L}$.

DEFINITION 2.11 A *variable assignment* α is a function, mapping all variables to elements of the domain D. By $F(X/d)$, we denote that formula obtained by replacing each free occurrence of the variable X by the domain element d in the formula F. Let J be a pre-interpretation with domain D. A *term assignment* $termeval_{J,\alpha} : t \mapsto d$ maps a term t to domain element d and is inductively defined as

- $termeval_{J,\alpha}(c) = c_J$ for any constant c (c_J is the corresponding domain element assigned by J_c)
- $termeval_{J,\alpha}(X) = \alpha(X)$ for any variable X
- $termeval_{J,\alpha}(f(t_1, \ldots, t_n)) = f_J(termeval_{J,\alpha}(t_1), \ldots, termeval_{J,\alpha}(t_n))$ for any function f/n (f_J/n is the function assigned to f/n by J_f)

DEFINITION 2.12 Let I be an interpretation of $\mathcal{L}$ with domain D and α be a variable assignment w.r.t. the same domain D. A truth function $eval_{I,\alpha} : \mathcal{L} \mapsto \{true, false\}$ assigns a truth value to a formula F. It is inductively defined as

- $eval_{I,\alpha}(p(t_1, \ldots, t_n) = p_I(termeval_{I,\alpha}(t_1), \ldots, termeval_{I,\alpha}(t_n))$ for any predicate p/n, where p_I is the relation assigned by the interpretation I to p.
- $eval_{I,\alpha}(F \wedge G) = eval_{I,\alpha}(F) \wedge eval_{I,\alpha}(G)$, for any formulas F and G. The rules for the formulas $F \vee G$, $F \rightarrow G$, $F \leftrightarrow G$ and $\neg F$ are defined analogously.
- $eval_{I,\alpha}(\exists\ X.F) = eval_{I,\alpha}(F(X/d))$ where $F(X/d)$ denotes the replacement of each occurrence of the variable X in the scope of the quantifier $\exists$ by a domain element $d \in D$.

- $eval_{I,\alpha}(\forall\, X.F) = \bigwedge_{d \in D} eval_{I,\alpha}(F(X/d))$

Note that for any closed formula the truth value does not depend on the variable assignment α. For a sentence F and interpretation I, we write $I \models F$ to denote $eval_{I,\alpha}(F) = eval_I(F) = \mathbf{true}$.

DEFINITION 2.13 A *model* M of a theory $\mathcal{T}$ based on the language $\mathcal{L}$ is an interpretation of $\mathcal{L}$ such that for each formula F of $\mathcal{T}$: $M \models F$.

An important class of interpretations are *Herbrand interpretations.*

DEFINITION 2.14 For a language $\mathcal{L}$, the *Herbrand Universe* $\mathcal{HU}(\mathcal{L})$ is the set of all ground terms constructible from $\mathcal{L}$. The *Herbrand Base* $\mathcal{HB}(\mathcal{L})$ is the set of all ground atoms that can be formed by $\mathcal{L}$.

DEFINITION 2.15 A *Herbrand pre-interpretation* for $\mathcal{L}$ is the pre-interpretation composed of

- the Herbrand Universe $\mathcal{HU}(\mathcal{L})$ as domain,
- the identity mapping for constants: $c \mapsto c$ for each constant c of $\mathcal{L}$,
- the mapping $\mathcal{HU}^n(\mathcal{L}) \mapsto \mathcal{HU}(\mathcal{L}) : (t_1, \ldots, t_n) \mapsto f(t_1, \ldots, t_n)$ for each function symbol f/n of $\mathcal{L}$.

A *Herbrand interpretation* $\mathcal{H}$ is an interpretation based on the Herbrand pre-interpre-tation of $\mathcal{L}$.

NOTATION 2.1.1 Depending on the needs, one finds different representations for an Herbrand interpretation. Consider a (propositional) language consisting of the propositional predicate symbols a, b, c. The first representation $I = \{a^{\mathbf{f}}, b^{\mathbf{t}}, c^{\mathbf{t}}\}$ denotes an interpretation I in which the atom a is false, and b and c are true. When only two-valued interpretations are considered, it is common to present only the true facts and assume that all others which are not mentioned are false. I is then represented as $\{b, c\}$. The empty interpretation $J = \emptyset$ denotes that every atom is false.

2.1.3 Logic Programming

Logic programs

Logic Programming considers logic theories of a specific form. The theories, called logic programs, are composed of the formulas called program rules.

DEFINITION 2.16 A *program rule* is a formula of the form

$$H \leftarrow B$$

where H is an atom[1] and B a formula.

H is called the *head* and B the *body* of the rule. When nothing is stated of the quantification of the variables, the variables of a program rule are implicitly universally quantified as: $\forall\ (H \leftarrow B)$. Conventionally, an empty head is equal to **false**. Such a program rule is called a *denial*. The empty body is **true**. In that case a program rule is called a *fact*. Note that a program rule with an empty head and body is inconsistent.

Usually B is restricted to a conjunction of literals: $l_1 \wedge \ldots \wedge l_n$. From now on, this is assumed, unless explicitly the contrary is specified. A *literal* is an atom or the negation of an atom. The notion *positive literal* refers to an atom, and a *negative literal* is a negated atom. In many Logic Programming systems the conjunction $\wedge$ is denoted by the ',' and a program rule is delimited with a dot '.' at the end.

When all body literals are positive literals the program rule is called *definite*. It is called *normal* when the body allows negative literals. A logic program is called definite if all its program rules are definite. When it consists of normal program rules it is a *normal* logic program.

REMARK 2.1.1 Sometimes program rules are referred to as *clauses*. Classically a clause is defined as a formula of the form $\forall\ \overline{X}.A_1 \vee \ldots \vee A_k \vee \neg B_1 \vee \ldots \vee \neg B_n$ where $\overline{X}$ are all variables of the literals A_i and B_i [187]. The formula is equivalent with $\forall\ \overline{X}.(A_1 \vee \ldots \vee A_k \leftarrow B_1 \wedge \ldots \wedge B_n)$ where $\leftarrow$ is the classical implication.

The latter notation has become the standard representation in Logic Programming for definitional knowledge, although that often $\leftarrow$ in a program rule does not correspond to the implication. In this dissertation we take the convention:

- '$\rightarrow$' denotes always classical implication.
- '$\leftarrow$' appearing in a program rule $p \leftarrow B$ with a *non-empty head* will *never* be interpreted as the implication.
- '$\leftarrow$' appearing in a program rule $\leftarrow B$ with an *empty head* will denote the implication. In order to distinguish with the previous case we will explicitly write the universal quantifier: $\forall\ \overline{X}. \leftarrow B$.

[1] In some Logic Programming formalisms, the head H is allowed to be a disjunction or a negated atom.

Formal semantics

The above remark already points to the possibility of different interpretations of the symbols. This is reflected in the vast amount of formal semantics that have been developed for Logic Programs. Commonly these are given in the form of fixpoint semantics: these define for the logic program an operator that acts on interpretations. A selection of the fixpoints of these operators are defined as the models of the logic program. Without going into details, we mention the most important ones. We start with definite logic programs.

DEFINITION 2.17 (THE IMMEDIATE CONSEQUENCE OPERATOR [187]) The immediate consequence operator T_P for a definite logic program P based on the language $\mathcal{L}$ is defined as: let I be an Herbrand interpretation of $\mathcal{L}$,

$$T_P(I) = \{p(\overline{t})^{\mathbf{t}} | p(\overline{t}) \leftarrow l_1 \wedge \ldots \wedge l_n \in P \text{ and each literal } l_i \text{ is true in } I, 1 \leq i \leq n\}$$

An interpretation I is more true than an interpretation J (denoted as $I \leq_t J$) if the set of all true atoms of I^t is a superset of that of J: $I^t \supseteq J^t$. Since $T_P(I)$ is monotonic for definite logic programs, i.e. for all Herbrand interpretations I and J, if $I \leq_t J$ then $T_P(I) \leq_t T_P(J)$, it has a fixpoint. This is due to the Knaster-Tarski theorem [247]. The least fixpoint $lfp(T_P(\emptyset))$ is given by $T_P \uparrow \omega(\emptyset)$.

DEFINITION 2.18 (DEFINITE LOGIC PROGRAMS) The formal semantics for a definite logic program P is given by the least Herbrand model of P.

The least Herbrand model is characterized as the least fixpoint of the immediate consequence operator $lfp(T_P(\emptyset))$. It holds that for definite logic programs this least Herbrand model is unique.

For logic programs containing negation the following three semantics are the most influential ones.

A) The *completion* semantics (Clark [78]) defines the models of P as the least Herbrand models of the first order logic theory consisting of

- The completion of the program rules. The completion is a set of iff-formulas obtained through the following transformation:
  (a) First a program rule normalization such that the arguments of the head are all variables. Consider the program rule

$$\forall\, \overline{Z}.p(t_1, \ldots, t_n) \leftarrow B$$

It is transformed to

$$p(X_1, \ldots, X_n) \leftarrow \exists\, \overline{Z}.X_1 = t_1 \wedge \ldots \wedge X_n = t_n \wedge B$$

where $X_1, \ldots, X_n$ are new variables not occurring in the original program rule. If a predicate has more than one defining rules, the newly introduced variables are chosen such that they are unique w.r.t. to all program rules of p, so that these can be used in each single program rule transformation.

(b) Followed by, for each predicate p appearing in the head of a program rule, the construction of the iff-formula

$$p(\overline{X}) \leftrightarrow B_1 \vee \ldots \vee B_k$$

where the predicate p appears in the head of k defining program rules. $\overline{X}$ are the variables introduced by the above transformation, and $B_1, \ldots, B_k$ are the resulting bodies.

- The Clark Free equality axioms ($\mathcal{FEQ}$)
  - for each pair of different constants c_1 and c_2, the axiom $c_1 \neq c_2$
  - for each pair of different function symbols f/n and g/m, the axiom $\forall\ f(X_1, \ldots, X_n) \neq g(Y_1, \ldots, Y_m)$.
  - for each function symbol f/n, the axiom

    $$\forall\ f(X_1, \ldots, X_n) = f(Y_1, \ldots, Y_n) \rightarrow X_1 = Y_1 \wedge \ldots \wedge X_n = Y_n$$

  - for each term t containing the variable X, the axiom $\forall\ t \neq X$.

We will use later a weaker variant of the completion semantics: the three-valued completion semantics. It is denoted as $comp_3$. Three-valued means that in addition to the truth values **true** and **false**, the value **undefined** is allowed. **undefined** means that there is lack of information to fix the truth. For this semantics the $\leftrightarrow$ is interpreted as the three-valued equivalence operator.

B) The *Stable Model semantics* are due to Gelfond and Lifschitz [134], who define it by means of a reduct for a propositional normal logic program.

Let P be a propositional normal logic program based on the language $\mathcal{L}$. For an interpretation I, P_I is the propositional logic program obtained by

- removing from P all rules with a literal $\neg l$ in the body such that l is true in I, and then
- removing from remaining rules the negative literals in the remaining bodies

The program P_I is a definite logic program. The Gelfond-Lifschitz operator $GL_P(I)$ is defined as the $lfp(T_{P_I}(\emptyset))$. Intuitively $GL_P(I)$ computes for an interpretation I all positive consequences under the assumption that the negative information of I is an over-estimate.

DEFINITION 2.19 (STABLE MODEL [134]) An interpretation I is a stable model for the propositional logic program P iff it is a fixpoint of $GL_P(I)$.

Note that $GL_P(I)$ is not monotone, but *anti-monotone*: if $I \leq_t J$ then $GL_P(I) \leq_t GL_P(J)$.

This definition can be extended to logic programs over a first order logic language, by first grounding the programs.

C) The last formal semantics is the *well-founded semantics* as defined by Van Gelder et al. [257]. It is also a three-valued semantics.

To define the well-founded semantics, we follow the alternative definition of Van Gelder in [133] based on the stable model operator. The operator $A_P(I)$ is defined as the operator $GL_P(GL_P(I))$. $A_P(I)$ is again monotone, and hence it has a least (and greatest) fixpoint. Consider the pair $(lfp(A_P(\emptyset)), gfp(A_P(\emptyset)))$, where the first argument is a under-estimate and the second as an over-estimate of all stable models. The well-founded model is defined as the interpretation assigning **true** to the atoms that are true in the under-estimate $lfp(A_P(\emptyset))$, **false** to the atoms that are false in over-estimate $gfp(A_P(\emptyset))$ and **undefined** to all other atoms. Well-founded refers to the fact that for an atom (formula) to be true there must be a well-founded support, e.g. a finite number of inference steps.

THEOREM 1 ([133])
Each logic program has a unique well-founded model.

PROOF 2
The result follows trivially from the definition. Since $A_P(I)$ is monotone it has a least and greatest fixpoint. Hence the pair exists and thus the well-founded model exists.

THEOREM 3 ([133])
The well-founded model is two-valued iff there exists a unique stable model.

PROOF 4
(The if-direction) The two-valuedness implies that the least fixpoint and greatest fixpoint of $A_P(I)$ are equal. It is known that $GL_P(lfp(GL_P(I))) = gfp(GL_P(I))$ and vice versa. (This is called an occilating pair.) Since $gfp(A_P(\emptyset)) = lfp(A_P(\emptyset))$, $GL(lfp(A_P(\emptyset))) = gfp(A_P(\emptyset)) = lfp(A_P(\emptyset))$. $lfp(A_P(\emptyset))$ is thus a fixpoint of $GL_P(I)$, proving the theorem.

(The only-if-direction) The other direction immediately follows from the uniqueness.

Procedural semantics

Having fixed the semantics of a logic program, the next goal is to solve tasks with the specified knowledge. In Logic Programming, a task is expressed by the notion of query answering. A first order formula $\mathcal{Q}$, called the query, can be related in different ways with respect to a logic program $\mathcal{P}$ and a semantics S:

- $\mathcal{Q}$ is *satisfiable* if there exists a S-model M of $\mathcal{P}$ such that $\mathcal{Q}$ is true w.r.t. M.
- $\mathcal{Q}$ is *valid* if $\mathcal{Q}$ is true in all S-models of $\mathcal{P}$.
- $\mathcal{Q}$ is *unsatisfiable* if for no S-model of $\mathcal{P}$ $\mathcal{Q}$ is true.
- $\mathcal{Q}$ is *nonvalid* if there exists a S-model M of $\mathcal{P}$ such that $\mathcal{Q}$ is false w.r.t. M.

One says that the logic program $\mathcal{P}$ *entails* $\mathcal{Q}$ if $\mathcal{Q}$ is valid w.r.t. $\mathcal{P}$. It is denoted as $\mathcal{P} \models_S \mathcal{Q}$ where S denotes the chosen semantics. When the semantics is clear from the context, the subscript is omitted.

Computing these relations with respect to expressive logic programming formalisms is in general undecidable. Therefore one limits the expressiveness of languages and develops for these efficient computational techniques. A milestone in these techniques is the refutation principle as described by Robinson [228] and refined by Kowalski [168]. As it also forms the basis of the $\mathcal{A}$system, we sketch it. First we need the notion of a variable substitution.

Definition 2.20 A *substitution* σ is a finite set of the form $\{X_1/t_1, \ldots, X_n/t_n\}$, where each X_i, i=1..n, is a variable, each t_i, i=1..n, is a term and $X_i \neq t_i$. The empty substitution is denoted as ϵ. The set $\{X_1, \ldots, X_n\}$ of a substitution σ is called the domain $Dom(\sigma)$. If all $t_1, \ldots, t_n$ are ground, σ is called a *ground substitution*. The application of a substitution σ on a formula F results in a formula denoted as $F\sigma$, which is obtained by the simultaneous replacement of all free occurrences of the variables $X_1, \ldots, X_n$ in F by their corresponding term given by σ. Also, substitutions can be composed: let $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$ and $\theta = \{Y_1/s_1, \ldots, Y_m/s_m\}$ two substitutions, then the composed substitution $\sigma\theta$ is defined as the set $\{X_i/t_i\theta | 1 \leq i \leq n \wedge X_i \neq t_i\theta\} \cup \{Y_i/s_i | 1 \leq i \leq m \wedge Y_i \notin Dom(\sigma)\}$

Note that the following properties hold for substitutions: (a) $\sigma\epsilon = \epsilon\sigma = \sigma$, (b) $(\theta\sigma)\gamma = \theta(\sigma\gamma)$ and (c) $(F\theta)\sigma = F(\sigma\gamma)$ for all formulas F.

A basic operation in resolution is unification, which is based on substitutions.

Definition 2.21 A substitution σ is a *unifier* for a set of expressions S if $S\sigma$ is a singleton. σ is the *most general unifier* (mgu) of S iff for each unifier θ of S, there exists a substitution γ such that $\theta = \sigma\gamma$.

The computation of the most general unifier for two terms is a common issue in proof procedures. For an overview of the different algorithms, see [29].

Intuitively, to prove that a formula F holds, resolution based procedures try to compute a contradiction from the assumption that $\neg F$ holds. This is based on the observation that $\mathcal{T} \models F$ if only if $\mathcal{T} \cup \{\neg F\}$ is unsatisfiable. To visualize this point-of-view, a goal F is denoted as $\leftarrow F$, a shorthand for **false** $\leftarrow F$. Thus, starting from the initial goal $\leftarrow F$, a resolution proof procedure attempts to find the empty goal $\leftarrow$ *true* (which is a contradiction).

These intuitions are made more precise in the following definitions. We will explain the process only for definite logic programs.

Definition 2.22 (SLD-resolution) An SLD-resolution (which stands for Linear resolution with Selection function for Definite clauses) can be described as follows: Let G be a definite goal $\forall\, \overline{X}. \leftarrow l_1 \wedge \ldots \wedge l_n$ in which l_i, for some $1 \leq i \leq n$, is selected. The selected literal l_i is an atom $p(\overline{s})$ and $p(\overline{t}) \leftarrow B_1 \wedge \ldots \wedge B_m$ is a selected program rule of the considered program (renamed if needed so that its variables are different from $\overline{X}$). Then G' is derived from G as the formula $\forall\, \overline{X}. \leftarrow (l_1 \wedge \ldots l_{i-1} \wedge B_1 \wedge \ldots \wedge B_m \wedge l_{i+1} \wedge \ldots \wedge l_n)\theta$ in which θ is the mgu of the atoms $p(\overline{s})$ and $p(\overline{t})$.

Often the term *unfolding* is used to refer to a resolution inference step. A predicate p is unfolded in a formula F means that the selected atom $p(\overline{t})$ is replaced by its body. In the case of SLD-resolution it is only by one of the clauses at a time, while in other contexts it might mean the replacement with the disjunction of all its clauses.

Definition 2.23 (SLD-derivation) Let $\mathcal{P}$ be a definite logic program and $\mathcal{Q}$ a definite query. An *SLD-derivation* is a sequence of goals $G_0 = \mathcal{Q}, G_1, G_2, \ldots$ in which each goal G_{i+1} is derived by a SLD-resolution step from the previous goal G_i based on a program rule C_{i+1} and a mgu θ_{i+1}. An *SLD-refutation* is a finite SLD-derivation of $\mathcal{P} \cup \{\mathcal{Q}\}$ which ends in the empty clause $\leftarrow$ *true* ($\Box$).

Recall that a definite logic program has a unique model, which is the least Herbrand model.

Definition 2.24 Let $\mathcal{Q}$ be a query and $\mathcal{P}$ a definite logic program. Then, a substitution σ is a *correct* answer for $\mathcal{Q}$ w.r.t. $\mathcal{P}$ iff $\mathcal{P}$ entails $\forall\, (\mathcal{Q}\sigma)$.

Using the SLD-proof procedure, answers substitutions can be computed. The answer substitution for the variables of $\mathcal{Q}$ is the composition of all mgu's $\theta_1 \circ \theta_2 \circ \ldots \circ \theta_n$ computed in the refutation for $\mathcal{P} \cup \{\mathcal{Q}\}$. Both notions of an answer coincide, as SLD is sound and complete for definite logic programs.

THEOREM 5 (SOUNDNESS OF SLD [187])
Let $\mathcal{P}$ be a definite logic program and $\mathcal{Q} = \leftarrow A_1 \wedge \ldots \wedge A_k$ a definite goal. Suppose that $\mathcal{P} \cup \{\mathcal{Q}\}$ has a SLD-refutation and σ is the computed answer substitution, then σ is a correct answer for $\mathcal{P} \cup \{\mathcal{Q}\}$.

THEOREM 6 (COMPLETENESS OF SLD [187])
Let $\mathcal{P}$ be a definite logic program and $\mathcal{Q}$ a definite goal. If σ is a correct answer for $\mathcal{P} \cup \{\mathcal{Q}\}$, then there exists a substitution γ such that for the computed answer θ of the SLD-refutation for $\mathcal{P} \cup \{\mathcal{Q}\}$, holds that $Q\sigma = Q\theta\gamma$.

SLD-resolution has several advantages which have led to the development of the Prolog system (*PROgrammation en LOgic*) [82]. The method allows to derive the necessary information only by applying a single inference step. That simplifies the computerized automation. Moreover it holds for a sufficient expressive language which makes it attractive for programmers. Over the years, the Prolog systems have been extended with a lot of features and other reasoning paradigms, but the core of Prolog is still SLD-resolution.

SLD-resolution is however limited because it does not handle negation. The extended variant of SLD that handles negation is called SLDNF, standing for SLD-resolution with Negation as Failure. For a survey on SLDNF, see [14]. Despite the many results in this area, the practice shows that most Prolog systems only offer a negation-as-failure rule which is only sound for ground negative literals.

2.1.4 Constraint Logic Programming

Constraint Logic Programming [149, 141, 234, 191, 13] is an extension of logic programming in which some parts of the modeling language are predefined. The predefined part is called the *constraint domain*. Some classical constraint domains are e.g. real numbers arithmetic, term equations and finite domain variables reasoning. Each constraint domain provides a set of relations, called *constraints*. A problem is modeled by a set of variables denoting the unknown elements of the problem and a set of constraints (the *constraint store*) restricting the possible values of the variables. Because the meaning of the constraint domain is fixed, special domain dependent solvers are constructed to compute the solutions of the problem. Such a *constraint solver* exploits the properties of the constraint domain in order to obtain an efficient construction of a solution. Typically the constraint based solution construction is some magnitudes faster than by a resolution based solver. Moreover, constraint solvers can handle some types of reasoning, e.g. arithmetic reasoning, which is difficult or impossible to be dealt with by resolution.

Constraint Logic Programming (CLP) merges in some sense two problem modeling/solving domains. The constraint solving domain mainly focuses on building efficient algorithms for specific languages, whereas Logic Programming merely aims at representing the problem knowledge in a declarative way in a general language. The CLP framework integrates both approaches in such a way that the user can take advantage of the representational capacities of Logic Programming and the computational efficiency provided by the constraint solvers.

Formally, a constraint domain $\mathfrak{D}$ is the triple $(\Sigma, \mathcal{D}, \mathcal{T})$. Σ is the alphabet (also called the *signature*) of constant, function and predicate symbols. The Σ-structure $\mathcal{D}$ is the pre-interpretation based on a set D of objects: it contains assignments of the constant symbols to the objects of D and for each function (predicate) symbol an assignment to a corresponding function (relation). The last part $\mathcal{T}$ is an axiomatization of (some) properties of $\mathcal{D}$ based on Σ. Most constraint domains require some extra properties e.g. that there is one equality predicate corresponding to the identity interpretation in $\mathfrak{D}$, or that Σ is closed under variable renaming, etc.

The Constraint Logic Programming framework CLP($\mathfrak{D}$) embeds the constraint domain $\mathfrak{D}$ in Logic Programming. Consider a user defined language Π, consisting of function symbols and predicate symbols, disjoint from the constraint domain $\mathfrak{D}$. Then a CLP($\mathfrak{D}$) program consists of program rules

$$p_0 \leftarrow c_1, \ldots, c_n, p_1, \ldots, p_m$$

where p_i, i=0..m, are atoms belonging to Π and c_j, j=1..n, are atoms belonging to $\mathfrak{D}$. The formal semantics of a CLP($\mathfrak{D}$) program is an extension of those of classical logic programs, but the constraint atoms are evaluated with respect to the constraint domain.

The mix of a flexible declarative modeling language and an efficient solution construction makes CLP attractive. It is successfully applied to solve many (industrial) problems e.g. scheduling problems such as resource allocation, tournament scheduling and timetabling problems. Typically, these problems are highly combinatorial without a clear efficient algorithm to solve them. Even when such a specific algorithm can be developed, it is often of limited use: in many applications the requirements of the problem vary regularly so that the single efficient algorithm is not suited anymore and must be adapted to the new needs which demands a serious effort. In [193], different problem solving methodologies are compared for a high school exam scheduling problem. The comparison learns that the CLP framework is able to cope effectively with the changes in the problem specification, which is a direct effect of its more declarative representation style. However, CLP is less adequate when the focus is on the construction of an optimal solution. In that case, the goal is not to find just a solution, but one which is as good as possible. For example, in exam scheduling problems it happens that the problem is over-constrained, i.e. the problem is unsatisfiable. Then, the goal is changed to find a schedule that violates as few as possible constraints. For such problems other paradigms such as genetic algorithms, are often more suited.

The CLP framework is implemented by extending Prolog with constraint libraries. Most Prolog systems provide nowadays a library for rational (real) numbers CLP(Q) (CLP(R)). Another popular constraint domain is the finite domain constraint domain (denoted by CLP(FD)). Because this constraint domain is applicable in many problem domains of our interest, e.g. planning and scheduling, the finite domain constraint domain is integrated in the $\mathcal{A}$system. Below we will highlight some properties of this constraint domain.

The addition of such efficient constraint solvers to Prolog leads to a shift in the modeling style. One could expect a "balanced" problem modeling style in which a problem is partially encoded by constraints and partially by the logic of the program rules, but it turns out that mostly the problem is formulated in terms of the constraints (notable in case of the FD-domain) and that the logic of the program rules is reduced to a variable generator and as the glue between the different constraints. This reduced use of the logic programming part is easily replaced by an imperative or object oriented programming language such as C or JAVA. In this way a smooth co-operation and integration of constraint solvers within more commonly used programming languages is obtained.

Finite Domain Constraint Logic Programming

Undoubtly, the Finite Domain constraint domain is one of the most important constraint domains that have been developed. With this constraint domain, complex problems such as scheduling, construction of tournament schedules, timetabling, resource planning, etc... are expressible and efficiently solvable by the implemented solvers.

Finite domain refers to the association of each variable with a finite set of values (typical integers), called the domain of that variable. Usually the following classes of constraints are supported[2]:

- membership: e.g. `X in 1..6`
- arithmetic constraints: e.g. `X#<Y` $(X < Y)$, `X*5#\=Y*Z+4` $(X \times 5 \neq Y \times Z + 4)$
- logical connectives: e.g. `X#=>Y` $(X \rightarrow Y)$, `X#>5 #\/Y#\=Y` $(X > 5 \vee Y \neq Y)$
- combinatorial or *global* constraints: e.g. `all_different([X,Y,Z])` $(X \neq Y \wedge X \neq Z \wedge Z \neq Y)$ These constraints encode a specific constraint network compactly.

A solution for a finite domain constraint problem is an assignment of a value from the variable's domain to each variable such that all constraints of the constraint store are satisfied. The construction of a solution is in general a hard task (some are NP-hard). Enumerating all possible candidate solutions and testing them afterwards is infeasible, because the search space exponentially grows with the number of variables. Fortunately, the application of pruning techniques reduces the search space and allows to construct a solution.

[2]According Sicstus Prolog syntax

Within the finite domain constraint domain, the pruning is usually performed by a consistency algorithm that is associated with a particular constraint. A consistency algorithm eliminates for a set of variables the inconsistent values from the domains of the involved variables. There exists many (variants of) consistency algorithms; an often used classification depends on the number of involved variables: node consistency (for constraints involving one variable), arc consistency (binary constraints) e.g. AC3 [189], etc. Because very good consistency algorithms are computationally expensive, usually the pruning in a finite domain constraint solver is limited to at most a variant of arc-consistency. Consequently the pruning will not remove all inconsistent values and as a consequence a final search procedure has to be applied. This is commonly called the *labeling*. Another consequence is that a finite domain solver is incomplete, where the notion of incompleteness means the following:

DEFINITION 2.25 A constraint solver is *complete* when it is able to decide at any time whether a given constraint store is satisfiable or not. If not, it is *incomplete*.

This means that a constraint store before the final labeling might be inconsistent without the detection by the solver (in case the applied consistency algorithms).

Many constraint solvers e.g. the one of Sicstus Prolog [66], offer different versions of the same constraint with a different kind of pruning. The amount of pruning depends, thus, on the chosen representation for a problem. That is also the reason of existence of the *global* constraints. These are not only compact notations for a certain constraint network, but also apply additionally a more effective algorithm such that more pruning is achieved with less work than the original constraint network (usually specified in the form of simple binary constraints). For example, the `all_different([X,Y,Z])` constraint represents the network $X\neq Y \wedge Y\neq Z \wedge Z\neq X$. When each domain of the variables `X,Y,Z` is the set $\{1,2\}$, the network is clearly inconsistent. This is not detected by an arc-consistency algorithm, but the global constraint does so. The applied algorithm is a maximal flow algorithm. This is typical for global constraints: the used algorithms often originate from other areas. Current on-going research goes even further: it designs hybrid problem solvers [115, 227] in which solvers from different areas e.g. linear programming, local search and first order constraint solvers are interconnected such that the solving efficiency is improved.

As explained, the last statement in a finite domain constraint program is the call to the labeling procedure, which searches for a consistent assignment for the set of to-be-instantiated variables. Subsequently, it selects a variable to instantiate and for this selected variable a value of its domain to which it is instantiated. If the assignment leads to a violation of a constraint, the process backtracks and explores another assignment. The labeling finishes when all variables have been assigned a consistent value. In this, the pruning power of the consistency algorithms is exploited by interleaving the assignments with a consistency propagation phase.

The assignment of a variable to a value allows to remove inconsistent values from the domains of the uninstantiated variables, leading to a smaller search space.

Unfortunately, an optimal labeling strategy for all problems does not exist. Therefore constraint solvers allow the user to formulate a problem specific strategy. Mostly, a good strategy is only discovered after a lot of experimentation.

The embedding of the finite domain solver is (Sicstus) Prolog is transparent for the user. The constraints are presented as a library of predefined Prolog predicates. With these the expert models the problem. As mentioned above, it often happens that the complete problem is modeled with the (finite domain) constraints. In that case Prolog is reduced to an interface language; the typical obtained structure for a finite domain problem is:

```
main(Vars):-
   setup_problem_variables(Vars), % definition of the search space
   post_constraints(Vars),        % problem description
   labeling(Vars).                % solution construction
```

First the constraint variables `Vars` are created and initialized with their domains. This sets the search space. Secondly all the constraints are posted to the constraint store. The constraint solver will catch up these and propagates a certain amount of implied information. The modeler often optimizes this phase (implicitly) by choosing efficient datastructures so that only relevant constraints are posted: e.g. a classical optimization is to remove symmetric constraints. And finally, a solution is constructed by the labeling process.

This has been a high level overview about the (finite domain) constraint solvers and their most important properties. For details about the implementation of a finite domain constraint solver, the reader is referred to [265] in which the system ROPE [267, 266] is described.

2.2 ID-Logic

The knowledge representation language ID-Logic [92, 94] is the modeling language for the domain knowledge of the problems. It is an extension of classical logic with non-monotone inductive definitions. By this extension, the language is able to support the modeling of the expert knowledge for a large class of practical problem domains, in such a way that the formal semantics of the expressed formulas corresponds in a natural and intuitive way to the human understanding of the problem domain.

Because of the expressiveness of the language, no general sound and complete solvers can be build. For this reason, and because ID-Logic aims to be independent of the procedural aspects connected with automated reasoning, an inference procedure will not be discussed.

2.2.1 The knowledge representation language ID-Logic

ID-Logic is an extension of classical (first order) logic with non-monotone inductive definitions. This integration points to the distinction between two fundamental sorts of knowledge, namely definitional and assertional, which are at the basis of ID-Logic. The distinction has been recognized firstly by Brachman and Levesque [57, 54].[3], who pointed out that the nature of much of our human knowledge is definitional.

In ID-Logic this idea is further elaborated by incorporating a more expressive, yet mathematically founded definition of the definitional knowledge. This definition has it roots in the study of mathematical induction. In [94], Denecker et al. show that Logic Programming can be understood as the study of a general and natural logic of inductive definitions. In this view, logic programs with negation are non-monotone inductive definitions. These definitions define a predicate based on the absence of knowledge. For example, consider the following three definitions:

Example 2.1

$$\left\{ \begin{array}{l} parent\_of(X,Y) \leftarrow father(X,Y) \\ parent\_of(X,Y) \leftarrow mother(X,Y) \end{array} \right\}$$

$$\left\{ \begin{array}{l} ancestor(X,Y) \leftarrow parent\_of(X,Y) \\ ancestor(X,Y) \leftarrow \exists\, Z.ancestor(X,Z) \wedge ancestor(Z,Y) \end{array} \right\}$$

$$\left\{ \begin{array}{l} even(X) \leftarrow X = 0 \\ even(X) \leftarrow \exists\, Y.X = s(Y) \wedge \neg even(Y) \end{array} \right\}$$

The first definition intuitively expresses that a parent is a father or a mother. This is the simplest case of a definition, where the defined predicate is defined in terms of other predicates that not depend themself on the defined predicate. The second example defines the transitive closure of the $parent\_of(X,Y)$ relation: the *ancestor* relation. The definition is an example of monotone induction. The last example has the most complex structure: it defines the even numbers only in terms of itself, and therefore to express the definition properly recursion through negation is needed. Because of the recursion through negation, it is an example of non-monotone induction.

Denecker argues in [91] that the well-founded semantics (in a generalized setting) captures the intended meaning of inductive definitions, and thus of logic programs. This is supported by *Approximation theory* [99], an algebraic fixpoint theory for general (monotone and non-monotone) operators in a lattice. Note that in the case of simple definitions, e.g. hierarchical or acyclic definitions, the well-founded semantics correspond to the completion semantics.

[3]This work became the basis of the family of representation languages: Description Logics [28]. See Section 2.4.1 for more information.

Another foundation of ID-Logic is the distinction between complete and incomplete knowledge. This stands orthogonal to the distinction between assertional and definitional knowledge. Intuitively completeness and incompleteness refer to the fact whether or not the expert is able to provide a definition for each predicate in his modeling language. If this is the case, the specification (or the domain knowledge) is complete, otherwise it is incomplete. In practice, the available domain knowledge is often incomplete. Although one is able to formally denote a certain relation, it is impossible to give a precise constructive definition in terms of the other predicates that exactly captures the meaning of that relation. This lack of definitional knowledge does not prevent that there is precise information available about the relation. For example, although we have no information about the parents of every person in the world (hence our knowledge is incomplete), it holds universally that one cannot be a parent of oneselves. This information can be specified as a first order sentence

$$\forall\, X.\neg parent\_of(X,X).$$

Note that the absence of knowledge about a relation is closely connected with the task the formal specification is aimed for. For the same domain of discourse, one task assumes complete knowledge (e.g. database querying task), while another task supposes incomplete knowledge for some relations (e.g. scheduling task). This implies that a problem specification is related to a (class of) task(s).

Formal definition of ID-Logic

Formally an ID-Logic theory is defined as follows:

DEFINITION 2.26 An ID-*Logic theory* $\mathcal{T}$ based on the first order logic language $\mathcal{L}$ is a pair ($\mathcal{D}$,$\mathcal{F}$). $\mathcal{D}$ is a set of definitions D_i (i=1..n) and $\mathcal{F}$ a set of first order formulas. A *definition* D is a set of rules

$$p(\bar{t}) \leftarrow B$$

where $p(\bar{t})$ is an atom and B any first order formula.

The predicates occurring in the head of a rule in a definition D are the *defined* predicates of D. $Defined(D)$ denotes the set of defined predicates of D. All the other predicates are the *open* predicates of D; the set of open predicates is denoted by $Open(D)$. By extension, the defined predicates of $\mathcal{D}$ are denoted by $Defined(\mathcal{D}) = \bigcup_{i=1..n} Defined(D_i)$. The open predicates of $\mathcal{D}$, those that have no definition at all, are $Open(\mathcal{D}) = \bigcup_{i=1..n} Open(D_i) - Defined(\mathcal{D})$. Finally, $Pred(D)$ is used to denote all predicates occurring in the definition D.

Although ID-Logic allows any first order formula as body, we will follow the convention of Logic Programming and restrict to conjunctions of literals.

EXAMPLE 2.2 (MULTIPLE DEFINITIONS) ID-Logic allows that more than one definition defines the same predicate. This formalizes the common situation that there exists distinct views on the same relation. The ancestor relation, presented before, can be expressed in different ways.

$$\mathcal{D} = \left\{ \begin{array}{l} \left\{ \begin{array}{l} ancestor(X,Y) \leftarrow parent\_of(X,Y) \\ ancestor(X,Y) \leftarrow \exists\ Z.ancestor(X,Z) \wedge ancestor(Z,Y) \end{array} \right\}, \\ \\ \left\{ \begin{array}{l} ancestor(X,Y) \leftarrow parent\_of(X,Y) \\ ancestor(X,Y) \leftarrow \exists\ Z.parent\_of(X,Z) \wedge ancestor(Z,Y) \end{array} \right\} \end{array} \right\}$$

The defined predicates of $\mathcal{D}$ are $Defined(\mathcal{D}) = \{ancestor/2\}$. $parent\_of/2$ is the only open predicate.

The formal semantics of an ID-Logic theory $\mathcal{T}$ =($\mathcal{D}$,$\mathcal{F}$) is as follows.

DEFINITION 2.27 (A MODEL OF A DEFINITION) An interpretation M is a model of a definition D iff there exists an interpretation I_o of $Open(D)$ such that M is the two-valued well-founded [257] model of D extending I_o. By extension, M is a model of a set of definitions $\mathcal{D} = \{D_1, \ldots, D_n\}$ iff M is a model for each definition $D \in \mathcal{D}$.

Recall that this is well-defined for non-monotone inductive definitions [94].

DEFINITION 2.28 (FORMAL SEMANTICS OF AN ID-LOGIC THEORY) An interpretation M is a model of the ID-Logic theory $\mathcal{T}$ =($\mathcal{D}$,$\mathcal{F}$) iff M is a model for $\mathcal{D}$ and satisfies all formulas of $\mathcal{F}$.

The collection of all models of $\mathcal{T}$ is denoted by $Mod(\mathcal{T})$.

Total models

ID-Logic only accepts two-valued well-founded models, i.e. models of which the atoms are *true* or $false$, also called *total* models. Three-valued models, i.e. that contain *undefined* atoms, are excluded. Underlying this choice is the assumption that the three-valuedness of a model is caused by some irregularity in the theory. For example, the definition

EXAMPLE 2.3

$$D = \left\{ \begin{array}{l} p \leftarrow \neg p. \\ q. \end{array} \right\}$$

defines simultaneously the predicates p and q. The unique well-founded model of this definition is $\{p^{\mathbf{u}}, q^{\mathbf{t}}\}$. Although q is well-defined, p is not. Because such models are usually not intentional, ID-Logic excludes them. Fortunately, for many important and frequent occurring classes of definitions, e.g. hierarchical definitions and acyclic definitions, the model is always two-valued [90].

General interpretations and Herbrand interpretations

The formal semantics of ID-Logic are based on general interpretations. In practice however, the semantics of ID-Logic is often limited to Herbrand interpretations by the (implicit) addition of the

- the Unique Names axioms [221]: two syntactic different terms denote a different object in the problem domain under consideration.
- the Domain Closure axiom: every element of the problem domain is denoted by at least one syntactic term. In [92] it is shown how to express this axiom using one definition and a single FOL statement.

The interpretations of a logic theory augmented with both axioms, are isomorphic with the Herbrand interpretations of the logic theory itself. Unless explicitly stated we assume from now on that each ID-Logic theory is augmented with both axioms.

Possible world semantics

Each model of an ID-Logic theory represents one possible state of the modeled domain of discourse. Hence, an expert who uses ID-Logic to model his domain knowledge has to represent those elements which are true in the problem domain. In this respect, ID-Logic differs from some other representation languages which model the believed states of the domain.

EXAMPLE 2.4 Consider for example Osama Bin Laden. We can model that he is alive iff he is not dead by the definition

$$\{alive \leftarrow \neg dead\}$$

Since we have no knowledge about his current state, the models of the ID-Logic theory consisting of this definition, namely $\{dead^{\mathbf{f}}, alive^{\mathbf{t}}\}$ and $\{dead^{\mathbf{t}}, alive^{\mathbf{f}}\}$ express this lack of knowledge correct. However, when this formula is used to model our believes, only the first model is the correct one. Because we have no means to believe that he is dead, it has to be that he is alive. This believe is expressed by the first model.

Incomplete knowledge

Another important point is that an ID-Logic theory $\mathcal{T}$ can have more than one model. Having more than one model expresses that the knowledge of $\mathcal{T}$ is *incomplete*, more precisely there is uncertainty about the exact state of the modeled domain. $\mathcal{T}$ has *complete* knowledge when $\mathcal{T}$ has only one model (modulo an isomorfism). It is said to be *inconsistent* when it has no models at all.

The open predicates are the source of incompleteness. The lack of a definition for them means that the domain expert was unable to fix his interpretation. In the best case, he is able to formulate general properties such that the interpretations are correct with respect to his domain knowledge.

Deriving information from ID-Logic theories

The ability to deal with incomplete knowledge reflects the fact that an ID-Logic theory can have more than one model. Therefore, it is useful to reformulate the most important relations between a formula and an ID-Logic theory. A formula F can be related to an ID-Logic theory $\mathcal{T}$ as follows:

1. A formula F is *satisfied* by $\mathcal{T}$ iff there exists a model M, $M \in Mod(\mathcal{T})$, in which F is satisfied.
2. A formula F is *entailed* by $\mathcal{T}$, denoted by $\mathcal{T} \models F$, iff every model M of $\mathcal{T}$ ($M \in Mod(\mathcal{T})$) satisfies F.

As explained earlier, the actual computation of these relations is a different issue. Without any extra restrictions, ID-Logic is very expressive and thus undecidable.

From the above notions, the satisfiability is the most important one in this thesis. Abductive reasoning can be seen as a computation method to compute a set of models for a theory $\mathcal{T}$ in which the formula F is satisfied. This relation is elaborated more in the next section.

Some methodological guidelines

Having a well-defined and structured knowledge representation language is not sufficient to get correctly formalized domain knowledge. Often the question arises what should be modeled as a definition and what as an assertion. In order to clarify the distinction, [102] presents two situations in which the distinction is subtle but of which the domain expert has to be aware. Each of the situations leads to a methodological guideline.

1. The first guideline: *rules of a definition follow the direction of causality.*

   For example, the following ID-Logic theories represent the knowledge that *streets are wet iff it rains*, and *it rains iff there are saturated clouds* by two definitions.

   $$\mathcal{T}_1 = (\{\{wet\_streets \leftarrow rain.\}, \{rain \leftarrow saturated\_clouds.\}\}, \emptyset)$$

   $$\mathcal{T}_2 = (\{\{rain \leftarrow wet\_streets.\}, \{saturated\_clouds \leftarrow rain.\}\}, \emptyset)$$

   Both theories have the same models and hence are logically equivalent. However, the observation that the streets are wet is in theory $\mathcal{T}_1$ explained by the assumption of saturated clouds and in the second by itself, as it is the open predicate on which the remaining part of the knowledge is built. The latter is counter-intuitive and the reason for this result is that the rules in $\mathcal{T}_2$ do not follow the methodological guideline.

2. The second guideline: *non-causal implications should not be added as a rule in a definition.*

Consider the following statements: *one is alive is implied by that one is walking* and *being born causes to be alive*. The ID-Logic theory

$$\mathcal{T}_1 = \left(\left\{ \left\{ \begin{array}{l} alive \leftarrow walking. \\ alive \leftarrow born. \end{array} \right\} \right\}, \emptyset \right)$$

formalizing the above statements, has as one of its models $\{alive^{\mathbf{t}}, walking^{\mathbf{t}}, born^{\mathbf{f}}\}$. This model is read as the possible world in which one is alive and walking but not born. Obviously this is not intended. The source of this anomaly is that the first expression is encoded as a definition's rule. The correct ID-Logic theory is thus

$$\mathcal{T}_2 = (\{\{alive \leftarrow born.\}\}, \{walking \rightarrow alive.\})$$

2.2.2 Properties of ID-Logic theories

Compositionality

Compositionality or modularity is an important property of a knowledge representation language. The ability to break down a large theory into smaller pieces or to compose a set of knowledge parts together to a larger unity is beneficial for many purposes. For example, experts can limit their scope of attention and perform local changes such that the rest of the knowledge base is affected in a controlled way.

For ID-Logic theories the following composition operation holds.

Definition 2.29 Let $\mathcal{T}_1 = (\mathcal{D}_1, \mathcal{F}_1)$ and $\mathcal{T}_2 = (\mathcal{D}_2, \mathcal{F}_2)$ be two ID-Logic theories based on the FOL language $\mathcal{L}$. Then the composition of $\mathcal{T}_1$ and $\mathcal{T}_2$ is the ID-Logic theory $\mathcal{T}$ based on $\mathcal{L}$ obtained by the pairwise union of $\mathcal{T}_1$ and $\mathcal{T}_2$:

$$\mathcal{T}_1 \circ \mathcal{T}_2 = (\mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{F}_1 \cup \mathcal{F}_2) = \mathcal{T}$$

Proposition 2.1 For two ID-Logic theories $\mathcal{T}_1$ and $\mathcal{T}_2$ it holds that

$$Mod(\mathcal{T}_1 \circ \mathcal{T}_2) = Mod(\mathcal{T}_1) \cap Mod(\mathcal{T}_2)$$

The proof follows immediately from the definitions 2.29 and 2.28. Note that this operation is monotone: the models of the composed theory are models of the composing theories. Consequently, adding new definitions or axioms to an ID-Logic theory is a monotone operation.

Accordingly, an ID-Logic theory $\mathcal{T}$ is decomposable in many different ways. We mention two special cases:

- $\mathcal{T}_1 = (\mathcal{D}, \emptyset)$ and $\mathcal{T}_2 = (\emptyset, \mathcal{F})$.
- $\mathcal{T}_1 = (\{D_1\}, \mathcal{F})$ and $\mathcal{T}_2 = (\{D_2, \ldots, D_n\}, \mathcal{F})$ given that $(\mathcal{D} = \{D_1, D_2, \ldots D_n\})$.

Changing Definitions

Updating a definition has a totally different effect. This is in general a non-monotonic operation. According to McCarthy [198], such non-monotone behavior is needed for a formal knowledge representational language to be elaboration tolerant. Elaboration tolerant means that it should be convenient to take into account new phenomena. Hence, small changes in the domain knowledge should be reflected by small changes in the formal specification in most cases.

The following example shows that ID-Logic has this property.

Example 2.5 Consider the definition that defines an elephant as a grey animal with big teeth.

$$D = \{elephant(X) \leftarrow animal(X) \wedge grey(X) \wedge bigTeeth(X).\}$$

On our safari, we see an animal, lets call it Clyde, which is grey but has no big teeth. According to our definition Clyde is not an elephant. However, our driver assures Clyde is an elephant because it has a trunk. We integrate this new information in our definition:

$$D' = \left\{ \begin{array}{l} elephant(X) \leftarrow animal(X) \wedge grey(X) \wedge bigTeeth(X). \\ elephant(X) \leftarrow animal(X) \wedge grey(X) \wedge hasTrunk(X). \end{array} \right\}$$

A bit further, we see Susanna. Susanna is also grey and has big teeth. According our definition, Susanna is an elephant. But, our driver denies that Susanna is an elephant, but a hippo. A hippo distinguishes from an elephant by its short legs. Again, we update our definition of an elephant:

$$D'' = \left\{ \begin{array}{l} elephant(X) \leftarrow animal(X) \wedge grey(X) \wedge bigTeeth(X) \wedge \neg shortLegs(X). \\ elephant(X) \leftarrow animal(X) \wedge grey(X) \wedge hasTrunk(X). \end{array} \right\}$$

Now, it follows from our theory that Clyde is an elephant and Susanna is not.

2.3 Abductive Logic Programming

2.3.1 Abductive reasoning

Abductive inference forms together with deduction and induction the main logical inferences. Already the ancient Greek philosophers, e.g. Aristotle, recognized the difference The philosopher C.S. Pierce (1839-1914) [118] was the first to establish a clear distinction between these modes of reasoning. He has formulated abduction as an inference method for producing plausible hypotheses that explain the observations. Deduction is characterized as deriving consequences that necessarily follow from a logical theory. The last inference method, induction, aims at deriving generalizations from a set of experimental data consistent with a given background theory. Induction is sometimes considered as a variant of abductive reasoning or vice versa. For this discussion (in the context of Logic Programming), we refer the interested reader to [121, 122].

Of these three modes of inference, deduction is the best known and studied inference in (computer) science. Due to the growing area of Machine Learning, inductive inference attracts more and more attention and acquires a widespread use. Compared to these two, abduction is far less known.

However, abduction is a natural form of human reasoning, as illustrated by the following scenario. Imagine that you are living in New-York and cooking dinner, when suddenly the electrical cooking plates drop out. Your first reaction is to check the fuses. Observing that they are still intact, you conclude that no short circuit has happened. Then you look through the window and see that all light has disappeared from the city. By this, you realize that something is wrong with the electricity provision which caused the cooking plates to stop heating.

Twice abductive inference has been applied in the example. A first time when deciding to check the fuses. We know (our background knowledge) that an electrical short circuit causes the drop out of a cooking plate, and that this normally has the effect that the fuses melt. This justifies our initial hypothesis of an electrical short circuit. The observation that the fuses are intact, rejects this initial hypothesis. Because there must be a cause, a second potential (abductive) explanation is that the electricity supplier has a problem. This hypothesis is verified by looking through the window. In general, when we are looking for a cause of a failure we apply abductive inference.

The example also shows that not every explanation is accepted. First of all, it has to be consistent with the background knowledge. In the example, the background knowledge states that a short circuit causes melted fuses and this is in contradiction with the observation that the fuses are intact. Secondly, the explanation is often required to be of a certain quality. In that context, abduction is referred to as "inference to the *best* explanation" [150]. "Best" is usually regarded as a minimality or maximality criterion.

Nowadays, abductive inference is studied and practiced in various disciplines

[148] such as Philosophy [275, 150], Psychiatry, Linguistics [237], and Artificial Intelligence [153] . Within Artificial Intelligence, abductive reasoning is intensively used and studied in the area of computational logic [98]. In that domain, an abductive solution for an observation with respect to a logic theory is commonly defined as:

DEFINITION 2.30 Giving a logic theory $\mathcal{T}$, an abductive solution for an observation Q is a formula $\mathcal{E}$ such that

- $\exists\ (\mathcal{E})$ is satisfiable w.r.t. $\mathcal{T}$, and
- $\mathcal{T} \models \forall\ (\mathcal{E} \rightarrow Q)$

This definition expresses the logical entailment view on abduction, in which the explanation formula $\mathcal{E}$ entails the observation Q.

Abductive reasoning can be regarded from different points of view. One is the *logical entailment view* (as above), in which an observation is explained by a formula iff the formula entails the observation. The other is the *causation view* in which the explanation formula *causes* the observation [150].

The difference is well illustrated by the Paresis disease example from [219]. The disease Paresis is caused by a latent untreated form of syphilis. But the chance that latent untreated form of syphilis leads to Paresis is only 25%. Thus, a doctor is able to explain Paresis by the hypothesis of a untreated form of syphilis, however the reverse doesn't hold; he can't explain of-course syphilis with the hypothesis of Paresis. In this example the directionality of logical entailment and causation is opposite: syphilis is a cause of Paresis but does not entail it, and Paresis entails syphilis but does not cause it.

The example illustrates that the causation view is the more correct view on abduction. Fortunately, situations where entailment and causation do not coincide are rare. In application domains such as temporal reasoning and diagnosis, both views clearly correspond. In those, the logic theories describe causality information. When the explanation formulas are restricted to predicates that describe the primitive causes of the modeled domain then entailment is exactly causation.

Abduction is strongly connected with incomplete knowledge. It is all about making hypotheses, and that is only sensible when there is uncertainty about the actual state. This uncertainty (by lack of information) is represented by the fact that the logic theory has more than one model. In this setting abduction is related with satisfiability checking and model generation. The existence of an explanation formula $\mathcal{E}$ that explains the observation Q is equivalent with the existence of a model of the theory $\mathcal{T}$ that satisfies Q. But the abductive solution $\mathcal{E}$ is more informative, because it only presents the *relevant* information in order to satisfy the query. A model fixes the complete state, hence a model generator is obliged to make choices on irrelevant parts. Abductive reasoning ignores everything that is irrelevant to explain the observation. In that view, one can say that the explanation formula presents a set of models that explain the observation. Note that deduction in the context of incomplete knowledge is deriving consequences that hold for all models.

2.3.2 Abductive Logic Programming

Abductive Logic Programming (ALP) [98, 153] is the extension of Logic Programming that aims at computing abductive solutions for logic programs representing an incomplete state of affairs. Its study started at the end of the eighties and consisted of two lines of research: On the one hand ALP was used to characterize the non-monotonic semantics of logic programs (with negation and incomplete knowledge) [117, 156, 108]. On the other hand it was applied as a computational method to solve a variety of problems; first for AI-planning problems [116, 202] and later broadened to problems such as diagnosis [216, 84], natural language understanding [237, 30, 71], multi-agent reasoning [249, 169], assignment and scheduling problems [159, 160, 238] and many others. For a near complete overview on the state of the art on Abductive Logic Programming, the interested reader is referred to [98, 97, 153].

This thesis follows the latter view according to which abduction is considered as a computational device. The other aspects, namely the representational and epistemological aspects, are provided by ID-Logic. But there is a strong connection between both, which will be discussed in the following sections.

Analogous to the evolution in Logic Programming, many different definitions of Abductive Logic Programming have been proposed. In [98], Denecker and Kakas present a unifying setting, which is generic with respect to the chosen semantics and syntax. We follow this setting.

An *Abductive Logic theory* is a triple ($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$) based on a first order language $\mathcal{L}$. $\mathcal{P}$ is a logic program consisting of program rules $p(\overline{t}) \leftarrow F$, where $p(\overline{t})$ is an atom and F is a first order formula. $\mathcal{A}$ is the set of ground abducibles[4]. Abducibles are atoms based on the predicates that do not appear in the head of a rule. In most abductive frameworks, only the predicate names are specified. In that case $\mathcal{A}$ is the grounding of these predicates with regard to the Herbrand Universe of $\mathcal{L}$. $\mathcal{IC}$ is a set of integrity constraints, which are first order statements over the modeled domain. Usually the integrity constraints express some knowledge about the abducibles.

The Abductive Logic theory is used to model the background knowledge which is used in the abductive reasoning. For instance, a specification of the power failure example is the following propositional Abductive Logic theory

[4]Later we will relate Abductive Logic theories with ID-Logic. Open predicates in the context of ID-Logic will correspond then to abducibles in Abductive Logic theories. For this reason will use them as synonyms.

EXAMPLE 2.6
The ALP theory $\mathcal{T}$ =

- $\mathcal{P} = \left\{ \begin{array}{l} failing\_cooking\_plate \leftarrow no\_current. \\ no\_current \leftarrow shortcircuit. \\ no\_current \leftarrow no\_electricity\_provision \end{array} \right\}$
- $\mathcal{A} = \{shortcircuit, melted\_fuses, no\_electricity\_provision\}$
- $\mathcal{IC} = \{shortcircuit \rightarrow melted\_fuses\}$

Given an Abductive Logic theory, we define now the abductive answers to queries with respect to the background knowledge. A query is any first order formula of $\mathcal{L}$.

DEFINITION 2.31 Given a semantics S, an *abductive explanation* for a query $\mathcal{Q}$, for an abductive logic theory ($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$) is a set $\Delta \subseteq \mathcal{A}$ and an answer substitution θ such that:

- $\mathcal{P} \cup \Delta \models_S \mathcal{Q}\theta$
- $\mathcal{P} \cup \Delta \models_S \mathcal{IC}$
- $\mathcal{P} \cup \Delta$ is consistent under the semantics S

$\mathcal{P} \cup \Delta$ denotes the logic program obtained by augmenting the program $\mathcal{P}$ with the logic program consisting of facts derived from Δ.

EXAMPLE 2.7 (EXAMPLE 2.6 CONTINUED) Recall our scenario, one answer of the query $\mathcal{Q} = failing\_cooking\_plate$ given $\mathcal{T}$ is $\Delta_1 = \{melted\_fuses, shortcircuit\}$. The other is $\Delta_2 = \{no\_electricity\_provision\}$. The verification of the fuses gives a more detailed observation, which is expressed by a changed query

$$\mathcal{Q}' = failing\_cooking\_plate \wedge \neg melted\_fuses$$

For this query the only abductive answer is $\Delta' = \{no\_electricity\_provision\}$.

Definition 2.31 defines an abductive answer, which can be constructed by applying abductive inference. However, it does not provide formal semantics for an abductive logic theory. The following definition defines a schema for an abductive logic theory [156], without fixing the actual semantics.

DEFINITION 2.32 Given a particular formal semantics S, a structure M is a model of an abductive logic theory ($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$) iff there exists a $\Delta \subseteq \mathcal{A}$ such that

- M is an S-model of the logic program $\mathcal{P} \cup \Delta$, and

- M is a classical model of $\mathcal{IC}$, i.e. $M \models_S \mathcal{IC}$

Entailment is defined as usually: F is FOL formula then

$$(\mathcal{P}, \mathcal{A}, \mathcal{IC}) \models F \text{ iff for each model M of } (\mathcal{P}, \mathcal{A}, \mathcal{IC}), M \models F$$

Fitting the ALP frameworks found in the literature in this semantic framework is a matter of selecting the right semantics and eventually of transforming the Abductive Logic theory to a corresponding logic theory. For example, the ALP framework of [85] under the completion semantics can be obtained by constructing the FOL theory composed of

- comp($\mathcal{P}$,$\mathcal{A}$), the completion of all non-abducible predicates of $\mathcal{P}$, and
- $\mathcal{IC}$, the integrity constraints, and
- UNA, the unique names axioms [221] (or Clark's equality axioms [78])

Similar embeddings can be defined for abductive extensions of Logic Programming based on the stable model semantics [156] and the well-founded semantics [215].

2.3.3 Knowledge Representation and Abduction

When starting from the aim of representing a problem domain for a given task in a declarative way and using this representation to solve the actual problem, the computational inference that corresponds to this goal is often abduction. The next example that designs a departmental duty system illustrates this.

Example 2.8 Suppose we like to model a departmental duty system. The purpose of such system is to maintain and support the distribution of the educational tasks that members of a university department have to perform. The process starts with the selection of an appropriate alphabet, e.g. consisting of the predicates $duty(\cdot)$, $person(\cdot)$, $skill(\cdot)$, $assignedTo(\cdot,\cdot)$,... Using the derived first order language, we formulate in the next phase the necessary information as a logic theory. Some pieces of the domain knowledge are

- A duty is only executed by a skilled person.

$$\begin{aligned}&\forall\, X, Y, S.duty(X) \wedge person(Y) \wedge \\ &\qquad assignedTo(X, Y) \wedge neededSkill(X, S) \rightarrow personalSkill(Y, S)\end{aligned}$$

- Every duty cost an amount time (the unit is hours). A duty with a cost larger as 20 hours is a large duty.

$$\forall\, X.duty(X) \rightarrow \exists\, C.dutyCost(X, C)$$

$$\{largeDuty(X) \leftarrow duty(X) \wedge dutyCost(X, C) \wedge C > 20\}$$

- A member who is writing a PhD thesis is excluded from duties.

  $$\forall\ X.person(X) \land writesPhD(X) \rightarrow \neg(\exists\ Y.assignedTo(Y, X)).$$

  Or one might use this auxiliary definition

  $$\{occupiedPerson(X) \leftarrow person(X) \land writesPhD(X)\}$$

  to obtain a more flexible (w.r.t. later changes) statement

  $$\forall\ X.occupiedPerson(X) \rightarrow \neg(\exists\ Y.assignedTo(Y, X))$$

Having formulated the basic structural knowledge about the domain, the last phase is that in which the represented knowledge is used to solve different tasks. One common task is to retrieve whether a duty is assigned and to whom. This task is solvable by deductive reasoning, and hence SLD-resolution or SQL-processing can be applied.

More of interest is another task in which the assignment table is constructed. This is clearly not a deductive task, as there is no constructive information specifiable (via definitions) which can infer an assignment from a set of duties and departmental members such that *all* integrity constraints are satisfied. A correct view for this task is that $assignedTo(\cdot, \cdot)$ is an open predicate in the ID-Logic specification for which a consistent interpretation is searched for. And this is exactly abductive inference.

Note that *changing* the representation, by representing the assignment relation not as a unknown binary relation (the declarative approach followed here) but as a finite table of free instances (i.e. consisting of variables[5]), the problem is transformed so that deductive inference can be applied. The new representation allows to solve the problem using CLP.

The example shows that abductive inference emerges naturally from the declarative problem specifications. As the initial goal of Declarative Problem solving is solving problems *with* a declarative specification, this forms a strong argument to study abductive inference.

2.3.4 ID-Logic and ALP

The above view reflects itself in the distinction we make between ID-Logic and Abductive Logic Programming. ID-Logic is a modeling, knowledge representational tool while ALP focuses on denoting/computing the right interpretations for the abducibles.

[5]Such a table of variables might be $\{v_{c_1}, v_{c_2}, \ldots\}$ where v_{c_i} denotes the person that is assigned to course c_i. The domain of the variables v_{c_i} are the available persons.

But there is a close connection between both because ID-Logic has emerged from research in the context of ALP. Abductive logic theories under the two-valued well-founded semantics form a special subclass of ID-Logic theories, namely those theories with a single definition. In earlier work on ID-Logic, those theories have been called OLP-FOL, *Open Logic Programming - First Order Logic*, that explicitly refers to the combination of Abductive (or Open) Logic Programs and First Order Logic.

In the other direction, we show below how ID-Logic theories can be transformed into OLP-FOL theories and how OLP-FOL theories can be interpreted as Abductive Logic theories. By using this transformation in a preprocessing phase, an ALP solver (in our case the $\mathcal{A}$system) can be used for abductive inference on ID-Logic theories.

Renaming of predicates

A first step in the transformation is the renaming of predicates in ID-Logic theories. The operation replaces in an ID-Logic theory a predicate with a fresh one such that knowledge is preserved, i.e. the resulted theory exactly contains the same information as the original one.

To define the operation, we need some new notions.

DEFINITION 2.33 A FOL language $\mathcal{L}$ *extends* the language $\mathcal{L}$' iff the set of constants, function symbols and predicate symbols of $\mathcal{L}$ are supersets of those of $\mathcal{L}$'.

NOTATION 2.3.1 Let the language $\mathcal{L}$ be an extension of the language $\mathcal{L}$'. The *restriction* of an interpretation I for a language $\mathcal{L}$ to a language $\mathcal{L}$' is the projection of I on the predicates of $\mathcal{L}$'. It is denoted as $I \downarrow_{\mathcal{L}'}$.

DEFINITION 2.34 Given a FOL theory $\mathcal{T}_1$ based on a language $\mathcal{L}_1$ and another theory $\mathcal{T}_2$ based on a language $\mathcal{L}_2$. Let the language $\mathcal{L}$ be $\mathcal{L} \subseteq \mathcal{L}_1 \cap \mathcal{L}_2$. The theories $\mathcal{T}_1$ and $\mathcal{T}_2$ are *equivalent in* $\mathcal{L}$ if

- for each model $M_1 \in Mod(\mathcal{T}_1)$, there exists a model $M_2 \in Mod(\mathcal{T}_2)$ such that $M_1 \downarrow_{\mathcal{L}} = M_2 \downarrow_{\mathcal{L}}$
- for each model $M_2 \in Mod(\mathcal{T}_2)$, there exists a model $M_1 \in Mod(\mathcal{T}_1)$ such that $M_2 \downarrow_{\mathcal{L}} = M_1 \downarrow_{\mathcal{L}}$

To define the renaming we need the auxiliary notion of the replacement of a predicate in a definition.

DEFINITION 2.35 A *replacement* θ of a predicate p by p^* applied on a definition D, is a transformation that replaces each occurrence of p in the rules of D by p^*. The replacement is denoted as $\theta = \{p \mapsto p^*\}$, and its application as $D\theta$. By extension, a replacement θ on a set of definitions $\mathcal{D} = \{D_1, \dots, D_n\}$ is defined as the set of replaced definitions by θ: $\mathcal{D}\theta = \{D_1\theta, \dots, D_n\theta\}$.

The application of the inverse replacement $\sigma = \{p^* \mapsto p\}$ on a definition $D\theta$ yields the original definition $D = (D\theta)\sigma$.

DEFINITION 2.36 Let $\mathcal{L}$' be the language that extends $\mathcal{L}$ with the predicate p^* and let θ be the replacement $p \mapsto p^*$. A *renaming* of a predicate p with a predicate p^* in an ID-Logic theory $\mathcal{T}$ =($\mathcal{D}$,$\mathcal{F}$) based on the language $\mathcal{L}$ is the theory $\mathcal{T}$'=($\mathcal{D}\theta$,$\mathcal{F}\theta$ $\cup\{\forall\ \overline{X} p(\overline{X}) \leftrightarrow p^*(\overline{X})\}$) based on the language $\mathcal{L}$'.

Note that the renamed theory $\mathcal{T}$' is logically equivalent with the theory

$$(\mathcal{D}, \mathcal{F} \cup \{\forall\ \overline{X} p(\overline{X}) \leftrightarrow p^*(\overline{X})\})$$

since p and $p*$ have the same interpretation and either the one or the other occurs in the rest of the formulas.

PROPOSITION 2.2 For an ID-Logic theory $\mathcal{T}$ based on the language $\mathcal{L}$, let $\mathcal{T}$' be the renaming of a predicate p with p^* based on $\mathcal{L}$'. Then, $\mathcal{T}$ is equivalent with $\mathcal{T}$' in $\mathcal{L}$.

PROOF 7
Given are the ID-Logic theory $\mathcal{T}$ =($\mathcal{D}$,$\mathcal{F}$) based on the language $\mathcal{L}$, and the renamed theory $\mathcal{T}$'=($\mathcal{D}\theta$,$\mathcal{F}\theta$ $\cup\{\forall\ \overline{X}.p(\overline{X}) \leftrightarrow p^*(\overline{X})\}$) based on the language $\mathcal{L}$'. θ is the replacement $p \mapsto p^*$.

According definition 2.34, we have to prove:

- for each model $M \in Mod(\mathcal{T})$, there exists a model $M' \in Mod(\mathcal{T}')$ such that $M \downarrow_{\mathcal{L}} = M' \downarrow_{\mathcal{L}}$:
  proof: Let $M\theta$ be the model M in which the predicate symbol p is replaced by p^*. It is a model for the theory $\mathcal{T}\theta$ based on the language $\mathcal{L}\theta$. Construct then the interpretation $M' = M \cup M\theta$. In M', the interpretation of all symbols different than p and p^* is the same as in M and it holds that the interpretation of the symbols p and p^* is the same. M' is an interpretation of the language $\mathcal{L} \cup \mathcal{L}\theta = \mathcal{L}'$. Also, it is a model for the theory $\mathcal{T}$': since it is a model of $\mathcal{D}\theta$ and $\mathcal{F}\theta$ based on $\mathcal{L}$' and due to that p and p^* have the same interpretation it is a model of $\{\forall\ \overline{X}.p(\overline{X}) \leftrightarrow p^*(\overline{X})\}$ based on $\mathcal{L}$'.

  Because the restriction of M' to the language $\mathcal{L}$ is M, it follows that for a model M of $\mathcal{T}$ there exists a model M' of $\mathcal{T}$', constructed as above, such that $M' \downarrow_{\mathcal{L}} = M = M \downarrow \mathcal{L}$. □

- for each model $M' \in Mod(\mathcal{T}')$, there exists a model $M \in Mod(\mathcal{T})$ such that $M' \downarrow_{\mathcal{L}} = M \downarrow_{\mathcal{L}}$:
  proof: The models of $\mathcal{T}' = (\mathcal{D}\theta, \mathcal{F}\theta \cup \{\forall\ \overline{X}.p(\overline{X}) \leftrightarrow p^*(\overline{X})\}$ are those of

$\mathcal{T}'' = (\mathcal{D}, \mathcal{F} \cup \{\forall\, \overline{X}.p(\overline{X}) \leftrightarrow p^*(\overline{X})\}$. By decomposing the ID-Logic theory $\mathcal{T}''$, it follows that a model M' of $\mathcal{T}$' is a model of $(\mathcal{D}, \mathcal{F})$ based on $\mathcal{L}$'. Since p^* does not occur in $(\mathcal{D},\mathcal{F})$, $M' \downarrow_{\mathcal{L}}$ is a model of $(\mathcal{D},\mathcal{F})$ based on $\mathcal{L}$. As this theory is $\mathcal{T}$, this finishes our proof. □

Transforming ID-Logic theories into OLP-FOL theories

Our goal is to show that every ID-Logic theory has an equivalent OLP-FOL theory. For this we now construct an equivalence preserving transformation. Given is an ID-Logic theory $\mathcal{T}$ =($\mathcal{D}$,$\mathcal{F}$) having multiple definitions $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ (n>1), based on the language $\mathcal{L}$. Let the to-be-constructed ID-Logic theory $\mathcal{T}$'=($\mathcal{D}$',$\mathcal{F}$')=({D},$\mathcal{F}$'). The goal is to find a transformation such that $\mathcal{T}$ is equivalent with $\mathcal{T}$' w.r.t. $\mathcal{L}$.

To get one definition from these n definitions the knowledge residing in each of them must be combined together. As explained earlier, the merging of definitions is a non-monotone operation. Hence, it must handled with care, because due to the merging new dependencies can be introduced that change the semantics.

For a related problem, Verbaeten et al. [270] prove the following theorem (reformulated in the ID-Logic formalism):

THEOREM 8
Given two ID-*Logic theories* $\mathcal{T}_1 = (\{D_1\}, \mathcal{F}_1)$ *and* $\mathcal{T}_2 = (\{D_2\}, \mathcal{F}_2)$ *based on the language* $\mathcal{L}$. *If* $Defined(D_1) \cap Pred(D_2) = \emptyset$, *then* $Mod((\{D_1 \cup D_2\}, \mathcal{F}_1 \cup \mathcal{F}_2)) = Mod(\mathcal{T}_1) \cap Mod(\mathcal{T}_2)$.

Based on this, we can proof the following property:

PROPOSITION 2.3 Compacting two definitions D_1 and D_2 of an ID-Logic theory $\mathcal{T} = (\{D_1, \ldots, D_n\}, \mathcal{F})$ into $D = \{p \leftarrow B | p \leftarrow B \in D_i, i = 1..2\}$ yields the ID-Logic theory $\mathcal{T}' = (\{D, D_3, \ldots, D_n\}, \mathcal{F})$. If none of the defined predicates of one definition, e.g. D_1, occurs in the other definition D_2 ($Defined(D_1) \cap Pred(D_2) = \emptyset$), then

$$Mod(\mathcal{T}) = Mod(\mathcal{T}')$$

PROOF 9
Now, consider the ID-Logic theory $\mathcal{T} = (\{D_1, D_2\}, \mathcal{F}_1 \cup \mathcal{F}_2)$. M is a model of $\mathcal{T}$ iff M is a model of D_1 and a model is of D_2 and a model of $\mathcal{F}_1$ and of $\mathcal{F}_2$. Or otherwise stated, when M is a model of $\mathcal{T}_1 = (\{D_1\}, \mathcal{F}_1)$ and $\mathcal{T}_2 = (\{D_2\}, \mathcal{F}_2)$. Then it follows that M is a model of $Mod(\mathcal{T}_1) \cap Mod(\mathcal{T}_2)$, and thus according the above theorem 8, a model of $Mod((\{D_1 \cup D_2\}, \mathcal{F}_1 \cup \mathcal{F}_2)$.

Based on this property, the transformation is defined. It first applies sufficient renamings of the predicates in the ID-Logic theory such that the resulting theory satisfies the above conditions. Afterwards the definitions are merged into each other.

DEFINITION 2.37 The transformation $\tau : (\mathcal{D} = \{D_1, D_2, \ldots, D_n\}, \mathcal{F}) \mapsto (\mathcal{D}' = \{D\}, \mathcal{F}')(n > 1)$ compact any ID-Logic theory by these consecutive steps:

1. Renaming of conflicting predicates.
   For all defined predicates p $\notin$ Open($\mathcal{D}$), rename each occurrence of p in every definition D_i with p^{D_i} (i=1..n). The resulting renamed definitions are D_i^r. This renaming of $\mathcal{T}$ is then:

   $$\mathcal{T}_{inter} = (\{D_1^r, \ldots, D_n^r\}, \mathcal{F}')$$

   where $\mathcal{F}' = \mathcal{F} \cup \{p \leftrightarrow p^{D_i} | p \notin Open(\mathcal{D})$ and p is a defined predicate of $D_i\})$

2. Compacting $\mathcal{T}_{inter}$ to one definition.
   $\mathcal{T}' = (\{D\}, \mathcal{F}')$ where $D = \{p \leftarrow B | p \leftarrow B \in D_i^r\}$

EXAMPLE 2.9 Example 2.2 revisited.
Since both definitions define the same predicate *ancestor*/2, first the renaming has to be applied:

$$\mathcal{D}_1 = \left\{ \begin{array}{l} \left\{ \begin{array}{l} ancestor^{D_1}(X,Y) \leftarrow parent\_of(X,Y) \\ ancestor^{D_1}((X,Y) \leftarrow \exists\ Z.ancestor^{D_1}((X,Z) \wedge ancestor^{D_1}((Z,Y) \end{array} \right\}, \\ \\ \left\{ \begin{array}{l} ancestor^{D_2}((X,Y) \leftarrow parent\_of(X,Y) \\ ancestor^{D_2}((X,Y) \leftarrow \exists\ Z.parent\_of(X,Z) \wedge ancestor^{D_2}((Z,Y) \end{array} \right\} \end{array} \right\}$$

Now, the theories are ready to be compacted in one definition. This results in the ID-Logic theory (suppose that $\mathcal{F}$ was empty)

$$\left(\begin{array}{l} \mathcal{D}' = \left\{ \left\{ \begin{array}{l} ancestor^{D_1}(X,Y) \leftarrow parent\_of(X,Y) \\ ancestor^{D_1}((X,Y) \leftarrow \exists\ Z.ancestor^{D_1}((X,Z) \wedge ancestor^{D_1}((Z,Y) \\ ancestor^{D_2}((X,Y) \leftarrow parent\_of(X,Y) \\ ancestor^{D_2}((X,Y) \leftarrow \exists\ Z.parent\_of(X,Z) \wedge ancestor^{D_2}((Z,Y) \end{array} \right\} \right\}, \\ \\ \mathcal{F}' = \left\{ \begin{array}{l} \forall\ X.ancestor(X) \leftrightarrow ancestor^{D_1}(X) \\ \forall\ X.ancestor(X) \leftrightarrow ancestor^{D_2}(X) \end{array} \right\} \end{array} \right)$$

Since *parent_of*/2 was an open predicate in both definitions, it is not affected by the transformation.

PROPOSITION 2.4 Let $\tau(\mathcal{T})$ be the transformed ID-Logic theory for the ID-Logic theory $\mathcal{T} = (\{D_1, \ldots, D_n\}, \mathcal{F})$ based on $\mathcal{L}$. Then, $\tau(\mathcal{T})$ and $\mathcal{T}$ are equivalent in $\mathcal{L}$.

PROOF 10
The proposition follows from the fact that both transformations in τ result in equivalent theories in $\mathcal{L}$. According to proposition 2.2, it holds for the renaming

part of the transformation. The second part results also in equivalent theories since by proposition 2.3 both sets of models (from the theories before and after the operation) are identical.

We are now able to formally present the main theorem.

THEOREM 11
For each ID-*Logic theory with multiple definitions an equivalent (in terms of knowledge)* ID-*Logic theory with one definition exists.*

PROOF 12
Let $\mathcal{T}$ be an ID-Logic theory. Then, the transformation $\tau(\mathcal{T})$ yields an ID-Logic theory containing a single definition, which is an OLP-FOL theory. According to proposition 2.4, this theory is equivalent with the original one in terms of knowledge.

OLP-FOL and ALP

LEMMA 1
An ID-*Logic theory with just one definition has an equivalent Abductive Logic theory (under the two-valued well-founded semantics).*

PROOF 13
Consider an ID-Logic theory $\mathcal{T}$ =($\mathcal{D}$,$\mathcal{F}$)=({D},$\mathcal{F}$). Then the equivalent ALP theory is (D,$\mathcal{A}$,$\mathcal{F}$) where $\mathcal{A}$ is the set of all open predicates of D. D is then the 'abductive logic program' and the expressions of $\mathcal{F}$ are the integrity constraints.

COROLLARY 2.1 In this setting, there is an one-to-one correspondence between open predicates in ID-Logic and abducibles in ALP. In the following, they are used as synonyms.

THEOREM 14
For each ID-*Logic theory an equivalent Abductive Logic Program (under the two-valued well-founded semantics) exists.*

PROOF 15
The theorem follows trivially from theorem 11 and lemma 1.

Abductive Normal Logic Programs

The above lemma and theorem hold under the assumption that the abductive logic program allows the same general program rules as ID-Logic. However, mostly the logic programs are restricted to normal logic programs. We show here how to obtain abductive normal logic programs from such a general representation. That transformation is correct for programs that do not introduce recursion through negation (or a syntactic criterion: it is correct for programs without a universal quantifier inside the body).

DEFINITION 2.38 An *abductive normal logic program* is a logic program consisting of normal program rules

$$p(\overline{t}) \leftarrow l_1 \wedge \ldots \wedge l_n$$

where the body literals l_i belong to one of the following types:

- defined atoms (based on predicates that appear in the head of some rule)
- abducible atoms (based on predicates that do not appear in the head of any rule)
- negated defined atoms.

The transformation of an Abductive Logic theory to an abductive normal logic program is based on the following Lloyd-Topor transformation [188] for First Order Logic sentences. This procedure transforms any sentence into a normal logic program.

The transformation acts on a set of program rules R. FOL statements (the integrity constraints), e.g. G, are transformed by converting them first into the auxiliary denial program rules, e.g. **false** $\leftarrow \neg G$. In the following, we denote with $F[A]$ the subformula A in the formula F. A and B denote formulas.

1. Replace the connectives $A \rightarrow B$ and $A \leftrightarrow B$ in each $F \in R$ with their equivalent formulas $\neg A \vee B$ and $(\neg A \vee B) \wedge (\neg B \vee A)$.
2. Propagate the negation and universal quantifiers as low as possible by applying on each rule $F \in R$ the rules
   - $F[\neg(A \vee B)] \rightsquigarrow F[(\neg A \wedge \neg B)]$
   - $F[\neg(A \wedge B)] \rightsquigarrow F[(\neg A \vee \neg B)]$
   - $F[\neg\neg A] \rightsquigarrow F[A]$
   - $F[\neg(\forall\, \overline{X}.A)] \rightsquigarrow F[\exists\, \overline{X}.\neg A]$
   - $F[\neg(\exists\, \overline{X}.A)] \rightsquigarrow F[\forall\, \overline{X}.\neg A]$
   - $F[\forall\, \overline{X}.(A \wedge B)] \rightsquigarrow F[A \wedge \forall\, \overline{X}.B]$ iff $vars(A) \not\subseteq \overline{X}$.
   - $F[\forall\, \overline{X}.(A \vee B)] \rightsquigarrow F[A \vee \forall\, \overline{X}.B]$ iff $vars(A) \not\subseteq \overline{X}$.
3. (a) Drop all existential quantifiers.

   (b) Replace every universal quantifier that is left by the introduction of auxiliary predicates. Each formula $q(\overline{t}) \leftarrow F[\forall\, \overline{X}.B(\overline{X},\overline{Y})] \in R$ is replaced by the two formulas $q(\overline{t}) \leftarrow F[\neg p^*(\overline{Y})]$ and $p^*(\overline{Y}) \leftarrow \neg B(\overline{X},\overline{Y})$, provided that none of the predicates in B depends via a program rule in R on the head predicate q.

4. Apply steps 3 and 4 on the newly introduced rules until all rules in R consist of conjunctions and disjunctions of literals.

5. Finally, normalize the rules by removing the disjunction: $H \leftarrow F[A \vee B]$ is replaced by $H \leftarrow F[A]$ and $H \leftarrow F[B]$

DEFINITION 2.39 The abductive normal logic program of an Abductive Logic theory ($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$) is the program $\mathcal{P}^*$ obtained by the application of the Lloyd-Topor transformation on each formula: $\mathcal{P}^* = \{P_R | R \in \mathcal{P}\} \cup \{P_F | F \in \mathcal{IC}\}$ where P_R (P_F) is the program obtained for the program rule R (the integrity constraint F) by applying the Lloyd-Topor transformation.

PROPOSITION 2.5 An Abductive Logic theory $\mathcal{T}$ based on $\mathcal{L}$ and its derived abductive normal logic program $\mathcal{P}^*$ are equivalent in $\mathcal{L}$ w.r.t. well-founded semantics.

PROOF 16
It is easy to see that the Lloyd-Topor transformation preserves the equivalence for all rules, except for 3.b), since it does not alters the language.

For the rule 3.b) it holds that under the given condition, i.e. that the selected subformula does not contain predicates that depend on the head predicate, the transformation will not introduce any recursion through negation. Therefore it is ensured that the programs before and after this rule assign the same truth-value to the shared predicates. Hence the final logic program is equivalent to the original program in the original language.

To show that the condition is needed, we present the following counter-example. Consider the program rule $p(X) \leftarrow \forall\, Y.p(Y)$. The transformation results in the program after applying step 3.b.

$$\begin{array}{l} p(X) \leftarrow \neg q \\ q \leftarrow \neg p(Y) \end{array}$$

For the language $\mathcal{L}$ consisting of a single constant a, the first program has the single model $\emptyset$. On the contrary the transformed program has two models $\{q\}$ and $\{p(a)\}$. When these models are projected on the language of the original program, it follows that the first model $\{q\}$ corresponds to the $\emptyset$ model. The second model has no corresponding model and hence must have been introduced by the transformation. This model is possible for the transformed program since one can assume that q is false, a possibility which is not available in the original program. Hence, in this case the transformation is not equivalence preserving.

Due to this transformation, the distinction between integrity constraints and definitional knowledge disappears. We have argued previously that this distinction is important at the level of knowledge representation. But, as the transformation shows, the distinction can be compiled away. This is important for the solver designers, because it gives the choice either to develop an abductive system that reasons directly with the Abductive Logic theory, with a separate treatment of integrity constraints and the logic program, or one can design a system for which this distinction is irrelevant. The former is followed e.g. by ACLP [161], the latter is followed by e.g. SLDNFA [96]. We will follow the latter approach.

NOTE 2.3.1
To simplify the abductive proof procedure, the truth-value **false** is usually replaced by an auxiliary predicate as head of the denials. When applying this, it is required to adapt each query: given a query Q and f the auxiliary head predicate, $Q \wedge \neg f$ is the updated query which is used in the query answering process.

2.3.5 Abductive Logic Programming frameworks

Over the years many Logic Programming systems that perform abductive reasoning have been developed. The first procedures are those of Eshghi-Kowalski [116, 117, 170] and Kakas-Mancarella [156, 157, 154]. This overview focuses on the most relevant, and more recent, systems for our purpose: SLDNFA [96], IFF [129] and ACLP [152, 161]. They share that they deal with the first order specifications directly, i.e. these systems use resolution to compute the abductive explanations that results in a top-down evaluation strategy. These three procedures form the foundations for the implementation of the $\mathcal{A}$system. There is a second category that is based on propositional languages and mostly applies bottom-up computations e.g. [235, 183, 184]. As these approaches are mostly based on the Stable Model semantics this track of research is absorbed by the Answer Set Programming domain. (See section 2.4.2.)

We give a short overview of each of the selected systems, together with their main applications.

- SLDNFA [95, 96, 90] is an abductive extension of SLDNF. The procedure computes answers for queries w.r.t. abductive normal logic programs. This procedure is proven sound and complete with respect to the three-valued completion semantics.

  The development of SLDNFA has been motivated by work on temporal reasoning and reasoning with incomplete knowledge. SLDNFA was an answer to questions raised by the early attempts [202, 100] of implementing AI-planning by the Event Calculus [170]. One of these contributions was the correct handling of non-ground abduction, i.e. abduction of atoms with variables. Also in context of AI planning [100, 102] describe an extension of SLDNFA with a constraint solver for total orders, which was one of the first integrations of abduction with constraint solving.

  The integration with constraint solvers has been further explored within the context of the Tractebel experiment [238, 104]. The goal of the experiment was the modeling and computation of an optimal solution of a complex scheduling problem: the maintenance schedule for the power producing units of the Belgian electricity provider Tractebel. The modeling resulted in a compact formal description of the problem only consisting of eleven constraints. Computationally this description was evaluated via a system composed of a loose coupling of SLDNFA with ROPE [267, 266, 265], a Finite Domain constraint solver. The compact high level description was reduced to a finite domain constraint store by SLDNFA, and this constraint store was searched for an optimal solution by ROPE. The example has also another important contribution: it showed the importance of aggregate expressions in declarative problem solving. It stimulated us to add aggregates to the $\mathcal{A}$system [263].

Continued research by Denecker on the logical level, has led to the definition of the knowledge representation language ID-Logic. One of the goals of ID-Logic is to provide a clear and founded explanation for the representational and epistemological aspects of Abductive Logic Programming. Our work, presented in this thesis, is a continuation of the research on the computational aspects of SLDNFA.

- The IFF-procedure [129] can be seen as a hybrid of the iff-abduction procedure of Console et al. [85] and SLDNFA. An IFF-theory consists of *if and only if*-definitions, representing the abductive program $\mathcal{P}$, and implications, representing the integrity constraints. Like SLDNFA, the IFF-procedure is sound and complete with respect to the three-valued completion semantics. A notable difference is the way it is formulated. Instead of an extension of SLD-resolution, it is described as a formula rewrite system. That yields a compact and intuitive representation of the inference rules.

  The IFF-procedure has been mostly applied in other problem areas than SLDNFA and ACLP. Whereas SLDNFA and ACLP mostly focus on planning and scheduling problems, the IFF-procedure has been applied in the context of multi-agents and (distributed) databases. For example, it is used to model the reasoning of agents and support the communication amongst agents [169]. [231] studies the problem of resource allocation in a multi-agent systems. Within the area of databases, the IFF-procedure has been applied to manage the active rules of a database [229, 230]. Another application is the design and reasoning of a mediator based system to integrate the information of multiple sources [232, 277].

- ACLP [4, 152, 162, 161] is some years younger than the other two. This system was the first to give constraint solvers a prominent place in an abductive system. It actually extends Constraint Logic Programming with abduction. Given a constraint logic program over a constraint domain $\mathfrak{D}$, a

ACLP($\mathfrak{D}$)-program consists of rules

$$p_0 \leftarrow c_1, \ldots, c_n, p_1, \ldots, p_m, a_1, \ldots, a_k$$

where p_i are user defined predicate atoms, a_j abducible atoms and c_l constraints on the domain $\mathfrak{D}$. The integrity constraints are kept separately, and are formulated in the form of denials (see Section 2.3.4).

The semantics of an ACLP-theory are the Stable Model semantics. No completeness theorem has been proven. This is probably provable by weakening the semantics to Partial Stable Models semantics (similar as the weakening of SLDNFA and the IFF-procedure to three-valued completion semantics) [161].

During experiments with the ACLP-system, Kakas et al. [158, 162] observed that the constraint store which is constructed during the derivation is able to detect inconsistent branches early in the derivation process. By exploiting this in the ACLP system, the search space is more efficiently traversed than e.g. SLDNFA does.

The merits of ACLP system include also the first large application of an abductive system in an industrial context: the scheduling of the crews for the flights of Cyprus Airways [159, 160, 200]. The ACLP system was able to produce solutions that were of good quality according to the company's experts. Especially the re-scheduling module that computes new schedules on the basis of the existing one with respect to changed needs was judged very useful. This was both due to the speed of finding an adequate solution and the flexibility of the system to express the new requirements.

Other abductive systems

A system that holds the middle between top-down and bottom-up is the AbDual system [11]. AbDual performs abduction for ground abductive programs by exploiting the tabling mechanism of XSB-Prolog. Simply stated, tabling is a memoization technique for SLDNF. It stores (intermediate) computed answers in tables so that, when needed, these answers can be reused in other computations. In this way redundant computations can be avoided, but more important, the termination behavior is greatly improved. It has been shown that SLG-resolution, i.e. the extension of SLDNF with tabling, is sound for the three-valued well-founded semantics.

AbDual computes for a ground abductive normal logic program a dual program consisting of the logic program and a set of rules $not(p) \leftarrow Fails_p$ for each defined predicate p where $Fails_p$ is an expression encoding that all the rules of p are false. An abductive solution is computed by the tabled execution of the query with respect to the original program augmented with its dual.

The multi-agent context is a fruitful application area for abductive reasoning, because incomplete information appears naturally. ALIAS [12, 75, 249] is a framework that supports abductive reasoning for agents in a multi-agent environment. It implements a distributed protocol to coordinate the reasoning of all the abductive agents in the system, inspired on the ALP framework of Kakas and Mancarella [156]. The global knowledge shared by all agents is represented by a set of abduced hypotheses posted in a global tuple space. The agents themself are equipped with a deductive and an abductive reasoner. When the agents perform actions, they are allowed to update the global knowledge provided that this yields a consistent state. This framework is used to study problems that arise in multi-agent systems, e.g. co-operation and competition [76].

2.3.6 Integration of CLP and ALP

We give here an overview of work in which Abductive Logic Programming and Constraint Logic Programming is combined.

The relationship between CLP and ALP has been recognized quite early. An overview of these first integrations and relationships is found in [153]. According to one view, constraints can be seen as a special kind of abducibles. In that view, the intended meaning of a constraint is given by a set of integrity constraints. E.g. the inequality $\cdot < \cdot$:

$$\begin{array}{c} \forall\, X, Y, Z. X < Y \wedge Y < Z \rightarrow X < Z. \\ \forall\, X, Y. (X < Y \vee Y < X). \\ \forall\, X. \neg X < X. \end{array}$$

An actual constraint problem corresponds to a (large and complex) query for the abductive program. A solution of the constraint problem is given by the abductive solution of the query w.r.t. the abductive program. Most applications of this view have been applied on equality (to deal with non-ground abducibles) [116] and inequality (in the context of planning) [102].

However, ALP is not really suited for designing constraint solvers. It lacks procedural operations, used in constraint solvers, such as simplification e.g. $3 < X \wedge 5 < X \equiv 3 < X$; therefore, this view has not been continued in later work, except of PROCALOG [273]. Other frameworks such as Constraint Handling Rules deal better with this procedural aspects in constraint solving and should be used for this purpose. Constraint handling rules (CHR) [127, 128] form a generic and flexible platform to define (special purpose) constraint solvers. Abdennadher et al. [2] show that by extending the CHR framework with explicit disjunction, abductive reasoning can be performed. However using CHR to model the abductive reasoning for a problem is *not* the same as using ALP (as defined above) to solve it. The reason is that CHR leads to less declarative programs than the original ALP. That is due to the strong procedural reading and execution of CHR-rules. Such a strong connection makes that the declarative reading of a CHR-program is completely superseded by the procedural interpretation of the rules. CHR is, thus, not a good knowledge modeling platform, but merely a framework to which an abductive task can be compiled or in which an abductive solver can be developed.

Bressan et al. [136, 60] implemented an abductive reasoning system based on [153] to compute query plans in a mediator-based system. The implemented abductive system (on top of Prolog) uses different inference techniques for the abductive program and the integrity constraints. On the one hand the abductive program is evaluated by a meta program; on the other hand the integrity constraints are formulated as CHR's and evaluated by the CHR-system. The result is a hybrid between ACLP [160] and the previous mentioned full CHR implementations.

In [172, 273, 171, 272], Kowalski et al. present a unifying framework PROCALOG for ALP, CLP and Semantic Query Optimization. PROgramming with Constraints and Abducibles in LOGic (PROCALOG) [273] is a glass-box approach to Constraint Programming, in which the constraints are expressed in a high level modeling language. In this respect PROCALOG is similar to CHR, although it didn't go so far in the glass-box vision as CHR. One can view this work as using abductive inference to build a constraint solver. The underlying procedure is a generalization of the IFF-procedure. This framework has been used to study a number of constraint problems, e.g. Job-shop scheduling [248].

The other view is to integrate CLP and ALP as autonomous identities cooperating in one system in order to solve a complex problem. One of the first attempts to integrate both in this view is the work by Codognet et al. [80, 79]. They propose an integration of abduction in the concurrent constraint language ccFD. In the same spirit, the most influential and known integration of abductive reasoning and constraint reasoning is that of ACLP by Kakas et al. [4, 152, 162, 161].

The abductive framework SLDNFA has been the subject of two independent integrations with CLP. We have already mentioned the extension with ROPE to solve the *Tractebel* problem. Independently, SLDNFA was extended with constraints to support multi-agent reasoning. Hayashi [140] presented an integration that extends the SLDNFA procedure in a similar fashion as we do, based on the integration of Kakas et al [161]. His procedure is extended with the constraint $<$ over the real numbers. These constraints are used to check the satisfiability of partial orderings of time points constructed in planning (using the Event Calculus). This procedure is sound and complete w.r.t. the completion semantics. Notably, he included constructive negation in the system. But, he defines his inference rules in a similar notation as [96], and therefore suffers from the same representational drawback as SLDNFA.

Recently Gavanelli et al. [132] repointed to the resemblance between constraint solving and abductive reasoning. Where in ACLP [161] and our $\mathcal{A}$system the constraint reasoning is mostly restricted to special constraint expressions that involve variables of the abducibles, they focus on redefining the abductive inference in a constraint processing style. In [132], they present a compiler for ground abductive programs to CHR-rules and suggest how this might be extended to the first order case.

2.4 Other knowledge representation frameworks

Here we present two related approaches, namely Description Logics and Answer Set Programming. We restrict ourselfs to the basic concepts.

Of both, Description Logics [28] has a longer history; its development started in the early eighties. The domain is known for its in-depth study of the expressiveness and complexity of its languages. This gives the user the potential to choose for a given problem of which the complexity is known the appropriate modeling language. The applications using Description Logics are mostly related to maintaining and manipulating the knowledge structure of a domain. For example, dealing with (large) ontologies, conceptual models, web-stored information, etc. Also the problem of data integration which is the subject of chapter 4, is tackled by means of Description Logics.

The other approach, Answer Set Programming (ASP) [31], is a new emerging domain that has high potential to be used for declarative problem solving. The language allows to formulate the domain knowledge in a precise and declarative way. More important, the domain has developed several reasoning systems with outstanding performance. Most applications of ASP are located in combinatorial search problems, such as AI planning, CSP problems and scheduling problems.

2.4.1 Description Logics

Description Logics [28] are a family of logics that originate from one of the first formal knowledge representation frameworks [174]: the semantic nets [220] and frame based systems [201]. Semantic nets represent knowledge as a network of nodes that represent concepts and links denoting the relationships between the nodes. In 1975, Woods [276] showed that semantics networks, although having a formal definition, were ambiguous which makes them unusable for knowledge representation. Following on that, the semantic nets framework was disambiguated and reformulated as a logic by Brachman and Levesque [57, 54]. In addition they were the first to point to the important difference in the type of knowledge: definitional[6] and assertional knowledge. They observed that semantic nets and frame-based languages [201] are well suited to represent definitional knowledge, while FOL is excellent to represent assertional knowledge. As already mentioned this is also fundamental to ID-Logic. Branchman and Levesque incorporated this

[6] In the first papers, this type of knowledge is also denoted as terminological knowledge.

in the first DL-system KRYPTON [54, 55] that combined definitional knowledge in a frame-based language and assertional knowledge in FOL.

One aspect which is truly studied in depth in Description Logics is the relationship between the expressiveness and the complexity of reasoning for a particular logic. Brachman and Levesque [58] showed that different combinations of language constructs might have different computational reasoning properties. One proposed way to use these results, is that the domain expert can select the best language to solve his problem based on ease of modeling (expressiveness) and (worst case) computational complexity (reasoning). Description Logics often will restrict the expressiveness of their language in order to have good computational complexity. This is done both at the level of syntax and at the level allowed formulas.

Description logics use a slightly different syntax than First Order Logic. The DL-syntax consists of unary predicates, called *concepts* and binary predicates, called *roles*. Predicates with higher arity are not included. Because only these relations are allowed, the arguments are made implicit. For instance, the concept FATHER is equivalent to the predicate $father/1$ and the role PARENT to the binary predicate $parent/2$. Constants, called individuals, are denoted by $a, b, c, \ldots$ Function symbols are not included. Each logical connective has an equivalent counterpart. For example, $A \sqcap B$ denotes the conjunction, $\neg A$ denotes the negation. Existential quantification is used in a restricted setting $\exists$ R.C, corresponding to the formula $\exists\ Y.r(X,Y) \rightarrow c(Y)$. Universal quantification is analogously. The truth-value **true** is represented by the universal concept $\top$, and **false** by the empty concept $\bot$. Description Logics were among the first logics to deal efficiently with cardinality expressions (called number restrictions): e.g. $\leq$ n C ($\geq$ n C) denotes that C has at least (at most) n elements.

A DL-theory describes the properties of the concepts. This is either by 'definitions' in which a concept is stated to be equivalent with a formula, or by stating that one concept is a subset of another. In general, these expressions do not refer to any particular constant. For this reason, the knowledge they represent is called terminological knowledge. The following concept definition defines the concept FATHER as a male (MALE) person (PERSON) that has at least one child ($\exists$ HASCHILD.$\top$).

$$\textsc{Father} \equiv (\textsc{Male} \sqcap \textsc{Person}) \sqcap \exists\ \textsc{HasChild}.\top$$

With regard to expressiveness, Description logics mostly limit definitions to hierarchical structures of concepts. The most expressive DL's have special constructs to define the transitive closure of a role.

Statements that involve constants are normally categorized as assertional knowledge. For example, the assertion HASCHILD(john,mary) states that John has Mary as child. The distinction between assertional and terminological knowledge is vague; each DL (and system) may draw the line at a different place. A DL-theory is then a pair (T,A) where T is a set of definitions, and A is a set of assertions.

T is called the terminological box or the T-Box. A is the assertional box or the A-Box.

The semantics (in particular w.r.t. the assertional knowledge) varies from one Description Logic to another, but most of them will accept the unique names axioms and do *not* impose the domain closure axiom. Consequently there is no closed-world assumption. For example, the above assertion HASCHILD(john,mary) must be interpreted as that John has at least Mary as child. Indeed, this knowledge does not prevent John to have other children than Mary, contrary to what would be inferred by the closed-world assumption. Note that as a consequence the basic concepts, i.e. those that have no definition, have an open interpretation. A DL-theory corresponds thus to a logic theory with incomplete knowledge. Due to the exclusion of inductive definitions, a Description Logic allows to specify only 'simple' theories for which all main formal semantics (completion/well-founded/stable) coincide. For this reason, the study of formal semantics for Description Logics has not draw much attention.

Since a DL-theory T can have multiple models, the entailment of a formula F w.r.t. T is defined (as usual) as T entails F ($T \models F$) iff for each model M of T holds that $M \models F$. In DL's the entailment often is expressed by means of the *subsumption* relation. A concept C is subsumed by a concept D with respect to a T-Box $\mathcal{T}$ if for every model M of $\mathcal{T}$ holds that $M \models$ C $\rightarrow$D. It is denoted by C $\sqsubseteq$ D. Logical entailment of a formula F w.r.t. a T-box $\mathcal{T}$ is then $F \sqsubseteq \perp$ w.r.t. $\mathcal{T}$.

To compute the entailment of a formula, first a classification graph is computed. In this graph all concepts are classified according to the subsumption relation. Based on this information the queries of the user are answered. Such compilation approach is beneficial if many queries must be answered against the same T-Box. Also, the simplified syntax can be exploited to optimize the reasoning. For some cases it reduces to syntactic manipulations which is much cheaper than resolution based reasoning. However, when the T-Box changes the whole compilation must be redone and this can be very time-consuming.

Van Belleghem et al. [255, 254] has demonstrated a strong relation between the early version of ID-Logic, OLP-FOL, and Description Logics. In order to do so, a basic DL, $\mathcal{ALCN}$, is embedded in OLP-FOL such that there is an one-to-one mapping between one interpretation of the DL and one of OLP-FOL. Every atomic DL-concept without a definition in the T-Box corresponds to an open predicate. The definitions in the T-Box are definitions in ID-Logic and all the rest of the knowledge is assertional and belongs in the FOL part of an ID-Logic theory. They also studied the correspondence between SLDNFA and some DL reasoning algorithms.

Their conclusions state that the OLP-FOL/SLDNFA approach has two severe drawbacks compared to Description Logics [254]. Firstly, the number restrictions of Description Logics has no compact counterpart in OLP-FOL. The notion has to be expressed in OLP-FOL as a number of (large) integrity constraints. Thus the corresponding OLP-FOL specification is polynomially larger as the original $\mathcal{ALCN}$ specification. Secondly, the prototype of SLDNFA of that moment had too poor efficiency to be used. This poor efficiency dropped even further when the SLDNFA-prototype had to deal with the number restriction encodings.

ID-Logic can be viewed as a very expressive Description Logic using the classic logic presentation. The formal semantics of ID-Logic allows non-monotone inductive definitions, which cover all notions of definitions defined in the context of Description Logics. As assertions in Description Logics are First Order formulas, they are identical in ID-Logic. Recent work (including this dissertation) addresses the parts which were not satisfactory according to Van Belleghem. The number restrictions are a special form of cardinality constraints. By the extension of ID-Logic with aggregates [101, 212], the number restrictions are as compact as by Description Logics formulated in ID-Logic. Moreover, it allows many more similar expressions. The complaints on the computational performance are tackled by the $\mathcal{A}$system [163, 263], an improved version of SLDNFA.

2.4.2 Answer Set Programming

Answer Set Programming [31] (ASP) is a new star at the AI-firmament. This research community stimulates declarative problem solving research at different levels: from knowledge representation to the implementation of more efficient reasoning algorithms, with an important eye on applications. The name refers to the fact that an answer to a query is a set of atoms instead of an answer substitution as in traditional Logic Programming. Like Description Logics, ASP is a family of different logics sharing the same formal semantics: Stable Model semantics [134].

The success of ASP is strongly connected with the existence two efficient solvers: sModels [208, 240] and DLV [112, 105]. sModels is an ASP solver for range restricted logic programs. A *range restricted program* is a normal logic program such that every variable in a rule is range restricted, i.e. is limited to a finite number of possible values by one of the positive literals in the rule. DLV is designed for extended disjunctive logic programs. A *disjunctive logic program* is a logic program that allows disjunction in the head. *Extended logic programs* make a distinction between explicit negation ($\neg$) and default negation (*not*). Default negation is negation by assumption; *notp* holds when the contrary p is not provable (similar to negation as failure). Explicit negation is classical negation.

In contrast to most Logic Programming systems, including LP, ALP and CLP, these solvers reason on a ground representation of the logic program. This is remarkable as it used to be accepted that reasoning with a ground representation is less efficient, because the whole program must be grounded. To obtain the ground representation, sModels introduces a preprocessing phase (the program lparse) which grounds the logic program. In DLV, the grounding is integrated in the system. Another distinguishable fact is that both are model generators: they completely construct the model of a program. This has as drawback that no goal directed search can be performed (as is done by SLD-resolution). Recently both solvers have been extended with aggregates (cardinality and sum), weak constraints and weight constraints.

At the representational level, ASP is more than sufficiently expressive, however it lacks probably even more than ALP epistemological foundations. ASP has only one kind of knowledge: a program rule. There is no distinction between definitional and assertional knowledge. Integrity constraints in denial form $\leftarrow I$ are expressed via the rule

$$p \leftarrow I, not\ p.$$

where p is a predicate that only occurs in the above rule. Because p cannot be true, I must fail.

Also the important difference in ALP between defined and open (abducible) predicates is somewhat blurred. First of all, each ASP language has its own way to represent defined and open predicates. In range restricted normal programs an open predicate p is represented by a set of *choice rules*:

$$\left\{ \begin{array}{l} p \leftarrow\ not\ p^*. \\ p^* \leftarrow\ not\ p. \end{array} \right\}$$

Disjunctive logic programs use the disjunctions in the head to express openness:

$$p \vee p^* \leftarrow .$$

In both formulations a new predicate p^* is introduced, representing the negation of p. Although precisely capturing the meaning of openness, from a methodological point of view it is a rather confusing way of expressing. In a propositional encoding, it is still often perceived as a reasonable representation. But, when modeling in First Order Logic, using one predicate to denote the negation of another predicate becomes counterintuitive. The first reason is consistency: without explicit statements by the user, the intended meaning and the formal meaning might not coincide. It becomes the user's responsibility to match them. Secondly, using two predicates that are each other's counterpart at the same time leads to less readable specifications.

ALP and ASP are closely connected. An embedding of ALP (based on the Stable Model semantics) in ASP is given by the following transformation [236]. In section 2.3.4, an Abductive Logic theory ($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$) has been transformed to an equivalent abductive normal program $\mathcal{P}^*$. In an analogous way, the ASP-embedding is constructed. Consider first a *propositional* Abductive Logic theory $\mathcal{T}_p = (\mathcal{P}_p, \mathcal{A}, \mathcal{IC}_p)$, then the corresponding ASP theory is the propositional normal program

$$\begin{array}{rl} \mathcal{T}_{asp} = & \mathcal{P}_p \cup \{p \leftarrow not\ p^*.p^* \leftarrow not\ p | p \in \mathcal{A}\} \cup \\ & \{false \leftarrow I | I \in \mathcal{IC}\} \cup \{f \leftarrow false, not f\} \end{array}$$

This is an elaboration of the Kakas and Mancarella definition of ALP [156]. Again, by adding a prior grounding, first order theories can be handled. The embedding can also be achieved by a more fine-grained transformation that ensures the groundability of the resulting normal logic program.

For some classes of programs the two-valued well-founded models coincide with the Stable Models. As a consequence via the above embedding, ASP-solvers can be used as ID-Logic model generators for theories that belong to those classes.

Chapter 3

The $\mathcal{A}$system, an abductive constraint system

3.1 Introduction

Although much fruitful research has already been done in the field of abductive reasoning, the acquired insights have not yet resulted in a system/toolbox available in a mainstream Logic Programming environment. Compared to Constraint Programming and the even more recent technology of Constraint Handling Rules, abduction is simply absent in any Logic Programming toolbox offered to a logic programmer at the moment of this dissertation (2004).

The $\mathcal{A}$system, the "physical" contribution of this dissertation, is the result of the study of abductive reasoning systems with the aim of providing extra insight in implementation aspects. From the logical inference perspective, the $\mathcal{A}$system is a mixture of three abductive proof procedures. The basis of the $\mathcal{A}$system is SLDNFA [96]. The $\mathcal{A}$system is a reformulation of SLDNFA as a state rewrite system, similar to the IFF-procedure [129]. We have added to these rules extra rules for pre-defined constraint domains. During an $\mathcal{A}$system execution, the constraint expressions will be evaluated by the corresponding constraint solvers. This idea originates from ACLP [161]. By extending the proofs of SLDNFA, we show that the $\mathcal{A}$system is sound and complete w.r.t. the three-valued completion semantics.

This multi-solver foundation determines largely the design of the $\mathcal{A}$system. It not only improves the performance of the abductive solver on some problem classes (since constraint solvers handle a problem more efficiently than a resolution based system), but also leads to a flexible framework in which the $\mathcal{A}$system's language and reasoning capacities are easily extended. The former is studied for two constraint solvers: the equality constraint solver over finite terms and the finite domain constraint solver. The latter is shown by one extension: namely for open functions.

Having solved the subproblems efficiently, is, however, not sufficient for an overall good performance. Therefore, the abductive inference is considered as a search problem. In contrast to SLD-resolution that relies on a fixed selection strategy, the $\mathcal{A}$system search is a mixture of active pruning and exhaustive systematic search. Generally spoken, the used strategy is a *least commitment* strategy. Choices are postponed as long as possible, until the system must select one. For a selected choice, the system commits to one of its branches and starts a propagation process in which the effects of this commitment are spread to all other choices. In this propagation process, the constraint solvers have an important role, since almost all propagation will pass through them. It is of importance that this propagation is done as well as possible. The derived information is able to cut down the search space.

Experimental results show that the $\mathcal{A}$system is a more robust and performant system than its SLDNFA, ACLP and other ALP ancestors. It is developed on top of Sicstus Prolog [239]. Currently version 3.10.1 or better is needed for correct executions due to some bugs in earlier versions of Sicstus Prolog. The system is freely available for research and educational purposes at `http://www.cs.kuleuven.ac.be/~dtai/kt/`.

The outline of this chapter is as follows. First the $\mathcal{A}$system's specification language will be presented, followed by the proof procedure of the basic version of the $\mathcal{A}$system with soundness and completeness theorems. (Sections 3.2 and 3.3) The next sections will focus in detail on different aspects in the system. Section 3.4 will discuss the general implementation framework. In particular, it will describe how the inference rules are organized so that the search for a solution is effective. The section also presents the main data structures and discusses two ways of implementing the inference rules as a meta-program. Sections 3.5 and 3.6 will focus on the interactions with the self implemented (dis)equality solver resp. the interactions with the finite domain solver. The implementation of the $\mathcal{A}$system is based on reification. Section 3.7 will present an extension of reification that is suited for the $\mathcal{A}$system. Section 3.8 will present an extension of the $\mathcal{A}$system with a special treatment for a special class of abducibles: open functions. This extension shows how the $\mathcal{A}$system can be extended with other constraint solvers. The Section 3.9 will present an experimental evaluation of the $\mathcal{A}$system and compares it to related systems. Finally, we will conclude with some general comments.

3.1.1 Our contributions

The first prototypes of the $\mathcal{A}$system, called SLDNFAC, extended SLDNFA in a similar fashion as ACLP. At that stage, SLDNFAC [103, 258] already contained some extra inference rules which were not present in the ACLP system. These rules checked for special constraint expressions in which values of the universally quantified variables are enumerable. For example, the first literal of the denial

$$\forall\, X. \leftarrow X \text{ in } 1..5, not(p(X)).$$

is a finite domain constraint expression limiting the possible values of the universally quantified variable to the set {1,2,3,4,5}.

Experiments with SLDNFAC and ACLP showed that a resolution style implementation was unable to solve many problems. For example the order of the goals in planning examples influenced directly the efficiency. Another observation was that in general neither the SLDNFAC, nor the ACLP system compute a solution for CSP problems without generating backtrack points, although they should. These experiences initiated the project of the $\mathcal{A}$system [163, 264].

Based on the initial experiences, we further developed and experimented with the $\mathcal{A}$system. It turned out that the equality reasoning as implemented in the version of [163] was the source of many deficiencies. Therefore we improved this equality reasoner to a full fledged (dis)equality constraint solver for terms. (See Section 3.5). Another improvement is the development of a special purpose meta-programming encoding for denials such that less computational overhead is produced. This is discussed in Section 3.4. Additionally, this chapter gives a complete overview over the ideas and problems that have been considered during the development of the $\mathcal{A}$system. Some of these have, unfortunately, failed or got no concrete solutions, and need further exploration. Especially, we like to mention the interaction with the finite domain solver (see Section 3.6). This integration has been troublesome from the beginning. The main reason is the incompleteness of the finite domain solver, i.e. its internal propagation mechanism is weaker than expected and needed for the $\mathcal{A}$system.

Section 3.9 compares the $\mathcal{A}$system with several other systems. The results affirm the conclusions from an early comparison by Pelov et al. [213, 214]. On particular examples, the abductive approach of the $\mathcal{A}$system results in an excellent performance compared to the others. In particular on CSP problems, like e.g. the N-queens problem, the approach to reduce a high level specification via abductive inference to a constraint store was very beneficial.

3.1.2 The use of Prolog as implementation language

Although Prolog is an obvious choice as implementation language for the $\mathcal{A}$system because of its roots in Logic Programming, it is useful to stress some other reasons as well. In addition, I will comment shortly on my experiences during the implementation.

Previous abductive prototypes showed that abductive reasoners can be easily implemented as a Prolog meta-program. Meta-programming is a technique to design relatively easy programs such as interpreters, debuggers and compilers that treat other programs as input. This technique is intensively exploited in the

$\mathcal{A}$system. It has resulted in a flexible implementation with respect to new ideas and experiments we wanted to perform. But, as we will show in Section 3.4 the traditional meta-programming approach leads to unacceptably high execution costs when evaluating denials. In order to reduce the overhead, we have introduced a ground meta program for denials.

Although Constraint Programming is nowadays more and more integrated with programming languages such as C, C++ and Java, Prolog is the language that provides the best integration. Many of the Constraint Programming libraries are originally developed in a Prolog context. Consequently, Prolog offers a wide range of constructs (e.g. attributed variables, coroutining, mutable terms) to support the implementation of other constraint domains. This aspect has been enormously valuable to develop the $\mathcal{A}$system.

The implementation of the $\mathcal{A}$system has not been an ideal software engineering process. As we were looking for the right concepts and techniques, several times we had to implement variants of the same logic. Fortunately, the compactness of Prolog code allowed to do this with a minimum of coding effort and time. On the other hand, the limited debugging aid by the Prolog system made it often difficult to find the source of a failure.

A last point we would like to mention is the dependence on the Prolog system. The $\mathcal{A}$system is correct only if the underlying Prolog executes the program correctly. And that is not always obvious. We have integrated many constructs from different libraries and in this process we encountered bugs in the used Prolog systems (ECL^{i}PS^e [110] and Sicstus [239]). Such a dependence is two-sided: if the underlying system is improving the $\mathcal{A}$system automatically improves, but if that system becomes buggy or makes incompatible changes, the $\mathcal{A}$system also suffers from it.

3.2 $\mathcal{A}$system Specifications

The $\mathcal{A}$system language

In this section, we briefly describe the kernel language of the $\mathcal{A}$system. The $\mathcal{A}$system syntax follows the Prolog convention: variables start with a capital; constant, function and predicate names are in lowercase.

The $\mathcal{A}$system language is composed of three logic domains (or constraint domains): the user defined domain, the equality domain and the finite domain domain.

- The user defined domain.
  This subdomain is defined by the alphabet (constants, function symbols and predicate symbols) introduced by the user to model the universe of discourse. The $\mathcal{A}$system supposes that the set of constants is a countably infinite set.

- The predefined domains.

– $\mathcal{E}$: this constraint domain provides term equality and disequality over the Herbrand Universe of the user defined domain. The equality reasoning follows the free equality axioms of Clark [78].

The following table shows the syntax (s and t are terms):

	$\mathcal{E}$	logic
equality	`s:= t`	$s = t$
disequality	`s:=/t`	$s \neq t$

– $\mathcal{FD}$: this domain provides finite domain constraint expressions. It is a subpart of the constraint language defined by Sicstus Prolog [239, 66]. In particular, it provides arithmetic expressions over the integers.

The constants of this sublanguage are the integers. As function symbols exist `+,-,*,/` denoting addition, subtraction, multiplication and integer division. The allowed basic relations are (t and s are $\mathcal{FD}$-terms):

relation type	CLPFD	logic	CLPFD	logic
equality	`s #= t`	$s = t$	`s #\= t`	$s \neq t$
inequality	`s #<t`	$s < t$	`s #=<t`	$s \leq t$
	`s #>t`	$s > t$	`s #>= t`	$s \geq t$
logical	`#\s`	$\neg s$	`s #<= t`	$s \Leftarrow t$
	`t #=>s`	$s \Rightarrow t$	`s #<=>t`	$s \Leftrightarrow t$

Plus also the more complex relations

* `Xin a..b`: shorthand for $\mathtt{X} \geq \mathtt{a} \wedge \mathtt{X} \leq \mathtt{b}$.
* `relation(X,l,Y)`: where X and Y are related according to the relation l. It is a compacted finite domain constraint version of the relation *l(X,Y)*. Although it is a redundant expression for the $\mathcal{A}$system specification language, it is very useful because it can speed up the reasoning.
  In the finite domain solver, l is represented as a list of pairs $\mathtt{a^i} - \{\mathtt{b^i_1}, \mathtt{b^i_2}, \ldots\}$. Informally, the expression reads as if X has value a^i then Y has a value in the set $\{b^i_1, b^i_2, \ldots\}$. Formally, the expression

$$\mathtt{relation}(\mathtt{X}, [\mathtt{a}^1 - \{\mathtt{b}^1_1, \mathtt{b}^1_2, \ldots\}, \mathtt{a}^2 - \{\mathtt{b}^2_1, \mathtt{b}^2_2, \ldots\}, \ldots], \mathtt{Y})$$

denotes the constraint network consisting of

$$\begin{array}{ll} X = a^i \Rightarrow Y \in \{b^i_1, \ldots\} & \text{and} \\ Y = b \Rightarrow X \in \bigcup_{b \in \{b^k_1, \ldots\}} \{a_k\} & \text{and} \\ X \in \bigcup_i \{a^i\} & \text{and} \\ Y \in \bigcup_i \{b^i_1, \ldots\} & \end{array}$$

where $\{b^i_1, \ldots\}$ denotes the set

> The finite domain solver provides many more constraints, which can be used 'in principle' in an $\mathcal{A}$system-specification. However, those constraints are often not meaningful from a declarative knowledge engineering point-of-view because they express properties over a set which is usually represented in DPS as a predicate.

Because each domain is independent, they define different symbols for the same notion. For example, all three define the negation or the complement of a predicate. Because the interpretation coincides, these representation are interchangeable. For example `not (X#>5)` $\equiv$ `#\ (X#>5)` $\equiv$ `X#=<5`. In order to avoid confusion, in this dissertation we will use the negation *not* only for user defined predicates.

The $\mathcal{A}$system problem specification

Having defined the $\mathcal{A}$system language, we now define $\mathcal{A}$system specifications. An $\mathcal{A}$system specification is an ALP theory ($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$), which is represented as an abductive normal logic program. It is composed of the following declarations and program rules.

$\mathcal{A}$ Each user defined predicate in an $\mathcal{A}$system-specification is required to be declared either defined or abducible. This is done using the declarations `defined/1` and `abducible/1`. For example,

```
defined(holds_at(_,_)).
abducible(act(_,_)).
```

declares the predicate $holds\_at/2$ to be defined, and $act/2$ to an abducible.

$\mathcal{P}$ The abductive logic program $\mathcal{P}$ is specified by means of the program rules. A program rule is conjunction of literals. A fact is a rule with an empty body.

For example, this extract from the event calculus [170]

$$holds\_at(P,T) \leftarrow initially(P,A) \wedge \neg clipped(0,P,T).$$
$$holds\_at(P,T) \leftarrow initiates(P,A) \wedge act(A,E) \wedge E < T \wedge \neg clipped(E,P,T).$$

$$initiates(on(A,B), move(A,B)).$$

is expressed in the $\mathcal{A}$system syntax as

```
holds_at(P,T):- initially(P,A),not(clipped(0,P,T)).
holds_at(P,T):- initiates(P,A),act(A,E),
                clp(E#<T),not(clipped(E,P,T)).

initiates(on(A,B),move(A,B)).
```

$\mathcal{IC}$ Integrity constraints are rules with an empty head (denials), denoting situations which are not allowed in the problem domain. To ease the reading (and parsing), the (empty) head of integrity constraints is made explicit by the reserved keyword `ic`.

For example, the FOL expression stating that every action must satisfy its preconditions

$$\forall\ A, T.act(A, T) \rightarrow precondition(A, T)$$

is represented as denial as

$$\forall\ A, T. \leftarrow act(A, T) \wedge \neg precondition(A, T)$$

which is represented in the $\mathcal{A}$system specification as

```
ic :- act(A,T),not(precondition(A,T)).
```

Recall that denials can be obtained via a Lloyd-Topor transformation on FOL formulas (see section 2.3.4).

A complete example specification is found in appendix A.2.

Like integrity constraints, *queries* have to be normalized (using again the Lloyd-Topor transformation) to a conjunction of literals. For example, the query

$$\exists\ T.holds\_at(on(a, b), T) \wedge \neg holds\_at(on(b, c), T)$$

is represented as

```
holds_at(obj_at(a,b),T), not(holds_at(obj_at(b,c),T))
```

REMARK 3.2.1 Our notation of a query differs from the common practice in Logic Programming to denote a query as the formula $\forall\ \overline{X}.\ \leftarrow l_1 \wedge \ldots \wedge l_n$, but as $\exists\ \overline{X}.k_1 \wedge \ldots \wedge k_m$. ($l_i$, i=1..n and k_i, i=1..m are literals.) This is done on purpose. Firstly, the chosen format has a more natural declarative reading. In this way, it reads as a natural language sentence, which is easier to understand for non-logic programmers that are in the (far) end the users of the $\mathcal{A}$system. Secondly, the $\mathcal{A}$system takes above formulas as input.

Preprocessing

The first phase in the $\mathcal{A}$system is a preprocessing step of the specification. The preprocessor analyses and transforms the specification to support and optimize the later solving process. One of these is annotating each literal in the body of a rule or integrity constraint with its sort. As one might have observed, the actual specification language does not make explicit the sorts. Sort explicitation is omitted because this can be done automatically. Other analyses and transformations are elaborated in later sections.

Facilitating the solver implementation is not the only goal of the preprocessing step. It forms a layer between the solver and the user which the $\mathcal{A}$system makes more robust to changes. In this way the solver can be adapted without changing the specification language. Likewise, one can add extra user support without changing the solver. For instance, we added a simple error detection. Each used predicate is required to be declared either as defined or as abducible. On this basis, the preprocessor tries to detect potential inconsistencies by reporting the use of non-declared predicates, the declared predicates which are not used and the not-recognized predefined expressions. Our own experiences show that this simple check detects many of the introduced errors that result from editing a specification.

(Intensional) formal semantics

Recall that it is our intension to build an abductive solver for ID-Logic. In Chapter 2, we have shown how an ID-Logic theory can be transformed into an abductive normal logic program. This relation requires that the models of the abductive normal logic program are *two-valued well-founded* models. Unfortunately, the $\mathcal{A}$system proof procedure is not sound w.r.t. the well-founded semantics, but w.r.t. a weaker semantics: the three-valued completion semantics (denoted as $comp_3$). (For examples and proofs see Section 3.3.4.) Fortunately, for a large class of programs the semantics coincide.

Hence, the formal semantics provided by $\mathcal{A}$system is

DEFINITION 3.1 An interpretation M is model of an $\mathcal{A}$system-specification ($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$) based on the language $\mathcal{L}$ iff

- M is the three valued completion model of $\mathcal{P} \cup \Delta$, $\Delta \subseteq \mathcal{A}$. M is also denoted as $comp_3(\mathcal{P} \cup \Delta)$.
- $M \models \mathcal{IC}$
- $M \models \mathcal{FEQ}(\mathcal{L})$ where the $\mathcal{FEQ}$ are the free equalities axioms.

As usual, the language $\mathcal{L}$ is derived from the specification. However $\mathcal{L}$ must contain at least two constants. Later when discussing the equality reasoning (see section 3.5) this assumption is important. Some of its inference rules rely on it.

3.3 The proof procedure

The basis of the $\mathcal{A}$system proof procedure is the SLDNFA [90, 96] proof procedure. As the name suggests, SLDNFA has been formulated as an extension of SLDNF. That representation lacks, however, readability and moreover it fixes the execution strategy implicitly. Both drawbacks are addressed by the representation of the proof procedure as a formula (state) rewrite system. This approach had been used by the IFF-procedure [129]. Formula rewrite rules simplify the understanding, because each rule describes a local transformation that can be understood without the knowledge of other rules. Also they leave the application order of the inference rules open. Since the order is an important factor for an efficient execution of the $\mathcal{A}$system, it is good not to fix this in advance. Although the strategy of SLDNFA is not fixed, all early prototypes followed the left-to-right, top-down selection strategy which is common in Logic Programming. The new presentation creates a visible distance with SLD(NF), which avoids that implicit assumptions on the evaluation strategy are made.

In addition to the combination of SLDNFA and IFF procedure, we incorporate the multi-solver idea of ACLP [161] in the $\mathcal{A}$system. For some predefined languages efficient first order constraint solvers exist. Their efficiency is exploited by the $\mathcal{A}$system. This is reflected in the existence of inference rules that select an expression of a constraint domain and move it to the corresponding constraint store, where it is evaluated by the constraint solver. The $\mathcal{A}$system proof procedure applies this 'optimization' for two types of expressions: equalities and finite domain expressions. For both, there is a special and efficient constraint solver available.

Notwithstanding the change in representation and the multi-solver extension, the $\mathcal{A}$system proof procedure is still very close to SLDNFA. Therefore, the soundness, completeness and termination conditions of SLDNFA are applicable for the $\mathcal{A}$system.

3.3.1 State Rewriting

An $\mathcal{A}$system execution, trying to answer a query, given an abductive program, can best be explained as the construction of a sequence of states, in which each state is derived from the previous one by the application of an inference rule. These inference rules reduce a formula into other, smaller formulas. Some of those formulas are 'simple' enough to determine their truth value and must not be reduced further, while others need further reduction by other inference rules. In this and the following subsections the order of the inference rules' applications is not considered. Section 3.4 discusses this issue.

To facilitate the inference, an $\mathcal{A}$system state distinguishes between reducible and non-reducible formulas.

Definition 3.2 An *$\mathcal{A}$system state* is a pair $(\mathcal{G}, \mathcal{ST})$ in which $\mathcal{G}$ is a set of *goal formulas* and $\mathcal{ST}$ a set of *basic formulas.*

A *goal formula*, or in short *goal*, is either a conjunction of literals $l_1 \wedge \ldots \wedge l_n$, or a denial $\forall \overline{X} \leftarrow l_1 \wedge \ldots \wedge l_n$. The inference rules take a goal formula as input and transform it to basic and goal formulas. They are discussed below in 3.3.2. *Basic formulas* are expressions that cannot be reduced by the $\mathcal{A}$system inference rules.

For some formulas such as abducible atoms, no further computation is possible, while others e.g. term equalities, can/must be evaluated by specialized solvers.

Because each kind of basic formula has its own treatment, the basic formulas store $\mathcal{ST}$ of a state has a substore for each kind of basic formula. These substores and their corresponding basic formulas are:

Δ for abducible atoms $a(\overline{t})$

Δ^* for denials with an abducible atom in the leftmost position: $\forall \overline{X}. \leftarrow a(\overline{u}) \wedge Q$

$\mathcal{E}$ for term equalities and disequalities

$\mathcal{FD}$ for finite domain expressions

From now on, $\mathcal{ST}$ is the abbreviation of the quadruple $(\Delta, \Delta^*, \mathcal{E}, \mathcal{FD})$.

The stores $\mathcal{E}^*$ and $\mathcal{FD}^*$ for the similar denials as in $\Delta^*$are not needed due to:

- The equality and finite domain solvers deal properly with negation. This allows to replace the general negation by the specific negation of the domain solvers.
- The denials $\forall\, \overline{X}. \leftarrow F \wedge Q$ where F is an atom from $\mathcal{E}$ or $\mathcal{FD}$ and $vars(F) \subseteq \overline{X}$, either are reducible or lead to floundering. Because floundering will abort the derivation, none of these denials needs to be stored.

Each state is associated with a FOL formula, denoting the meaning of the state.

Definition 3.3 The *meaning* $\mathcal{M}$ $(\mathcal{S})$ of a state $\mathcal{S}$=$(\mathcal{G},\mathcal{ST})$ is the conjunction of all formulas in $\mathcal{S}$:

$$\mathcal{M}(\mathcal{S}) = \bigwedge_{F \in \mathcal{G}} F \wedge \bigwedge_{F \in \mathcal{ST}} F$$

The following properties of states are useful.

Definition 3.4 An $\mathcal{A}$system state $\mathcal{S}_=(\mathcal{G},\mathcal{ST})$ is *consistent* if $\mathcal{G} \cup \mathcal{ST}$ is consistent.

A weaker notion of consistency is quasi-consistency.

Definition 3.5 An $\mathcal{A}$system state $(\mathcal{G},\mathcal{ST})$ is *quasi-consistent* when every store in $\mathcal{ST}$ is consistent.

Definition 3.6 A (quasi-)consistent state with an empty goal set $\mathcal{G}$ is a *solution state.*

Query answering

Given an $\mathcal{A}$system-theory ($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$) and a query $\mathcal{Q}$, an $\mathcal{A}$system-derivation starts from the initial state $\mathcal{S}_0$= ($\{\mathcal{Q}\ \}\cup\mathcal{IC}$,$\mathcal{ST}^{\emptyset}$)[1]. $\mathcal{ST}^{\emptyset}$ is the empty store $(\emptyset, \emptyset, \emptyset, \emptyset)$. From this initial state a sequence of states $\mathcal{S}_0$,$\mathcal{S}_1$,... is constructed by subsequent rewritings. One single step consists of selecting a goal formula from $\mathcal{G}$ and applying the corresponding inference rule to obtain the next state. The derivation stops when a solution state is reached; it backtracks when the current state is inconsistent. This process can be described by a derivation tree.

Definition 3.7 An $\mathcal{A}$system derivation tree is a tree in which

- every node is a state and
- the root is the initial state $\mathcal{S}_0$ and
- the children of a node are all the states that can be constructed from that node for the selected formula F.

An $\mathcal{A}$system derivation tree is finite if all branches are finite.

A leaf node is labeled either failed (due an inconsistency) or successful (a solution state).

Definition 3.8 An $\mathcal{A}$system-derivation for a query $\mathcal{Q}$ *fails* when all leaf nodes are labeled failed.

Definition 3.9 An $\mathcal{A}$system-derivation for a query $\mathcal{Q}$ is *successful* when it reaches a solution state after a finite number of steps.

Definition 3.10 (An $\mathcal{A}$system answer) For a query $\mathcal{Q}$ to the $\mathcal{A}$system-theory ($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$) having the successful derivation $\mathcal{S}_0, \ldots, \mathcal{S}_{sol}$, an *abductive answer* is the pair $(\theta, \mathcal{M}(\mathcal{S}_{sol}))$ where θ is the variable substitution for all free variables of $\mathcal{Q}$ induced by $\mathcal{S}_{sol}$.

Usually the user has most interest in the set of abducted atoms Δ_{sol} and the answer is restricted to (θ, Δ_{sol}). When Δ_{sol} is ground then the abductive answer is called ground.

The following (simple) properties are important when discussing the relation with the constraint solvers (see section 3.6).

Proposition 3.1 In a successful branch of a derivation tree all states are consistent.

[1] In practice, $\mathcal{S}_0 = (\{\mathcal{Q}, \neg ic\}, \mathcal{ST}^{\emptyset})$, where ic is the auxiliary predicate introduced by the Lloyd-Topor transformation and which is the head of each integrity constraint.

Since no inference rule will turn an inconsistent state into a consistent state, it holds trivially. Consequently the following forms a necessary condition.

PROPOSITION 3.2 In a successful branch of a derivation tree all states are quasi-consistent.

Hence, it is the responsibility of each underlying solver to inform the $\mathcal{A}$system derivation process that it has become inconsistent such that the derivation process can backtrack. We will see that this is not always the case.

3.3.2 The inference rules

Having elucidated the principles of the derivation, we now present the inference rules of the $\mathcal{A}$system. The following conventions are used:

- $\mathcal{G}_i^- = \mathcal{G}_i - \{F\}$, where F is the selected goal formula from the goal stack $\mathcal{G}_i$ of state $\mathcal{S}_i$.
- OR and SELECT denote nondeterministic choices in an inference rule.
- Q is a conjunction of literals, possibly empty. Since an empty conjunction is equivalent to `true`, the denial $\leftarrow Q$ with empty Q is equivalent to `false`.
- Only changes are reported. If a store is not affected by the rule, the store is the same as in the previous state.
- Important for the $\mathcal{A}$system is the quantification of the variables. The $\mathcal{A}$system proof procedure deals with universally quantified and free variables. The free variables can be understood as globally existentially quantified variables.From now on, we will explicitly denote the universal quantification. The free variables have no explicit quantification.

  To enhance the reading, an additional convention is used: the variables of the terms denoted by the letters s and t will never be universally quantified, whereas those of u and v can be universally quantified. $vars(u)$ denotes the set of variables of the term u. $\overline{X}$,$\overline{Y}$ denotes a set of variables, possibly empty.

The inference rules are classified in several groups, named after the leftmost literal in the selected formula (shown in bold). Each group contains one rule for a (positive) conjunction of literals and one rule for a denial. Later, when we discuss extensions to the $\mathcal{A}$system, this locality and modularity will be exploited.

1. *Basic truth values:* The definition of the conjunction ($\wedge$):

   - **true** $\wedge$ **Q**: $\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{\mathrm{Q}\}$
   - **false** $\wedge$ **Q**: fail
   - $\forall\overline{\mathbf{X}}. \leftarrow$ **true** $\wedge$ **Q**: $\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{\forall\overline{X}. \leftarrow Q\}$

- $\forall\overline{\mathbf{X}}. \leftarrow \mathbf{false} \wedge \mathbf{Q}$: $\mathcal{G}_{i+1} = \mathcal{G}_i^-$

The removal of superfluous variables:

- $\forall\overline{\mathbf{X}}, \mathbf{Y}.\mathbf{Q}$ and $Y \notin vars(Q)$: $\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{\forall\, \overline{X}.Q\}$

2. *Defined predicates:*
The inference rules unfold the bodies of a defined predicate. For positive conjunctions this corresponds to standard resolution with a selected clause, whereas in the denial all clauses are used because every clause leads to a new denial.

D.1 $\mathbf{p}(\overline{\mathbf{t}}) \wedge \mathbf{Q}$:
Let $p(\overline{s}_i) \leftarrow B_i \in \mathcal{P}$ $(i = 1, \ldots, n)$ be n clauses with p in the head. Then:
$\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{\overline{t} = \overline{s}_1 \wedge B_1 \wedge Q\}$ OR ... OR $\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{\overline{t} = \overline{s}_n \wedge B_n \wedge Q\}$

D.2 $\forall\, \overline{\mathbf{X}}. \leftarrow \mathbf{p}(\overline{\mathbf{u}}) \wedge \mathbf{Q}$:
$\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{\forall\, \overline{X}, \overline{Y}. \leftarrow \overline{u} = \overline{v} \wedge B \wedge Q \mid p(\overline{v}) \leftarrow B \in \mathcal{P}$ and $\overline{Y} = vars(B) \cup vars(\overline{v})\}$

3. *Negations:*
Resolving negation corresponds to 'switching the mode of reasoning' from a positive literal to a denial and vice versa. This is similar to the idea of negation-as-failure in logic programming.

N.1 $\neg\mathbf{p}(\overline{\mathbf{t}}) \wedge \mathbf{Q}$:
$\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{Q, \leftarrow p(\overline{t})\}$[2]

N.2 $\forall\, \overline{\mathbf{X}}. \leftarrow \neg\mathbf{p}(\overline{\mathbf{t}}) \wedge \mathbf{Q}$ where vars$(\overline{t}) \not\subseteq \overline{X}$:
$\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{p(\overline{t})\}$ OR $\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{\neg p(\overline{t}), \forall\, \overline{X}. \leftarrow Q\}$

4. *Abducibles:*
The first rule is responsible for the creation of new hypotheses. Both rules ensure that the elements in Δ are consistent with those in Δ^*.

A.1 $\mathbf{a}(\overline{\mathbf{t}}) \wedge \mathbf{Q}$:
SELECT an arbitrary $a(\overline{s}) \in \Delta_i$ such that
$\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{Q\} \cup \{\overline{s} = \overline{t}\}$ OR
$\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{Q\} \cup \{\forall\, \overline{X}. \leftarrow \overline{u} = \overline{t} \wedge R \mid \forall\, \overline{X}. \leftarrow a(\overline{u}) \wedge R \in \Delta_i^*\} \cup$
$\{\overline{s} \neq \overline{t} | a(\overline{s}) \in \Delta_i\}$ and
$\Delta_{i+1} = \Delta_i \cup \{a(\overline{t})\}$

A.2 $\forall\, \overline{\mathbf{X}}. \leftarrow \mathbf{a}(\overline{\mathbf{u}}) \wedge \mathbf{Q}$:
$\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{\forall\, \overline{X}. \leftarrow \overline{s} = \overline{u} \wedge Q \mid a(\overline{s}) \in \Delta_i\}$ and
$\Delta_{i+1}^* = \Delta_i^* \cup \{\forall\, \overline{X}. \leftarrow a(\overline{u}) \wedge Q\}$

[2]The variables of $\overline{t}$ become free variables in the resulted denial.

5. *Equalities*[3]*:*
   These inference rules isolate the (in)equalities so that the equality solver can evaluate them.

   E.1 $\mathbf{s} = \mathbf{t} \wedge \mathbf{Q}$:
   $\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{Q\}$ and $\mathcal{E}_{i+1} = \mathcal{E}_i \cup \{s = t\}$

   E.2 $\forall\ \overline{\mathbf{X}}.\leftarrow \mathbf{v} = \mathbf{u} \wedge \mathbf{Q}$:
   $\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{\forall\ \overline{X}. \leftarrow E_s \wedge Q\}$ where E_s is the equational solved form of $v = u$.
   This rule reduces the denial to the basic case:

   E.2.b $\forall\ \overline{\mathbf{X}}, \overline{\mathbf{Y}}. \leftarrow \mathbf{Z} = \mathbf{u} \wedge \mathbf{Q}$ where $Z \notin \overline{X} \cup \overline{Y}$ and $vars(u) \subseteq \overline{Y}$:
   $\mathcal{E}_{i+1} = \mathcal{E}_i \cup \{\forall\ \overline{Y}.Z \neq u\}$ OR
   $\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{\forall\ \overline{X}, \overline{Y}. \leftarrow Q(Z/u)\}$

   A special case of this rule is

   E.2.c $\forall\ \overline{\mathbf{X}}, \mathbf{Y}. \leftarrow \mathbf{Z} = \mathbf{Y} \wedge \mathbf{Q}$ where $Z \notin \overline{X}$:
   $\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{\forall\ \overline{X}. \leftarrow Q(Y/Z)\}$

6. *finite domain expressions:*
   These inference rules isolate the finite domain expressions so that the finite domain constraint solver can evaluate them. The allowed finite domain constraint expressions $c(\overline{t})$ are those that have been presented before. We suppose that the finite domain solver handles both the positive expression $c(\overline{t})$ aswell the negated expression $\neg c(\overline{t})$.

   F.1 $\mathbf{c}(\overline{\mathbf{t}}) \wedge \mathbf{Q}$:
   $\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{Q\}$ and $\mathcal{FD}_{i+1} = \mathcal{FD}_i \cup \{c(\overline{t})\}$

   F.2 $\forall\ \overline{\mathbf{X}}. \leftarrow \mathbf{c}(\overline{\mathbf{t}}) \wedge \mathbf{Q}$ where vars$(\overline{t}) \not\subseteq \overline{X}$:
   $\mathcal{FD}_{i+1} = \mathcal{FD}_i \cup \{\neg c(\overline{t})\}$ OR
   $\mathcal{FD}_{i+1} = \mathcal{FD}_i \cup \{c(\overline{t})\}$ and $\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{\forall\ \overline{X}. \leftarrow Q)\}$

Floundering

Floundering refers to the effect that resolving a formula has to spawn an infinite number of new formulas. This explosion happens when a predicate is resolved that contains a universally quantified variable. For instance, consider $\forall\ X.p(X)$ and a FOL language $\mathcal{L}$ with an infinite set of constants. To evaluate the formula an infinite number of instances must be checked: $\{p(c) \mid c \in \mathcal{HU}(\mathcal{L})\}$. And that is obviously infeasible. Therefore a formula which leads to floundering, should not be selected. If there is no choice except selecting such a formula, the derivation process is aborted.

[3]Only the rules for the equalities are shown here, other rules of the $\mathcal{E}$-solver are shown in section 3.5.

There are two ways to handle the issue of floundering. One is to ensure that floundering goals are never selected during a derivation. In the context of SLDNF [14, 187], such a search strategy (computation rule) is called *safe*. The other approach is to impose syntactic restrictions that ensure the formula is not floundering (called the admissible and allowedness criteria [35, 187]). These criteria restrict the quantification of variables in the goal formulas. In the inference rules above, these are given by the quantification conditions associated with each selected formula. If a formula is selected that does not satisfy these conditions, e.g. $\forall X.p(X)$ and $\forall X \leftarrow not(p(X))$, it is not allowed. Consequently the procedure flounders.

Because floundering usually originates from a bug in the specification and it is a unintended situation, it happens rarely in practice. We have, therefore, decided not to design an independent verification tool to check for potential sources of floundering. (Note that finding out that a SLDNF-derivation does not flounder for a given query is in general undecidable [14].) To ensure correct behavior, runtime checks for floundering are performed. (When one is sure that floundering does not happen, these checks can be disabled; saving in this way some execution time.) When it encounters a floundering goal, the $\mathcal{A}$system aborts and reports an error to the user.

3.3.3 An example trace of the $\mathcal{A}$system proof procedure

The specification

This section presents an example of the $\mathcal{A}$system proof procedure. As example domain we have selected the blocks world planning problem.

This problem domain describes an artificial world in which there is a table, a number of blocks and a single robot. The blocks are located either on the table or on top of another block. They can be moved from one spot to another by the robot, provided the selected block is clear (i.e. there is no other block on it) and the destination is also clear. It is supposed that the table is sufficiently large, so that each block can be placed on the table.

We represent the location Y of a block X at time T as $on(X, Y, T)$. The next program rules describe the temporal relations, using the Event Calculus [170].

$$on(X, Y, T) \leftarrow initially(X, Y) \wedge \neg moved(X, Y, 0, T).$$
$$on(X, Y, T) \leftarrow move(X, Y, E) \wedge E < T \wedge \neg moved(X, Y, E, T).$$

$$moved(X, Y, E, T) \leftarrow move(X, Z, C) \wedge Z \neq Y \wedge between(C, E, T).$$

$$between(C, E, T) \leftarrow E \leq C \wedge C < T.$$

A block X is on Y at time T, either if it was initially on this location and it has not been moved to another location ($\neg moved(X, Y, 0, T)$) between the initial time point (0) and time T; or the block X has been moved by the robot ($move(X, Y, E)$)

to location Y before T and this move has not been undone ($\neg moved(X, Y, E, T)$) in the meantime. A block X is moved away from Y between time points E and T if there is an action $move(X, Z, C)$ that moves X to another location Z, different from Y ($Z \neq Y$), that happens between E and T ($between(C, E, T)$).

The domain specifies that moving a block requires that the block and the destination are clear. This is expressed by the integrity constraint

$$\forall\ X, Y, E.\ \leftarrow move(X, Y, E) \wedge \neg succeeds\_move(X, Y, E)$$

and the auxiliary predicate $succeeds\_move/3$:

$$\begin{array}{l} succeeds\_move(X, Y, E) \leftarrow ablock(X) \wedge location(Y) \wedge time(E) \wedge \\ X \neq Y \wedge clear\_block(X, E) \wedge clear\_location(Y, E). \end{array}$$

The preconditions are expressed by the $clear\_block/2$ and $clear\_location/2$ predicates. Additionally, $succeeds\_move/3$ ensures also that the first argument is a block, the second a location and the third a time point. Since we do not want to consider trivial moves, we exclude the possibility that the robot picks up a block and places it back on the same location $X \neq Y$.

A block is clear when there is no other block on it.

$$\begin{array}{l} clear\_block(X, E) \leftarrow \neg something\_on(X, E). \\ something\_on(X, E) \leftarrow on(Y, X, E). \end{array}$$

Note that in order to get the logic right, an auxiliary predicate $something\_on/2$ is necessary. Using the previous definition, we can define a clear location as either the table (denoted by 0), or as a clear block.

$$\begin{array}{l} clear\_location(0, E). \\ clear\_location(Y, E) \leftarrow Y \neq 0 \wedge clear\_block(Y, E). \end{array}$$

Finally, we require that each different action happens on a different time point. In other words, we require a sequential plan.

$$\begin{array}{l} \forall\ X, Y1, Y2.\ \leftarrow move(X, Y1, T) \wedge move(X, Y2, T) \wedge Y1 \neq Y2. \\ \forall\ X1, Y, X2.\ \leftarrow move(X1, Y, T) \wedge move(X2, Y, T) \wedge X1 \neq X2. \end{array}$$

For a given initial situation and a goal description, a planning problem is finding a sequence of actions such that when executing these actions in the specified order the initial situation is turned into a situation in which the goal holds. For the above specification, the planning problem is the search for an interpretation of the $move/3$ predicate such that in the final state the goal formula holds. Therefore, the action $move/3$ is an open predicate.

An $\mathcal{A}$system execution

Suppose we have three blocks 1, 2 and 3, which are stacked upon each other. Initially, 1 is on top of 2, 2 is on 3 and 3 is on the table (0). Consider a goal tower, where 2 is on the table and 1 is on 2 and 3 is on 1. Intuitively, a successful plan is obtained by first putting block 1 on the table, then 2 also on the table, and then stacking 1 on 2 and 3 on 1.

We show how the $\mathcal{A}$system constructs this solution using depth first search. The initial situation is given by the following definitions:

$$\begin{array}{l} initially(1,2). \\ initially(2,3). \\ initially(3,0). \\ \\ ablock(X) \leftarrow X \text{ in } 1..3. \\ location(X) \leftarrow X \text{ in } 0..3. \\ time(X) \leftarrow X \text{ in } 1..6. \end{array}$$

Note that it is assumed that the initial situation is consistent: e.g. that each block is positioned on one and only one location. If this must be guaranteed, extra integrity constraints must be added such as

$$\begin{array}{l} \forall\ X,Y,Z.initially(X,Y) \wedge initially(X,Z). \\ \forall\ X,Y,Z.initially(Y,X) \wedge initially(Z,X). \end{array}$$

The query describes the final state that must be reached; it is for our example scenario

$$\exists\ T.time(T) \wedge on(2,0,T) \wedge on(1,2,T) \wedge on(3,1,T).$$

The $\mathcal{A}$system derivation starts from the initial state

$$\mathcal{S}_0 = (\{time(T) \wedge on(2,0,T) \wedge on(1,2,T) \wedge on(3,1,T)\} \cup \mathcal{IC}, \mathcal{ST}^{\emptyset})$$

The $\mathcal{A}$system selects as first formula the query

$$F_0 \equiv time(T) \wedge on(2,0,T) \wedge on(1,2,T) \wedge on(3,1,T)$$

The applicable inference rule is D.1. Since the predicate $time/1$ has a single clause, the unfolding of $time/1$ with this clause yields the state

$$\mathcal{S}_1 = (\{T = T' \wedge T' \text{ in } 1..6 \wedge on(2,0,T) \wedge on(1,2,T) \wedge on(3,1,T)\} \cup \mathcal{IC}, \mathcal{ST}^{\emptyset})$$

Since we are following a depth first strategy, the updated formula is selected again:

$$F_1 \equiv T = T' \wedge T' \text{ in } 1..6 \wedge on(2,0,T) \wedge on(1,2,T) \wedge on(3,1,T)$$

Inference rule E.1 is applicable and posts the equality $T = T'$ to the $\mathcal{E}$-store. The $\mathcal{E}$-solver solves this equality by unifying both.

$$\mathcal{S}_2 = (\{T \text{ in } 1..6 \wedge on(2,0,T) \wedge on(1,2,T) \wedge on(3,1,T)\} \cup \mathcal{IC}, (\emptyset, \emptyset, \{T = T'\}, \emptyset))$$

To simplify the presentation, we will skip from now on all applications of E.1. Also the selected formula is left implicit, since this is clear from the context. Continuing the evaluation, the next applicable rule is F.1. This updates the $\mathcal{FD}$-store.

$$\mathcal{S}_3 = (\{on(2,0,T) \wedge on(1,2,T) \wedge on(3,1,T)\} \cup \mathcal{IC}, (\emptyset, \emptyset, \{T = T'\}, \{T \text{ in } 1..6\}))$$

Next, D.1 is used to unfold $on(2,0,T)$ with its first clause to

$$\begin{array}{ll} \mathcal{S}_4 & = (\mathcal{G}_4, \mathcal{ST}_4) \\ \mathcal{G}_4 & = \{initially(2,0) \wedge \neg init\_moved(2,0,T) \wedge on(1,2,T) \wedge on(3,1,T)\} \cup \mathcal{IC} \\ \mathcal{ST}_4 & = (\emptyset, \emptyset, \{T = T'\}, \{T \text{ in } 1..6\}) \end{array}$$

By unfolding (D.1), $initially(2,0)$ is found unsatisfiable since none of the given clauses will match. Therefore, the $\mathcal{A}$system must backtrack and select the second clause to unfold $on(2,0,T)$. This yields

$$\begin{array}{ll} \mathcal{S}_{4'} & = (\mathcal{G}_4, \mathcal{ST}_4) \\ \mathcal{G}_4 & = \{move(2,0,E) \wedge E < T \wedge \neg moved(2,0,E,T) \wedge on(1,2,T) \wedge on(3,1,T)\} \\ & \quad \cup\, \mathcal{IC} \\ \mathcal{ST}_4 & = (\emptyset, \emptyset, \{T = T'\}, \{T \text{ in } 1..6\}) \end{array}$$

Now inference rule A.1 is applicable. Since $\Delta = \emptyset$, the first branch is not applicable (i.e. the reuse of already abduced atoms). Because also $\Delta^* = \emptyset$, no extra denials have to be added to the goals.

$$\begin{array}{ll} \mathcal{S}_5 & = (\mathcal{G}_5, \mathcal{ST}_5) \\ \mathcal{G}_5 & = \{E < T \wedge \neg moved(2,0,E,T) \wedge on(1,2,T) \wedge on(3,1,T)\} \cup \mathcal{IC} \\ \mathcal{ST}_5 & = (\{move(2,0,E)\}, \emptyset, \{T = T'\}, \{T \text{ in } 1..6\}) \end{array}$$

Again F.1. can be applied, which results in

$$\begin{array}{ll} \mathcal{S}_6 & = (\mathcal{G}_6, \mathcal{ST}_6) \\ \mathcal{G}_6 & = \{\neg moved(2,0,E,T) \wedge on(1,2,T) \wedge on(3,1,T)\} \cup \mathcal{IC} \\ \mathcal{ST}_6 & = (\{move(2,0,E)\}, \emptyset, \{T = T'\}, \{T \text{ in } 1..6, E < T\}) \end{array}$$

The next inference rule that must be applied is N.1. This adds the denial $\leftarrow moved(2,0,E,T)$ (E and T are free variables!) to the goal stack.

$$\begin{array}{ll} \mathcal{S}_7 & = (\mathcal{G}_7, \mathcal{ST}_7) \\ \mathcal{G}_7 & = \{\leftarrow moved(2,0,E,T)\} \cup \{on(1,2,T) \wedge on(3,1,T)\} \cup \mathcal{IC} \\ \mathcal{ST}_7 & = (\{move(2,0,E)\}, \emptyset, \{T = T'\}, \{T \text{ in } 1..6, E < T\}) \end{array}$$

For this denial, inference rule D.2 is applicable. Since $moved/4$ has only one clause, the updated goal stack is

$$\begin{array}{ll} \mathcal{S}_8 & = (\mathcal{G}_8, \mathcal{ST}_8) \\ \mathcal{G}_8 & = \{\forall\ Z, C. \leftarrow move(2, Z, C) \wedge Z \neq 0 \wedge between(C, E, T)\} \cup \\ & \quad \{on(1, 2, T) \wedge on(3, 1, T)\} \cup \mathcal{IC} \\ \mathcal{ST}_8 & = (\{move(2, 0, E)\}, \emptyset, \{T = T'\}, \{T \text{ in } 1..6, E < T\}) \end{array}$$

Inference rule A.2 is now applicable. Since $\Delta = \{move(2, 0, E)\}$ there is an abducible that must be verified. The denial, itself, is stored in Δ^*.

$$\begin{array}{ll} \mathcal{S}_9 & = (\mathcal{G}_9, \mathcal{ST}_9) \\ \mathcal{G}_9 & = \{\forall\ Z, C. \leftarrow move(2, Z, C) = move(2, 0, E) \wedge Z \neq 0 \wedge between(C, E, T)\} \cup \\ & \quad \{on(1, 2, T) \wedge on(3, 1, T)\} \cup \mathcal{IC} \\ \mathcal{ST}_9 & = (\{move(2, 0, E)\}, \{\forall\ Z, C. \leftarrow move(2, Z, C) \wedge Z \neq 0 \wedge between(C, E, T)\}, \\ & \quad \{T = T'\}, \{T \text{ in } 1..6, E < T\}) \end{array}$$

Using the $\mathcal{E}$-solver (inference rule E.2), the equality $move(2, Z, C) = move(2, 0, E)$ is simplified to $Z = 0 \wedge C = E$. Then by using inference rule E.2.c, the reasoning continues.

$$\begin{array}{ll} \mathcal{S}_{10} & = (\mathcal{G}_{10}, \mathcal{ST}_{10}) \\ \mathcal{G}_{10} & = \{\leftarrow 0 \neq 0 \wedge between(E, E, T)\} \cup \{on(1, 2, T) \wedge on(3, 1, T)\} \cup \mathcal{IC} \\ \mathcal{ST}_{10} & = (\{move(2, 0, E)\}, \{\forall\ Z, C. \leftarrow move(2, Z, C) \wedge Z \neq 0 \wedge between(C, E, T)\}, \\ & \quad \{T = T'\}, \{T \text{ in } 1..6, E < T\}) \end{array}$$

F.2 is applicable, since it is a disequality between finite domain variables. Because $0 \neq 0$ fails, the denial is satisfied and can be removed from the goal stack.

$$\begin{array}{ll} \mathcal{S}_{11} & = (\mathcal{G}_{11}, \mathcal{ST}_{11}) \\ \mathcal{G}_{11} & = \{on(1, 2, T) \wedge on(3, 1, T)\} \cup \mathcal{IC} \\ \mathcal{ST}_{11} & = (\{move(2, 0, E)\}, \{\forall\ Z, C. \leftarrow move(2, Z, C) \wedge Z \neq 0 \wedge between(C, E, T)\}, \\ & \quad \{T = T'\}, \{T \text{ in } 1..6, E < T\}) \end{array}$$

We have shown now the application of most inference rules. After a lot more inference steps, a state $\mathcal{S}_f$ with an empty goal stack is created $\mathcal{G}_f = \emptyset$. One such state is given by the next store

$$\Delta_f = \{move(2,0,E_1), move(1,0,E_2), move(3,1,E_3), move(1,2,E_4)\},$$

$$\Delta_f^* = \left\{ \begin{array}{l} \forall\ Z, C. \leftarrow move(2,Z,C) \wedge Z \neq 0 \wedge between(C, E_1, T) \\ \forall\ Z, C. \leftarrow move(1,Z,C) \wedge Z \neq 0 \wedge between(C, E_2, T) \\ \forall\ Z, C. \leftarrow move(3,Z,C) \wedge Z \neq 1 \wedge between(C, E_3, T) \\ \forall\ Z, C. \leftarrow move(1,Z,C) \wedge Z \neq 2 \wedge between(C, E_4, T) \\ \forall\ X, Y, C. \leftarrow move(X,Y,C) \wedge \neg succeeds\_move(X,Y,C) \\ \forall\ X, Y1, T1, Y2, T2.\ \leftarrow move(X,Y1,T1) \wedge move(X,Y2,T2) \wedge \\ \qquad\qquad Y1 \neq Y2 \wedge T1 = T2. \\ \forall\ X1, Y, T1, X2, T2.\ \leftarrow move(X1,Y,T1) \wedge move(X2,Y,T2) \wedge \\ \qquad\qquad X1 \neq X2 \wedge T1 = T2. \\ \forall\ X, Y, C. \leftarrow move(X,3,C) \wedge C < E_3 \wedge \neg moved(X,3,C,E_3). \\ \forall\ X, Y, C. \leftarrow move(X,2,C) \wedge C < E_2 \wedge \neg moved(X,2,C,E_1). \\ \forall\ X, Y, C. \leftarrow move(X,1,C) \wedge C < E_1 \wedge \neg moved(X,1,C,E_2). \\ \forall\ X, Y, C. \leftarrow move(X,1,C) \wedge C < E_3 \wedge \neg moved(X,1,C,E_3). \\ \forall\ X, Y, C. \leftarrow move(X,1,C) \wedge C < E_4 \wedge \neg moved(X,1,C,E_4). \\ \forall\ X, Y, C. \leftarrow move(X,2,C) \wedge C < E_4 \wedge \neg moved(X,2,C,E_4). \end{array} \right\}$$

The computed set Δ_f represents a plan of four actions that leads to the goal state. Note that the time points are not ground. Their ordering is determined by the finite domain constraint store. (See below.)

The Δ_f^* presents all integrity constraints that are constructed during the derivation. The first four result from the requirement that a move should not undo the required effects from another move ($\neg moved(X,Y,E,T)$). The next three are exactly the three denials in the original problem specification. The last six result from the requirement that the location and blocks must be clear.

We have left out the $\mathcal{E}$-store since it does not contain information of interest for this explanation. The $\mathcal{FD}$-store contains important information: the ordering between the different time points. The presented store is a simplification of the real store. In reality, it contains far more constraints but they are either trivially satisfiable or reducible to one of those in the presented store.

$$\mathcal{FD}_f = \left\{ \begin{array}{l} T \text{ in } 1..6, \\ E_1 < T, E_1 \text{ in } 1..6, \\ E_3 < T, E_3 \text{ in } 1..6, \\ E_4 < T, E_4 \text{ in } 1..6, \\ E_2 < E_1, E_3 < E_2, E_2 \text{ in } 1..6 \\ E_2 < E_4, E_1 < E_4 \end{array} \right\}$$

The state $\mathcal{S}_f$ forms a solution if the constraint stores $\mathcal{E}_f$ and $\mathcal{FD}_f$ are satisfiable. This holds since an assignment for the free variables T, E_1, E_2, E_3 and E_4 can be constructed. One solution is

$$\Delta_f = \{move(2,0,2), move(1,0,1), move(3,1,4), move(1,2,3)\}$$

for $T = 5$. This is exactly the intuitive solution we had in mind.

3.3.4 Soundness and completeness of the $\mathcal{A}$system proof procedure

We now present the soundness and completeness theorems for the $\mathcal{A}$system procedure, which are extended versions of those for SLDNFA [90]. These results are with respect to the three-valued completion semantics $comp_3$. In this way, for all $\mathcal{A}$system theories a general soundness and completeness result can be proven. However, it also means that, in general, the $\mathcal{A}$system is not a solver for ID-Logic, since ID-Logic is based upon the two-valued well-founded semantics.

The weakening to the completion semantics is due to the fact that the definitions are interpreted as iff-formulas and the $\mathcal{A}$system inference rules do not detect loops. The need for the change from two-valued to three-valued semantics is shown by the following example (from Denecker [90]):

EXAMPLE 3.1 Consider the propositional abductive logic theory $(\{p \leftarrow a \wedge \neg p, q \leftarrow a\}, \{a\}, \emptyset)$. An $\mathcal{A}$system-derivation for the query q results in the solution state $(\emptyset, (\{a\}, \emptyset, \emptyset, \emptyset))$. However under the two-valued completion semantics, $\mathcal{P} \cup \Delta = \{p \leftarrow a \wedge \neg p, q \leftarrow a, a\}$ has no model. The fact that a is **true** reduces the first rule to $p \leftarrow \neg p$ which is a loop through negation. But under the three-valued completion semantics, the extended program $(\mathcal{P} \cup \Delta)$ has a model: $\{p^u, q^t, a^t\}$.

Such a program is called locally inconsistent for the predicate p. For a detailed discussion on this issue, we refer to [90]. It has been proven that some classes of theories always have a two-valued model. For those, the results can be strengthened to the two-valued semantics. Some of these are [90]:

- definite logic programs
- hierarchical logic programs
- stratified logic programs
- locally stratified logic programs with Strong Domain Closure Axiom and $\mathcal{FEQ}$
- acyclic logic programs with Strong Domain Closure Axiom and $\mathcal{FEQ}$

Soundness

Consider a solution state $(\emptyset, (\Delta, \Delta^*, \mathcal{E}, \mathcal{FD}))$, then I_Δ is the interpretation of the abducibles defined as $I_\Delta = \{a(\overline{t})^{\mathbf{t}} | a(\overline{t}) \in \Delta\} \cup \{a(\overline{t})^{\mathbf{f}} | a(\overline{t}) \notin \Delta\}$. This interpretation is the unique model of the logic program D_Δ constructed as the enumeration of facts for all atoms of Δ: $D_\Delta = \{a(\overline{t}) \leftarrow \mathbf{true}. | a(\overline{t}) \in \Delta\}$. Then, $P + \Delta$ denotes the normal logic program obtained by augmenting P with the enumerative definition D_Δ.

THEOREM 17 (SOUNDNESS OF THE $\mathcal{A}$SYSTEM)
Let P be the abductive normal program derived from an $\mathcal{A}$system theory ($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$). If $(\theta, \mathcal{M}(\mathcal{S}_{sol}))$ is an $\mathcal{A}$system-answer for a query $\mathcal{Q}$ then

$$comp_3(P + \Delta) \models \mathcal{Q}\theta$$

where Δ is the set of computed abducibles in $\mathcal{S}_{sol}$. Furthermore, it holds that $comp_3(P + \Delta)$ is consistent.

The proof is based on the soundness of the inference rules.

LEMMA 2 (THE SOUNDNESS OF THE INFERENCE RULES)
Let $\mathcal{T}$ be an $\mathcal{A}$system-theory. For each inference rule that transforms a state $\mathcal{S}_i$ into a state $\mathcal{S}_{i+1}$, it holds that

$$comp_3(\mathcal{T}) \models (\mathcal{M}(\mathcal{S}_i) \Leftarrow \mathcal{M}(\mathcal{S}_{i+1}))$$

Notation: $\Leftarrow$ denotes here classical implication and $\Leftrightarrow$ the equivalence. We use a different notation than which is used in our abductive logic programs and ID-Logic theories, since the statement is at a different level.

PROOF 18
The inference rules D,A,N and E (the equations) are part of the original presentation of the SLDNFA proof procedure. The lemma has been proven in [90]. It remains to show the lemma for the rules F, since these are newly added w.r.t. original SLDNFA.

F.1 The rule holds trivially, because it is just a move from one set to another.

F.2 The proposition that must be proven reduces to these two statements

$$comp_3(\mathcal{T}) \models (\forall\, \overline{X}. \leftarrow c(\overline{t}) \wedge Q) \Leftarrow \neg c(\overline{t})$$
$$comp_3(\mathcal{T}) \models (\forall\, \overline{X}. \leftarrow c(\overline{t}) \wedge Q) \Leftarrow c(\overline{t}) \wedge \forall\, \overline{X}. \leftarrow Q$$

provided that vars$(\overline{t}) \not\subseteq \overline{X}$. The quantification condition on the rules allows to move the atom $c(\overline{t})$ out of the scope of the universal quantifier in the selected formula. Then the proof is as follows

$$
\begin{array}{llr}
\forall\, \overline{X}. \leftarrow c(\overline{t}) \wedge Q & \Leftrightarrow \forall\, \overline{X}. \neg(c(\overline{t}) \wedge Q) & \text{(equivalent formula)} \\
& \Leftrightarrow \forall\, \overline{X}. \neg c(\overline{t}) \vee \neg Q & \text{(De Morgan)} \\
& \Leftrightarrow \neg c(\overline{t}) \vee \forall\, \overline{X}. \neg Q & \text{(c}(\overline{t})\text{ outside scope)} \\
& \Leftrightarrow \neg c(\overline{t}) \vee (c(\overline{t}) \wedge \forall\, \overline{X}. \neg Q) & \text{(binary choice on c}(\overline{t})\text{)} \\
& \Leftrightarrow \neg c(\overline{t}) \vee (c(\overline{t}) \wedge \forall\, \overline{X}. \leftarrow Q) & \text{(to denial representation)}
\end{array}
$$

Finally, the disjunction is made explicit, proving our propositions:

$$\models (\forall\, \overline{X}. \leftarrow c(\overline{t}) \wedge Q) \Leftarrow \neg c(\overline{t})$$

$$\models (\forall\, \overline{X}. \leftarrow c(\overline{t}) \wedge Q) \Leftarrow c(\overline{t}) \wedge \forall\, \overline{X}. \leftarrow Q$$

PROOF 19 ((OF THEOREM 17))
Suppose we have a successful $\mathcal{A}$system derivation starting from the initial state $\mathcal{S}_0 = (\{Q\} \cup \mathcal{IC}, \mathcal{ST}^{\emptyset})$ and terminating in the solution state $\mathcal{S}_f$. Assume that the solution state $\mathcal{S}_f$ is not an answer for the query $\mathcal{Q}$. Hence $comp_3(P + \Delta_{sol}) \not\models \mathcal{Q}$, or equivalently $comp_3(P + \Delta_{sol}) \models (\neg\mathcal{Q} \Leftarrow \mathcal{M}(\mathcal{S}_f))$.

Since each inference rule is sound (lemma 2), it holds that for any Δ_{sol} which is part of an answer, $comp_3(P + \Delta_{sol}) \models (\mathcal{M}(\mathcal{S}_0) \Leftarrow \mathcal{M}(\mathcal{S}_f))$. Making the meaning of the initial state explicit, this yields $comp_3(P + \Delta_{sol}) \models ((Q \wedge IC) \Leftarrow \mathcal{M}(\mathcal{S}_f))$. More specifically, $comp_3(P + \Delta_{sol}) \models (Q \Leftarrow \mathcal{M}(\mathcal{S}_f))$, which is in contradiction with our assumption that $\mathcal{M}(\mathcal{S}_f)$ does not imply the query Q.

The completeness of the $\mathcal{A}$system

THEOREM 20 (COMPLETENESS OF THE $\mathcal{A}$SYSTEM)
Given an $\mathcal{A}$system-theory $\mathcal{T}$ =($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$) and a query $\mathcal{Q}$. Let P be the corresponding abductive normal logic program of $\mathcal{T}$. Suppose that $\mathcal{Q}$ has a finite $\mathcal{A}$system-derivation tree W.

(a) if all branches of W are failed, then $comp_3(P) \models \forall\, (\neg\mathcal{Q})$

(b) if $comp_3(P + \exists\, (\mathcal{Q}))$ is satisfiable, then W contains a successful branch.

Analogous to the soundness theorem, the proof of the completeness is based on lemma over the individual inference rules.

LEMMA 3
Let $\mathcal{T}$ be an $\mathcal{A}$system-theory. For any $\mathcal{A}$system-state $\mathcal{S}_i$, let $\mathcal{S}_{i+1}^1, \ldots, \mathcal{S}_{i+1}^m$ be the states that can be computed from $\mathcal{S}_i$ for a selected formula F. Then the equivalence holds

$$comp_3(\mathcal{T}) \models \forall\, (\mathcal{M}(\mathcal{S}_i) \Leftrightarrow \mathcal{M}(\mathcal{S}_{i+1}^1) \vee \ldots \vee \mathcal{M}(\mathcal{S}_{i+1}^m))$$

PROOF 21
Again, it suffices to prove this lemma for the new rules F.1. and F.2. See [90] for the proof of the others.

F.1 The to-be-proven equivalence is

$$comp_3(\mathcal{T}) \models \forall\ (c(\bar{t}) \wedge Q \Leftrightarrow Q \wedge c(\bar{t}))$$

which holds trivially.

F.2 The to-be-proven equivalence is

$$comp_3(\mathcal{T}) \models \forall\ (\forall\ \overline{X}. \leftarrow c(\bar{t}) \wedge Q \Leftrightarrow (\neg c(\bar{t})) \vee (c(\bar{t}) \wedge \forall\ \overline{X}. \leftarrow Q))$$

provided that vars$(\bar{t}) \not\subseteq \overline{X}$. The condition on the rule allows to move the atom $c(\bar{t})$ out of the scope of the universal quantifier in the selected formula. Then the proof is as follows

$$\begin{array}{lll} \forall\ \overline{X}. \leftarrow c(\bar{t}) \wedge Q & \Leftrightarrow \forall\ \overline{X}. \neg c(\bar{t}) \vee \neg Q & \text{(De Morgan)} \\ & \Leftrightarrow \neg c(\bar{t}) \vee \forall\ \overline{X}. \neg Q & \text{(c}(\bar{t})\text{ outside scope)} \\ & \Leftrightarrow \neg c(\bar{t}) \vee (c(\bar{t}) \wedge \forall\ \overline{X}. \neg Q) & \text{(binary choice on c}(\bar{t})\text{)} \\ & \Leftrightarrow \neg c(\bar{t}) \vee (c(\bar{t}) \wedge \forall\ \overline{X}. \leftarrow Q) & \end{array}$$

PROOF 22 ((OF THEOREM 20))
From [90], it is known that this theorem holds for all rules, except for the F rules. According to the above lemma, all rules, including the F rules, preserve completeness, i.e. they do not lose solutions since all derived states together are equivalent to the original state.

Now, it is given that W is the finite derivation tree for the query $\mathcal{Q}$. (a) If W is finitely failed and since all inference rules are completeness preserving, there exists no successful derivation for $\mathcal{Q}$. Consequently, it follows that $comp_3(P) \not\models \forall\ (\mathcal{Q})$. (b) If $comp_3(P + \exists\ (Q))$ is satisfiable, there must exist a successful derivation. Otherwise, by (a), $comp_3(P + \exists\ (\mathcal{Q}))$ must be unsatisfiable, which is in contradiction with the initial assumption.

Termination condition

The finiteness of the derivation-tree for a query $\mathcal{Q}$ is a strong requirement for the completeness theorem. Obviously it is impossible to verify in general if this condition holds before answering the query. In [268], Verbaeten studies termination of abductive inference in SLDNFA. She proves termination for *abductive nonrecursive* programs w.r.t. any bounded query. And thus, any bounded query w.r.t. those abductive nonrecursive programs will have a finite derivation tree, so that the completeness theorem is satisfied.

Abductive recursiveness is a notion that formalizes loops through abduction. Exactly these loops are the extra source of non-termination that is introduced by SLDNFA compared to SLDNF. These loops are of course not the only source of non-termination, but the other sources are not specific to abduction, and thus outside our scope.

EXAMPLE 3.2 Consider the $\mathcal{A}$system-theory $\mathcal{T} = (\emptyset, \{a/1, b/1\}, \{\forall\ X \leftarrow a(X) \wedge \neg b(X+1), \forall\ X. \leftarrow b(X) \wedge \neg a(X)\})$ and the query a(1). For this query the $\mathcal{A}$system won't terminate because the abduction of a(1) will introduce b(2) and this adds a(2), starting the loop again. The $\mathcal{A}$system constructs in this way the infinite model $\{a(n)|n \in \mathbb{N} \text{ and } n > 0\} \cup \{b(n)|n \in \mathbb{N} \text{ and } n > 1\}$.

The formalization of these loops uses a dependency graph.

DEFINITION 3.11 For an abductive normal logic program P, a defined predicate p depends directly on another (abductive/defined) predicate q

- $p \rightarrow^+ q$ if there exists a positive literal $q(\cdot)$ in a body of a defining clause of p.
- $p \rightarrow^- q$ if there exists a negative literal $q(\cdot)$ in a body of a defining clause of p.

Then,

- $p \leadsto^+ q$ if there exists a chain of dependencies between p and q consisting of only positive arcs ($\rightarrow^+$).
- $p \leadsto^- q$ if there exists a chain of dependencies between p and q containing an odd number of negative arcs ($\rightarrow^-$).

Intuitively, one can interpret $p \leadsto^+ q$ as that p produces q when evaluated in a conjunction, or consumes q when p is evaluated in a denial. Analogous, $p \leadsto^- q$ reads as p consumes q in a conjunction and produces q in a denial.

EXAMPLE 3.3 Example 3.2 continued The abductive normal logic program P of $\mathcal{T}$ is

$$ic \leftarrow a(X) \wedge \neg b(X+1).$$
$$ic \leftarrow b(X) \wedge \neg a(X).$$

The table of direct dependencies is

rule 1	rule 2
$ic \rightarrow^+ a$	$ic \rightarrow^+ b$
$ic \rightarrow^- b$	$ic \rightarrow^- a$

The derived chains are

rule 1	rule 2
$ic \leadsto^+ a$	$ic \leadsto^+ b$
$ic \leadsto^- b$	$ic \leadsto^- a$

According to the intuitive reading (in denial evaluation), ic will produce b and consume a via rule 1, and consume b and produce a via rule 2. This indicates a cycle, in which a is produced by one rule and consumed by another. These cycles are often non-terminating.

Formally,

DEFINITION 3.12 (ABDUCTIVE RECURSIVE) An abductive normal logic program is *abductive recursive* iff for defined predicates (including those of the predefined domains) $p_1, \ldots, p_n$, $n \geq 1$, and for abducible predicates $a_1, \ldots, a_n$, $n \geq 1$, the dependency graph contains the following cyclic situation:

$$\begin{array}{cccc} p_1 \leadsto^+ a_1 & p_2 \leadsto^+ a_2 & & p_n \leadsto^+ a_n \\ & & \cdots & \\ p_1 \leadsto^- a_2 & p_2 \leadsto^- a_3 & & p_n \leadsto^- a_1 \end{array}$$

An abductive normal logic program is called *abductive nonrecursive* iff it is not abductive recursive.

EXAMPLE 3.4 (EXAMPLE 3.3 CONTINUED) The dependency graph $ic \leadsto^+ a$ and $ic \leadsto^- a$ forms a cycle. Hence the theory $\mathcal{T}$ is abductive recursive.

This dependency graph uses only the predicate name as distinguishable element. This limited level of detail makes that some programs are always classified as abductive recursive. In particular meta-theories, e.g. the Event Calculus [170], suffer from this. It is recommended in such cases to include the principal functors in the analysis to get a more detailed result.

To conclude this section, we give the termination theorem of Verbaeten [268] for SLDNFA extended to the $\mathcal{A}$system case. We need first some auxiliary definitions.

DEFINITION 3.13 A *level mapping* for a logic program P based on the language $\mathcal{L}$ is a function $|.| : \mathcal{HB}(\mathcal{L}) \cup \neg\mathcal{HB}(\mathcal{L}) \mapsto \mathbb{N}$, with $|\neg A| = |A|$ for all $A \in \mathcal{HB}(\mathcal{L})$. ($\neg\mathcal{HB}(\mathcal{L}) = \{\neg A | A \in \mathcal{HB}(\mathcal{L})\}$) A literal l is called *bounded* w.r.t. $|.|$ iff $|.|$ is bounded on the set $[l]$ of $\mathcal{L}$-ground instances of l. If l is bounded w.r.t. $|.|$ then we define $|l|$ as the maximum $|.|$ takes on $[l]$. A query $\mathcal{Q}$ is *bounded* w.r.t. $|.|$ if all its literal are bounded. A logic program P is *semi-acyclic* w.r.t. $|.|$ iff for each ground instance $p \leftarrow l_1 \wedge \ldots \wedge l_n$ of the rules in the program P holds that

$|p| > |l_i|$ if p and l_i depend mutually on each other
$|p| \geq |l_i|$ otherwise

THEOREM 23 (TERMINATION OF THE $\mathcal{A}$SYSTEM [268])
Let P be the abductive normal program derived from the $\mathcal{A}$system-theory ($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$). If P is semi-acyclic w.r.t. a level mapping $|.|$ and P is abductive nonrecursive, then for all bounded queries $\mathcal{Q}$ w.r.t. $|.|$, the $\mathcal{A}$system is terminating w.r.t. $\mathcal{Q}$.

3.3.5 How the $\mathcal{A}$system is derived from SLDNFA, IFF and ACLP

As mentioned, the $\mathcal{A}$system proof procedure is a mixture of the three most influential abductive procedures: SLDNFA [96], IFF [129] and ACLP [161]. In short, we have reformulated the SLDNFA proof procedure as a state (formula) rewriting system as done by IFF and added a finite domain constraint domain as ACLP.

To give a clearer view on this mixture, we present here these procedures in short.

- SLDNFA [95, 96, 90] is an abductive extension of SLDNF. The procedure computes answers for queries w.r.t. abductive normal logic programs. SLDFNA is proven sound and complete with respect to the three-valued completion semantics.

  To answer this query, SLDNFA performs the following derivation process. (We only sketch it.) From the query, SLDNFA constructs two goal sets: a set of the positive goals that must succeed and a set of negative goals that must fail. Succeeding a goal is done via reducing it to the empty goal. A goal is failed by the construction of a finitely failed tree. Negation is handled by moving the goal to the other goal set: e.g. if $\neg A$ is selected as a positive goal, it is added as $\leftarrow A$ to the set of negative goals.

  A recognized drawback of the procedure is its presentation. The complex formulation lacks clarity and hence it is difficult to overview and understand SLDNFA.

- The IFF-procedure [129] is closely related to SLDNFA. More precisely, it can be seen as a hybrid of the iff-abduction procedure of Console et al. [85] and SLDNFA. An IFF-theory consists of *if and only if*-definitions, representing the abductive program $\mathcal{P}$, and implications, representing the integrity constraints. Like SLDNFA, the IFF-procedure is sound and complete with respect to the three-valued completion semantics.

  The procedure is presented as a formula rewrite system: starting from the query the IFF-procedure computes an equivalent formula in terms of the abducibles, by applying a number of inference rules:

  1. unfolding of defined predicates: a defined predicate is replaced by its definition.

2. factoring of abducibles: given two abducibles, it is either the case that they are identical or different.
3. splitting: the non-deterministic selection of a disjunct in a disjunction.
4. equality rewrite rules based on the Clark's Free Equality axioms [78].
5. logical simplification: e.g. $\mathbf{false} \wedge Q$ is equivalent with **false**, etc.
6. forward propagation: for an implication $p \rightarrow Q$, if p is derived true then Q is added.
7. case analysis: for an implication $x = t \rightarrow Q$, it holds either $x = t \wedge Q$ or $x \neq t$.

Negation, e.g. $\neg A$, is dealt with by replacing it with an equivalent implication, $\mathbf{false} \leftarrow A$. In general, the formulation as a rewrite system makes the procedure more elegant than SLDNFA. Moreover, whereas in the earliest versions of SLDNFA Skolem constants were introduced to get the logic right, the IFF-procedure used free variables. This has been also adopted by SLDNFA in its later formalizations, which considerably simplified the presentation.

- ACLP [4, 152, 162, 161] is extends Constraint Logic Programming with abduction. Given a constraint logic program over a constraint domain $\mathfrak{D}$, a ACLP($\mathfrak{D}$)-program consists of rules

$$p_0 \leftarrow c_1, \ldots, c_n, p_1, \ldots, p_m, a_1, \ldots, a_k$$

where p_i are user defined predicate atoms, a_j abducible atoms and c_l constraints on the domain $\mathfrak{D}$. The integrity constraints are kept separately, and are formulated in the form of denials. The semantics of an ACLP theory are given by the Stable Model semantics.

Negation is treated by ACLP via the introduction of auxiliary abducibles. For each defined predicate p an auxiliary abducible predicate p^* is constructed. The original ACLP theory is then transformed according to the next rules:

  - Each negative literal $\neg l$ occurring in the bodies of the program rules and integrity constraints is replaced by the auxiliary predicate l^*.
  - The theory is augmented with the integrity constraints

$$\forall\, \overline{X}.\neg(p(\overline{X}) \wedge p^*(\overline{X}))$$

and

$$\forall\, \overline{X}.p(\overline{X}) \vee p^*(\overline{X})$$

This treatment of negation has already been used in earlier abductive procedures [117, 156].

The inference procedure consists of two kinds of derivations: an *abductive derivation* that unfolds (positive) goals to a set of constraints and abducibles (this corresponds to the evaluation of the positive goals in SLDNFA) and a *consistency derivation* for an abducible computing the necessary conditions for the consistency of the abducible. The latter corresponds to the evaluation of the negative goals invoked by the abduction of an abducible atom in SLDNFA. In order to deal correctly with non-ground abducibles ACLP uses a *naming* of variables, which corresponds to the skolemization in SLDNFA.

Kakas et al. [158, 162] observed that the constraint store which is constructed during the derivation is able to detect inconsistent branches early in the derivation process. By exploiting this in the ACLP system, the search space is more efficiently traversed than e.g. SLDNFA does.

One can see that the $\mathcal{A}$system proof procedure has inherited the separated reasoning for positive and negative goals from SLDNFA. The positive inference of SLDNFA corresponds to the $\mathcal{A}$system's inference rules for conjunctions; the negative inference is related with the inference rules for the denials. Observe the difference with the IFF-procedure: it does not act on denials but on general implications. That avoids some Lloyd-Topor normalization compared to the $\mathcal{A}$system. Also the classification used by IFF is slightly different: some rules in the $\mathcal{A}$system are combinations of those of IFF. For example, IFF considers the unfolding of a defined predicate as a separate rule from the selection of a disjunct in a disjunction. In the $\mathcal{A}$system both are combined into one rule D.1.

Another inherited part of SLDNFA is the way to ensure the consistency of the abduced atoms by keeping track of the derived denials in a separated store Δ^*. Such auxiliary structure is absent in the IFF-procedure. It deals in an elegant way with this issue. Suppose that state formula F contains the following implication $a(X) \rightarrow p(X)$. For a selected abducible atom $a(s)$, this formula is replaced by the formula $(a(X) \wedge X \neq s \rightarrow p(X)) \wedge X = s \rightarrow p(X)$. The extended implication $a(X) \wedge X \neq s \rightarrow p(X)$ expresses compactly that the implication has been evaluated for the abducible atom $a(s)$. Similarly, the IFF-procedure has no explicit representation of the set of abduced atoms Δ, and the set of equalities $\mathcal{E}$. The IFF-procedure shows that those auxiliary structures are not necessary. However, since these structures can be used to optimize the reasoning and visualize the existence of the subsolvers, we find it sensible to select this representation.

The $\mathcal{A}$system inference rules are presented using a left-to-right evaluation of the formulas. (Both SLDNFA and ACLP follow this strategy.) In the actual implementation, this strategy is mostly followed for the evaluation of denials. For conjunctions, the evaluation order is more complex due to the goal suspension mechanisms. As the IFF-procedure shows, one could leave the selection strategy free. The drawback of such freedom is that not all selections are safe. The verification of that would lead probably to a reduction in computational efficiency.

However, the $\mathcal{A}$system inference rules do not fix the selection strategy of the formulas, in contrast to the formulations of SLDNFA and ACLP. Both are presented as a top-down left-to-right procedure. As we will show in section 3.4, this strategy is very ineffective in general.

Another difference w.r.t. SLDNFA and IFF is the absence of a unification algorithm (in case of SLDNFA) or an explicit set of equality rewrite rules (IFF). In the $\mathcal{A}$system equality reasoning is done by a special equality solver $\mathcal{E}$. This decision fits in the multi-solver idea underlying the $\mathcal{A}$system. Note that in the ACLP-system, all equalities are handled by the finite domain constraint solver.

The latter points to a difference between the $\mathcal{A}$system and ACLP. Where ACLP defines for each constraint domain an enclosing abductive procedure, the $\mathcal{A}$system treats the constraint domain as a predefined logic, having a domain specific solver. The $\mathcal{A}$system has the ability to deal with several domains in one specification. On the one hand that complicates the reasoning, e.g. it is often impossible to determine for a variable to which domain it will belong when it is encountered for the first time. When there is a single constraint domain as in the ACLP system, where all variables are finite domain variables, some inference rules can be optimized, e.g. the reuse of abducibles. On the other hand, the multi-solver structure enhances both efficiency and expressiveness of the $\mathcal{A}$system.

3.3.6 Summary

This section has presented the proof procedure of the $\mathcal{A}$system, which is a combination of three abductive procedures SLDNFA [96], ACLP [161] and IFF [129]. It is a reformulation of the SLDNFA proof procedure as a state (formula) rewriting system as done by IFF and augmented with a finite domain constraint domain as ACLP. Since the changes are small w.r.t. the logic of SLDNFA, the soundness, completeness and termination results for SLDNFA are reused for the $\mathcal{A}$system.

3.4 Implementation of the $\mathcal{A}$system

Implementing the $\mathcal{A}$system procedure involves decisions at different levels: the used data structures, the evaluation of the inference rules and how the system searches a solution. In this and the followings sections we will present the key ideas that make the $\mathcal{A}$system a powerful abductive reasoner.

In a software system have data structures an important role: an important part of the actual performance of a system depends on the used data structures. The $\mathcal{A}$system generates and processes a lot of data during its execution. Based on a study of the occurring data transfers, we will design appropriate structures that support the $\mathcal{A}$system's reasoning.

With the chosen data structures, the inference rules are encoded in the form of a meta-program. This is a classical technique to build fast interpreters for new languages. It was also applied in the ancestors of the $\mathcal{A}$system: SLDNFA and ACLP. It turns out that the classical meta-programming in Prolog is not completely successful; for the evaluation of the denials, it generates a certain overhead which is measurable. In order to solve this problem, a more cost effective ground meta-program [53] is developed for the evaluation of the denials. This representation is a first step towards the compilation of an abductive logic program.

The above problems concern "low-level" issues which address the basic efficiency of the reasoning performed by the $\mathcal{A}$system. The most important factor w.r.t. the computational efficiency, however, is the organization of the search such that the $\mathcal{A}$system finds a solution relatively fast. A bad search strategy (one that uses a lot of backtracking) leads inevitably to a system with a poor performance. Therefore much work has been spent in developing and evaluating techniques to improve the search. This work has often revealed shortcomings in other parts of the solver, e.g. in the equality reasoning and finite domain constraint solver. These implementation aspects are presented in their respective sections.

Our goals with the development of the $\mathcal{A}$system are twofold: (1) the construction of a robust publicly available abductive reasoning system and (2) the study of efficient implementations of an abductive inference process. To fulfill the first goal, some implementation effort is spent in the user interface. It includes: a preprocessor which generates (simple) error messages for some common mistakes, a help functionality and a small user manual. Other features are the debugger and the large number of parameters which can be used e.g. to retrieve information about a system's run or to change the system's behavior. These are both useful for the user, who can tune the $\mathcal{A}$system to its needs, and for the developer to study the system's behavior. For the latter goal and in order to keep the code maintainable, the program code is modularized as much as possible. We leave most of these aspects out of the dissertation, except the parameters that involve the search.

The section is organized from the high level efficiency issues to the more low level. It starts with the search process. We will discuss the application order of the inference rules and how the evaluation of the inference rules can be optimized in order to reach a solution faster. Next the supporting data structures are discussed. As a last topic, the actual implementation of the inference as a meta-program is discussed. At the end, we summarize our contributions presented in this section.

3.4.1 The first prototypes

The first abductive prototypes followed closely the decisions of Prolog, an implementation of SLDNF. They were implemented as a meta-program applying a left-to-right selection rule with depth-first search. This organization results in a rather unpredictable behavior of the systems. For one representation the problem instances are solved efficiently; but minor changes to this specification, e.g. the reordering of goals in the query or the clauses of rules, degrades its performance. Hence the system behavior depends too strongly on the problem specification.

Another problematic aspect of the Prolog-style is the blind chronological search. At each choice, branches are selected and evaluated without checking whether the chosen branch is a good candidate.

Example 3.5 (Big goal) Consider the following abductive logic theory

$$
\begin{array}{ll}
\mathcal{P} & = \left\{ \begin{array}{l} p(X) \leftarrow a(X). \\ b(X) \leftarrow ... \end{array} \right\} \\
\mathcal{A} & = \{a(\cdot)\} \\
\mathcal{IC} & = \left\{ \begin{array}{l} \forall\, X. \leftarrow a(X) \wedge \neg b(X). \\ \leftarrow a(0). \end{array} \right\}
\end{array}
$$

Suppose that the body of $b(X)$ is hard to evaluate, i.e. it requires a long derivation until a consistent state is reached. In a top-down left-to-right evaluation strategy the query $p(0)$ leads to the following derivation (the bold formula is selected):

$$
\begin{array}{ll}
 & \mathcal{S}_0 = (\{\mathbf{p(0)}\} \cup \{\forall\, X. \leftarrow a(X) \wedge \neg b(X)\,, \leftarrow a(0)\}, \mathcal{ST}^{\emptyset}) \\
\leadsto_{D.1} & \mathcal{S}_1 = (\{\mathbf{a(0)}\,, \forall\, X. \leftarrow a(X) \wedge \neg b(X)\,, \leftarrow a(0)\}, \mathcal{ST}^{\emptyset}) \\
\leadsto_{A.1} & \mathcal{S}_2 = (\{\forall\, \mathbf{X}. \leftarrow \mathbf{a(X)} \wedge \neg \mathbf{b(X)}\,, \leftarrow a(0)\}, (\{a(0)\}, \emptyset, \emptyset, \emptyset)) \\
\leadsto_{A.2} & \mathcal{S}_3 = (\{\mathbf{b(0)}, \leftarrow a(0)\}, (\{a(0)\}, \{\forall\, X. \leftarrow a(X) \wedge \neg b(X)\}, \emptyset, \emptyset)) \\
\leadsto_{D.1} & \ldots \\
\leadsto_{\ldots} & \mathcal{S}_i = (\{\leftarrow \mathbf{a(0)}\}, (\{a(0), \ldots\}, \{\forall\, X. \leftarrow a(X) \wedge \neg b(X)\,, \ldots\}, \mathcal{E}_i, \mathcal{FD}_i)) \\
\leadsto_{A.2} & \text{fail and backtrack}
\end{array}
$$

In the first steps, the atom $p(0)$ is evaluated by unfolding, leading to the addition of $a(0)$ to Δ. At step 2, two constraints remain to be further evaluated. The top-down selection strategy first selects the constraint $\forall\, X. \leftarrow a(X) \wedge \neg b(X)$. Evaluated from left to right, $b(0)$ has to be proven next. When after a long time

$b(0)$ is proven, the second constraint is selected and immediately fails. Together with chronological backtracking, this leads to a full exploration of the derivation tree of $b(0)$. Selecting the second constraint first would avoid the exploration of $b(0)$.

Of course this is a well-known problem, but abductive reasoning is particularly vulnerable to it. Since integrity constraints often specify *simple* restrictions on abducibles, early checks can avoid useless computations. A related problem is thrashing, where the system explores a (infinitely) large subtree before backtracking to the origin of the unsatisfiability.

To avoid these problems, the general strategy of delaying the evaluation of complex goals is often applied. Only after resolving all simple goals, one of the postponed goals is selected for evaluation. Applied on example 3.5, delaying the evaluation of $b(0)$ allows the second constraint to be selected. This constraint fails and without exploring $b(0)$ the answer to the query is generated.

Because we observed that the abductive systems made too much uninformed choices, we applied this suspension technique more and more in the subsequent versions of the $\mathcal{A}$system. Another observation during the experiments with early versions was that the Prolog-like systems made too many small commitments. For example, in inference rule A.1 a selected abducible atom is either unified with one of the elements in the current Δ or it is added as a new hypothesis to Δ. Because we had no way to express that the atom is unified with one of the elements in Δ as a single statement, each possibility had to be tried one by one. Such commitments lead to a large variation in execution times: a small change to the input data of the problem (e.g. goal ordering) can turn an effective execution into a very ineffective one, and vice versa.

A technique that turned out to be useful for both problems is *reification*: the association of a boolean variable with a formula. This boolean denotes the truth of the formula w.r.t. the current store. Originally, the technique was only available for finite domain expressions, but it was of such a great help to simplify the $\mathcal{A}$system implementation, that we have extended it to equality reasoning over Herbrand terms. Section 3.7 presents in more detail reification.

3.4.2 The organization of the search

One important aspect we have touched upon above, is the difference between complex and simple inferences. Applied on the $\mathcal{A}$system's inference rules (see Section 3.3.2), we can classify those rules as either *deterministic* or *non-deterministic*. Deterministic rules transform the input state to another unique state. Non-deterministic rules are not deterministic rules and embody a choice point in the $\mathcal{A}$system derivation. Based on this classification the $\mathcal{A}$system's search is organized in subsequent phases: a deterministic phase where all deterministic inference rules are applied until only non-deterministic ones are left, followed by one non-deterministic inference rule application. This application invokes a new deterministic phase, and so on.

Deterministic inference rules

This phased search is possible due to the following properties of a set of applicable deterministic rules $\{r_1, r_2, \dots\}$ on an $\mathcal{A}$system state $\mathcal{S}_i$:

- The deterministic rules are *confluent*. The state $\mathcal{S}_{i+2}$ obtained via the chain $r_1 \prec r_2$ is the same as via $r_2 \prec r_1$. This can be verified easily for each pair of deterministic rules. Hence the application order of the rules is irrelevant.
- The order has little impact on the overall efficiency. If $\mathcal{S}_i$ is part of a derivation leading to a solution state then all rules $\{r_1, r_2, \dots\}$ must be evaluated successfully. Only if $\mathcal{S}_i$ is not extendable to a consistent state $\mathcal{S}_{i+n}$, then the time to find out the inconsistency is order-dependent. From our experimental experience we have observed that the set of applicable deterministic rules is usually small and hence inconsistency is almost immediately detected.

These properties justify to compact deterministic rules into one transition step $\rightarrow_d$ that transforms an $\mathcal{A}$system state deterministically into the next state. This is denoted as $\mathcal{S}_i \rightarrow_d \mathcal{S}_{i+1}$. The effect of this transition is formally described via the following notions.

DEFINITION 3.14 The *deterministic consequence state* of a formula F for a state $\mathcal{S}$=($\mathcal{G}$,(Δ,Δ^*,$\mathcal{E}$,$\mathcal{FD}$)) is the state ($\mathcal{G}$',(Δ',Δ^*',$\mathcal{E}$',$\mathcal{FD}$')) obtained by executing all applicable deterministic rules for the state[4] ({F},(Δ,Δ^*,$\mathcal{E}$,$\mathcal{FD}$)) until no deterministic inference rule is applicable. The *deterministic unfolding* of a formula F for a state $\mathcal{S}$ is the difference between the state ({F},(Δ,Δ^*,$\mathcal{E}$,$\mathcal{FD}$)) and the computed deterministic consequence state $\mathcal{S}_c = (\mathcal{G}_c, (\Delta_c, \Delta^*_c, \mathcal{E}_c, \mathcal{FD}_c))$:

$$\begin{aligned}(\mathcal{G}_u, (\Delta_u, \Delta^*_u, \mathcal{E}_u, \mathcal{FD}_u)) &= (\mathcal{G}_c, (\Delta_c, \Delta^*_c, \mathcal{E}_c, \mathcal{FD}_c)) \setminus (\{F\}, (\Delta, \Delta^*, \mathcal{E}, \mathcal{FD}))\\ &= (\mathcal{G}_c \setminus \{F\}, (\Delta_c \setminus \Delta, \Delta^*_c \setminus \Delta^*, \mathcal{E}_c \setminus \mathcal{E}, \mathcal{FD}_c \setminus \mathcal{FD}))\end{aligned}$$

Note that this deterministic consequence state might not be reachable due to the existence of loops in the program. The notion of a deterministic unfolding of a formula w.r.t. a given state will be used intensively as part of the forward propagation techniques to improve the search.

EXAMPLE 3.6 Consider the program

$$\begin{array}{l} p(X) \leftarrow a(X) \wedge r(X).\\ r(1).\\ r(2).\\ \leftarrow a(X) \wedge X < 2.\end{array}$$

[4]The goal stack of the state is limited to F because we want only to derive (directly related) information about F w.r.t. the state $\mathcal{S}$.

The deterministic unfolding of $p(Y)$ for some state

$$\mathcal{S}_i = (\{a(Z)\}, (\emptyset, \{\forall\, X. \leftarrow a(X) \wedge X < 2\}, \emptyset, \emptyset))$$

is computed as:

1. Its deterministic consequence state $\mathcal{S}_c$ is obtained by applying all deterministic inference rules on $(\{p(Y)\}, (\emptyset, \{\forall\, X. \leftarrow a(X) \wedge X < 2\}, \emptyset, \emptyset))$. This yields the state $\mathcal{S}_c = (\{r(Y)\}, (\{a(Y)\}, \{\forall\, X. \leftarrow a(X) \wedge X < 2\}, \emptyset, \{Y \geq 2\}))$.

2. The deterministic unfolding $\mathcal{S}_u$ of $p(Y)$ is then

$$\begin{aligned}
\mathcal{S}_u &= \mathcal{S}_i \setminus \mathcal{S}_c \\
&= (\{p(Y)\}, (\emptyset, \{\forall\, X. \leftarrow a(X) \wedge X < 2\}, \emptyset, \emptyset)) \\
&\quad \setminus (\{r(Y)\}, (\{a(Y)\}, \{\forall\, X. \leftarrow a(X) \wedge X < 2\}, \emptyset, \{Y \geq 2\})) \\
&= (\{r(Y)\}, (\{a(Y)\}, \emptyset, \emptyset, \{Y \geq 2\}))
\end{aligned}$$

NOTATION 3.4.1 $\mathcal{S}_i \rightarrow_d \mathcal{S}_{i+1}$ denotes a (deterministic) transition from a state $\mathcal{S}_i$ to a state $\mathcal{S}_{i+1}$, where $\mathcal{S}_{i+1}$ is the result of an iterative process in which deterministic rules are applied until the goal stack contains only choice points.

EXAMPLE 3.7 Example 3.6 continued The deterministic transition $\mathcal{S}_i \rightarrow_d \mathcal{S}_{i+1}$ from $\mathcal{S}_i = (\{a(Z), p(Y)\}, (\emptyset, \{\forall\, X. \leftarrow a(X) \wedge X < 2\}, \emptyset, \emptyset))$ result in the state

$$\mathcal{S}_{i+1} = (\{a(Z)\}, (\{a(2)\}, \{\forall\, X. \leftarrow a(X) \wedge X < 2\}, \emptyset, \{Y \geq 2\}))$$

First, $p(Y)$ is selected in $\mathcal{G}_i$ since it has only one clause. The addition of the deterministic unfolding to $\mathcal{S}_i$ yields $(\{a(Z), r(Y)\}, (\{a(Y)\}, \{\forall\, X. \leftarrow a(X) \wedge X < 2\}, \emptyset, \{Y \geq 2\}))$. Since $Y \geq 2$ and the only clause of r which satisfies this condition is $r(2)$, the evaluation can be continued and the state $\mathcal{S}_{i+1}$ is reached. Further deterministic reasoning is not possible.

The deterministic inference rules do not make choices and, thus, they are as such not relevant for the actual search. One is tempted to spend all attention to the choice points and forget about these rules. However, the majority of the reasoning in an $\mathcal{A}$system derivation is performed by these rules and thus the inference must be sufficiently effective to get an overall good performance. This aspect is studied in Section 3.4.4.

The choice points

From the inference rules of Section 3.3.2, only five rules are non-deterministic and thus lead to a choice point. They are listed below:

1. a selected conjunction

   (a) positive definitions (pos): inference rule D.1.
   (b) reuse of abducibles (reu): A.1.

2. a selected denial
   (a) negation constraints (neg): N.2.
   (b) equality constraints (equ): E.2.b.
   (c) finite domain constraints (clp): F.2.

The abbreviations are used in the implementation to denote the code that is connected to these choice points.

Each choice point represents logically a disjunction $d_1 \vee \ldots \vee d_n, n > 1$, where each disjunct d_i is a formula. When the choice point is selected for evaluation, one of the disjuncts is chosen to proceed with. The other disjuncts are suspended until backtracking activates the need for their evaluation. Similarly as above, the selection and evaluation of a choice point is denoted as $\mathcal{S}_i \rightarrow_{nd} \mathcal{S}_{i+1}$, where $\mathcal{S}_{i+1}$ is the result from the application of the choice on the $\mathcal{S}_i$. It is called an *non-deterministic* transition.

The scheduler

An $\mathcal{A}$system derivation is then visualized by the following chain of alternating deterministic and non-deterministic transitions.

$$\mathcal{S}_0 \rightarrow_d \mathcal{S}_1 \rightarrow_{nd} \mathcal{S}_2 \rightarrow_d \mathcal{S}_3 \rightarrow_{nd} \ldots$$

Each choice point evaluation is followed by a (large) propagation step of the newly added information by the evaluation of the choice.

This process is implemented by the scheduler in Figure 3.1. If the set of choices is empty, the search is finished. Otherwise, the scheduler checks first if there is a deterministic (implied) goal. If so, it is evaluated. In the other case, a backtrack point is created by selecting and evaluating one of the choice points.

The deterministic transition also affects the disjuncts of the choice points. Suppose we can observe that a disjunct is surely inconsistent w.r.t. the current state store. Due to the deterministic propagation of information, it may happen that all disjuncts but one of a choice point are observed to be inconsistent w.r.t. the current state store. Then this choice point represents no choice anymore and it forms an excellent candidate to be evaluated next. Another interesting case is when all disjuncts of a choice point are inconsistent w.r.t. the current state store. The state is then unsatisfiable and the $\mathcal{A}$system has to backtrack. These example cases of information propagation show that some choice points do not lead to a backtrack point in a derivation. It is well-known that the more backtrack points are created during the search, the higher is the chance for a long and time consuming search process. Therefore the use of a detection mechanism, that reduces the number of

```
evaluate(Choices,_State):-
   empty(Choices),!.
evaluate(Choices,State):-
   exists_det_choice(Choices,Ch1,RestChoices),!,
   evaluate_det(Ch1,RestChoices,State).
evaluate(Choices,State):-
   select_choice(Choices,Ch1,RestChoices),
   evaluate_sel(Ch1,RestChoices,State).

evaluate_det(Ch1,RestChoices,State):-
   % do propagation of Ch1: no backtracking possible
   propagate(Ch1,RestChoices,NewChoices,State),!,
   evaluate(NewChoices,State).

evaluate_sel(Ch1,RestChoices,State):-
   % do propagation of Ch1: backtracking possible
   propagate(Ch1,RestChoices,NewChoices,State),
   evaluate(NewChoices,State).
```

Figure 3.1: The scheduler

backtrack points, such as the one explained before, may make the search process more effective.

Forward propagation

Choice points are in the $\mathcal{A}$system first class citizens, which require a special treatment in order to detect the above discussed situations. The $\mathcal{A}$system transforms them to suspended goals. Let G be a choice point with n disjuncts $d_1 \vee \ldots \vee d_n$. A suspended goal (choice point) associates with each disjunct d_i a condition c_i that represents its (dis)entailment w.r.t. the current state store. The goals are suspended, i.e. they wait for execution, until the conditions reflect a situation that implies one single action. Since each choice point type has its own characteristics, the applied reasoning is different for all of them. Before discussing each choice point separately, we introduce the technique, called reification, that forms the basis of our detection implementation.

Reification is the association of a boolean variable B to a constraint C: it is denoted as $C \Leftrightarrow B$. This expression is evaluated w.r.t. the current constraint store $\mathcal{C}$. B represents the truth value of C with respect to $\mathcal{C}$. Procedurally, if C is entailed **false** (**true**) w.r.t. $\mathcal{C}$, B is 0 (1), and if B is assigned 0 (1) then $\neg C$ (C) is added to the constraint store. In finite domain constraint solvers such as the one of Sicstus [66], reification is used to implement logical connectives. For example, the disjunction $C_1 \vee C_2$ is expanded to

$$\begin{array}{l} B_1 \vee B_2 \\ C_1 \Leftrightarrow B_1 \\ C_2 \Leftrightarrow B_2 \end{array}$$

In the first prototype of the $\mathcal{A}$system [103], only reification of finite domain expressions was used. Since it turned out to be useful at other places in the $\mathcal{A}$system, it was one of the motivations to implement an equality solver $\mathcal{E}$ with reification support. For more details on reification and the (needed) improvements we made to the original mechanism, see Section 3.7.

A) **Choice points originating from denials**
The choice points equality constraints, finite domain constraints and negation constraints originate from a denial. These denials share the same general form $\forall\, \overline{X}. \leftarrow C \wedge Q$ where the literal C determines the type of the choice point. The choice point represents the binary choice

$$\neg C \vee (C \wedge \forall\, \overline{X}. \leftarrow Q)$$

The literal C is the hinge of the choice; its entailment or disentailment determines exactly which disjunct is to be selected.

In the case of equality constraints and finite domain constraints, the literal C is an expression that is not evaluated by the $\mathcal{A}$system itself, but by the equality solver or the finite domain solver. Since these solvers provide reification, the choice point can be suspended by the combination of reification and coroutining, i.e. a Prolog built-in that is able to delay the execution of a goal until a condition is satisfied.

The next piece of code describes the construction of a suspended goal using reification and coroutining:

```
suspended_goal(denial(C,Q),sus_denial(B,C,Q)):-
   negation(C,NegC), % compute the negation of C = NegC
   NegC <=> B,       % reify NegC
   when(ground(B),   % wait until B is ground and then execute
      (B = 0               % the suspended goal
         -> evaluate(Q)    % if B=0, then evaluate Q
         ;  true           % otherwise the denial is satisfied
         )).
```

When C contains no universally quantified variables (as required for finite domain expressions) the computation of the negation can be omitted. In that case, the above code must be adapted accordingly.

For negation constraints, the suspension is slightly more complex. Depending on the literal that is negated, the suspension mechanism of that literal is used. Given the corresponding boolean, the correct reasoning is implemented.

B) **Positive definitions**
This choice point is characterized by the potential of having many disjuncts, one for each clause in the definition of the defined predicate.

For each disjunct a guard is constructed. A *guard* is the equality and finite domain constraint store given by the deterministic unfolding of the disjunct. The guard forms a necessary condition for the success of the disjunct. If a state disentails the guard, the connected disjunct cannot lead to a successful state and thus this branch can be excluded from future exploration. The entailment, however, is not sufficient to guarantee success. To find out the (dis)entailment, the guard is reified and it is added to the appropriate constraint store.

Checking if a positive definition choice point has become deterministic requires to verify that just one guard is left. This check can be reduced to a single value check: namely, the sum of the booleans of the guards. This sum determines the amount of potentially successful disjuncts if the value of all booleans is known. However, this sum reduces the usefulness of the guards, since it requires that the value of each boolean is known. By a small change to the definition of the condition test, we obtain a useful test: if the maximal value that the sum can take is 1, the choice point is deterministic. This version notifies the scheduler whenever only one disjunct is remaining, independent of the state that the associated boolean of the guard of this disjunct has (ground or non-ground).

```
suspended_goal(pos(Pos),sus_pos(B,Guards,Pos)):-
   create_guards(Pos,Guards), % construct all reified guards
   sum(Guards,B),             % the sum of the guard booleans
   when(max(B)=<1,            % wait until the max-value of B=<1
      (max(B)=1
         -> execute(Guards,Pos) % evaluate the left-over branch
         ;  fail
         )).

create_guards(Pos,Guards):-
   number_of_clauses(Pos,N),  % the number of clauses of Pos
   create_guards(N,Pos,Guards).

create_guards(0,Pos,[]).
create_guards(N,Pos,[G|Gs]):-
```

```
    N>0,
    clause(N,Pos,Body),        % select the Nth clause
    deterministic_unfolding_ClpEq(Body,CLPEQ),
    CLPEQ <=> G,               % reify the guard
    N1 is N-1,
    create_guards(N1,Pos,Gs).
```

When a positive definition choice point is selected for an explicit choice[5], the reified guards are also informative. Some of the disjuncts may already have been detected as unsatisfiable, consequently they must not be explored. However this explicit choice-making imposes a special requirement on the reification procedures. It has to be possible to remove the reified formula from the constraint store. Making an explicit choice for a choice point removes the need for forward propagation on that choice point. Moreover when the formula stays in the data structures it will just waste computational time and computer memory. This operation is not supported by the original reification mechanism (in CLP(FD)), hence we have implemented it ourselves. (See Section 3.7.)

C) **Reuse of abducibles**
The name of this choice point refers to the attempt in inference rule A.1 to unify a new hypothesis with the already derived ones. The goal of the rule is to ensure that each abduced atom is consistent with the integrity constraints. For an abducible atom, this can be achieved either by unifying the atom with an already known consistent abducible atom or by verifying explicitly the consistency. That is reflected in the inference rule.

Let us recall the rule: for the selected formula $\mathbf{a}(\overline{\mathbf{t}}) \wedge \mathbf{Q}$ in state $\mathcal{S}_i$

- SELECT an arbitrary $a(\overline{s}) \in \Delta_i$ such that $\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{Q\} \cup \{\overline{s} = \overline{t}\}$ OR,
- $\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{Q\} \cup \{\forall\, \overline{X}. \leftarrow \overline{u} = \overline{t} \wedge R \mid \forall\, \overline{X}. \leftarrow a(\overline{u}) \wedge R \in \Delta_i^*\} \cup \{\overline{s} \neq \overline{t} \mid a(\overline{s}) \in \Delta_i\}$ and $\Delta_{i+1} = \Delta_i \cup \{a(\overline{t})\}$

In the first branch an already abduced atom $a(\overline{s})$ of Δ_i is unified with the selected atom $a(\overline{t})$. Note that this choice itself contains a choice, namely the selection of an element out of Δ_i. The second branch expands the set of abduced atoms by adding $a(\overline{t})$ to Δ_i. This requires that $a(\overline{t})$ is consistent with all constraints, i.e. the formulas stored in Δ_i^*, and that it is unique, i.e. different from all other elements of Δ_i. The latter ensures that there is no overlap with the first branch and that Δ forms a set. Note that this inference rule is the only one that affects the set Δ.

[5]This is a non-deterministic step, creating a backtrack point in the system's execution.

EXAMPLE 3.8 Suppose a state $\mathcal{S}_i$ containing $\Delta = \{a(1), a(S)\}$ and that $a(X)$ is the selected formula. The generated choice point is either $a(1) = a(X)$, or $a(S) = a(X)$ or $a(X)$ is a new hypothesis different from the existing ones: $a(X) \neq a(1) \wedge a(X) \neq a(S)$.

The example shows that the decision which branch to take is covered by the formula

$$\bigvee_{\{a(\overline{s}) \in \Delta_i\}} a(\overline{t}) = a_i(\overline{s})$$

The formula is called the *reuse formula*. If it is entailed by a state, then the abducible $a(\overline{t})$ is equal to one of the existing hypotheses by Δ_i. Disentailment on the contrary requires that $a(\overline{t})$ is treated as a new hypothesis w.r.t. the hypotheses in Δ_i.

Again, reification is useful here. Using

$$\bigvee_{\{a(\overline{s}) \in \Delta_i\}} a(\overline{t}) = a_i(\overline{s}) \Leftrightarrow B$$

the choice point A.1. is suspended until (dis)entailment is detected or an explicit choice is forced. Note that the reuse formula only involves equations of the Herbrand Universe. Since the reuse of abducibles is a vital choice point in the abductive system, the equality reasoning and reification in that constraint domain are vital as well.

The next example shows that in case of disentailment the awakening of a suspended reuse choice point does not ensure the removal of the choice point.

EXAMPLE 3.9 Suppose that the selection of the abducible atom $a(X)$ at stage i has led to the suspended reuse choice point based on the reified expression $a(X) = a(1) \vee a(X) = a(S) \Leftrightarrow B$. The $\mathcal{A}$system proceeds and after some derivation steps state $\mathcal{S}_j$,$i < j$, is reached. Suppose that $\mathcal{S}_j$ disentails the reuse formula and thus $B = 0$. Then it is known that $a(X)$ is different from $a(1)$ and $a(S)$, and according to the inference rule A.1. $a(X)$ should be added as a new hypothesis when this was detected in $\mathcal{S}_i$. However, the state $\mathcal{S}_j$ has a larger $\Delta_j \supseteq \Delta_i$ than state $\mathcal{S}_i$. So it might be that some abduced atom $a(s) \in \Delta_j \setminus \Delta_i$ could be unified with $a(X)$. This has to be verified and again the reuse choice point can be suspended. Only if all hypotheses are checked then the addition of the abducible as a new hypothesis in Δ is allowed.

To support this reasoning the abduced atoms get a unique identifier (an integer) assigned. The highest identifier of the abduced atoms used in the reuse formula is stored in the suspended goal. When the goal is woken and the integer is the same as the highest of the current state, then all existing hypotheses are checked and a real abduction must happen.

The determination if an abductive atom will be reused or not, is a slow process. When deriving an abducible for the first time, it is often very general. That means that the atom contains a lot of free variables on which very few restrictions are imposed. Consequently, such an atom seems unifiable with many of the elements in Δ, and it takes a long time to find out if it is really reusable or not. This process may be speeded up by adding redundant information.

We have observed that in many problem specifications integrity constraints encode simple information that limit the possible values of the variables in abducibles. A typical example is the precondition requirement in AI planning. Recall the specification of the blocks world domain from section 3.3.3.

Example 3.10

$$\forall\ X, Y, E.\ \leftarrow move(X, Y, E) \wedge \neg succeeds\_move(X, Y, E)$$

$$\begin{array}{l} succeeds\_move(X, Y, E) \leftarrow \\ \qquad\qquad ablock(X) \wedge location(Y) \wedge time(E) \wedge \\ \qquad\qquad X \neq Y \wedge clear\_block(X, E) \wedge clear\_location(Y, E). \end{array}$$

If e.g. an abducible atom $move(X_1, Y_1, T_1)$ is selected, the information of $ablock(X_1)$, $ablock(Y_1)$, $time(T_1)$ and $X_1 \neq Y_1$ is already useful to limit possible unifications.

Irrespective of whether the abducible atom is reused or not, the abducible atom will satisfy the integrity constraints in a solution state. Therefore, it is safe to add the *certain* information of the integrity check, namely the information that follows deterministically from the integrity check of the abducible atom under consideration. In example 3.10, this information includes e.g. the type restrictions $ablock(X_1)$, $ablock(Y_1)$ and $time(T_1)$. Formally, the *lookahead* for an abducible atom $a(\overline{t})$ in a state $\mathcal{S}_i$ is the (finite domain or equality) constraint store $\mathcal{C}$ computed by the deterministic unfolding of the integrity constraints of $\mathcal{S}_i$, i.e. unfolding Δ_i^* w.r.t. $a(\overline{t})$. As argued, $\mathcal{C}$ will be satisfied in every solution state that is an extension of $\mathcal{S}_i$, and can be added to the current constraint stores.

The lookahead is *redundant* information. If $a(\overline{t})$ is unifiable with an element $a(\overline{s})$ of Δ_i then $\mathcal{C}$ was already added to the constraint store by the integrity check of $a(\overline{s})$. In the other case, when $a(\overline{t})$ must be abduced, the integrity checking is recomputed because when it is detected, the state $\mathcal{S}_j$, $i < j$, contains more integrity constraints than the one ($\mathcal{S}_i$) in which the lookahead was computed. This redundancy is a knife that cuts both ways. It allows to detect the (dis)entailment of the abducible atom under consideration earlier, but it also increases the complexity and size of the constraint stores. The latter becomes visible in the time needed by the constraint solver to construct a solution. For more on this issue, see Section 3.6.

Further optimization

The search process can be subject to many more improvements. Here we will present two simple optimizations that are incorporated in the $\mathcal{A}$system. Other potential extensions are collected in the section on future work.

Improving determinism In many specifications, definitions occur with only one clause. Such definitions never create a choice point, hence the $\mathcal{A}$system reasoning can continue immediately by unfolding the definition. Because it is a property of the predicate, only depending on the specification, this meta-information is derived when preprocessing the specification file. The $\mathcal{A}$system implementation of the inference rule D.1 checks this meta-information when it is triggered to decide if either a choice-point must be created or the reasoning can continue.

Another optimization is for the denial choice points. When such a denial consists of a single literal, this literal must fail. It is easy to see that the second branch of the inference rules E.2b, F.2 and N.2 is unsatisfiable since the empty denial is **false**. In case of equality (E.2b) and finite domain constraints (F.2), the negation of the expressions is added to the constraint store. For inference rule N.2., the mode of reasoning is switched and the goal is (positively) evaluated.

The $\mathcal{A}$system as pure constraint generator Since a backtrack point at the level of the $\mathcal{A}$system search is more costly than one in the constraint store, the constraint choice points are optimized as follows. Let c_1 and c_2 be to constraint literals of the same constraint domain. If the denial $\forall\,\overline{X}. \leftarrow c_1 \wedge c_2 \wedge Q.$ is selected, the $\mathcal{A}$system will consider the conjunction of $c_1 \wedge c_2$ as one constraint. This view allows to move one $\mathcal{A}$system choice point to the constraint store. Applying the original inference rules, first the satisfiability of c_1 is verified and only then when necessary c_2 is evaluated. The optimization requires that the constraint solver can handle (reified) disjunctions.

Later in Section 3.8, we will show that for one class of specifications in which all abducibles belong to a special class, i.e. open functions, the $\mathcal{A}$system derivation collapses to one $\rightarrow_d$ transition. All computational complexity is then handed to the underlying constraint solvers. In principle this is beneficial because normally the constraint solvers have a better (due to a more specific) search than the general search by the $\mathcal{A}$system. However, the constructed constraint stores may be too complex for the constraint solvers so that the performance actually decreases. At this moment, only by experimentation a balance can be found. For more on this topic, see Section 3.6.

Search strategies

The above considerations structure the $\mathcal{A}$system search so that as few choices as possible have to be made. This section concerns the problem which choice point to select to proceed when only choice points are left in the goal stack of the state, i.e. a non-deterministic transition.

We cannot define one universal strategy such that all problems are solved efficiently. Therefore we focus on the one hand on strategies that result in a good average performance. On the other hand, the search in the $\mathcal{A}$system is parametrized such that the characteristics of a given problem can be exploited to reach a solution faster.

We will stick to complete search strategies. Complete strategies ensure that all branches that eventually lead to a solution are explored; while incomplete strategies skip some. Incomplete search is often faster, but of course it does not guarantee that if no solution is found there is no solution. For many problems this property is important and hence we have limited our attention to complete search strategies. Incomplete search strategies are left for future work.

Our experiences with solving abductive tasks have led to formulate some general heuristic principles. In general it is good to evaluate the positive definitions early because they contain a large amount of unexplored knowledge. More concrete knowledge leads to more detailed states and hence more forward propagation is achieved. On the other hand it is good to postpone the reuse of abducibles quite long. Reuse typically tightens the constraint stores strongly. In applications such as AI-planning, the evaluation of the positive definitions leads to the introduction of almost all necessary abducibles of a solution, and thus the reuse has more chances to succeed. For the other three types of choice points it is less obvious what is the best place. In many of our applications the equality and finite domain constraint choice points were eliminated by the extra information added by the evaluation of the positive definitions.

The above heuristics make no statement about which choice point to select among the choice points of the same type. Therefore we have implemented different selection strategies that can be used to improve the efficiency of the $\mathcal{A}$system on a particular problem:

- **oldest first (FIFO)**: select the choice point that is created first. This is breadth-first.

- **youngest first (LIFO)**: select the choice point that is created last. This is a form of depth-first.

- **number of branches**: select the choice point with the least number of potential successful branches. The chance to select the right branch is the highest here.

- **number of variables**: select the choice point with the least number of variables. Variables denote lack of knowledge, hence selecting the choice point with the least number of variables corresponds to selecting the one of which the most is known. Again the chance to make a right choice is higher.

- **hierarchy of predicates**: For some problems, e.g. in hierarchical planning problems, the expert has meta-knowledge about the choices in the problem domain. It might be known that the decision taken for some choices has important (positive) effects on the traversed search space if they are taken before others.

  This ordering has to be given by the expert as an ordered list of predicates. (It is called a hierarchy since it was introduced in the context of planning problems.) Based on this ordering, the first goal of the most preferred predicate is selected.

The above strategies are all fixed by the human expert. Unfortunately, it requires a lot of experimentation to finetune the strategy since no good general guidelines are available that direct the human expert towards the 'best' strategy. Therefore, it would be of great help if the system is able to derive a strategy automatically that leads faster to a solution. That is future work.

3.4.3 Data structures

The main $\mathcal{A}$system data structure is the $\mathcal{A}$system state $\mathcal{S} = (\mathcal{G}, \mathcal{ST})$. Its structure must support the search that is explained above. For this, the operations on each part of $\mathcal{S}$ are analyzed and an appropriate data structure is constructed.

The goal stack

The goal stack $\mathcal{G}$ stores the (suspended) choice points. The main operations are addition and deletion of choice points. Retrieval of choice points seems less important, however to implement the different selection strategies of the search[6], the retrieval of the elements is important. Therefore we have selected AVL-trees (provided by the library `assoc` in Sicstus Prolog). The insertion, retrieval and deletion cost at most $O(log(N))$ operations, where N is the number of elements in the AVL-tree. To support the type based evaluation of the choice points, the goal stack is encoded as a tuple of $(Pos, Reu, Neg, Equ, Clp)$ in which each argument is an AVL-tree storing the corresponding elements.

[6]Some strategies are dynamical e.g. number of vars, and thus the exact value can only be computed when it is requested.

A choice point itself is encoded as an instance of the term

$$choice(Type, ID, Head, Tail, B, LocalInfo, GlobalInfo)$$

where the arguments denote

- *Type*: the type of the choice point.
- *ID*: the unique identifier of the choice point.
- *Head*: the atom on which the choice depends.
- *Tail*: the logical expression that must be evaluated if the *Head* is successful.
- *B*: the boolean that is associated with the reified expression.
- *LocalInfo* and *GlobalInfo*: information stored about the choice point. *LocalInfo* stores information that is used to support the forward propagation e.g. the sum of the guards of positive definitions. *GlobalInfo* stores information about the choice point inferred by the $\mathcal{A}$system e.g. the history of the choice point.

Not every argument is sensible for each type of choice point. In that case the argument is assigned a null-value.

The store of the state

From the store $\mathcal{ST}$= (Δ,Δ^*,$\mathcal{E}$,$\mathcal{FD}$), only Δ and Δ^* require a special data structure because $\mathcal{E}$ and $\mathcal{FD}$ are maintained by their respective constraint solvers. Recall that Δ stores abducibles $a(\overline{t})$ and Δ^* denials of the form $\forall\ \overline{X}. \leftarrow a(\overline{t}) \wedge Q$ where $a(\overline{t})$ is an abducible.

The only inference rules that interact with Δ and Δ^* are the rule A.1 and A.2 (see Section 3.3.2). Their analysis yields the following observations about the data structure operations.

- insertion
  The inference rules show only addition of a single element to the data structure.
- retrieval
  This is the most important operation. Both inference rules retrieve large amounts of information from Δ and Δ^*.

  A.1 For a new abducible $a(\overline{s})$
  - ∗ get all formulas $\{\forall\ \overline{X}. \leftarrow a(\overline{u}) \wedge Q | \leftarrow a(\overline{u}) \wedge Q \in \Delta^*\}$
  - ∗ get all formulas $\{a(\overline{t}) | (\overline{t}) \in \Delta\}$

A.2 For a formula $\forall\, \overline{X}. \leftarrow a(\overline{u}) \wedge Q$ get all $\{a(\overline{t}) | a(\overline{t}) \in \Delta\}$

Observe that the retrieval of elements out of Δ and Δ^* is completely determined by the (head) abducible of the formula. This similarity allows to define the same data structure for both.

Another observation is that each retrieval of an element is followed by a check if the element is relevant. This is expressed by generated equalities between the selected abducible and the elements of Δ and Δ^*. Hence it suffices to retrieve only the relevant elements from Δ and Δ^*.

- deletion
  Deletion is only needed when an inconsistent state is encountered and the system has to return to the previous state. This operation is automatically supported by the backtracking of Prolog, and hence must not be considered.

Summarizing the analysis: the data structures Δ and Δ^* must support efficient (bulk) retrieval.

The simplest design of Δ (Δ^*) is a flat list. When queried to retrieve some elements, all elements are returned and the inference rule implementation has to take care of the selection of the appropriate elements. That is computationally very costly (see also next Section 3.4.4). And thus, we would like to have a data structure that returns a set which is as small as possible. The following present such a datastructure. To explain it, we introduce the following auxiliary notions:

Definition 3.15 An atom A is *relevant* for another atom B if they are renamings of each other.

Example 3.11 Let $\Delta = \{a(1,x), a(2,y), a(z,4)\}$. The retrieval of the relevant elements of Δ for $a(1,v)$ yields the set $\{a(1,x), a(z,4)\}$. The result for $a(v,1)$ is $\{a(1,x), a(2,y)\}$.

Computing if two atoms are relevant for each other requires the computation of a unifier. Because many of the elements stored in Δ (Δ^*) are not relevant for the query, the accumulated time needed for those unifier computations forms a substantial overhead for the $\mathcal{A}$system. Therefore we define a syntactic approximation of this computation which is used to determine if a (set of) element(s) is relevant for the query.

Definition 3.16 An *abstraction* of an atom is an instance of the schema

$$\begin{array}{l} value - arg(n) - value \\ value - var \end{array}$$

The abstraction starts with the predicate name of the atom. *value* is the name of the predicate symbol, function symbol or constant at that location. $arg(n)$

denotes a selected argument position n which is not a variable. If all arguments are variables then var is used to denote this.

Note that, because the argument position is not fixed, an atom might have more than one abstraction.

EXAMPLE 3.12 The abstraction of some atoms at different depth.

atom	depth 1	depth 2	depth 3
a(x,y)	a	a-var	a-var
a(1,y)	a	a-arg(1)-1	a-arg(1)-1
a(x,2)	a	a-arg(2)-2	a-arg(2)-2
a(f(z),y)	a	a-arg(1)-f	a-arg(1)-f-var
a(f(a),3)	a	a-arg(2)-f	a-arg(2)-f-arg(1)-a
		a-arg(2)-3	a-arg(2)-3

Depth 1 corresponds to the predicates name, depth 2 also considers the argument positions, and depth 3 includes information about the arguments of the arguments.

The abstraction of an atom will be used as the key in our data structure. The current implementation uses as key the left-most abstraction for depth 3. That is sufficient for all applications we have considered.

The data structure for Δ and Δ^* is composed of two alternating data structures. It starts with an AVL-tree which uses the first value part of the abstraction, i.e. the predicate name, as key to separate the elements. The nodes of this tree are pairs $(ArgList, All)$ where All is a list of all element in this node and $ArgList$ a structured argument list. The elements of the argument list are triples (arg, t, all), where arg is a argument position (e.g. `var`, `arg(0)`, `arg(1)`, $\ldots$) and all a list storing all elements that have the corresponding argument position. These elements also are stored in a structured way in the tree t. t is an AVL tree of the same structure where the next value of the abstraction is used as key. In this way the abstraction is step by step traversed, until the last value is reached. At this point the element is stored in the data structure. When the abstraction ends with `var`, then the element is stored in the all list of the triple (var, t, all). Since this is the end of the abstraction, the tree t in the triple is superfluous and can be omitted. In the case that the abstraction ends in a value, the element is stored in the node of an AVL-tree which is just a list of elements.

EXAMPLE 3.13 (EXAMPLE 3.12 CONTINUED) We show the data structure for some of the atoms of example 3.12. A node in an AVL-tree is denoted as $t(k, v)$ where k is the key and v is the value associated with the key k. Entries in the same tree are denoted as $t(k_1, v_1)\&t(k_2, v_2)$. We start with the empty tree t.

the added atom	the state of data structure after addition
a(x,y)	t(a,([(var,t',[a(x,y)])],[a(x,y)]))
a(1,y)	t(a,([(var,t',[a(x,y)]), (arg(1),t"(1,[a(1,y)]),[a(1,y)])]), [arg(x,y),a(1,y)])
a(f(z),2)	t(a,([(var,t',[a(x,y)]), (arg(1),t"(1,[a(1,y)])&t"(f,([(var,t"',[a(f(z),2)])],[a(f(z),2)])), [a(1,y),a(f(z),2)])]), [arg(x,y),a(1,y),a(f(z),2)])

The retrieval of the relevant elements for an atom a is the following procedure. First the abstraction k of a is computed. Then based on this abstraction k, the elements are retrieved as the union of the sets collected as follows:

1. if the first part of k is a value and t is the current tree, then select the corresponding node $(ArgList, All)$ in the AVL tree t
2. Let $(ArgList, All)$ be the current node. If the first part of k is
   - var, then add All and stop
   - $arg(N) - k'$, then add each All_i from $(arg(i), t_i, All_i) \in ArgList$ where $i \neq N$ and set t_N of $(arg(N), t_N, All_N$ as current tree and continue at 1 with k' and t_N

EXAMPLE 3.14 Retrieving all relevant elements for $a(1, b)$ of the last data structure instance of example 3.13 is done as follows: the abstraction of $a(1, b)$ is $a - arg(1) - 1$. According to step 1, the pair

$$\begin{aligned}(Arglist, All) \quad &= ([([(var, t', [a(x, y)]),\\ &(arg(1), t''(1, [a(1, y)])\&\\ &t''(f, ([(var, t''', [a(f(z), 2)])], [a(f(z), 2)])),\\ &[a(1, y), a(f(z), 2)])]), [arg(x, y), a(1, y), a(f(z), 2)])\end{aligned}$$

is selected. Since the next part of the key is $arg(1)$, step 2.b applies. The argument of the first element of $ArgList$ is var and that is different from $arg(1)$, hence $[a(x, y)]$ is one part of the to be constructed list. The argument of the second element of $ArgList$ matches. The relevant elements are found in the tree $t''(1, [a(1, y)])\&t''(f, ([(var, t''', [a(f(z), 2)])], [a(f(z), 2)]))$. We have only the value 1 left of the key, according to step 1 we have to select the list of node for 1 which is $[a(1, y)]$. Since we have evaluated the whole key, all relevant elements are $[a(x, y), a(1, y)]$. Observe that the third element that is stored $a(f(z), 2)$ never can unify with $a(1, b)$, and that it is not in the answer of the retrieval which was the intension of this design.

Cost analysis of the retrieval

To give an idea of the gain this complex data structure offers compared to the simple approach, we present the following cost analysis. This analysis makes some rough approximations, but is sufficiently detailed to get a first sight on the trade-off between both.

- We first consider the retrieval of the relevant elements for a very detailed atom e.g. $a(f(3, g(a)), 6)$.

  For the simple approach the retrieval cost is linear in the size of the elements. Suppose that there are n elements of which $m(m < n)$ are *not* relevant. The cost of verifying if two atoms are not relevant is e. Then there is an overhead of $m \times e$ unification computations for the simple approach.

  To estimate the cost of our data structure we suppose that all the AVL-trees storing values contain f elements and that each argument list contains a elements. ($a - 1$ corresponds to the average number of arguments of a predicate or functions symbol.) Then for a depth d, there are d lookups in an AVL-tree in the worst case. The total cost is then $1+d\times\log(f)+(d-1)\times a$.

  For example, if $f = 10, a = 5, d = 3$, then it costs 14 operations to retrieve the relevant elements for the complex data structure. If the cost of verifying an equality is small, e.g. 2, then the designed data structure is already more efficient with 8 not relevant elements.

- When retrieving the relevant element for a very general atom e.g. $a(x, y)$, the gain is even clearer: $1 + \log(f)$ for the complex data structure compared to the same overhead $m \times e$. In our example the complex data structure needs only 2 operations, meaning that from one not relevant element, it is better than the simple approach.

This indicates that the presented data structure for Δ and Δ^* reduces the retrieval time of the relevant elements. This effect increases if the amount of stored elements increases, or if the request is very general. But the gain in computational time imposes higher memory demands. One way to reduce the space consumption is to store instead of the atoms (or denials) a corresponding unique identifier of the atom (denial). The relation between this identifier and the atom (denial) is stored in another datastructure. In this way the retrieval cost in number of operations increases by $(n - m) \times \log(n)$ in case of an additional AVL tree. For large m, the designed datastructures perform better even with this additional cost. This turned out to happen in practice.

We have chosen not to fix the abstraction computation to a particular argument, e.g. the first argument as in most Prolog systems. That would simplify the data structure and probably yield some (small) performance gain. But, such a fixation often leads to a shift in the problem formulation style. Instead of letting the readability prevail, the user will try to position the arguments of its (abducible) predicates so that the most determining argument is at the indexed position. This threatens the declarativeness of the specification. When possible, we avoid the introduction of such influences in the system.

REMARK 3.4.1 (PRACTICAL NOTE) The concept of a global large data structure is alien to (pure) Prolog. A global data structure is obtained by threading the data structure as an argument through every predicate which is involved in the computation. For small programs this approach works fine, however for large programs as the $\mathcal{A}$system this leads to an overhead. In the $\mathcal{A}$system, approximately 70% of the code which contains the global data structure as argument, does not access that data structure.

By applications/extensions such as CLP, Prolog developers have recognized the need for global data structures, but most of these structures will not keep the references to the original variables, which is required by the $\mathcal{A}$system. (Remember that some variables in the $\mathcal{A}$system are free variables.) For instance, for an atom $a(x, y)$ stored in such a global structure, only an independent variant (a copy) is retrievable, e.g. $a(x_1, y_1)$. The references to the variables x and y are lost.

3.4.4 Meta programming for the inference rules

Because the majority of the applied inference rules are deterministic, is it important to provide a solid evaluation of them. This part of the encoding is responsible for the basic efficiency of the system.

More specifically, the section presents a study of meta-programming for abductive inference. A meta-program is any program that treats another program, called the object-program (and specifically here an $\mathcal{A}$system-specification) as data. Meta-programs are used to build easily interpreters, debuggers, compilers, etc... The advantages of the meta-programming approach are the ease of construction and the flexibility to change the inference structure. The classical example is the *vanilla meta-interpreter* (see Figure 3.2) that encodes SLD(NF) resolution with a left-to-right selection rule and a depth-first search strategy.

```
demo(true).
demo((P,Q)) :- demo(P),demo(Q).
demo(P)     :- atomic(P),clause(P:-Q),demo(Q).
demo(not P) :- not demo(P).
```

Figure 3.2: Vanilla meta-interpreter

The implementation of the inference in the $\mathcal{A}$system is based on this simple meta-program. Unfortunately, this meta-program contains some inadequate implicit assumptions that complicate the implementation of an abductive solver. Let us illustrate these assumptions with the vanilla meta-program.

SLDNF corresponds to the $\mathcal{A}$system inference rules D, E and N (see Section 3.3.2). Comparing those rules with the rules of the vanilla meta-program, one observes that the $\mathcal{A}$system has a separate rule set for failing goals in the form of the evaluation of denials, while SLDNF suffice with an implementation of SLD-resolution and some extra control to deal with a negated literal (the last program rule). The latter observation is supported by the SLDNF survey of [14] which shows that most work on SLDNF (implicitly) focuses on how to transform the negation in such a way that the existing SLD-resolution can be reused. The use of a different rule set for evaluating denials is a first incentive that the design decisions of the SLD(NF)-resolution might not be completely appropriate for the $\mathcal{A}$system.

Another distinguishable property is the quantification of variables. SLD(NF) uses only one quantification: all variables of a clause are universally quantified. This uniformity allows to simplify the procedure by making the quantification implicit, as done in the vanilla meta-program. This is impossible for the $\mathcal{A}$system. A denial contains variables with a different quantification: universally quantified and free variables. Hence we have to provide an efficient way to find out the quantification of the variables.

In Logic Programming, the object programs are commonly interpreted in their *non-ground* representation such as is done by the presented vanilla meta-interpreter [59]. This style is called *non-ground meta-programming*. Typical is that the variables of the object program are represented and interpreted by variables of the meta-program. In turn, this enables the use of the underlying logic programming environment for the maintenance of the bindings of the variables, which simplifies the meta-program considerably. In contrast to this approach stands the *ground* meta-programming style. This style is less popular. According to this approach, the object program is encoded as a ground term by replacing each variable by a unique identifier. Consequently it requires that the meta-program explicitly maintains the bindings of the variables.

This subsection presents two meta-programming encodings for the deterministic inference rules of the $\mathcal{A}$system. We start with the classical non-ground encoding. An analysis of the computational behavior of this encoding will reveal some important inefficiencies. To resolve these, we present a second encoding for the evaluation of denials using a ground meta-programming. To conclude we present the results of a small experiment.

Non-ground representation

The non-ground encoding is a straightforward extension of the vanilla meta-interpreter. Figures 3.3 and 3.4 sketch the general layout of the meta-programs in the $\mathcal{A}$system. The meta-programs contain two auxiliary data structures: the actual

```
eval_pos([],State,Choices).
eval_pos([H|Q],State,Choices):-
   eval_pos(H,State,Choices),
   eval_pos(Q,State,Choices).

eval_pos(definition(H),State,Choices):-
   inference_rule_encoding_D_1.
...
```

Figure 3.3: The meta-program of the positive inference

```
eval_neg([H|Q],State,Choices):-
   eval_neg(H,Q,State,Choices).

eval_neg(definition(H),Q,State,Choices):-
   inference_rule_encoding_D_2(H,Q,NewQs),
   eval_neg_all(NewQs,State,Choices).
...

eval_neg_all([],State,Choices).
eval_neg_all([IC|ICs],State,Choices):-
   eval_neg(IC,State,Choices),
   eval_neg_all(ICs,State,Choices).
```

Figure 3.4: The meta-program of denial evaluation

store of an $\mathcal{A}$system state (represented by the variable `State`) and the set of choice points (`Choices`). Depending on the considered inference rule, the state store is consulted and/or updated (for deterministic rules) or the set of choices is updated (in the case of a non-deterministic inference rule). For the correct implementation of the suspended goals, the data structures `State` and `Choices` are implemented by means of *mutable terms*. Mutable terms can be understood as pointers to a data structure. They implement a backtrackable destructive assignment: new values can be assigned, discarding the old value (i.e. by a pointer redirection); on backtracking the old value is restored. By this implementation's choice return values e.g. `NewState` are not needed. Each meta-program consists of a part that traverses the object program rules and a part that encodes the individual inference rule. Remark the difference between the first part of both meta-programs. In the case of `eval_pos/2` the empty list is equivalent with **true**; while this for `eval_neg/2` is **false** (expressed by the absence of this basic rule). Note that both meta-programs are connected via the negation (inference rules N) and the consistency derivation of abducibles in A.1.

This meta-program encodes the deterministic extension of an $\mathcal{A}$system state. No choice points are expected and thus, the meta-program must be deterministic as well.[7] This information can be exploited to optimize the meta-program so that Prolog will not construct any choice point when executing the meta-program. One optimization is given by the principal functor indexation.

Principal functor indexation refers to a specific compiler optimization. It constructs at the level of the WAM (the virtual machine of Prolog) a switch on the first argument of a predicate. This switch directs calls to the predicate immediate to the right clauses. In this way the evaluation of irrelevant clauses can be avoided. In the best case when all functors of the first argument are different from each other, no backtrack point is created in the execution of program.

In the meta-program this principal functor indexation is obtained by adding an annotation to each literal of the object program. For example, an atom $p(\overline{t})$ of a defined predicate in the body of a rule is annotated as $definition(p(\overline{t}))$. If it is an abducible atom, the annotation is $abducible(p(\overline{t}))$, etc. In the meta-program, the annotation forms the principal functor. Hence, by assigning a unique annotation to each literal, the principal functor indexation makes the meta-program deterministic.

The computational cost of the denial evaluation

Sources of inefficiencies The non-ground meta-interpreter has the notable advantage that the clauses of the object-program are usable both for the positive and negative part of the meta-program. That allows to keep the preprocessing

[7]This behavior can be verified by special analysis tools. In the language *Mercury*, this information must be declared for each predicate. The compiler will verify this statement and reports the user if it finds an inconsistency.

rather simple. But as earlier mentioned, the evaluation of the denials is a complex issue. In particular the unfolding of the definitions and the consistency checking of abducibles (which can be seen as a form of unfolding with respect to a dynamic definition) has to be implemented with care.

Let us illustrate the difficulties by an example. Consider the denial $\forall\ X. \leftarrow p(X,Y) \wedge Q$ where p is a defined predicate. According to the inference rule D.2. for each clause of the definition of p a new denial is generated. Let p have two clauses $p(V,W) \leftarrow V = 1 \wedge W = 1.$ and $p(V,W) \leftarrow V = 2 \wedge W = 2.$ Then the inference rule generates two new denials

$$\forall\ X,V,W. \leftarrow X = V \wedge Y = W \wedge V = 1 \wedge W = 1 \wedge Q$$
$$\forall\ X,V,W. \leftarrow X = V \wedge Y = W \wedge V = 2 \wedge W = 2 \wedge Q$$

Logically, the fact that the denials share the same variables is not problematic. Their scope is correctly limited by the universal quantification to one single denial. In a Prolog system this sharing is troublesome. Prolog knows only one implicit quantification, having the effect that variables sharing the same name cannot be independent. Because the meta-program uses its variables (thus Prolog's) to represent the variables of the object program (the $\mathcal{A}$system specification), it introduces an incorrect connection between logical expressions at the object level. A simple solution is the addition of a renaming for all *universally quantified* variables when generating the new denials from one denial. Note that free variables should not be renamed, because they are allowed to be shared by multiple denials.

The renaming of the universally quantified variables leads to these (logically equivalent) denials:

$$\forall\ X1,V1,W1. \leftarrow X1 = V1 \wedge Y = W1 \wedge V1 = 1 \wedge W1 = 1 \wedge Q'$$
$$\forall\ X2,V2,W2. \leftarrow X2 = V2 \wedge Y = W2 \wedge V2 = 2 \wedge W2 = 2 \wedge Q''$$

The renaming also has to be applied on the tail of the original denial Q, otherwise the logical link between both is broken. One source of the computational inefficiency of the non-ground approach is this renaming process. This is discussed in more detail below.

The second source of inefficiency is the generation of a huge amount of simple equations e.g. $X1 = V1$, $Y = W2$, $\ldots$, between a universally quantified variable and another term. According to inference rule E.2.c, this equality always succeeds. Because there is no syntactic method to distinguish between an equality of two free variables and an equality in which a universally quantified variable is involved, the evaluation of these simple equations requires the analysis of each term and that is rather costly. This analysis is a small overhead compared to a standard Prolog unification for one equality, but due to the huge amount of equations the accumulated application forms a substantial overhead in an $\mathcal{A}$system derivation. Hence when possible, the unfolding should avoid the generation of new universally quantified variables. For example, the ideal unfolding of the above denial would be

$$\forall\, X1. \leftarrow X1 = 1 \wedge Y = 1 \wedge Q'$$
$$\forall\, X2. \leftarrow X2 = 2 \wedge Y = 2 \wedge Q''$$

In practice, the generation of 'superfluous' universally quantified variables cannot be avoided.

Unfolding in a denial The unfolding of an defined atom in a denial $\forall\, \overline{X} \leftarrow H \wedge Q$ is encoded as the following program. (This is the implementation of inference rule D.2, see Section 3.3.2.)

```
eval_neg(defintion(H),Q,State,Choices):-
   get_skolems((H,Q),Skolems),           % get all free variables
   number_clauses_comp(H,N),             % get the number of clauses
   select_each_body(N,H,Skolems,Q,List), % all denials after unfolding
   eval_neg_all(List,State,Choices).     % continue evaluating

select_each_body(0,_,_,_,[]).
select_each_body(N,H,Skolems,Q,[IC|List]):-
   N>0,
   copy_term(Skolems,(H,Q),(HFree,QFree)), % copy denial
   unfold(HFree,N,QFree,IC)             % IC=[HFree=H_spec|Clause|QFree]
   N1 is N-1,
   select_each_body(N1,H,Skolems,Q,List).
```

It first selects all free variables of the denial. Then it constructs a list of all unfoldings of the denial. In that process, each time the denial is renamed by a copying process (`copy_term/3`) and this renamed denial is used for the unfolding w.r.t. the Nth clause.

The construction of a renamed term Most Prolog implementations use a recursive function to copy a term. This `copy_term(Term,Copy)` function constructs a copy by traversing the original term; the process copies the ground elements and replaces each variable with a new one. An optimization, used e.g. in Sicstus Prolog, is not copying ground sub-terms. These will be shared by the original term and its new variant. The complexity of this algorithm is in the worst case $O(n)$ where n is the number of cells that the term consumes on the heap of Prolog.

For our application, a variant of this function is needed. Namely it should not replace the free variables by a fresh variable. One option is to extend the standard algorithm by adding an extra check when processing a variable. A simpler solution is mimicking this behavior by first creating a fresh copy and afterwards unifying the free variables with their counterparts in the copy.

PROGRAM 3.4.1

```
copy_term(Free,OriginalTerm,Copy):-
   copy_term((Free,OriginalTerm),(Free,Copy)).
```

The complexity of the new algorithm is slightly larger, namely $O(n + k)$ where k is the number of free variables in the term.

REMARK 3.4.2 (A PRACTICAL NOTE) The Sicstus Prolog `copy_term/2` implementation is not used in the implementation of the $\mathcal{A}$system. This procedure is unsound and unreliable when the original term contains finite domain constraint variables (which is the case for the $\mathcal{A}$system). In this case Sicstus Prolog requires to use a special variant called `fd_copy_term/3` that returns, next to a fresh copy, a copy of the constraint store connected with the finite domain variables. This information is useless for our purposes and its construction leads to an extra overhead. It introduces namely multiple copies of the same finite domain constraint, increasing the size of the finite domain constraint store without any search benefit. Therefore we have implemented an $\mathcal{A}$system-suited `copy_term/3` procedure ourselves, that directly deals with the free variables. Of-course, this is at the cost of computation time.

The complete cost of the evaluation of a denial

The above insights allow to estimate the total cost of the deterministic evaluation of a denial. Consider the denial $\forall\, \overline{X}.\leftarrow H \wedge Q$. Suppose that after u unfolding steps the first part H is completely unfolded and resolved so that m choice points of the form $\forall\, \overline{Y}.\leftarrow Q'$ are obtained. The formula Q' in each generated choice point is an instance of the formula Q in the original denial. The total estimated cost of the deterministic unfolding of the denial is $O((u \times m) \times (n + k))$ where n is the heap size of Q and k the number of free variables in Q. Let's also suppose that each unfolding requires the evaluation of e equations. Then, there are $O(u \times e)$ equations solved in this deterministic unfolding process.

Although the estimate is linear in the number of unfoldings, their effect is measurable. We profiled the execution of the $\mathcal{A}$system for some problems. This showed that the predicates involved in the copying and the evaluation of the simple equality evaluation, belong to the most frequently executed predicates.

Recall that the reason for copying is the occurrence of universally quantified variables *and* free variables in denials. If every variable in a denial would be free, then no copying is needed. Thus, an increased $\mathcal{A}$system's efficiency is already obtained with a denial representation that only uses Prolog variables to represent the free ($\mathcal{A}$system) variables. A method to achieve this, is to build a ground meta-program for denials.

Ground meta-programming

Within Logic Programming the use of a ground representation has been advocated by the designers of the logic programming language Gödel [143, 142]. To support meta-programming, Prolog is augmented with non-logical features such as `assert/1`,`retract/1`, `var/1`, `=..`, `arg/3`, etc. Gödel gives these a logical counterpart and therefore the ground representation for meta-programming is adopted. In this way meta-programming is more declarative in Gödel than in Prolog.

In [53], Bowers mentions three reasons why the ground representation appears to be a bad choice and why it is thus rarely used.

- The representation of object programs as terms is too complex.
- Meta-programming using the ground representation is too laborious. Simple object-operations like unification require large procedures.
- The assignment of the variables must be maintained explicitly. This decreases the performance of the meta-program (to a unacceptable level).

In our opinion, the lack of interest for this approach might also be the lack of applications for which a ground representation is a real benefit. Our problem, a (semi-)ground meta-program for the evaluation of denials might be such an application. Despite of the mentioned drawbacks, the authors of Gödel stress that a ground representation when provided with the right components is well-suited to build meta-programs and not more complex than using the non-ground representation.

With regard to our problem, the above drawbacks have to be compared to the overhead introduced in the non-ground meta-programming. For example, the equality reasoning in a denial is already more complex than the classical unification, and hence the ground representation may add no additional overhead. The most important reason to explore this encoding is the possibility to avoid the copying which is needed in the non-ground encoding.

The ground representation of an $\mathcal{A}$system program rule For a ground representation for $\mathcal{A}$system program rules, it suffices to define a ground representation for the variables together with a maintenance procedure for the bindings. We have chosen to represent each variable of a clause by a unique numerical index. This index refers to a location in an auxiliary data structure $\mathcal{B}$ that stores the actual value of the variable. In our implementation, we have chosen an AVL-tree for $\mathcal{B}$.

The keys of the elements $\mathcal{B}$ are the indexes, while the actual value is one of the following terms:

- `undef` : a location with no value.
  This variable is universally quantified.

- `link(Var)` : The value of this variable is stored at the cell of Var. Universally quantified variables might get aliased[8] during the evaluation. Aliasing happens when an equality between two universally quantified variables is evaluated. The link is placed from the variable with the highest index to the one with the lowest. In this way the value is always accessible for later use, because a lower index refers to a variable that is created earlier and hence it might be used by other literals that are not yet evaluated.

- `val(Value)`: An actual value of the cell.
  `Value` is a term which contains no universally quantified variables.

- `term(Functor,Args)`: this is used to represent a term that might contain universally quantified variables.

  `Functor` is the function symbol of the term and `Args` is a list of indexes of the argument values.

Observe that $\mathcal{B}$ only stores information that is not subject to renaming, which is intentional for our goal.

EXAMPLE 3.15 Consider the following actual state of a binding $\mathcal{B}$:

$$\mathcal{B} \mapsto [1 - undef, 2 - val(X), 3 - term(father, [2, 1]), 4 - link(1)]$$

(The AVL-tree is for presentational purposes replaced by a list.) $\mathcal{B}$ stores four variables having indexes 1,2,3 and 4. 1 is a universally quantified variable since its value is $undef$. The second variable 2 is the free variable X. The third represents the term $\forall\, Y.father(X, Y)$, where X is the free variable to which index 2 refers and Y is a name for the universally quantified variable referred to by index 1. The last variable is an alias of the first variable.

To limit the space consumption, the cells of universally quantified variables are left undefined, instead of storing the value $undef$. When the retrieval for the value of a variable from $\mathcal{B}$ fails, the value `undef` is returned. The following program returns the actual value of an index stored in $\mathcal{B}$, where universal variables (also within compound terms) are denoted by `undef`.

```
get_value(Ref,Value,Bindings):-
   (get_assoc(Ref,Bindings,Val)
   -> (nonvar(Val),Val = link(Var))
        -> get_value(Var,Value,Bindings)
   ;  (nonvar(Val),Val = term(F,Args))
        -> get_values(Args,Values,Bindings),
```

[8]When two variables are unified with each other, they are *aliased*. Both variables represent the same unknown value, but use a different name.

```
            Val =..[F|Values]
        ;   Value = Val
        )
    ;   Value = undef
    ).
```

Let $\mathcal{P}$ be an abductive normal logic program. The set $Def_p \subseteq \mathcal{P}$ consists of the program rules that have the predicate p as head: $Def_p = \{p(\overline{t}) \leftarrow B_i \,|\, i = 1..n\}$ Then the following transformation computes the ground representation for all program rules of the predicate p.

1. For each rule $p(t_1, \ldots, t_n) \leftarrow B \in Def_p$:
   (a) Normalization of the head: $p(x_1, \ldots, x_n) \leftarrow y_1 = t_1, \ldots, y_n = t_n, x_1 = y_1, \ldots, x_n = y_n, B_1$ where x_i and y_i are fresh variables. The x_i variables represent the values of the arguments when an actual call to this predicate is made in some body. (Recall that the evaluation by the $\mathcal{A}$system is top-down.) The y_i are introduced to eliminate any non-variable term in the head of the rule.
   (b) Normalization of the body literals of B:
      i. a positive literal $p(s_1, \ldots, s_n)$:
      $$y_1 = s_1, \ldots, y_n = s_n, p(y_1, \ldots, y_n)$$
      ii. a negative literal $\neg p(s_1, \ldots, s_n)$:
      $$y_1 = s_1, \ldots, y_n = s_n, \neg p(y_1, \ldots, y_n)$$
   (c) Transformation of the body to a ground list.
      i. Since all introduced auxiliary variables y_i are always universally quantified, each equality $y_i = t$ is transformed to `setval(y_i,t)`. The solver will interpret this as: insert the value t in the bindings $\mathcal{B}$ at the location of index y_i.
      ii. The actual index of the local variables y_i is dependent on the actual state of bindings structure $\mathcal{B}$. Hence their index must be computed during the execution. To do this the following statements are added at the start of the transformed body:
      `offset(`k`,Offset),assign(`y_1`,1,Offset),...,assign(`y_k`,`k`,Offset)`
      where k is the number of introduced auxiliary variables.
      The auxiliary statements are interpreted by the solver as:
      - `offset(Count,Offset)`: `Offset` is the current number of cells in $\mathcal{B}$ and this number must be increased by `Count`.
      - assign(Y,Ref,Offset): Y is Ref + Offset.

The resulting ground representation T for the program rule $p(t_1, \ldots, t_n) \leftarrow q(s_1, \ldots, s_m) \wedge B$ is represented as a Prolog list

$$\begin{aligned}\mathtt{T} = [&\mathtt{offset(n,0)}, \mathtt{assign(Y_1,1,0)}, \ldots, \mathtt{assign(Y_k,n,0)},\\ &\mathtt{setval(Y_1,t_1)}, \ldots, \mathtt{setval(Y_n,t_n)},\\ &\mathtt{setval(Y_{n+1},s_1)}, \ldots, \mathtt{setval(Y_{n+m},s_m)},\\ &\mathtt{q(Y_{n+1}, \ldots, Y_{n+m})}\\ &\ldots]\\ \mathtt{Args} = [&\mathtt{Y_1}, \ldots, \mathtt{Y_n}]\end{aligned}$$

k is the number of auxiliary variables that are introduced.

2. Since it is our goal to simplify the unfolding during the evaluation of the denials, all transformed rules of Def_p are connected together as follows.

   - Let the variable `Q` denote the tail Q of the select denial $\forall \leftarrow p(\overline{t}) \wedge Q$ in an $\mathcal{A}$system derivation. `Q` is added at the end of every transformed program rule as follows: $\mathtt{T(Q)} = [\mathtt{Body|Q}]$. (`Body` represents the transformed body.) The value of `Q` must thus be a list.
   - Let $T_1, \ldots, T_n$ be the transformed bodies for each of the clauses of the predicate p. The ground representation of the definition is

     $$\mathtt{p(Args, Q, [T_1(Q), \ldots, T_n(Q)])}$$

     where *Args* is a list of variables representing the arguments of p. These variables are shared by each transformed body $T_i(Q)$. Observe also that each transformed body has the same tail Q.

The transformation can be optimized so that the number of auxiliary variables y_i is reduced. This is done by iteratively evaluating and removing equalities between universally quantified variables.

Example 3.16 Consider the following program rules of the Event Calculus.

$holds\_at(P,T) \leftarrow initially(P) \wedge \neg clipped(0,P,T).$
$holds\_at(P,T) \leftarrow initiates(P,A,E) \wedge act(A,E) \wedge E < T \wedge \neg clipped(E,P,T).$

Its transformed definition (after the removal of some auxialiary variables) is

```
holds_at([A1,A2],Tail,[
   [offset(1,0),assign(Y1,1,0),def(initially,[A1]),
    setval(Y1,0),not(def(clipped,[Y1,A1,A2]))|Tail],
   [offset(2,0),assign(Y1,1,0),assign(Y2,2,0),
    def(initiates,[A1,Y1,Y2]),abd(act,[Y1,Y2]),
    clp(<,[Y2,A2]),not(def(clipped,[Y2,A1,A2]))|Tail]
   ]).
```

The annotations $\mathtt{def}(\cdot)$, $\mathtt{abd}(\cdot)$, ... allow to exploit the principal functor indexation of Sicstus Prolog.

The interpretation of the ground denials The interpreter of the denials has a similar structure as the non-ground version. Likewise we need iterators that evaluate the denial from left to right.

```
eval_neg([E|Es],Bindings,State):-
   eval_neg(E,Es,Bindings,State).

eval_neg_all([],Bindings,State).
eval_neg_all([IC|ICs],Bindings,State):-
   eval_neg(IC,Bindings,State),
   eval_neg_all(ICs,Bindings,State).
```

The following calls perform the update of the bindings structure.

```
eval_neg(setval(VarRef,Value),Es,Bindings,State):-
   setval(VarRef,Value,Bindings,NewBindings),
   eval_neg(Es,NewBindings,State).
eval_neg(offset(Count,Offset),Es,Bindings,State):-
   offset(Count,Offset,Bindings,NewBindings),
   eval_neg(Es,NewBindings,State).
```

The calculation of the reference for a variable.

```
eval_neg(assign(Ref,Offset,N),Es,Bindings,State):-
   assign(Ref,Offset,N),
   eval_neg(Es,Bindings,State).
```

Compared to the non-ground version, the evaluation of the defined predicates and abducibles is considerably simplified.

```
eval_neg(def(H,Args),Es,Bindings,State):-
   P =.. [H,Args,Es,ICs],  % compute the compiled goal
   call(P),                % execute the goal
   eval_neg_all(ICs,Bindings,State).

eval_neg(abd(H,Args),Es,Bindings,State):-
   abd_inference(H,Args,Es,Bindings,State,ICs),
   eval_neg_all(ICs,Bindings,State).

abd_inference(H,Args,Q,Bindings,State,ICs):-
   % construct the atom
   construct_atom_univ(H,Args,Bindings,Atom,_Univ),
   % store the denial
   put(ic,(Atom,(H,Args,Q,Bindings)),State),
   % retrieve all abducibles
   get(nondest,abd,Atom,HList,State),
```

```
   % remove redundant information
   structure_to_plainlist(HList,Abducibles),
   abd_construct_ics(Abducibles,N,Args,Q,ICs).        % unfold

abd_construct_ics([],_N,_Args,_,[]).
abd_construct_ics([A|As],N,Args,Q,[IC|ICs]):-
   abd_construct_ic(A,N,Args,Q,IC),
   abd_construct_ics(As,N,Args,Q,ICs).

abd_construct_ic(A,N,Args,Q,IC):-
   % derived at compile time from the specification
   abducible_equality_comp(A,Args,Q,IC).
```

The code for other types of literals is similar.

Comparison and experimental evaluation The motivation for this ground implementation was the complexity of the code for evaluating denials using non-ground meta-interpretation. The non-ground implementation requires copying which has to take into account the quantification of the variables. The derivation of this quantification information is the source of yet another overhead since it must be done for each copying request. A connected problem is the generation of superfluous equalities, i.e. an equality between a universal variable and a term. These equalities always succeed, however their evaluation as normal equalities reauires to check the quantification of the involved variables. This consumes extra computation time.

These three problems disappear or happen less in the ground implementation. Since no copying is needed, the derivation of quantification information is only needed when equalities must be solved or when atoms are reconstructed. As a side effect, since only free variables are stored in the bindings, the attribute which is used to distinguish between free variables and universally quantified variables in the non-ground version, can be removed. These simplifications give rise to the expectation that there will be a speed up when using the ground implementation.

The next tables show the constraint setup times, i.e. the time to reach the final node, for two experiments, the N-queens problem and the blocks world AI planning problem. Both problems generate a finite domain constraint store which is searched for a solution at the end of a derivation. Since the latter process is not relevant here, the corresponding timings are not included in the tables. The experiments has been performed using the same parameter values for each implementation, ensuring that the differences in timings are entirely due to the differences in meta-programming approach (plus some necessary changes in the data structures.)

- The N-queens problem:

size	ground	non-ground
10	20ms	20ms
20	70ms	110ms
30	90ms	200ms
40	330ms	350ms
50	510ms	510ms
60	810ms	770ms
70	1s 70ms	960ms
80	1s 370ms	1s 280ms
90	1s 710ms	1s 640ms
100	2s 240ms	2s 30ms

  The size is the number of queens.

- The blocks world problem:

size	ground	non-ground
5	100 ms	170 ms
10	1s 570 ms	1s 310 ms
17	700 ms	2s 990 ms
30	3s 520 ms	34s 880 ms
50	11s 920 ms	2m 8s 960 ms

  The size is the number of blocks that have to be moved.

In the first experiment, the ground version is a bit slower than the non-ground. On the contrary, the second experiment shows an impressive improvement of the ground version w.r.t. to non-ground. These results indicate that more than one factor is involved.

The first experiment shows that the manipulation of the bindings structure is a costly operation. In particular, the ground version has to access this structure in order to construct atoms (abducibles and finite domain constraints). According to the table, this takes as much as or even more time than copying.

The second experiment shows that the ground meta-interpreter pays off. In our opinion, it is due to a combination of the following factors. The blocks world problem requires more unfolding than the N-queens, hence much more copying happens in the non-ground version. Also by some necessary changes, the cost to retrieve an integrity constraint from Δ^* is cheaper in the ground version. Since this happens a lot in the blocks world experiment, this contributes much to the reduction of the computation time. A final factor is that the non-ground version must decompose a lot of atoms and terms during its execution. In the ground version, all terms and atoms are already decomposed (once by the preprocessor) and hence the ground version avoids this process. This corresponds to the results of the N-queens experiment. Because in the ground version all terms are decomposed, and for some inference steps, e.g. finite domain constraints, the atoms (terms) are needed they must be reconstructed from the bindings. For example the following arithmetic expression $x + (z \times 3) \leq 4$ will be decomposed to these equalities $y_1 = x, y_2 = 3, y_3 = z, y_4 = \times(y_3, y_2), y_5 = +(y_1, y_4), y_6 = 4, \leq (y_5, y_6)$ by the transformation to a ground representation. The decomposition consumes clearly more space than the non-ground version, both in the size of the clause and in the binding maintenance data structure. When evaluating the ground representation, none of these equalities will fail so that step by step the arithmetic expression is reconstructed. This consumes clearly more time than in the none-ground version which immediately has the arithmetic atom available. To make the ground evaluation more efficient, one could design for each constraint a special ground version so that this decomposition and composition of the terms is avoided. This is left for future work. The combination of these factors explain the impressive improvement in the blocks world problem.

We conclude this comparison with a short note on the differences in space requirements. The ground version consumes more space than the non-ground version for storing denials in $\forall\ \overline{X}. \leftarrow a(\bar{t}) \wedge Q$ in Δ^*, mainly due to (1) longer clause representations and (2) the auxiliary data structures for maintaining the bindings of each denial.

From meta-interpretation to compilation

A meta-program is a flexible way to encode new inference patterns, but at the cost of a lower performance. When speed matters, the meta-program has to be replaced by a compilation step. This might result in a performance improvement by a factor of 3 or more for this part of the $\mathcal{A}$system.

Compilation translates a program (rule) into an equivalent procedure in a lower level language. Here, it would be the translation of a definition (a set of abductive normal program rules with the same head predicate) and the corresponding $\mathcal{A}$system execution to a specific Prolog program. Because of the two modes of reasoning, two programs will be constructed during compilation. For the denial evaluation, we have to choose on which meta-programming version the compilation will be based. In our opinion, the ground version is more suited for compilation due to its simpler design. Moreover, since this implementation resembles the operations of the WAM (Warren Abstract Machine [7, 271]), it might be possible to extend the WAM with denial specific operations. Especially the support for the maintenance of the bindings will have a large impact to reduce the computation time aswell to reduce the space consumption since the maintenance is one of the main bottlenecks according to our measurements.

3.4.5 Conclusions and future work

This section has presented the main implementation issues of the $\mathcal{A}$system. To our knowledge, this is the first survey and study of all those aspects for an abductive first order top-down reasoner. In short, we have presented the following improvements to the general implementation structure of the $\mathcal{A}$system.

- We have organized the search so that information which can be derived deterministically, is immediately added to the state of the system.
- We have proposed and successfully implemented forward propagation for each choice point such that more deterministic information can be derived.
- We have recognized that this forward propagation is facilitated by the general methodology of reified expressions.
- We have designed a special purpose data structure for Δ and Δ^* such that the retrieval cost of abducibles and denials is reduced.
- We have studied the implementation of the inference rules. In particular, we have discussed and compared two implementations for the evaluation of denials.

All these elements together make the $\mathcal{A}$system more robust than its ancestors, allowing it to solve larger problem instances. For an experimental evaluation of the $\mathcal{A}$system, see Section 3.9.

Note that the issues discussed here only concern the general structure and reasoning performed by the $\mathcal{A}$system. A large part of the reasoning is performed by external solvers, i.e. by the equality solver and the finite domain constraint solver. Even if each subsolver is very efficient on its own, the integration and collaboration in one system generates new difficulties and problems that reduce the performance of the system. The following sections will discuss these issues.

This basic framework of the $\mathcal{A}$system can be subject of various improvements and extensions. In short, the main themes are the design of more intelligent search heuristics that take more domain knowledge into account, the continuation of the work towards low-level support for the evaluation of denials and the improvement of the forward propagation reasoning. These topics are discussed in more detail in the section on future work (Section 3.10).

3.5 The equality reasoning solver $\mathcal{E}$

The reasoning which is most applied in the $\mathcal{A}$system is equality reasoning. Except for the finite domain inference rules (see Section 3.3.2), all rules produce or resolve equalities. Hence, to construct an efficient system, the equality reasoning must be extremely efficient.

The first design of the equality reasoning was an extension of standard unification [90]. As it turned out, equality reasoning only based on unification (i.e. equality reasoning that only considers equality) is unsatisfactory for the $\mathcal{A}$system. It is too limited, at the level of expressiveness as well as at the level of computational efficiency. Therefore, a new equality solver that allows a more expressive language (more in particular, disequality is in the language), has been implemented: the (dis)equality constraint solver $\mathcal{E}$. The use of the constraint solver $\mathcal{E}$ improves the $\mathcal{A}$system's efficiency considerably in comparison to $\mathcal{A}$system versions using unification based equality solvers.

The solver $\mathcal{E}$ can also be used independently from the $\mathcal{A}$system as an equality solver for (Sicstus) Prolog. Especially, we see potential use as a subsolver in an implementation of constructive negation in Prolog. That is not surprising, since negation and equality reasoning (in particular, when allowing disequalities) are closely related. This relation is explored in the related work discussion.

We start this section with an overview of the earlier approaches and based on this overview we motivate the need for a new equality solver. Next the theoretical foundations, adopted from the survey of Comon and Lescanne [83], are presented. Based on these notions the actual implementation is presented. The section concludes with the illustration of the effects of the new $\mathcal{E}$-solver on the $\mathcal{A}$system and the positioning of our work in relation to others.

3.5.1 The earlier approaches

In the original version of SLDNFA [90] the equality relation was treated by an extended version of unification. In the positive context, the standard unification algorithm [228, 192] was applied, which efficiently applies the Clark free equality axioms ($\mathcal{FEQ}$). It is shown by Figure 3.5. In these rules, the following notational conventions are used: the symbols c and d represent different constants, f and g different functors, X and Y variables, s and t terms, and $\overline{s}$ and $\overline{t}$ tuple of terms. $\overline{s}=\overline{t}$ is the shorthand for $s_1 = t_1 \wedge \ldots \wedge s_n = t_n$. $vars(t)$ denotes the set of variables occurring in the term t. We make abstraction of the interaction with the $\mathcal{A}$system by the use of the next notations. $\perp$ means failure and will invoke backtracking; $\top$ means true and does not need further evaluation. σ denotes a global substitution, i.e. it must be applied on the complete $\mathcal{A}$system state such that every occurrence of the variables in its domain is replaced by the corresponding term given by σ.

For the negative context (equality reasoning in denials), the algorithm has been extended to deal correctly with the free variables and universally quantified

- $c = d \mapsto \perp$
- $c = c \mapsto \top$
- $f(\overline{s}) = f(\overline{t}) \mapsto \overline{s} = \overline{t}$
- $f(\overline{s}) = g(\overline{t}) \mapsto \perp$
- $X = t \mapsto \perp$, when $X \in vars(t)$ and t is not a variable (occur check)
- $X = t \mapsto \sigma(X/t)$ when $X \notin vars(t)$

Figure 3.5: The standard equality inference rules.

- $\forall\, \overline{X}. \leftarrow c = c \wedge Q \mapsto \forall\, \overline{X}. \leftarrow Q$
- $\forall\, \overline{X}. \leftarrow c = d \wedge Q \mapsto \top$
- $\forall\, \overline{X}. \leftarrow f(\overline{s}) = f(\overline{t}) \wedge Q \mapsto \forall\, \overline{X}. \leftarrow \overline{s} = \overline{t} \wedge Q$
- $\forall\, \overline{X}. \leftarrow f(\overline{s}) = g(\overline{t}) \wedge Q \mapsto \top$
- $\forall\, \overline{X}, X. \leftarrow X = t \wedge Q \mapsto \top$ when $X \in vars(t)$
- $\forall\, \overline{X}, X. \leftarrow X = t \wedge Q \mapsto \forall\, \overline{X} \leftarrow Q(X/t)$ provided that $X \notin vars(t)$

Figure 3.6: The equality inference rules applied in denials.

variables. The extension is described by the rules in Figure 3.6. Given a denial $\forall\, \overline{X}. \leftarrow s = t \wedge Q$, the rules of Figure 3.6 reduce it to an *equality constraint* which is a denial of the form $\forall\, \overline{X}. \leftarrow Y = t \wedge Q$ where Y is a free variable ($Y \notin \overline{X}$) and t a term different from a universally quantified variable. In the case that Q is empty, the equality constraint is $\forall\, \overline{X}. \leftarrow Y = t$ which is equivalent with $\forall\, \overline{X}.Y \neq t$.

As explained in the previous section the equality constraints are choice points. In the current implementation they are handled by reification. That was not the case from the beginning.

The first implementation treated these residual equality constraints as passive elements and stored them as elements of a list. When a unification had happened, the list of equality constraints was searched for implied equalities. For each implied equality, the corresponding rule was fired. This process was early identified as one of the main reasons for the poor performance of the original SLDNFA implementation.

The second implementation removed partially the passiveness of these constraints. It used the coroutining facilities of Prolog (the when/2 built-in) for the implementation. Coroutining allows to suspend the execution of a Prolog-goal until some conditions on a set of variables are fulfilled.

```
post_pone_equ(X,Y,Q):-
   when((ground(X),ground(Y)),
      (X=Y -> evaluate(Q); true)).
```

Thus, when a unification happens, immediately the relevant equality constraints are awakened and the corresponding action is undertaken. This implementation has improved the efficiency of the $\mathcal{A}$system for the first time to an acceptable level. But it has several drawbacks:

- The disequality information is not propagated. The existence of the suspended goal $\leftarrow X = Y$ does not have the effect that another suspended goal $\forall\, \overline{Z}. \leftarrow X = Y \wedge Q$ is triggered: $\leftarrow X = Y$ means that $X \neq Y$ and thus it follows that $\forall\, \overline{Z}. \leftarrow X = Y \wedge Q$ is $\forall\, \overline{Z}. \leftarrow Q$. Clearly this is an intended behavior because it avoids the construction of a choice point and it directs the search to the important choices. This propagation mechanism improves the $\mathcal{A}$system's efficiency.

- Each occurrence of a disequality ($\leftarrow X = Y$) introduces a new, but already existing, suspended goal in the (Prolog) memory. But this only consumes extra memory without any benefit. As unwanted side-effect, it decreases the efficiency since every suspended goal is evaluated. When there are n suspended redundant goals, the same equality is evaluated n times.

- The high complexity of the code makes it difficult to maintain. Errors that showed up, were very difficult to trace back because Sicstus Prolog has very limited capabilities to trace through a system that uses coroutining.

The first drawback turns out to be critical for the $\mathcal{A}$system. Experiments with relative small instances of the *logistics* planning problem were unsuccessful, even after fine-tuning the search strategy. The $\mathcal{A}$system performed (infinite) long derivations in which many failing branches were explored. The examination of the execution traces showed that one of the main reasons was the absence of propagation of the disequality information.

Improving the propagation of the disequality information is, however, more complex than just providing a new implementation. The above approaches consider only the equality relation explicit. Disequality is implicitly available via the residual equality constraints. Another limitation of the above implementations is that they consider single equalities. More complex formulas that combine equalities and disequalities in disjunctions and conjunctions are not handled. These combinations show up when for the evaluation of denials more than one literal in one inference step are evaluated. E.g. the denial $\leftarrow X = a \wedge \neg(Y = b)$ is evaluated according the $\mathcal{A}$system inference rule in two steps: first a choice is made on the left most literal $X = a$, only when $X = a$ holds (or must hold) the following literal

$\neg(Y = b)$ is considered. Taking both into account, the disjunction $X \neq a \vee Y = b$ has to be evaluated. Thus, when the $\mathcal{E}$-solver can reason with disjunctions then it is possible to remove choice points from the $\mathcal{A}$system inference level to the more efficient constraint level.

3.5.2 Theoretical foundations

The above considerations directed us to search for new foundations to build our equality reasoning on. Our theoretical foundations are based on the excellent survey work of Comon and Lescanne [83]. They present equality reasoning for equational problems in the context of generalized term algebra's (over finite and rational trees). In particular, they prove for sets of transformation rules soundness, completeness and termination with regard to the type of algebra and the type of solved form. From this extensive study, we have selected the most appropriate inference rules (sufficient to be complete) and solved form to base our implementation on.

Since our goal is to build an equality constraint solver for the $\mathcal{A}$system, the presented notions in [83] must be adapted such that they suit better our goals. Therefore, the resulting $\mathcal{E}$-solver is not an exact instantiation of the equality reasoners discussed by [83]. The main differences are the addition of reified expressions and the used solved form which is a less detailed form than the ones Comon and Lescanne discuss.

We first summarize the work of Comon and Lescanne that is relevant for us. Then based on the introduced notions the design of the logical inference in the $\mathcal{E}$-solver is presented.

Preliminaries

We consider a many sorted first order language. The set of sorts S is a finite set of symbols S=$\{s, s_1, \ldots, s_n\}$. Each function symbol f (collected in the set F = {f/n,g/m,...}) with arity m is prescribed as $f : s_1 \times \cdots \times s_m \rightarrow s$. Constants are defined as function symbols with arity 0; by this, they also are sorted. In this section, the predicate symbols are restricted to $\{=, \neq\}$, where $=$ denotes the equality and $\neq$ the disequality. The first order languages based on this alphabet are denoted as $\mathcal{L}^=$.

For a set of sorted variables $\mathfrak{X}$=$\{X_1 : s_1, X_2 : s_2, \ldots\}$, T(S,F,$\mathfrak{X}$) is the set of all terms that can be constructed from F and $\mathfrak{X}$ according to the sort restrictions of S. A term is either a variable, or a compound term $f(t_1, \ldots, t_n)$ where f/n is a function symbol and each t_i, i=1..n, is itself a term and such that the sort of t_i corresponds with its arguments position. The sort of a term t is denoted with $sort(t)$ and defined as the symbol s of $t : s$. The elements of T(S,F,$\emptyset$) are called ground terms.

An alternative view of a term is to describe it as a tree-structure (which is done in Comon and Lescanne [83]). This is useful in the case of infinite terms. Many of the properties we will discuss holds also for rational trees (these represent well-structured infinite terms). Since that is beyond our needs (at the moment), the above definitions are sufficient to deal with finite terms. In that case, T(S,F,$\emptyset$) is exactly the Herbrand Universe of $\mathcal{L}^=$.

The semantics of a logic theory $\mathcal{T}$ based on $\mathcal{L}^=$ are given by Herbrand interpretations of $\mathcal{T}$, where the equality and disequality symbols are interpreted as follows:

Definition 3.17 Let s and t be two terms

- s and t are equal, denoted as $s = t$, iff $s \equiv t$ (the syntactic equality between terms).
- s and t are not equal, denoted as $s \neq t$, iff $s \not\equiv t$ (the syntactic disequality between terms).

Note that other definitions than syntactic equality exists, e.g. based on the notion of congruence. But that is outside our scope.

An important notion in equality reasoning is a *substitution*. In [83], a generalized notion is used in order to deal properly with infinite terms. For our purposes, the definition 2.20 suffices: A *substitution* $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$ is a morfism from T(S,F,$\mathfrak{X}$) to T(S,F,$\mathfrak{V}$), where $\mathfrak{X}$ is a finite subset of $\mathfrak{V}$ and each variable $X_i \in \mathfrak{X}$, $1 \leq i \leq n$ and each term $t_i \in T(S, F, \mathfrak{V})$. The application of a substitution σ on a formula F is the replacement of every occurrence of a variable X_i,$1 \leq i \leq n$, in F by the corresponding term t_i in σ. (This is denoted as $F\sigma$.) $\mathfrak{X}$ is called the domain of σ, denoted as Dom(σ). The composition $\theta \circ \sigma$ of two substitutions σ and θ is denoted as $\sigma\theta$. A substitution σ:T(S,F,$\mathfrak{X}$) $\mapsto$ T(S,F,$\mathfrak{V}$) is *away from* a set of variables V if V$\cap\mathfrak{X}$ = $\emptyset$ and V $\cap\mathfrak{V}$ = $\emptyset$.

Equational Problems

Definition 3.18 An *equation* $s = t$ is an expression where $s, t \in T(S, F, X)$ and sort(s)=sort(t). A *disequation* is an expression of the form $s \neq t$. The trivial equation is s=s and the trivial disequation is s$\neq$s.

A basic $\mathcal{E}$-formula is an equation or a disequation. If F and G are $\mathcal{E}$-formulas then F$\wedge$G and F$\vee$G are also $\mathcal{E}$-formulas. The negation is not included in the language because it can completely removed by pushing it to the atomic expressions (using De Morgan laws), where $\neg(s = t) \Leftrightarrow s \neq t$ and $\neg(s \neq t) \Leftrightarrow s = t$.

Definition 3.19 An *equational problem* is an expression of the form

$$\exists\, W_1, \ldots, W_n . \forall\, Y_1, \ldots, Y_m . P$$

where P is an $\mathcal{E}$-formula and $W_1, \ldots, W_n$,$Y_1, \ldots, Y_m$ are distinct variables.

Each variable in an equational problem is classified either as *free* (no quantification), as *auxiliary* (existentially quantified) or as *parameter* (universally quantified).

NOTATION 3.5.1 $s =^* t$ in an equational problem denotes the fact that at the place of $=^*$ both $=$ as $\neq$ can be placed.

Intuitively, a solution for an equational problem EP is a substitution that satisfies EP. The formal definition is based on the notion of a substitution that validates an $\mathcal{E}$-formula.

DEFINITION 3.20 A substitution σ *validates* a $\mathcal{E}$-formula F if

- F is an equation s=t and $s\sigma \equiv t\sigma$,
- F is a disequation s≠t and **not** such that $s\sigma \equiv t\sigma$,
- F is a conjunction H∧G and σ validates H **and** G,
- F is a disjunction H∨G and σ validates at least one of the disjuncts,
- F $\equiv \top$

DEFINITION 3.21 A *solution* of an equational problem

$$EP = \exists\, W_1, \ldots, W_n . \forall\, Y_1, \ldots, Y_m . P$$

is a substitution σ such that

1. σ is a substitution which domain Dom(σ) is the set of free variables of EP, and
2. there exists a substitution θ with domain Dom(θ)=$\{W_1, \ldots, W_n\}$ and θ is away from Dom(σ)$\cup\{Y_1, \ldots, Y_m\}$, and such that for all substitutions ρ, which domain Dom(ρ)=$\{Y_1, \ldots, Y_m\}$ and ρ is away from Dom(σ)$\cup$Dom(θ), $\rho\theta\sigma$ validates EP.

An equational problem P is satisfiable if there exists a solution σ for P. It holds that $P\sigma$ is equivalent with $\top$. Note that every substitution for the trivial equation is a solution. It follows that $s = s$ is equivalent with $\top$. P is unsatisfiable if there is no solution for P. In that case P is equivalent with $\perp$. The trivial disequation is unsatisfiable.

Computing solutions for equational problems

We present now a number of inference rules that can be used to reason with equational problems. Unification algorithms such as [228, 192] can be regarded as implementations for special classes of equational problems. By applying the inference rules on an equational problem, the equational problem can be transformed into particular kinds of equational problems from which a solution can be trivially extracted. These special equational problems are called *solved forms.*

From the different solved forms in [83], we have selected the one which is called *Definition with constraints* [83], because it contains equations as well as disequations.

DEFINITION 3.22 An equational problem EP is a *definition with constraints* if it is either $\top$,$\perp$ or a conjunction of equalities and disequalities

$$\exists\, W_1, \ldots, W_n : X_1 = t_1 \wedge \ldots \wedge X_m = t_m \wedge X_1' \neq s_1 \wedge \ldots \wedge X_p' \neq s_p$$

where

- all variables are either free (the X_i and X_j') or auxiliary (the W_k), and
- all variables $X_1, \ldots, X_m$ occur only once, and
- every variable X_i', i=1..p, is distinct from s_i.

To compute this solved form, the reasoning uses on the one hand general properties of the connectives $\wedge$ and $\vee$, and the equality relations $=$ and $\neq$: $\wedge$ and $\vee$ are associative, commutative and idempotent; $\top$ is the identity of $\wedge$ and the absorbing element of $\vee$, and vice versa for $\perp$; $\wedge$ and $\vee$ are distributive w.r.t. each other; $=$ and $\neq$ are commutative; ... and on the other hand specific equality inference rules. The latter are depicted in Figures 3.7 and 3.8. The rules follow the notational conventions: the symbols s, t and u denote terms; X and Y denote variables; P and R are arbitrary equational problems; $|sort(t)|$ denotes the number of elements of the sort of t and $P[F]$ denotes that F is a subformula of P.

In their survey [83], Comon and Lescanne prove the following theorems for our language over finite terms $\mathcal{L}^=$:

- All rules are *sound.*
- All rules in Figure 3.7 are *preserving*, i.e. the resulting equational problem (i.e. after the application of the rule) has exactly the same solutions as the original one. The rules of Figure 3.8 are *globally preserving.* Their application results in an equational problem which contains only a subset of the solutions of the original one. By combining several applications of the rule together, they preserve all solutions.

- elimination of trivial equations and disequations (T)
  - $t = t \implies \top$
  - $t \neq t \implies \bot$ (failure)
- clash of compound terms (C)
  - $f(t_1, \ldots, t_m) = g(s_1, \ldots, s_n) \implies \bot$ (failure)
  - $f(t_1, \ldots, t_m) \neq g(s_1, \ldots, s_n) \implies \top$
- decomposition of compound terms (D)
  - $f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n) \implies t_1 = s_1 \wedge \cdots \wedge t_n = s_n$
  - $f(t_1, \ldots, t_n) \neq f(s_1, \ldots, s_n) \implies t_1 \neq s_1 \vee \cdots \vee t_n \neq s_n$
- occur check (O)
  - $X = t \implies \bot$ when X occurs in t (if $X \not\equiv t$).
  - $X \neq t \implies \top$ when X occurs in t
- elimination of universal quantifier (EU)
  - $\forall\, \overline{Y}, Y.P \implies \forall\, \overline{Y}.P$ if $Y \notin vars(P)$
- merging (M)
  - $s = t \wedge s = u \implies s = t \wedge t = u$
  - $s \neq t \vee s \neq u \implies s \neq t \vee t \neq u$
  - $s = t \wedge s \neq u \implies s = t \wedge t \neq u$
  - $s = t \vee s \neq u \implies t = u \vee s \neq u$
- replacement (R)
  - $X = t \wedge P \implies X = t \wedge P(X/t)$
  - $X \neq t \vee P \implies X \neq t \vee P(X/t)$
- universal quantified variables (U)
  - $\forall\, \overline{Y}, Y.P \wedge Y \neq t \implies \bot$
  - $\forall\, \overline{Y}, Y.Y \neq t \vee R \implies \forall\, \overline{Y}.R(Y/t)$
  - $\forall\, \overline{Y}, Y.P \wedge Y = t \implies \bot$ if $|sort(t)| \geq 2$.
  - $\forall\, \overline{Y}.X = t \vee R \implies \forall\, \overline{Y}.R$ where the variable X and $X \notin \overline{Y}$ and t is a term containing a universally quantified variable ($X \not\equiv t$).

Figure 3.7: The inference rules for equational reasoning

- explosion (E)
  - $\forall \overline{Y}.P[s =^* t] \implies \exists W_1, \ldots, W_n \forall \overline{Y}.P[s =^* t \wedge s = f(W_1, \ldots, W_n)]$ where W_i are new variables and $f \in F$ and s is a term without an occurrence of a universally quantified variable.
- elimination of disjunctions (ED)
  - $\forall \overline{Y}.P \wedge (P_1 \vee P_2) \implies \forall \overline{Y}.P \wedge P_1$ provided that $vars(P_1) \cap \overline{Y} = \emptyset$ or $vars(P_2) \cap \overline{Y} = \emptyset$.

Figure 3.8: Globally preserving rules

- All rules, except R, are *complete* for the solved form *definition with constraints*. (Using these rules any equational problem is transformable in the desired solved form. The union of all solutions of all derivable solved forms is the set of all solutions of the original equational problem.)
- The non-deterministic application of variants[9] of the rules EU,U,M,T,C,D,O,E to any equational problem always *terminates*. The obtained irreducible equational problem is then a parameterless equational problem.
- The non-deterministic application of variants of the rules T,R1,M,C,D,O,E,ED to any parameterless equational problem terminates. The obtained irreducible equational problem is then a *definition with constraints*.

When equational problems are restricted to only contain the equality, existing unification algorithms such as Robinson's [228] and Martelli-Montanari's [192] can be obtained. Hence it is safe to replace unification by the computation of a solution based on these rules.

The $\mathcal{E}$-constraint solver

Based upon the above notions defined by Comon and Lescanne, we present here the ones that are used in the $\mathcal{E}$-solver. Our goal is to develop a constraint reasoner for equational problems. As usual, our $\mathcal{E}$-solver acts on a set of $\mathcal{E}$-formulas $\{F_1, \ldots, F_n\}$, called the $\mathcal{E}$-store. This set represents the equational problem $F_1 \wedge \ldots \wedge F_n$.

Since a constraint store is setup incrementally (a program will add step by step new $\mathcal{E}$-formulas), the equational problem is not available at once. Therefore constraint solvers process the newly added constraints as far as possible, without making any choice. The obtained formula has typically some desired properties that e.g. allow easy storage. In the $\mathcal{E}$-solver, the formulas of the $\mathcal{E}$-store are in

[9] Some rules have extra conditions on their application. For them we refer to [83].

pre-definition with constraints solved form. We shorten the name to the *pre-solved form*.

DEFINITION 3.23 (THE PRE-SOLVED FORM) An equational problem is in pre-*definition with constraints* solved form if it is $\top$ or $\bot$, or a conjunction of the following $\mathcal{E}$-formulas

$$\begin{array}{ll} (E) & X = t \\ (DE) & \forall\, \overline{Y}.X' \neq t' \\ (DJ) & \forall\, \overline{Y}.\bigvee_{i=1..m} X''_i =^* t''_i \end{array}$$

where

- every X, X', X'' is a free or auxiliary variable (thus not a parameter)
- every variable X never appears in another formula
- each term t, t', t''_i is not a parameter
- every pair (X', t') defined by a disjunction $X' \neq t'$, never appears in a DJ-formula as $X' =^* t'$.

It is called the pre-definition with constraints solved form because the application of the globally preserving rules E and ED of Figure 3.8 transforms this pre-solved form to a *definition with constraints* solved form.

- $\forall\, \overline{Y}.X' \neq t'$:
  1. apply E: $\exists\, Z_1, \ldots, Z_n \forall\, \overline{Y}.X' \neq t' \wedge X' = f(Z_1, \ldots, Z_n)$
  2. apply M: $\exists\, Z_1, \ldots, Z_n \forall\, \overline{Y}.f(Z_1, \ldots, Z_n) \neq t' \wedge X' = f(Z_1, \ldots, Z_n)$
  3. if f/n is **not** the principal functor of the term t':
     (a) apply C: $\exists\, Z_1, \ldots, Z_n \forall\, \overline{Y}.X' = f(Z_1, \ldots, Z_n)$
     (b) apply EU: $\exists\, Z_1, \ldots, Z_n.X' = f(Z_1, \ldots, Z_n)$
     which is a formula of type E in the pre-solved form.
  4. if f/n is the principal functor of the term t':
     simplify the disequation $\forall\, \overline{Y}.f(Z_1, \ldots, Z_n) \neq t'$ by using e.g. D and U2.
- $\forall\, \overline{Y}.\bigvee_{i=1..m} X''_i =^* t''_i$:
  1. apply ED: $\forall\, \overline{Y}.X''_j =^* t''_j$ $(1 \leq j \leq m)$
  2. if $=^* \equiv \neq$ then the above reasoning applies.
  3. if $=^* \equiv =$ then
     (a) if $vars(t'') \subseteq \overline{Y}$ then apply U3: $\bot$
     (b) else $\forall\, \overline{Y}.X''_j =^* t''_j$ is equivalent with $X''_j =^* t''_j$ and that is a formula of type E in the pre-solved form.

REMARK 3.5.1 The rule E is always applicable in order to remove parameters since it is assumed (unless explicitly stated) that there exists always an appropriate function symbol. Usually, the set of function symbols is only implicitly defined as the set of function symbols that appear in the logical theory. Our adopted assumption is however weaker: the set of function symbols is a superset of the set that contains all used function symbols. Finiteness of the set of function symbols is only obtained by the addition of an explicit enumeration of the possible values for a variable: e.g. $\exists\, \overline{Z}.(X = f_1(Z_1, \ldots, Z_{m_1}) \vee \ldots \vee X = f_n(Z_1', \ldots, Z_{m_n}'))$.

For that reason, the rule E is not included in the implementation and the equational formulas are reduced to a more general variant of the *definition with constraints* solved form:

DEFINITION 3.24 An equational problem EP is a *generalized definition with constraints* if it is either $\top$,$\bot$ or a conjunction of equalities and disequalities

$$\exists\, W_1, \ldots, W_k : X_1 = t_1 \wedge \ldots \wedge X_m = t_m \wedge \forall\, \overline{Y}.X_1' \neq s_1 \wedge \ldots \wedge \forall\, \overline{Y}.X_p' \neq s_p$$

where

- the variables X_i and X_j' are free, and
- all variables $X_1, \ldots, X_m$ occur only once, and
- every variable X_i' is distinct from s_i, i=1..p .

EXAMPLE 3.17 The $\mathcal{E}$-formula $X = f(a,b) \wedge Y = f(a,Z) \wedge Z \neq b \wedge \forall\, U.Z \neq g(U)$ is in *generalized definition with constraints* solved form. From this solved form, every substitution that is derived from this formula contains $X/f(a,b), Y/f(a,Z)$. For the variable Z many substitutions apply. Since the presentation of each substitution is normally less informative than the compact presentation by the disequalities, these substitutions are not constructed.

Given the above notions, the $\mathcal{E}$-solver implements the following framework:

- The equational problem is stored as a set of formulas in the $\mathcal{E}$-store. The initial $\mathcal{E}$-store is empty, which is equivalent to $\top$.
- Adding an $\mathcal{E}$-formula to the $\mathcal{E}$-store, invokes a process that brings the $\mathcal{E}$-store back in the pre-solved form. This reasoning process uses (some of) the presented inference rules.
- The *generalized definition with constraints* solved form is constructed when the user activates the search for it. (Similar to the labeling call in the finite domain solver). If no such activation happens the $\mathcal{E}$-store in pre-solved form is returned as a (conditional) answer.

The main concern of the $\mathcal{E}$-solver is the satisfiability of its store.

DEFINITION 3.25 An $\mathcal{E}$-store is *satisfiable* if it is transformable to a *generalized definition with constraints* different from $\perp$.

Unfortunately (as already indicated by the existence of a 'labeling' process), the $\mathcal{E}$-solver is incomplete (in the sense of constraint solvers, see Section2.1.4), i.e. the constraint solver is not aware at any time of the satisfiability of its store. This is shown by the following example:

EXAMPLE 3.18 The following three constraints $(X = a \vee X \neq b), (X = b \vee X \neq a)$ and $(X = a \vee X = b)$ result in the following $\mathcal{E}$-store in pre-solved form: $\{(X = a \vee X \neq b), (X = b \vee X \neq a), (X = a \vee X = b)\}$. However as easily verified, the store is not satisfiable.

Reification

The above defined $\mathcal{E}$-solver is able to solve any equational problem defined in $\mathcal{L}^{=}$. In the previous Section3.4, we have argued that the $\mathcal{E}$-solver has to support reification for our implementation of the $\mathcal{A}$system. That requirement implies the extension of the above solver: reified formulas require new inference rules and the definitions of an equational problem and solved form must be updated.

DEFINITION 3.26 A *reified* $\mathcal{E}$-formula is a formula of the form

$$F \Leftrightarrow B$$

where F is an $\mathcal{E}$-formula and B a boolean. An *extended equational problem* EP is a conjunction of an equational problem S and a conjunction of reified $\mathcal{E}$-formulas R.

New inference rules are needed to be able to reason with reified expressions. The relationship between the formula F and the boolean B are given by the inference rules in Figure 3.9. Rule RT2 requires the computation of the negation of an expression. At the beginning of this section, we pointed out that negation is superfluous and can easily be removed by pushing it down to the atomic equality relations. It was reasonable to assume that this process was done by the user, before posting the $\mathcal{E}$-formula to the constraint store. Now, by this rule RT2, the $\mathcal{E}$-solver must support this computation. Although not very complicated, one has to pay attention to switch the quantifiers correctly. For example $\forall\, Y.X \neq f(Y)$ has to become $\exists\, Y.X = f(Y)$.

To provide more intuition for the information consumption rules RC, we consider now two special cases.

- Trivial reified expressions (RT)
  - $P \Leftrightarrow 1 \implies P$
  - $P \Leftrightarrow 0 \implies \neg P$
  - $\top \Leftrightarrow B \implies B = 1$
  - $\bot \Leftrightarrow B \implies B = 0$
- Negation (N)
  The negation is computed by pushing the negation to basic equations, which are the dual of each other. We mention here the most important ones.
  $$\begin{array}{llll} \neg\bot \implies \top & & \neg(P \wedge R) \implies \neg P \vee \neg R \\ \neg\top \implies \bot & & \neg(P \vee R) \implies \neg P \wedge \neg R \\ \neg s = t \implies s \neq t & & \neg(\forall\, \overline{Y}.P) \implies \exists\, \overline{Y} \neg P \\ \neg s \neq t \implies s = t & & \neg(\exists\, \overline{Y}.P) \implies \forall\, \overline{Y} \neg P \end{array}$$
- Inside the reified formula (IR)
  Each preserving inference rule from Figure 3.7 can be applied inside the formula F of a reified formula $F \Leftrightarrow B$. In these rules F is considered as an equational problem, independent from other formulas in the extended equational problem. We illustrate this with some rules:
  - $(f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m) \vee F) \Leftrightarrow B \implies F \Leftrightarrow B$
  - $(\forall\, \overline{Y}.Y \neq t \vee F) \Leftrightarrow B \implies F(Y/t) \Leftrightarrow B$
  - $\vdots$
- Consumption of information (RC)
  These rules relate the state of a reified formula with the current state in the extended equational problem.
  - $X = t \wedge (F \Leftrightarrow B) \implies X = t \wedge (F(X/t) \Leftrightarrow B)$
  - $\forall\, \overline{Y}.X \neq t \wedge (F[X = s] \Leftrightarrow B) \implies \forall\, \overline{Y}.X \neq t \wedge (F[X = s \wedge \forall\, \overline{Y} s \neq t] \Leftrightarrow B)$
  - $\forall\, \overline{Y}.X \neq t \wedge (F[X \neq s] \Leftrightarrow B) \implies \forall\, \overline{Y}.X \neq t \wedge (F[X \neq s \vee \forall\, \overline{Y} s = t] \Leftrightarrow B)$

Figure 3.9: The inference rules for reified expressions

1. Consider $\forall\,\overline{Y}.X \neq t \wedge F[X = t] \Leftrightarrow B$

$$\begin{array}{lll}
\Longrightarrow & \text{RC.2}: & \forall\,\overline{Y}.X \neq t \wedge F[X = t \wedge \forall\,\overline{Y} t \neq t] \Leftrightarrow B \\
\Longrightarrow & \text{IR.T.2}: & \forall\,\overline{Y}.X \neq t \wedge F[X = t \wedge \bot] \Leftrightarrow B \\
\Longrightarrow & \text{IR.}\wedge: & \forall\,\overline{Y}.X \neq t \wedge F[\bot] \Leftrightarrow B
\end{array}$$

2. Consider $\forall\,\overline{Y}.X \neq t \wedge F[X \neq t] \Leftrightarrow B$

$$\begin{array}{lll}
\Longrightarrow & \text{RC.3}: & \forall\,\overline{Y}.X \neq t \wedge F[X \neq t \vee \forall\,\overline{Y}.t = t] \Leftrightarrow B \\
\Longrightarrow & \text{IR.T.1}: & \forall\,\overline{Y}.X \neq t \wedge F[X \neq t \wedge \top] \Leftrightarrow B \\
\Longrightarrow & \text{IR.}\wedge: & \forall\,\overline{Y}.X \neq t \wedge F[\top] \Leftrightarrow B
\end{array}$$

EXAMPLE 3.19 Consider the following extended equational problem

$$\begin{array}{c}
S \wedge F \Leftrightarrow B \text{ where} \\
S = \forall\, Y.X \neq f(Y) \wedge X \neq a \text{ and } F = (X = f(a) \vee X = Z)
\end{array}$$

The reified $\mathcal{E}$-formula F can be simplified as follows

$$\begin{array}{lll}
\Longrightarrow & \text{RC2}: & S \wedge (X = f(a) \wedge \forall\, Y.f(a) \neq f(Y)) \vee X = Z) \Leftrightarrow B \\
\Longrightarrow & \text{D}: & S \wedge (X = f(a) \wedge \forall\, Y.a \neq Y) \vee X = Z) \Leftrightarrow B \\
\Longrightarrow & \text{U2}: & S \wedge (X = f(a) \wedge \bot) \vee X = Z) \Leftrightarrow B \\
\Longrightarrow & \wedge, \vee: & S \wedge X = Z \Leftrightarrow B \\
\Longrightarrow & \text{RC2}: & S \wedge (X = Z \wedge \forall\, Y.Z \neq f(Y)) \Leftrightarrow B \\
\Longrightarrow & \text{RC2}: & S \wedge (X = Z \wedge \forall\, Y.Z \neq f(Y) \wedge Z \neq a) \Leftrightarrow B
\end{array}$$

If now the store is augmented with the equality $Z = a$ (S' is $S \wedge Z = a$), the reified formula is further reduced to

$$\begin{array}{lll}
\Longrightarrow & \text{RC1}: & S' \wedge (X = Z \wedge \forall\, Y.a \neq f(Y) \wedge a \neq a) \Leftrightarrow B \\
\Longrightarrow & \text{T}: & S' \wedge (X = Z \wedge \forall\, Y.a \neq f(Y) \wedge \bot) \Leftrightarrow B \\
\Longrightarrow & \wedge: & S' \wedge \bot \Leftrightarrow B \\
\Longrightarrow & \text{RT}: & S' \wedge B = 0
\end{array}$$

By using the above inference rules, an extended equational problem can be transformed to our goal formula: a *definition with constraints* solved form. To make the reasoning more uniform, the pre-solved form is extended with representations for the reified expressions.

DEFINITION 3.27 A reified expression is in pre-solved form if it is of the form

$$\begin{array}{ll}
(RDJ) & (\forall\,\overline{Y}.\bigvee_{i=1..n} X_i =^* t_i) \Leftrightarrow B, \text{ or} \\
(RCO) & (\forall\,\overline{Y}.\bigwedge_{i=1..m} X_i =^* t_i) \Leftrightarrow B
\end{array}$$

3.5.3 The implementation

We have implemented the $\mathcal{E}$-solver as an equality *constraint* solver, which means that $\mathcal{E}$-formulas are used actively to search for a solution. The $\mathcal{E}$-solver uses an incremental process to evaluate its $\mathcal{E}$-store: whenever new information about the variables in its store becomes available (i.e. new constraints are added to the store), the solver continues with the reasoning either until it is found to be inconsistent and invokes backtracking, or until it results in a new irreducible state.

Subsequently, we present the syntax of the implemented language, the used data structures and how the inference rules are implemented. Thereafter, we discuss the universal quantification and how it is integrated in the solver.

The language

The $\mathcal{E}$-solver supports the following expressions:

- basic expressions: `X:=Y` (equality) and `X:=/Y` (disequality), where `X` and `Y` are terms.
- conjunctions and disjunctions of basic expressions: `or(Exprs)` and `and(Exprs)` where `Exprs` is a list of basic expressions.
- reified expressions: `Expr :<=> B` where `Expr` is one of the previous expressions and `B` a boolean variable.
- universal quantification of variables: `forall(Vars,Expr)` for normal expressions and `forall(Vars,Expr) :<=> B` for reified expressions, where `Vars` is a list of variables of `Expr`, `Expr` is an expression and `B` is a boolean variable. The universally quantified variables must be fresh variables that occur only in the $\mathcal{E}$-formula.

EXAMPLE 3.20 Some expressions and their corresponding representation our implementation:

Equational problem	Epsilon notation
$X \neq a$	`X:=/a`
$f(X,Y) \neq f(b,c)$	`f(X,Y) :=/ f(b,c)`
$(g(X,Y) = g(a,Z) \Leftrightarrow B_1)$	`g(X,Y):=g(a,Z):<=> B1`
$\forall\, U.X \neq f(U,U)$	`forall([U],X:=/f(U,U))`
$\forall\, V.g(X,a) = g(f(a,a),V) \Leftrightarrow B_2$	`forall([V],g(X,a):=g(f(a,a),V)):<=>B2`

(X, Y are variables and a, b, c are constants)

The data structures

Developing efficient constraint solvers in Prolog is not a trivial task. Therefore various researchers have designed several techniques to facilitate the implementation. One of these is the special construct called *attributed variables* [146]. The idea behind attributed variables is to attach variable specific information via an attribute to the variable. In this way, a direct access to the relevant parts in the constraints can be modeled, e.g. when the variable becomes unified with a value, the constraint verification can be initiated immediately.

As explained before, the intention is to keep the constraint store in pre-solved form. The next observations and decisions simplify the work that must be done.

- Disjunctions are equivalent to reified disjunctions with the boolean B set to **true**.
$$\forall\, \overline{Y}.\ \bigvee_{i=1..m} X_i =^* t_i \equiv (\forall\, \overline{Y}.\ \bigvee_{i=1..m} X_i =^* t_i) \Leftrightarrow 1$$
Consequently, the data structure does not need a separate representation of a disjunction. The same holds for conjunctions, but since a constraint store is a set of expressions which is interpreted as the conjunction of these expressions, the conjunction can be made implicit.

- The evaluation of reified expressions by the inference rules of figure 3.9 leads to an arbitrary nesting of conjunctions in disjunctions, and vice versa. To turn such a formula in its pre-solved form, properties like distributivity can be applied. Since that is a costly operation, another approach is applied. Each conjunct in a disjunction will be replaced by the boolean that is obtained by the reification of the conjunct (formula 3.1). An analogous process is applied for disjuncts in reified conjunctions (formula 3.2). Hence, the data structures must maintain formulas of the form:
$$\left(\forall\, \overline{Y}.\ \bigvee_{i=1..m} X_i =^* t_i \vee \bigvee_k B_k\right) \Leftrightarrow B \tag{3.1}$$
$$\left(\forall\, \overline{Y}.\ \bigwedge_{i=1..m} X_i =^* t_i \wedge \bigwedge_k B_k\right) \Leftrightarrow B \tag{3.2}$$

- Because the semantics coincide, the maintenance of the equalities $X = t$ is outsourced to the internals of Prolog.

Example 3.21 (Example 3.20 continued) The formulas of example 3.20 can be (deterministically) transformed to the next formulas: $X \neq a, (X \neq b \vee c \neq Y \Leftrightarrow 1), (X = a \wedge Y = Z \Leftrightarrow B_1), \forall\, U.X \neq f(U,U)$. The last equation is reducible to $B_2 = 0$.

Maintaining the constraint store as one large formula is computationally inefficient. Each reasoning step may require to traverse the whole formula. This process is (partially) eliminated by separating the formula in such a way that direct access is provided to the relevant parts for the application of an inference rules. For this, attributed variables are used. The following process cuts an equational problem into parts. Each part is associated with one variable, and that association is made explicit by storing this part in the attribute of the variable. Of course, the process takes care that no information is lost.

1. Select a variable X from the $\mathcal{E}$-formula
2. Select the relevant constraint information for the variable X occurring in the $\mathcal{E}$-formula:
   - for any $X =^* t$, it is the formula $X =^* t$ itself
   - for any subformula $ID : \forall\, \overline{Y}.D \vee X =^* t \Leftrightarrow B$, it is the triple $(\vee, X =^* t, ID)$
   - for any subformula $ID : \forall\, \overline{Y}.C \wedge X =^* t \Leftrightarrow B$, it is the triple $(\wedge, X =^* t, ID)$

   where ID is a unique identifier of the reified expression.
3. Remove the selected formulas from the $\mathcal{E}$-formula and store them in the attribute of X.

   We have chosen to associate a variable with a ternary attribute *eps(DIS,DJ,CO)*, which is denoted as: $X \mapsto (DIS, DJ, CO)$. In this attribute, each argument is set for one type of the above selected formulas:
   - DIS is a set of disequalities: $\{X \neq t_1, X \neq t_2, \ldots\}$.
   - DJ is a set of disjuncts $\{(\vee, X = t_1, O_1), (\vee, X \neq t_2, O_2), \ldots\}$ and O_i are the identifications of the disjunctions to which that disjunct belongs.
   - CO is a set of conjuncts $\{(\wedge, X = s_1, A_1), (\wedge, X \neq s_2, A_2), \ldots\}$ and A_i are the identifications of the disjunctions to which that disjunct belongs.

   (In the current implementation the sets are represented as lists.)
4. Continue from start until the whole $\mathcal{E}$-formula is stored in the attributes of the variables.

From now on, we use the following shorthands: the term *junct* refers to any triple of the form $(\wedge/\vee, X =^* t, ID)$, and a *junction* refers to a reified conjunction or a reified disjunction.

In this splitting of junctions, the identification ID is the glue between the independent juncts. It must contain therefore information about the global state of the junction. In the implementation, an identification is a quadruple (B, N, ID, BL). In this structure, B is the boolean of the original reified expression; N is the number of active juncts in this junction and ID a unique identifier of the junction. The last argument is an auxiliary boolean BL, which is true when only one active junct is left and the reified boolean B is 1 (0) when the junction is a disjunction (conjunction). Due to this last argument the $\mathcal{E}$-solver can achieve more propagation.

The identification is shared by all juncts from the same junction. During the reasoning of the $\mathcal{E}$-solver, each update to the identification must be visible in all juncts. If all arguments can be represented as variables or as constants, a normal Prolog term suffices. However, the number of active juncts (which is a constant in each state) varies during the reasoning and this change cannot be modeled by a standard Prolog term. Therefore an identification is encoded by means of a backtrackable destructive assignment, called a *mutable term.* A mutable term can be seen as a pointer to a term-structure. The value of a mutable term can be changed by the construction of a new term and redirecting the pointer to this term. This creates a backtrack point. On backtracking the old value is restored.

EXAMPLE 3.22 (EXAMPLE 3.21 CONTINUED) The attributes of variables are

X $eps(\{a, f(U,U)\}, \{(\vee, \neq, b, O1)\}, \{(\wedge, =, a, A2)\})$

```
eps([a,f(U,U)],[or(:=/,b,O1)],[and(:=,a,A2)])
```

Y $eps(\emptyset, \{(\vee, \neq, c, O1)\}, \{(\wedge, =, Z, A2)\})$

```
eps([ ],[or(:=/,c,O1)],[and(:=,Z,A2)])
```

where $O1$ is the mutable term $(1, 2, id1, BL_1)$ and $A2$ is $(B_1, 2, id2, BL_2)$. The variable Z gets no attribute since the junct in which it is involved is stored in the attribute of Y. When Z would be selected before Y it would have received an attribute.

Propagating information

The next step in the design of the $\mathcal{E}$-solver is the implementation of the inference rules. There are two ways to invoke reasoning by the $\mathcal{E}$-solver. One invocation happens when a Prolog program adds a new $\mathcal{E}$-formula in the constraint store. This formula will be simplified and distributed over the attributes of the involved variables as explained above, by the $\mathcal{E}$-solver. This triggers further reasoning because The newly added information triggers further reasoning, leading to simplification of other expressions in the $\mathcal{E}$-store.

The second invocation of the $\mathcal{E}$-solver happens on unification. Logically, this means that an equality is added to the constraint store, but since the equalities are maintained by the unification mechanism of Prolog, unification is a different way to trigger the $\mathcal{E}$-solver than the previous one. For the unification of an attributed variable, the Prolog engine calls a special unification handler. In this handler, the programmer specifies what must happen with the information stored in the attributes of the involved variables. The handler for the $\mathcal{E}$-variables is presented below in the program 3.5.1.

In the following, we only present how a unification (equality) is propagated through the whole $\mathcal{E}$-store. The addition of a disequation has an analogous effect, and will not be presented here. Both will use the same auxiliary procedures.

The unification handler of the $\mathcal{E}$-solver must take care of two situations. Let X be the variable for which the unification handler is called and t the term to which X will be unified after the evaluation of the handler. In the first case, t is a variable with an $\mathcal{E}$-attribute. The unification of X with t is correct if all formulas in the attribute of X are satisfiable w.r.t. t, and if all formulas in the attribute of t are satisfiable w.r.t. X. In all other cases the formulas in the attribute X must be satisfiable w.r.t. the term t.

The handler also has to ensure that no information is lost. After the handler has been executed, the attribute of X is lost because the value of X will be overwritten by t. In the handler X and t are not allowed to be unified with a(nother) value: just after the handler is evaluated successfully, the unification of X and t happens. Since the formulas in the attributes of X and t must be checked when X (and t) have their new value, all formulas are selected and put in an action list. The action list is evaluated after the unification. This way of checking also ensures that no information is lost, since each formula is not reduced to $\bot$ or $\top$ is added again to an attribute of a variable. This may be again t (when t was a $\mathcal{E}$-variable), but it also may be another variable. Because the disequations are sure to be part of the attribute after unification (in case t is an $\mathcal{E}$-variable), the union of the disequations of X and t are immediately stored in the attribute of t.

Program 3.5.1 (The unification handler)

```
handle_unification(X,T,eps(XDE,XDJ,XCO),Actions):-
   ((var(T), get_atts(T,eps(TDE,TDJ,TCO))
      -> % also T is an epsilon-var
         union(XDE,TDE,DE),
         put_atts(T,eps(DE,[],[])), % recue disequalities
         put_atts(X,-eps(_,_,_))    % remove attribute
         Actions = [propagate_all(=,T,eps(XDE,XDJ,XCO)),
                    propagate_all(=,X,eps(TDE,TDJ,TCO)]
      ;  % T is normal term
         Actions=[propagate_all(=,T,eps(XDE,XDJ,XCO))]
      ).
```

After unification, Prolog executes the actions that have been returned by the unification handler. In the $\mathcal{E}$-solver, these actions are the verification of the constraints in the attribute of a variable. There are two checks to be performed.

- The disequality check: for each term $s \in DE_X$, it must hold that $t \neq s$. This disequality is further simplified until none of the inference rules in Figure 3.7 can be applied.
- The evaluation of juncts: e.g. for each junct $(\vee, =^*, s, ID) \in DJ_X$, it must be verified whether $t =^* s$. This evaluation starts with the application of one of

the inference rules RC of Figure 3.9. Again, this newly obtained subformula is further simplified until no inference rules is applicable anymore.

The obtained $\mathcal{E}$-formulas are stored in the attributes of the involved variables according to the process explained before.

PROGRAM 3.5.2 (THE EVALUATION OF A JUNCT)
The presented program is an extract of the evaluation process of the juncts. It evaluates a junct $(\vee, =, s, O)$ that has been stored in the attribute of the variable X, w.r.t. the term t which has been unified with the variable X. This situation is formally described by the formula $X = t \wedge (X = s \vee F_O) \Leftrightarrow B_O)$. F_O and B_O represent the rest of the junction which are part of the identification O in the junct.

```
propagate(X,:=,T,or(:=,S,O)):-
   % check T=S
   (reduce(S:=T,ES)
      -> (ES = []
            -> % successful
               disjunction_entailed(O)
            ;  % replace junct with a new one
               and(ES) :<=> B,
               addToJunction(B,O)
            )
      ;  % fails
         decrease_active_juncts(O)
      ).
```

In this case, the inference rule RC1 is applicable: $X = t \wedge (X = s \vee F_O) \Leftrightarrow B_O \implies_{RC.1} X = t \wedge (X = s \vee F_O)(X/t) \Leftrightarrow B_O \implies X = t \wedge (t = s \vee F_O) \Leftrightarrow B_O$. It results in the equality junct $s = t$. If this equality is unsatisfiable (hence it is reducible to $\perp$), then this disjunct is reduced to $\perp$. Since the junct is part of a larger disjunction, this information is propagated by decreasing (by one) the number of juncts in the disjunction. If the equality $s = t$ is reducible to $\top$, then the whole disjunction is satisfied ($B_O = 1$). The last case is when the equality is simplified to another formula. This formula must be a conjunction. This conjunction is reified and the associated boolean will replace the original junct in the disjunction.

In total there are eight of these programs to evaluate all possible combinations $(|\{=, \neq\}| \times |\{\vee, \wedge\}| \times |\{=, \neq\}| = 8)$. Each program depends on the (dis)equality reduction procedures to make its decisions. These procedures, i.e. `reduce(X:=Y,Es)` and `reduce(X:=/Y,DES)`, perform the basic inference in the solver. They simplify a (dis)equality $s =^* t$ to a list of (dis)equalities $[X_i =^* u_i, \ldots]$ where every X_i is a variable and u_i a term, using the inference rules of Figure 3.7 and the syntactic equality $\equiv$ and the properties of $\wedge$ and $\vee$. The answers they return, are one of the following:

- the procedure fails when $s =^* t \equiv \bot$
- the procedure returns an empty list when $s =^* t \equiv \top$
- the procedure returns a non-empty list, which is a simplification of the equivalent junction. In the case of the reduction of an equality $s = t$, the list represents a conjunction; otherwise when $s \neq t$ is reduced, the list represents a disjunction.

Finally, observe that the decrease of the amount of juncts can invoke further propagation. If the value becomes 1, then there is just one active junct in the junction. If the boolean B of the junction has the appropriate value then it follows that this last junct must be evaluated. This reasoning is triggered by instantiating the boolean BL in the identification of the junct. For example, if there is just one active junct in a disjunction (not a reified disjunction!), then this last junct must be true in order to ensure that the disjunction is satisfied.

Universal quantification

Up to now, we did not discuss parameters (universally quantified variables). To identify these variables, we again exploit attributed variables. Each universally quantified variable is associated the attribute `univ/0`; we denote this as $Y \mapsto \mathit{univ}$. Recall that these are the variables appearing in the argument `Vars` of the expressions `forall(Vars,Expr)` and `forall(Vars,Expr) :<=> B`.

The only place where the solver explicitly has to deal with universally quantified variables, is in the reduction procedures:

- Equality reduction:
  Since we assume that our language has more than one constant, the equality $\forall\, Y.Y = X$ will always fail. Hence, when encountering a universally quantified variable during the reduction of an equality, the procedure fails.
- Disequality reduction:
  The reduction procedure for a disequality $s = t$ must be extended to handle universally quantified variables. First the disequality $s = t$ is simplified to a disjunction of the form

  $$\forall\, \overline{Y} \left(\bigvee_{0\leq i\leq n} X_i \neq s_i \vee \bigvee_{0\leq j\leq m} Y_j \neq t_j\right)$$
  $$\text{where } Y_j \in \overline{Y}, 0 \leq j \leq m, \text{ and } X_i \notin \overline{Y}, 0 \leq i \leq n$$

  using the inference rules T,D,C and O and the simplification rules of disjunction ($\vee$). (It may be that the same variable occurs more than once in the left hand side of a disequality.) If there are disequalities $Y_j \neq t_j$ then the formula is further reduced in a second phase. This reduction applies inference rules U2 until all disequalities for the universally quantified variables Y_j are removed.

EXAMPLE 3.23 Suppose the following execution. As first constraint $\forall Y.X \neq f(Y,Y)$ is added to the store. That results in $X \mapsto ([f(Y,Y)],[\,],[\,])$ and $Y \mapsto univ$. Next, the constraint $X = f(v,w)$ is added. v and w are free variables. Via the unification handler, the reduction call

$$\texttt{reduce(f(V,W)} := /\texttt{f(Y,Y),Reduct,Univ)}$$

is invoked. The first phase reduction computes that `Reduct=[ ]` and `Univ=[Y:=/V,Y:=/W]`. Stepwise assignment of `Univ`, selects `Y:=/V` and then unifies Y with V. That transforms the second disequality in `V:=/W`. Since this contains no universally quantified variable, and no further disequalities must be evaluated, the reduction process succeeds. This reduction is added to the constraint store as $u \mapsto ([w],[\,],[\,])$. The derived disequality is the expected condition that must be fulfilled in order to keep the constraint store consistent. If v and w have the same value, the addition of the equality $X = f(v,w)$ to the constraint store $\mathcal{E} = \{\forall\, Y.X \neq f(Y,Y)\}$ must fail.

In our implementation we have assumed that there are only free variables and universally quantified variables. This suffices for the $\mathcal{A}$system since only unfolding of definitions can introduce new universally quantified variables. The evaluation of equations does not generate new universally quantified variables. Free variables are interpreted as globally existentially quantified in the $\mathcal{A}$system. Hence when carefully done, they can take over the role of auxiliary variables (i.e. existentially quantified). Indeed, the negation of an $\mathcal{E}$-formula containing only parameters and free variables will have only auxiliary and free variables. When as in the $\mathcal{A}$system case, $\mathcal{E}$-formulas are not negated twice, the distinction between auxiliary and free variables can be ignored. This assumption allows to state that in order to get the correct quantifications when negating an $\mathcal{E}$-formula, it suffice to remove the attribute *univ* from all universal variables in the $\mathcal{E}$-formula. This must only happen for inference rule RT2. For a reified expression $F \Leftrightarrow B$, the following suspended goal will remove the univ/0 attribute when B becomes false.

```
when(ground(B),
   (B = 0
      -> remove_univ_attribute(Vars)
      ;  true
      )
   ).
```

Enumeration

The propagation reasoning ensures that the $\mathcal{E}$-store is in a pre-solved form. In order to reach a generalized *definition with constraints* solved form, we need an extra procedure: `eps_labeling(Vars)` where `Vars` is a list of variables. This procedure removes systematically all juncts stored in the attributes of the variables in `Vars`. This procedure will traverse the list `Vars` from left to right to select the next to be evaluated variable. For each selected variable X the stored juncts $(\wedge/\vee, =^*, t, ID)$ are evaluated: either by unifying X with t or by adding the disequation $X \neq t$. When the list `Vars` contains all variables and the procedure terminates, then the $\mathcal{E}$-store is in generalized *definition with constraints* solved form.

3.5.4 Related work and discussion

Equality reasoning is without doubt the most fundamental kind of reasoning a (first order) logic inference system has to perform. This has resulted in a vast amount of literature on the subject.

In this domain, unification [29] is the most studied and applied equality reasoning technique. It can be characterized as the process to compute a substitution for an equality between two terms $s = t$ such that both become (syntactically) equal. The first formal account was given by Robinson [228], who introduced unification as a basic operation in his resolution principle. Since then, various researches proposed improvements to his unification algorithm. Notable is the one by Martelli and Montanari [192] who showed that a linear time algorithm exists.

With regard to our problem, unification is just a part of it. In our case, the equational problems that must be solved, include disequations, conjunctions and disjunctions such that the unification algorithms are not directly applicable anymore. A first study of this extended situation was done in the context of PROLOG II. Colmerauer [81] introduced disequations to provide a semantic definition of PROLOG II. Then he showed that some transformations on an equational problem lead to irreducible forms (cfr. solved form). This transformational approach is also used in [173], where Lassez, Maher and Marriott demonstrated the fundamental mechanisms of equational problems. The survey of Comon and Lescanne [83], on which our work is based, aims at unifying all previous work (up to that moment) on equational problems.

Our contribution to the area of equality reasoning is the implementation of a (dis)equality constraint solver that *allows* reification. To our knowledge, no Prolog system provides such solver. One of the reasons for the absence of such solver is that no Prolog system implements a general form of negation, which requires to reason with disequalities. They only provide the *negation as failure* principle [78] to compute the negation of a goal. This can easily be described by the Prolog program

PROGRAM 3.5.3 (NEGATION AS FAILURE)
```
not(P):- call(P),!, fail.
not(P).
```

Shortly after its introduction, it was proven that it is in general unsound when the negated goal contains variables[10].

To compute correctly the negation of a goal, the need for disequalities appear naturally [81]. For example, given the Prolog program

```
p(a).
q(X):- not(p(X)).
```

the expected result for the query `q(Y)`, is $Y \neq a$. Such answers cannot be provided by SLDNF-resolution since it considers only substitutions as answers. For a survey on SLDNF we refer to [14]. Another technique to compute the negation, called *constructive negation*, was proposed by Chan [69, 70]. The idea behind it is the following procedure: for a negated goal, compute all solutions to the positive goal and collect them in a disjunction. The negation of this disjunction is then equivalent to the negated goal and can be used to extract all answers for the negated goal. However, his proposal was considered hard to implement [205], and therefore no Prolog system uses it. More recently Stuckey [244] suggested to use Constraint Programming techniques to implement constructive negation. He argues that this framework is a more natural environment to implement constructive negation. This claim is supported by Barták [36] who presented an implementation of a constraint solver clp(H) over Herbrand terms and integrated it into a constructive negation procedure. Our solver $\mathcal{E}$ can be seen as an improved and more expressive version (as it also allows conjunction, disjunction and reification) of this solver.

The most recent work is by Muñoz et al. [204, 205, 206]. They make consecutive steps towards the introduction of a sound negation in CIAO Prolog [77]. More specifically, they aim at the integration of a sound negation in a Prolog compiler (in casu constructive negation) with a minimal loss of efficiency. For this, Muñoz et al. analyze the uses of negation in Prolog programs [207] and try to derive automatically the appropriate computation of the negated goal. Indeed in many cases simple techniques like *negation as failure* are applicable and more complicated techniques e.g. constructive negation, are not necessary.

Constructive negation has another similarity to our work in the context of the $\mathcal{A}$system. In order to compute efficiently the answers of the negated goal, the notion of a *frontier* of a negated goal is introduced. Informally, this is the

[10] A sample program to show it is:

```
q1 :- not(p(X)).
q2 :- X=a,not(p(X)).
p(a).
```

q1 will fail, while q2 is true.

formula obtained after the application of the constructed negation procedure for a limited number of resolution steps. We illustrate the concept by the next example program.

```
even(0).
even(s(s(X)):-even(X).
```

The frontier of depth 1 for the negated goal *not(even(Y))* is

$$Y \neq 0 \wedge \neg(Y = s(s(Y1)) \wedge even(Y1))$$

Frontiers form a compact representation of the negated goal. Some disjuncts are evaluated completely, i.e. they contain only equalities and disequalities, and can be used to return an answer immediately. If these answers are exhausted, a disjunct that is not evaluated completely, can be extended.

For the same program, the $\mathcal{A}$system will reduce first the query to two residual denials: $\leftarrow Y = 0$ and $\forall\ Y_1.\ \leftarrow Y = s(s(Y_1)) \wedge even(Y_1)$. The first denial will be added immediately to the $\mathcal{E}$-store as the disequality $Y \neq 0$. The second forms a choice point and is stored in the goal stack. The resulting $\mathcal{A}$system state corresponds exactly to the above frontier. This is not surprising, as the negation in the $\mathcal{A}$system is intended to be classical negation[11] and SLDNFA encloses SLDNF. This correspondence shows that the $\mathcal{A}$system implements SLD augmented with *constructive negation.* This is an interesting observation and coincidence since the extension of SLD with constructive negation was not the aim of the abductive procedure SLDNFA.

3.5.5 The $\mathcal{E}$-solver in the $\mathcal{A}$system

At the beginning of this section, we have motivated the development of the $\mathcal{E}$-solver mainly by the observation that a large part of the reasoning performed by the $\mathcal{A}$system is equality reasoning. Here, we present several effects and improvements on the $\mathcal{A}$system which are possible due to the $\mathcal{E}$-solver.

Effects on the language

Using $\mathcal{E}$ imposes some restrictions on the language of the $\mathcal{A}$system. Underlying the inference rules is the assumption that the unique names axioms hold. Also it requires that the $\mathcal{A}$system language has at least two constants for each sort (see inference rule U.3 in Figure 3.7). Suppose there was only one constant and no function symbols, then the expression $\forall\ Y.Y = a$ is equivalent with $\top$ and not $\bot$ as is encoded in the rules U.3.

[11] The name SLDNFA is a bit misleading in this respect.

For some problems, the domain of a sort is closed, i.e. that sort has a finite number of domain elements. This domain closure must be stated explicitly. For every variable X, it is expressed as the disjunction

$$\bigvee_i X = t_i$$

where t_i, i=1..n, are all the possible terms in the Herbrand Universe of the language for the sort of X.

A closed domain can be regarded as a finite domain. For many problem domains this is a natural view, and moreover it leads to more compact representations and to a more efficient reasoning. Since any finite set of elements can be mapped to a finite set of integers, the the finite domain variable solver of Sicstus Prolog limits its domain elements to integers. That limits a natural encoding (what is the aim of Declarative Problem Solving) since sensible names must be replaced by numbers.

Example 3.24 Consider a graph coloring problem: the color of a vertex is given by the abducible $color/2$; the problem instance is given by a graph description ($edge/2$ and $vertex/1$) and the set of colors $color/1$. The general problem description is

$$\begin{array}{l}
\forall\ X,Y,CX,CY. \leftarrow edge(X,Y) \wedge color(X,CX) \wedge color(Y,CY) \wedge CX = CY. \\
\forall\ X,Y,CX,CY. \leftarrow color(X,CX) \wedge color(X,CY) \wedge CX \neq CY. \\
\forall\ X. \leftarrow vertex(X) \wedge \neg exists_color(X). \\
exists_color(X) \leftarrow color(X,CX) \wedge color(CX).
\end{array}$$

A problem instance encoding a triangular graph that must be colored with three colors is

$$\begin{array}{lll}
vertex(v1). & edge(v1,v2). & color(red). \\
vertex(v2). & edge(v2,v3). & color(blue). \\
vertex(v3). & edge(v3,v1). & color(yellow).
\end{array}$$

This instance encoded using finite domain expressions yields

$$\begin{array}{l}
vertex(X) \leftarrow X \text{ in } 1..3. \\
color(X) \leftarrow X \text{ in } 1..3. \\
edge(X,Y) \leftarrow relation(X,[1-\{2\},2-\{3\},3-\{1\}],Y).
\end{array}$$

The $\mathcal{A}$system computes for both problems the same solutions, but the computation is totally different. The first problem instance is solved by generating a number of equality constraints for the first two denials. Since the tail is empty, these lead to equality expressions that are posted in the $\mathcal{E}$-store. The domain closure for the colors is expressed by the $color(CX)$ literal in the definition of $exists\_color/1$. The $\mathcal{A}$system cannot simplify this predicate, hence it forms a choice point. To find a solution, the $\mathcal{A}$system makes a choice for each instance of *color*/1. For the next solution the $\mathcal{A}$system backtracks and makes a different choice.

The finite domain encoding avoids the backtracking at the level of the $\mathcal{A}$system inference rules since the whole problem is reduced to a finite domain constraint store. Instead of an enumerative definition of *color*/1, a single finite domain expression X in 1..3 suffice to express that the colors are either 1 or 2 or 3. This transfers the choice point of *color*/1 from the $\mathcal{A}$system inference rules to the finite domain constraint store. Since that are the only choice points for this problem, all backtracking must be performed by the finite domain constraint solver. The backtracking of the finite domain solver is much faster than that of the $\mathcal{A}$system; thus, this encoding constructs the solutions faster.

The following small experiment in which a graph is colored shows the efficiency differences between the different equality reasoners in the $\mathcal{A}$system. We have implemented a simple Prolog program that searches a solution by *generate and test*. This resembles the behavior of the equality reasoning in the earliest prototypes of SLDNFA. The efficiency of the $\mathcal{E}$-solver is tested by two programs that construct an $\mathcal{E}$-store. In the first variant the constraint store contains only disequalities. The second variant adds also the domain closure as a disjunction to the $\mathcal{E}$-store. The last tested solver is the finite domain solver ($\mathcal{FD}$-solver). The test program constructs the same constraint store as the second variant of the $\mathcal{E}$-solver. The $\mathcal{FD}$-solver allows to experiment with the search strategy: therefore we present the results for a left-to-right selection strategy and for the *first fail most constrained* strategy (ffc). The latter is usually the most efficient strategy. The problem instances are four-colorable graphs that have been used in the experiments of [213]. The size in the table refers to the number of vertices in the graph. The entries with a dash (-) denote that these could not be solved in a reasonable amount of time (i.e. some minutes).

size	Prolog	$\mathcal{E}$-solver		$\mathcal{FD}$-solver	
		only $\neq$	$\neq$ + domain closure	left to right	ffc
20	60ms	40ms	10ms	0ms	0ms
25	310ms	20ms	20ms	10ms	0ms
29	290ms	140ms	30ms	10ms	10ms
30	360ms	160ms	30ms	10ms	0ms
100	-	-	120ms	30ms	20ms
300	-	-	-	-	160ms

The table shows an improvement in efficiency from the simple generate-and-test approach to the approach using the $\mathcal{FD}$-solver with the ffc search strategy. That generate-and-test is the worst approach is to be expected. The differences between the different constraint encodings is explained as follows. The first variant of the $\mathcal{E}$-solver takes the least information into account of the four constraint encodings. In particular, the constraint solver cannot detect that a vertex cannot be colored when it must be different from all colors in the domain. That is why the second variant of the problem is solved more efficiently by the $\mathcal{E}$-solver. By the domain closure disjunction, that situation is detected and backtracking happens immediately. The improvement from the $\mathcal{E}$-solver to the $\mathcal{FD}$-solver is explained by more efficient data structures for finite domain constraint problems. The $\mathcal{E}$-solver represents the domain as a disjunction which consumes a lot of space. In the $\mathcal{FD}$-solver, this disjunction can be a list of intervals that encode the current domain. This is much more efficient. In addition, the equality constraints are evaluated more efficiently by the $\mathcal{FD}$-solver since they can act on this compact domain representation. For the disequality, it means that if the term from which the variable must be different is ground, then this term can be removed from the domain of the variable. The first variant for the $\mathcal{FD}$-solver uses the left-to-right selection strategy. This selection strategy is also applied in the $\mathcal{E}$-solver encodings. Hence, the improvement is almost completely due to this more compact data representation in the $\mathcal{FD}$-solver. Finally, the efficiency of the $\mathcal{FD}$-solver is improved by a more intelligent search strategy (ffc).

In general, our modeling advice is to use finite domain constraint expressions when the domain is finite and can be enumerated as a set of constants. When the domain is open (i.e. no domain closure), or function symbols are needed, the $\mathcal{E}$-solver must be used. The equality solver and finite domain store are complementary in this respect.

Optimization of $\mathcal{A}$system inference rules

Comparing the inference rules of the $\mathcal{A}$system with those of the $\mathcal{E}$-solver, one observes that the $\mathcal{A}$system rule E.2c is very similar to the rule U.2 of the $\mathcal{E}$-solver. Inference rule E.2c can be viewed as a generalization of U.2 where the disequality is embedded in a conjunction of arbitrary literals. This suggests that it is possible to combine the reasoning of the $\mathcal{A}$system and the $\mathcal{E}$-solver so that a more efficient computation is obtained for the E.2c rule.

Consider the denial $\forall\, \overline{Y}. \leftarrow u = v \wedge Q$. Let $\overline{Z} \subseteq \overline{Y}$ be the universally quantified variables of u and v. The inference rules E.2 evaluate this denial is two phases: first simplify the equation to a conjunction of equations between a variable and a term and then evaluate these equations one by one. Eventually, if all equations are satisfied the denial $\forall\, \overline{Y}'. \leftarrow Q\sigma$ s reached where σ is the substitution resulting from the evaluation of $u = v$. Our implementation of the reification allows to handle this process in one step, due to a side effect in the evaluation of the parameters, namely it computes the substitutions of the parameters that are the reasons for the unsatisfiability. The used reified expression is $\forall\, \overline{Z}.u \neq v \Leftrightarrow B$. By this side effect, the $\mathcal{E}$-solver computes the necessary substitutions for the parameters $\overline{Z}$, so that the correct denial $\forall\, \overline{Y}'. \leftarrow Q\sigma$ is computed. Thus, in this way the rules E.2 are compacted into one inference rule.

An extension of the $\mathcal{A}$system inference rules

The $\mathcal{A}$system inference rules that have been presented, required to express the disequality as the negation of the equality. Since the $\mathcal{E}$-solver offers disequality expressions, it is sensible to extend the $\mathcal{A}$system inference rules so that the $\mathcal{A}$system deals explicitly with the disequality. This gives the following rules:

E.1 $\mathbf{s} \neq \mathbf{t} \wedge \mathbf{Q}$:
$\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{Q\}$ and $\mathcal{E}_{i+1} = \mathcal{E}_i \cup \{s \neq t\}$

E.2 $\forall\, \overline{\mathbf{X}}. \leftarrow \mathbf{s} \neq \mathbf{t} \wedge \mathbf{Q}$: $vars(s) \cup vars(t) \not\subset \overline{X}$
$\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{$s=t$\}$ OR
$\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{\forall\, \overline{X}. \leftarrow Q\} \cup \{s \neq t\}$

Propagation of disequality information

The main improvement that this implementation yields w.r.t. earlier versions of the equality solver, is the propagation of disequality information. This means that the $\mathcal{E}$-solver is able to infer from a given constraint store e.g. $\{X \neq Y\}$ and an observing reified expression e.g. $X = Y \Leftrightarrow B$ that $B = 0$. (The reified expression represents a choice point at the level of the $\mathcal{A}$system.) This effect is visible in the evaluation of equality constraints $\forall\, \overline{Y}. \leftarrow s = t \wedge Q$, reuse of abducibles and the evaluation of positive definitions. The last is explained below. Early versions of the equality solvers ignored the disequality information, hence the versions of the $\mathcal{A}$system using those solvers made more choices, consequently exploring a larger search tree in order to find the same solutions.

This propagation of disequality information is especially important when the formulas that the $\mathcal{A}$system encounters during a derivation, have a lot of uninstantiated arguments. Not exploiting all available information in such case, is deadly for the efficiency of the system since it implies a higher chance to make a wrong decision.

Principal functor indexation

In Section 3.4, we have exploited the principal functor indexation (i.e. using the functor of the first argument to build an index, so that on selection of an atom in a derivation, the right clause immediately can be selected) to optimize the $\mathcal{A}$system meta-program. A similar optimization is useful at the level of the $\mathcal{A}$system specifications.

We will show how the $\mathcal{E}$-solver results in (a generalization of) indexation at the level of the $\mathcal{A}$system-specifications. Recall that the suspension for a defined predicate atom is done by constructing a guard for each clause in the definition of the predicate. A guard for a clause is the reification of the finite domain and equality expressions the clause contains.

EXAMPLE 3.25 Consider the following extract from a 'logistics' planning specification

$$\forall\, A, T. \leftarrow act(A,T) \wedge \neg preconditions(A,T).$$

$$preconditions(move(Obj, Vec, From, To), T) \leftarrow \ldots$$
$$preconditions(move\_vehicle(Vec, From, To), T) \leftarrow \ldots$$

The derivation of the atom $act(move\_vehicle(truck1, loc1, loc2), T_1)$ leads to positive definition choice point $preconditions(move\_vehicle(truck1, loc1, loc2), T_1)$. The constructed guards[12] are

$$G_1 \Leftrightarrow move(Obj, Vec, From, To) = move\_vehicle(truck1, loc1, loc2) \wedge T = T_2$$
$$G_2 \Leftrightarrow move\_vehicle(Vec', From', To') = move\_vehicle(truck1, loc1, loc2) \wedge T' = T_2$$

The equality solver shall infer that the first guard is unsatisfiable ($G_1 = 0$), and hence the (deterministic) search can be continued by evaluating the second clause.

A similar effect may happen due to the propagation of disequality information: for instance, when an atom $act(A_1, T_2)$ is derived similar guards are constructed. Suppose that later $\forall\, V', Fr', To'. A_1 \neq move\_vehicle(V', Fr', To')$ is added to the constraint store, then the $\mathcal{E}$-solver infers that the second guard is false.

This 'indexation' effect is very important for problem specifications that use functors. A class of problems for which this is important are planning problems. Instead of compiling the Event Calculus into fluents as done for the blocks world example in Section3.3.3, the Event Calculus can be specified as a general (meta-)theory. In that case, the domain specific information (actions and fluents) is specified as functors for the general predicates.

Combining finite domain and equality suspension

The $\mathcal{A}$system inference rules treat finite domain and equality expressions separately. By reification both can be integrated into one evaluation step. That removes some backtrack points from the $\mathcal{A}$system level. Additionally, it makes the reasoning more dynamic since it breaks the left to right evaluation of the denials.

EXAMPLE 3.26 Consider the denial with three free variables X, Y and Z:

$$\leftarrow X < Y \wedge Z = f(a).$$

First the inference rule F.2 of the $\mathcal{A}$system applies, and, when needed, inference rule E.2 thereafter. The effect is an explicit choice point at the level of the $\mathcal{A}$system search. Because of the least-commitment strategy of the $\mathcal{A}$system, the choice is suspended by reifying $X < Y$. The evaluation scope is in this way limited to the finite domain constraint, ignoring the rest of the formula.

[12] We ignore the body of the clause, since this is not relevant for our explanation.

Another approach treats this denial as the disjunction

$$X \geq Y \vee Z \neq f(a)$$

which can be evaluated by reification (for each expression the appropriate version)

$$X \geq Y \Leftrightarrow_{fd} B_1 \qquad Z \neq f(a) \Leftrightarrow_{eq} B_2 \qquad B_1 \vee B_2$$

Note that this combination is only possible due to a common view on the associated boolean in a reification.

Suppose that the $\mathcal{A}$system derives $Z = f(a)$. This information has no effect in the first approach, since the choice only concerns the variables X and Y. Hence, it may take a long time for the $\mathcal{A}$system to find out that it must be $X \geq Y$. In the combined approach, this information is consumed immediately, forcing $X \geq Y$.

3.5.6 Summary

The $\mathcal{E}$-solver is an efficient constraint solver for equalities *and* disequalities between terms. The development of this solver has been motivated by the need for efficient equality reasoning in order to enhance the efficiency of the $\mathcal{A}$system. For that reason, the $\mathcal{E}$-solver handles conjunctions and disjunctions of (dis)equalities and allows reification of the expressions. To our knowledge, the $\mathcal{E}$-solver is the first equality constraint solver with support for reification.

The improved reasoning abilities of the $\mathcal{E}$-solver have a large impact at several places in the $\mathcal{A}$system. It improves not only the pure equational reasoning, but also it allows to implement the forward propagation in the $\mathcal{A}$system and it is able to remove choice points from the $\mathcal{A}$system search.

3.6 Interaction with the finite domain constraint solver

The sections before have presented an efficient implementation of the necessary components of an abductive logic system (i.e. it is implementation of SLDNFA). In this section we discuss the integration of another constraint solver in the $\mathcal{A}$system: a finite domain constraint solver ($\mathcal{FD}$-solver).

As mentioned in the previous section, the $\mathcal{FD}$-solver can be viewed as an efficient equality solver for problems with a finite domain. An important additional reason for its integration in the $\mathcal{A}$system is the extended reasoning capabilities it offers. The $\mathcal{FD}$-solver gives the $\mathcal{A}$system the ability to do arithmetic reasoning (on integers). Without the finite domain solver, (efficient) arithmetic reasoning is cumbersome since each operation requires either many derivation steps (e.g. when modeling with the successor function symbol) or the specification of the arithmetic operation as an infinite enumeration (e.g. $plus(0,0,0)$, $plus(0,1,1)$, $\ldots$). This tedious and complex formulation is avoided by the integration of the finite domain solver in the $\mathcal{A}$system.

This section discusses a new inference rule (Section 3.6.1), how to construct a constraint store of a 'good quality' (Section 3.6.2), satisfiability checks (Section 3.6.3) and the relation with the $\mathcal{E}$-solver (Section 3.6.4).

3.6.1 Extended inference rules for finite domain expressions

The inference rules F in Section 3.3.2 show general inference rules that apply to any $\mathcal{FD}$-expression. Because we are dealing with *finite domain* constraint expressions, we can add an additional inference rule.

Let $Domain(X)$ denote the domain of a variable X. The domain is *bounded* if its domain has a lower bound and upper bound different from infinity. (Finite domain solvers have a representation for infinite domains.) It follows that all elements from a bounded domain can be enumerated. That property is exploited by the next inference rule in order to evaluate finite domain constaints that contain universally quantified variables. Without loss of generality, we present the inference for one universally quantified variable. $c[Y]$ denotes that the variable Y occurs in the finite domain expression c. $Domain(Y)\downarrow_c$ denotes the domain of Y determined by the constraint c.

F.2b $\forall\, \overline{\mathbf{X}}, \mathbf{Y}. \leftarrow \mathbf{c}[\mathbf{Y}] \wedge \mathbf{Q}$: $vars(c) \setminus \{Y\} \not\subset \overline{X}$ and $Domain(Y)\downarrow_c$ is bounded
$\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{\forall\, \overline{X}.(\leftarrow c \wedge Q)(Y/\overline{d}) \mid d \in Domain(Y)\downarrow_c\}$

Intuitively the inference rule generates a set of denials in which the universally quantified variable is substituted by one of its domain elements, determined by the constraint c. The boundness condition ensures that this process terminates.

EXAMPLE 3.27 Consider the graph coloring problem. It specifies that all vertices must have a color. This is represented by the assertion

$$\forall\ X.vertex(X) \rightarrow \exists\ Y.colorOf(X,Y) \wedge color(Y).$$

which is in denial form

$$\begin{array}{l}\forall\ X. \leftarrow vertex(X) \wedge \neg exists\_color(X) \\ exists\_color(X) \leftarrow colorOf(X,Y) \wedge color(Y).\end{array}$$

Without using the finite domain constraints the vertices are enumerated as

$$\begin{array}{l}vertex(1). \\ vertex(2). \\ \vdots \\ vertex(n).\end{array}$$

This can be expressed more compactly by using a finite domain constraint as

$$vertex(X) \leftarrow X \text{ in } 1..n.$$

When the $\mathcal{A}$system evaluates the above denial, it unfolds the atom $vertex(X)$ by its definition:

$$\forall\ X. \leftarrow X \text{ in } 1..n \wedge \neg exists\_color(X).$$

The extended inference rule F.2b applies since the domain of X is {1,2,. . . ,n} by the constraint expression X in $1..n$. By substitution, e.g. for 2, the denial

$$\leftarrow 2 \text{ in } 1..n \wedge \neg exists\_color(2).$$

is obtained. Now inference rule F.2 applies and the derivation can continue.

The $\mathcal{A}$system inherits at this moment the restriction from the finite domain solver of Sicstus Prolog that only integer values are allowed. This makes the specification for many problem domains less readable because the elements must be mapped into integers. For that reason it is necessary to provide an automated mapping in the future. (In the ECL^iPS^e system [110] the finite domain provides this mapping already.)

The $\mathcal{A}$system is the first abductive system that does this inference [103]. In ACLP, such assertions must be expressed as a recursive predicate that is included in the query.

3.6.2 The quality of the constructed constraint store

One can wonder if the result of the reduction process of an abductive constraint logic specification to a constraint store yields the same or a comparable constraint store as one directly encoded in CLP. If so, then an abductive framework is an excellent candidate to be the modeling language for CSP problems.

For comparing constraint stores, a measure is needed. Since ultimately a solution must be constructed for the constraint problem in the constraint store, a good measure is the amount of pruning that is achieved. The more pruning, the faster the final assignment process will find a solution. As explained in the introductory chapter, finite domain constraint solvers offer different variants of the same constraint with a different amount of pruning. Recall also that the variants with more pruning often are more costly, so that then gain in terms of the removed domain values is undone by the cost to find this out. Therefore, in order to solve the constraint problem fast, a good balance must be found between the pruning and the cost of doing this pruning.

Comparing with hand-written CLP solutions

Comparing the constraint stores generated by the $\mathcal{A}$system and by a CLP-program which use the same binary constraints to express the relations between the problem variables, the quality is almost the same. In Section 3.8, a special variant of abductive theories is presented, where abducibles are restricted to *open functions*. Those specifications can be transformed to an equivalent CLP program. For example, it includes Constraint Satisfaction Problems such as the N-queens, the graph coloring and the traveling salesman problem. For these problems, the quality of the generated constraint stores resulting from equivalent encodings is similar.

However, in practice, the constraints stores generated by a CLP-encoding and by the $\mathcal{A}$system will differ a lot in quality. Basically, the CLP-framework allows to set up a more specific constraint store since the expert has access to the individual variables. Also, the CLP-framework offers special constraints: *global constraints*, that efficiently deal with a particular constraint network. A typical example is the `all_different(Vars)` constraint, that states that all variables of the list `Vars` are mutually distinct. Global constraints yield two benefits: the problem encoding is more compact and the solution construction improves since more values of the domains will be pruned. Unfortunately, these global constraints cannot easily incorporated in the high level specification language of the $\mathcal{A}$system. They resemble to aggregate expressions in the fact that they express a property of a set of objects. The major problem for an integration is that the global constraints require that all information (i.e. the set of objects) are known at call time. In general this is not the case in an $\mathcal{A}$system derivation: by abduction new objects are added to the store and these must be included in the global constraints.

Disjunctions as constraints

One of the main reasons why the $\mathcal{A}$system generated constraint store is of less quality are disjunctions. It is well-known that in general disjunctions have little pruning power. Consequently many finite domain constraint solvers such as Sicstus Prolog have a simple implementation for it: they are passive constraints in the store. Therefore, in the CLP-practice disjunctions are avoided as much as possible by reformulating the constraints using *global constraint* preferentially. However, the simplicity of the disjunction implementation has as consequence that some pruning potential is not exploited. For example, consider the constraint store

$$\{X \text{ in } 1..2, X > 1 \vee X \neq 1\}$$

Although the disjunction implies that X must be different from 1, the Sicstus solver will not eliminate the value 1 from the domain of X.

The $\mathcal{A}$system can treat disjunctions in two ways: either by a backtrack point in the $\mathcal{A}$system search, or by posting the disjunction to the finite domain constraint store. Each possibility may increase or decrease the efficiency of the problem solving. The effect depends entirely on the actual problem. We illustrate both possibilities with a job-shop experiment [103]. In this problem tasks must be sequentially ordered in time. Let $task(T, S, E)$ denote that task T starts at time S and stops at time E. The following denial expresses the sequential ordering of the tasks:

$$\begin{array}{l} \forall\, T1, S1, E1, T2, S2, E2. \\ \qquad \leftarrow task(T1, S1, E1) \wedge task(T2, S2, E2) \wedge \neg sequential(S1, E1, S2, E2). \end{array}$$

1. The representation for backtracking by the $\mathcal{A}$system

$$\begin{array}{l} sequential(S1, E1, S2, E2) \leftarrow E1 \leq S2. \\ sequential(S1, E1, S2, E2) \leftarrow E2 \leq S1. \end{array}$$

   Using this representation, the $\mathcal{A}$system will create a backtrack point for $sequential/4$ atoms. This execution constructs simple constraint stores which can easily be evaluated. However, many of these constraint stores are unsatisfiable. Before the $\mathcal{A}$system finds a solution it will have backtracked many times. As backtracking is costly and the detection of inconsistency is not always simple, this approach is often undesirable.

2. The representation for backtracking by the finite domain constraint solver

$$sequential(S1, E1, S2, E2) \leftarrow (E1 \leq S2 \vee E2 \leq S1).$$

   Here both clauses are combined into one disjunctive finite domain constraint expression. That removes the backtracking from the $\mathcal{A}$system level, and shifts all backtracking to the finite domain constraint solver. As explained, the disjunctions do not contribute much to the pruning in the constraint store. Also, a constraint store with a lot of disjunction may be hard to evaluate for the $\mathcal{FD}$-solver.

There is a trade-off between both approaches. Our experiments learned that it is difficult to judge in advance which execution model is better. The choice is up to the user.

We have added one improvement to the $\mathcal{A}$system w.r.t. the disjunctions. We will check each disjunct if it is already satisfied. This can be easily done for ground disjuncts. When one satisfied disjunct is found, the disjunction is surely true and hence is not added to the constraint store. That keeps the constraint store smaller.

Redundant constraints

Since the $\mathcal{A}$system derives automatically constraints from a high level specification, it happens that syntactic equal or equivalent constraints are added to the constraint store. Such redundancy in the constraint store slows down the search process for a solution because each constraint is verified by the constraint solver. This does not happen in hand-written constraint logic programs because the programmer has taken care of that. This problem is still open, and is left for future work.

3.6.3 Incomplete solvers

A finite domain constraint solver has the property of being incomplete. The incompleteness has the effect that during the constraint setup phase the constraint store is only kept locally consistent. To find out if a constraint store is overall consistent, one has to construct a solution. The incompleteness is easily shown by these variants of the same program[13]:

```
p1 :- X in 1..10, Y in 1..10, X#=Y, X#\=Y.
p2 :- X in 1..10, Y in 1..10, X#=Y, X#\=Y, labeling([],[X,Y]).
```

Program p1 will succeed while p2 will fail. In program p2, the labeling will enumerate all possible assignments of X and Y. As none of them lead to a solution the program fails.

When a human encodes a problem directly as a Constraint Logic program, the situation is acceptable. Mostly, the logic program (in Prolog) is only used to manipulate data structures in order to set up the correct constraints. The actual problem itself is encoded in the constraints. Therefore only after all constraints are posted, the search for a solution can start.

[13]We use the Sicstus Prolog syntax; For this example, $\mathcal{E}$-solver will give the intended behavior. It is the simplest example to show the phenomenon of incompleteness.

For reasoning systems such as the $\mathcal{A}$system, that generate the constraint store in an automated way, and rely on the consistency of the store for its reasoning, the above sketched procedure is troublesome. Consider the following simplified $\mathcal{A}$system-derivation:

$$\begin{array}{l} \mathcal{S}_0 = (\mathcal{G}_0, \mathcal{FD}_0) = (\{X = Y, X \neq Y\} \cup \mathcal{G}', \emptyset) \\ \rightarrow \mathcal{S}_1 = (\mathcal{G}_1, \mathcal{FD}_1) = (\{X \neq Y\} \cup \mathcal{G}', \{X = Y\}) \\ \rightarrow \mathcal{S}_2 = (\mathcal{G}_2, \mathcal{FD}_2) = (\mathcal{G}', \{X = Y, X \neq Y\}) \\ \rightarrow \ldots \end{array}$$

where $\mathcal{G}$ is the goal stack and $\mathcal{FD}$ the finite domain constraint store. Initially, the goal stack contains two constraints between the free variables X and Y. In step 1, the first one is selected and added to the constraint store. Step 2 does the same for the other constraint. Obviously the store $\mathcal{FD}$ is inconsistent. But the finite domain solver shall not detect this due to its incompleteness. Thus, it will not prevent the $\mathcal{A}$system to continue its search along the (inconsistent) branch.

The latter should be avoided as much as possible. One solution is the execution of a satisfiability check each time after updating the constraint store. That makes the constraint solver complete. In general, it is infeasible: (1) the computational complexity of the constraint stores is in general NP, hence checking the satisfiability is a costly operation. In the worst case of a unsatisfiable constraint store, the whole search space is traversed. (2) If a solution state contains N constraints then the constraint store is checked at least N times during the $\mathcal{A}$system-derivation. (At least N, because also failing branches may have to be explored.) In the worst case the consistency check requires finding N times a solution for a NP problem.

Another drastic solution is being optimistic: do nothing, hoping that unsatisfiable constraint stores are rare. For many experiments this is a good option. But it depends on the application. For example, in Section 3.9 we presents experimental results for two AI planning problems. The $\mathcal{A}$system constructs satisfiable intermediate constraints stores, for the blocks world while for the logistics planning it suffers quite a lot from unsatisfiable intermediate constraints. Without a satisfiability check, the $\mathcal{A}$system is unable to solve any reasonable large problem instantiation of that problem.

Since in general none of both options is a good option, we have studied the problem in more detail. That has resulted in several different strategies that lie in between both options. All proposals simplify the consistency check, hoping that this suffices to find out the undetected inconsistencies earlier than the optimistic one without the cost of a complete check.

- *time-out*
  The time-out solution gives a limited amount of time to the satisfiability check. If the check is completed within this time limit, then the result of the check is used. An unsatisfiable store invokes backtracking, for a satisfiable store the search is continued. A time-out is less informative: the constraint

store can as well be satisfiable as unsatisfiable. Therefore, the $\mathcal{A}$system search is continued, but the time limit is increased. That grants the next check more time to find a solution. If the number of consecutive time-outs is over a certain threshold, then the $\mathcal{A}$system will cut the search along that branch and backtracks. Obviously this will also cut good branches, hence the use of the time-out solution makes the $\mathcal{A}$system incomplete.

A variant (which is implemented in the $\mathcal{A}$system) is not to backtrack to the previous choice point, but to the choice point which has hit first the time limit. This version takes the experimental observation into account that if the satisfiability checking lasts long, it is mostly unsatisfiable. Another indication is that usually the check before the first time-out was easy, which means that the addition of some constraints makes the constraint store suddenly harder to evaluate.

- incremental labeling
  In the time-out solution, it is implicitly assumed that subsequent constraint stores are not connected with each other. Each satisfiability check starts from zero to find a solution. Clearly this is not the case. Each constraint store is the extension of the previous one by the addition with some constraints. This allows to design an incremental check, i.e. the check will not explore the search space a previous check traversed.

  Two possible designs can be made.

  1. Suppose S is a solution for a list of variables Var using the labeling strategy lab, computed for the previous constraint store. This information (S, Var and lab) is stored, and retrieved for a new check of the next (extended) constraint store. The search for a solution for the new constraint store is started from the old solution S, as if the search for the new constraint store had already explored some part of the search space.

     The advantage of this method is that the constraint store is not affected and hence does not become more complex. Also, when reaching a potential solution state, it is known that the last constructed solution must be close to a solution assignment. In the best case, it is the solution.

     The drawback is that dynamic labeling strategies, e.g. finite fail (ff), that base their selection of the next variable to label on the state of the constraint store, cannot be used since they lead to different selection orders for different constraint stores. As these dynamic strategies are often more efficient, the restriction to fixed strategies, e.g. left-to-right, implies also a lower efficiency of the intermediate check. Moreover it may be that an incremental labeling satisfiability check using a fixed strategy is slower than a satisfiability check that start from scratch but uses a more intelligent labeling strategy.

2. The search space that is traversed to find a solution is computed and removed by adding extra constraints to the store. A next check of the constraint store cannot explore that part any more.

   Cutting the search space by adding constraints has the advantage that the search space is effectively reduced and therefore it increases the amount of pruning the finite domain constraint solver is able to perform. Another advantage is that dynamic labeling strategies can be applied. In that case, the search space that is removed by each check, is not necessarily connected with each other.

   The difference between the use of a dynamic and a fixed strategy can be visualized as the cutting of a sheet of paper (which visualizes the whole search space). Using a fixed strategy will remove connected parts of the sheet, e.g. starting from the left upper corner. The first check (cut) removes the left upper corner, the second, removes an additional part at the same corner, etc. A dynamic strategy, however, cuts holes in the paper: each check removes somewhere a piece of paper.

   The main disadvantage of this method is the increasing size of the constraint store. The more constraints have to be verified, the longer it takes for the finite domain solver to find a solution. For large executions this approach can add so many constraints to the constraint store that it results in a degradation of the performance of the solution construction instead of a speed up.

- Partitioning the constraint store.
  The constraints and variables in the constraint store do not form a random problem, but are the result of a reduction process from a high level specification. This satisfiability check tries to exploit this information to obtain a problem dependent and more effective labeling strategy.

  Observe in the first place that the automated derivation of a labeling strategy is possible because there exists another (high level) specification of the problem. This is more complicated and almost impossible for a pure CLP encoding since the high level knowledge is hidden in the meaning the human expert assigns to the variables. That information is not available for the CLP system, hence the CLP system can base its labeling strategy not on such semantic information.

  We experimented with this idea by exploiting one structuring component: type information. Types can be regarded a (finite) set of values. In a well-specified problem, constraints express relations between variables of the same type. Inter-type constraints are only possible via disjunctions, in which each disjunct is an expression consisting of variables of the same type.

This allows to split the set of problem variables in several subsets, one for each type. Instead of performing a search for a solution for all problem variables at once, this strategy looks independently for a solution for each subset. Since each check consists of few variables, the total time to construct a solution is expected to be less. (Of course this depends on the size of the subsets.) This check is not complete, i.e. it might be that the constraint store is unsatisfiable but not detected by this check. But it is sufficient: when one of the checks fails the constraint store is unsatisfiable.

The typing is one splitting criterion. Each partitioning of the problem variables has the (potential) effect to reduce the checking time.

A variant is the following. Instead checking a semantically connected set of variables, the $\mathcal{A}$system will check the constraint store for the variables occurring in the newly added constraint. (Variables occurring in the same (disjunctive) constraint may belong e.g. to different types according to the previous criterion.) It gives many small checks since the satisfiability check will be executed immediately after the addition of the constraint to the store.

Each of the presented strategies reduces the amount of work for the intermediate constraint store checks. Since they exploit independent properties, it is possible to combine them. The best performance we found in the combination of the incomplete satisfiability check for the constraints (the last discussed one) combined with a reduction of the search space by adding additional constraints. Unfortunately, none of these (combined) approaches is satisfactory in general. For many large problem instances, they are not sufficient and new strategies are needed.

3.6.4 Interaction with the $\mathcal{E}$-solver

During an $\mathcal{A}$system derivation, finite domain constraints and $\mathcal{E}$-constraints are imposed on the same variables. For example, the reuse of abducibles may impose the $\mathcal{E}$-formula for an abducible atom e.g. $act(A, B)$

$$\bigvee_{act(X,T)\in\Delta} a(A,B) = a(X,T)$$

where $act/2$ denotes an action for an AI planning problem. The arguments of the abducible $act/2$ also participate in finite domain constraints. For example, the second argument of abducible is a time point, which is expressed by the assertion

$$\forall\ X, T.act(X,T) \rightarrow T \text{ in } 1..10.$$

Here, the time points are restricted between 1 and 10.

Both solvers can be regarded as completely independent; both will ensure that in a solution state all constraints will be satisfied. But this neglects the fact that both have different and distinct information about the same variable. E.g. when the $\mathcal{E}$-solver derives that T should differ from 5, this is valuable information for the $\mathcal{FD}$-solver, since it can remove 5 from the domain of the variable T. The exchange of information can improve the efficiency of the $\mathcal{A}$system.

Currently, the implemented information exchange is asymmetric. The $\mathcal{E}$-solver informs the $\mathcal{FD}$-solver about each disequality it has in its store, if the variables are finite domain variables. In the other direction, there is a passive information exchange. The $\mathcal{E}$-solver uses in its inference the current domain of a variable to decide whether an equality involving a finite domain variable can be true. There is no active version implemented, in which the $\mathcal{FD}$-solver invokes the constraint check when the domain of a variable changes. Of course, when the domain becomes a singleton, the $\mathcal{E}$-store is checked.

The information exchange can be improved for one particular constraint: the reuse of abducibles. Without loss of generality, consider the reuse of the abducible atom $a(X)$, where X is also a finite domain variable:

$$\bigvee_{a(Y)\in\Delta} a(X) = a(Y)$$

which is equivalent with

$$X = Y_1 \vee \ldots \vee X = Y_n$$

for $\Delta = \{a(Y_1), \ldots a(Y_n)\}$. This allows us to derive the necessary consistency condition that

$$dom(X) \subseteq dom(Y_1) \cup \cdots \cup dom(Y_n)$$

where $dom(X)$ denotes the domain of variable X. When this condition is violated the reuse formula is surely violated.

We have implemented this idea as the constraint `in_domain(X,Vars)`, denoting that the domain of X must be a subset of the union of the domains of Vars.

EXAMPLE 3.28 The effect of this condition can be observed when solving an instance of the blocks world planning domain. For a 10-block problem[14], the $\mathcal{A}$system derives that there must be two actions

$$\begin{array}{c} move(8, 4, T_1) \\ move(8, 0, T_2) \end{array}$$

The first moves block 8 on block 4 and the second moves 8 on the table (0). In order to satisfy all preconditions the $\mathcal{A}$system derives also other actions $move(8, Y, T)$. It can unify with any of both. In the implementation without the `in_domain(X,Vars)`, the $\mathcal{A}$system derives that the domain of Y is `(0..6)v(9..10)`, and that of T is `1..18`. The application of the `in_domain(X,Vars)` restriction, will result in a reduction of the domain of Y to `{0}v{4}`. Such domain reductions are obviously beneficial.

[14]This is problem 10-0 from the AIPS planning competition [5]

3.6.5 Summary and contributions

The integration of a finite domain solver in the $\mathcal{A}$system yields two main advantages: arithmetic reasoning over integers and an improvement of the efficiency. The latter is due to the shift of backtrack points from the $\mathcal{A}$system level to the finite domain store.

By this integration, the $\mathcal{A}$system is well equipped to deal with the class of problems for which the finite domain solver is designed: finite domain constraint satisfaction problems. Also for other problem domains such as AI planning, the integration of the finite domain constraint domain is beneficial. When experimenting with the latter problems, we observed that due to the incompleteness of the finite domain constraint solver, some inconsistent stores were not detected. Therefore we have designed several satisfiability checks in order to detect these situations fast with a minimal overhead cost. By these checks, the $\mathcal{A}$system can avoid the exploration of (a part of) the branches which finite domain constraint store is inconsistent.

The constraint stores that are considered, are generated automatically. These have different characteristics than the hand-engineered constraints, to which the finite domain constraint solver is oriented. In particular, the $\mathcal{A}$system will generate many redundant and weakly pruning constraints, i.e. disjunctions. We present some simple, but restricted, solutions to improve the generated constraint stores. The design of a complete layer in between the $\mathcal{A}$system and the finite domain solver that takes care of all those issues is for the future.

Finally, we have presented the integration of the $\mathcal{E}$-solver and the finite domain solver. At this moment the information flow is one-directional: from the $\mathcal{E}$-solver to the finite domain solver. We also have presented an improvement of the pruning for a disjunction consisting of equalities. Although it is just for one particular form of a disjunctive constraint, the impact is high since the constraint will occur frequently during an $\mathcal{A}$system derivation.

3.7 Reification

The implementation of the non-deterministic $\mathcal{A}$system inference rules is based on reification. Reification is a technique which associates a boolean variable B with a formula F denoting the truth value of the formula w.r.t. a given theory $\mathcal{C}$ (i.e. constraint store). The reified expression is denoted as $F \Leftrightarrow B$. Procedurally, if F is entailed **false** (**true**) w.r.t. $\mathcal{C}$, B is 0 (1), and if B is assigned 0 (1) then $\neg F$ (C) is added to the constraint store $\mathcal{C}$.

Using this technique, we have designed a general mechanism for active participation of choice points in the search process. In short, (a part of) each disjunct of a choice point is reified and the set of associated booleans is used to suspend the execution of the choice point. When certain conditions are fullfilled, e.g. all associated booleans are false, a suspended choice point is awakened and its execution is resumed. For more information on this, we refer to Section 3.4.

Originally, reification was only available for a small number of finite domain constraint expressions. In order to make this technique also available for other expressions, we have (re)implemented the reification ourselves: both for finite domain constraints and equality reasoning. (See sections 3.6 and 3.5 for more information.)

In this section, we focus on a general improvement with respect to the implementation of reification: namely the removal of a reified expression from the constraint store. Note that the removal of constraints is not supported by a constraint solver in general so we cannot reuse any existing method. Removing reified expressions from a constraint store is a necessary operation in the $\mathcal{A}$system since otherwise the constraint stores become crowded with irrelevant constraints. It happens in an $\mathcal{A}$system derivation that the system selects a suspended choice point and creates a backtrack point (i.e. it makes an explicit choice) for it. At this point the reified expressions that have been added to the constraint stores to observe the state, have become irrelevant. Hence, they can be discarded and retracted from the constraint store. If they are left in the constraint store, the constraint solver will continue to reason on them. As a consequence, these irrelevant reified expressions consume valuable space and computation time.

This section presents an extension to the classical reification reasoning so that these expressions can be easily removed from the constraint store. Note that it is impossible to remove the reified constraints by assigning **true** (1) to the boolean corresponding to the selected branch and **false** (0) to the booleans of the other branches, because that will insert the associated constraints in the constraint store. Since these reified constraints from the different branches of one choice point, do not exclude each other, this is an invalid operation.

Although our initial motivation is to resolve the above problem, this problem occurs also in another context: the evaluation of disjunctions. The finite domain constraint solver provided by Sicstus, evaluates disjunctions by reifying the disjuncts. For example, consider the disjunction

$$(x < 4 \vee y < z)$$

The solver expands this disjunction to the next three expressions

$$(B_1 \vee B_2) \qquad x < 4 \Leftrightarrow B_1 \qquad y < z \Leftrightarrow B_2$$

When $x < 3$ is added to the constraint store, the constraint solver infers that B_1 is 1 and hence the disjunction $B_1 \vee B_2$ is satisfied. This removes the first two expressions form the store, but leaves the third constraint untouched. The third has become a ghost constraint, of which the final value is irrelevant. If no special inference is foreseen for such cases, the use of reification enlarges the constraint store with ghost constraints. As mentioned, these constraints hinder the normal inference instead of contributing to search space reduction.

Our solution for this problem is based on the extension of the set values for the booleans with a new value 2 denoting *obsolete*. When a boolean gets assigned this value, the associated reified expression can safely be removed from the constraint store. Formally, for a basic constraint F which is reified $F \Leftrightarrow B$, the following rules apply

- if B gets the value 0, then add $\neg F$ to the constraint store.
- if B gets the value 1, then add F to the constraint store.
- if B gets the value 2, then $F \Leftrightarrow B$ is removed form the constraint store.
- if F is entailed by the constraint store, then B is assigned 1.
- if F is disentailed by the constraint store, then B is assigned 0.

More complex constraints which are compositions of basic constraints by logical connectives, can be handled by reifying all basic constraints and then transforming the composition of booleans to a number of reified conjunctions and disjunctions. (As shown in the example above.) This requires the definition of new truth-tables for reified conjunctions and disjunctions. The symmetric cases ($B_1 > B_2$) are omitted.

- A reified disjunction

$B_1 \vee B_2 \Leftrightarrow B$		
B	B1	B2
0	0	0
1	0	1
1	1	1
1	1	2
2	0	2
2	2	2

The table shows that it is sufficient that one disjunct is true to make the whole disjunction true (row 4). The values of the other disjuncts do not matter. In the case that the complete disjunction has to be removed (B gets value 2, rows 5-6) some disjuncts may already be false while other are still undetermined and have to be removed as well.

- A reified conjunction

$B_1 \wedge B_2 \Leftrightarrow B$		
B	B1	B2
0	0	0
0	0	1
0	0	2
1	1	1
2	1	2
2	2	2

This table is the dual of the previous one. When one conjunct is derived to be false, all other conjuncts can be discarded (row 3). By removing the complete conjunction (B is assigned 2, rows 5-6), some of the conjuncts may already be true, while others are still undetermined. These must be discarded.

Those truth tables are implemented in the reification procedures. The key idea in the implementation is to assign the maximal possible value to a boolean B_i in the conjunction (disjunction) when the associated boolean B is 0 (1). This ensures that when the conjunction (disjunction) is false (true), all conjuncts (disjuncts) that have a undetermined value, get the value 2, having the effect that they are removed from the store. For example, reconsider these reified expressions

$$(B_1 \vee B_2) \Leftrightarrow B \qquad x < 4 \Leftrightarrow B_1 \qquad y < z \Leftrightarrow B_2$$

When now $x < 3$ is added to the store, the solver will infer that B_1 is 1, and hence B is 1. The latter has as effect that B_2 must get a value. Since the domain of B_2 is $\{0,1,2\}$, its maximal value is 2. Assigning 2 to B_2 will remove the last reified expression from the constrain store, which is our intended behavior. For the finite domain constraint solver, this has been implemented as a new (global) constraint using the extension interface of the solver. In the $\mathcal{E}$-solver, it has been integrated in the structure of the solver. The encoding for the finite domain constraint solver is found in appendix A.3.

3.8 Open Functions

The modeling of a constraint satisfaction problem in Constraint Logic Programming involves as first step the creation of the necessary variables. These variables represent the unknowns of the problem instance; more precisely they represent the value of an function which is left implicit. This function gives the meaning to the variables.

[61] introduces *open functions* which allow to make the implicit meaning of variables explicit. Moreover, it eliminates a 'procedural' aspect of the representation of a Constraint Satisfaction Problem in Constraint Logic Programming: namely that a set of variables must be given in advance.

DEFINITION 3.28 An *open function* is a function, having a domain $\mathcal{D}$ and range $\mathcal{R}$ but of which the actual function is unknown.

EXAMPLE 3.29 (GRAPH COLORING) A graph consists of a set of vertices $\mathcal{V} = \{v1, v2, \ldots\}$ and a set of edges {edge(v1,v2),...}. The goal is to find a coloring of the graph with the colors $\mathcal{C}$ ={red,blue,yellow,...} such that each vertex has a different color than the vertices it is directly connected to. To represent the problem, we define the open function $color : \mathcal{V} \mapsto \mathcal{C}$, denoting the color of a vertex. The problem is then expressed as the constraint

$$\forall\ X, Y.edge(X, Y) \rightarrow color(X) \neq color(Y).$$

Note that a set of variables is not needed anymore. The function description takes care of that. The encoding leads also to a more precise definition of the task to be solved. The goal is to find a satisfying function description for the open functions used in the specification. One way is to construct a full enumerated description, that for each domain element gives the corresponding function value. That exactly corresponds to looking for a satisfying assignment of the variables. As the example 3.29 illustrates the resulting problem description is almost self explanatory.

In the context of Abductive Logic Programming, open functions can be regarded as a special class of abducibles [103]. An open function $f : \mathcal{D} \mapsto \mathcal{R}$ is represented by the binary abducible predicate $f/2$ for which the following axioms hold:

$$\forall\ X, Y_1, Y_2.f(X, Y_1) \wedge f(X, Y_2) \rightarrow Y_1 = Y_2 \tag{3.3}$$

$$\forall\ X.X \in \mathcal{D} \rightarrow \exists\ Y.f(X, Y). \tag{3.4}$$

$$\forall\ X, Y.f(X, Y) \rightarrow Y \in \mathcal{R}. \tag{3.5}$$

EXAMPLE 3.30 (EXAMPLE 3.29 CONTINUED) Let $vertex/1$ denote the set of vertices and $color/1$ denote the colors. The open function encoding of the graph coloring problem is encoded as the abductive logic theory ($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$):

- $$\mathcal{P} = \left\{ \begin{array}{l} vertex(v1). \\ vertex(v2). \\ edge(v1, v2). \\ color(red). \\ color(blue). \\ color(yellow). \end{array} \right\}$$

- $\mathcal{A} = \{color(\cdot, \cdot)\}$

- $\mathcal{IC} =$

$$\left\{ \begin{array}{l} \forall\ X, Y_1, Y_2.\ color(X, Y_1) \wedge color(X, Y_2) \rightarrow Y_1 = Y_2 \\ \forall\ X.\ vertex(X) \rightarrow \exists\ Y. color(X, Y). \\ \forall\ X, Y.\ color(X, Y) \rightarrow color(Y). \\ \forall\ X, Y.\ edge(X, Y) \rightarrow \exists\ CX, CY.\ color(X, CX) \wedge color(Y, CY) \wedge CX \neq CY. \end{array} \right\}$$

The last integrity constraint is usually written as

$$\forall\ X, Y, CX, CY. edge(X, Y) \wedge color(X, CX) \wedge color(Y, CY) \rightarrow CX \neq CY.$$

This formula is only equivalent with the previous one if $color/2$ is a function.

Although a simple transformation allows to deal with open functions, it has computational advantages to implement a special treatment. For open functions the set of corresponding abducibles can be constructed immediately. This avoids the verification of the function condition 3.3. After the setup, no new abductions of an atom are allowed, and thus only reuse of abducibles can happen. That simplifies the inference rules A.1 and A.2 for abducibles (see Section 3.3.2). In rule A.1 the branch of a new hypothesis can be removed. Also inference rule A.2 is simplified because the set of abducibles is complete and thus the selected denial $\forall\ \overline{X}. \leftarrow a(\overline{u}) \wedge Q$ must not be stored for later consistency checks. The simplified variants of these inference rules for open functions are:

OF.1 $\mathbf{of(X, Y)} \wedge \mathbf{Q}$:
SELECT one of the $of(s, t) \in \Delta_i$ such that
$\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{\mathrm{Q}\} \cup \{X = s \wedge Y = t\}$

OF.2 $\forall\ \overline{\mathbf{Z}}. \leftarrow \mathbf{of(X, Y)} \wedge \mathbf{Q}$:
$\mathcal{G}_{i+1} = \mathcal{G}_i^- \cup \{\forall\ \overline{Z}. \leftarrow X = s_i \wedge Y = t_i \wedge Q | of(s_i, t_i) \in \Delta_i\}$

In order to get the reasoning correct, the $\mathcal{A}$system has to start from the initial state $\mathcal{S}_0 = \{\{\mathrm{Q}\} \cup \mathcal{IC}, \mathcal{ST}^{of}\}$ where the open functions are already expanded available in

Δ. $\mathcal{ST}^{of}$ is the store which can be reached as the result of the $\mathcal{A}$system derivation for the following program w.r.t. the query **true**:

$$\mathcal{P} = \{exists_f(X) \leftarrow f(X,Y) \wedge range_f(Y) | f \text{ is an open function } \}$$
$$\mathcal{IC} = \{\forall\ X. \leftarrow dom_f(X) \wedge \neg exists_f(X) | f \text{ is an open function}\}$$
$$\mathcal{A} = \{of_f(\cdot,\cdot) | f \text{ is a open function}\}$$

in which $dom_f(X)$ ($range_f(Y)$) is a defined predicate encoding the domain (range) of the open function f.

Solving the graph coloring problem with this extended version of the $\mathcal{A}$system, yields still a one step transformation reducing the high level specification into a finite domain constraint store. Compared to the not extended version, the open functions variant also consumes less memory because the constraints (and it derivatives) must not be stored. This improves the scalability of the $\mathcal{A}$system.

3.9 Experimental verification and comparison

In the previous sections, we have discussed the theoretical foundations and many implementational issues that have impact on the behavior and the solving efficiency of the $\mathcal{A}$system. Here, we present an experimental verification of the $\mathcal{A}$system. We have decided not to present a large benchmark consisting of many problems. Instead, we have selected a limited number of problems to show some specific properties. Since these problems are representatives for one problem class, we can draw some general conclusions from this small experiments.

To position the $\mathcal{A}$system w.r.t. related systems, we have selected one system for each related computational paradigm. The finite domain solver of Sicstus Prolog [66], denoted in the following as $\mathcal{FD}$-solver, is the representative for Constraint Logic Programming [191, 13]. This is the same solver that the $\mathcal{A}$system uses. As representative for Answer Set Programming [31], we have selected DLV [105, 112]. Since our set of problems contains planning problems, we also have selected an efficient strips planner: the heuristic planner Fast Forward (FF) [145].

Next to the experimental verification, we discuss also some more high level topics. We compare the application area of the reasoning systems and the differences in representation between some systems. Finally we discuss some limitations of the $\mathcal{A}$system.

All timings, presented in this section and elsewhere in the thesis, are computed on a Linux machine, 800Mhz, 512MB.

3.9.1 CSP problems

Constraint Satisfaction Problems over a finite domain form an important class of problems. For example, this class includes most of the scheduling problems such as exam timetabling and job shop scheduling. Finite domain constraint solvers are designed especially for these problems. Since the $\mathcal{A}$system has an integrated finite domain constraint solver, the expectation is that these problems can be efficiently solved.

For our experiments we have chosen two easily scalable problems: the *N-queens* problem and the *Pigeon hole* problem. The first is a satisfiable problem and the second is unsatisfiable.

The systems which are included in this comparison are the $\mathcal{A}$system, a pure CLP solution using the $\mathcal{FD}$-solver and DLV. The search strategy that has been used by the $\mathcal{FD}$-solver, both in the $\mathcal{A}$system as in the pure CLP encoding, is *first fail most constrained* (ffc). DLV offers no possibility to change the search strategy.

N-queens problem

The problem is to position N queens on a N by N chess board so that no queen is threatened by another queen. This problem is evaluated for problem sizes from

10 until 100 queens. The columns denoted with *model* present the time needed to compute a solution. The column titled *setup* present the time that is needed to construct the constraint store (in case of the $\mathcal{A}$system and the $\mathcal{FD}$-solver) or to ground the problem (DLV). The blank entries denotes not executed experiments due to too long computation times.

N-queens						
size	$\mathcal{A}$system		$\mathcal{FD}$-solver		DLV	
	setup	model	setup	model	setup	model
10	30ms	0ms	0ms	0ms	123ms	16ms
20	70ms	10ms	30ms	10ms	1s 232ms	490ms
30	160ms	40ms	30ms	20ms	5s 842ms	4s 376ms
40	300ms	80ms	60ms	30ms	27s 991ms	22s 852ms
50	460ms	100ms	100ms	270ms	54s 754ms	95s 601ms
60	670ms	200ms	130ms	310ms		
70	880ms	270ms	170ms	50ms		
80	1s 180ms	380ms	240ms	70ms		
90	1s 490ms	460ms	300ms	520ms		
100	1s 840ms	610ms	340ms	110ms		

Pigeon-hole problem

Suppose we have N pigeons and M holes. Each hole has just space for one pigeon. The problem is to assign each pigeon a hole to sleep in for the night. When there is one hole less than the number of pigeons (M = N-1), the problem becomes unsatisfiable.

The size in the problem instances refers to the number of pigeons. Because the setup times are really small here, we did not include them in the table.

Pigeon hole			
size	$\mathcal{A}$system	$\mathcal{FD}$-solver	dlv
4	20ms	20ms	6ms
5	10ms	10ms	41ms
6	10ms	10ms	68ms
7	40ms	40ms	380ms
8	190m	190m	2s 420ms
9	1s 450ms	1s 450ms	20s 250ms
10	12s 680ms	12s 680ms	

Discussion of the results

The computational results show that the $\mathcal{A}$system results follow closely the $\mathcal{FD}$-solver results. In general, the $\mathcal{A}$system requires more time to setup the constraint store than the $\mathcal{FD}$-solver. This is to be expected. A comparison of the setup times shows that the overhead of the $\mathcal{A}$system is about a constant factor of 5 for the N-queens problem, which is acceptable.

Note that the timings for the $\mathcal{A}$system are the result of a very particular, yet important, behavior on these problems. It constructs one single constraint store without creating any backtrack point in the $\mathcal{A}$system search. Hence, solving a CSP problem with the $\mathcal{A}$system has the same behavior as a standard CLP(FD) program. Using the $\mathcal{A}$system has two advantages: the problem can be specified in a more declarative way and the expert has not to spend time in the design of the right data structures and corresponding formulations of the constraints. However, when speed matters, CLP(FD) offers more possibilities. The expert can design a domain specific search strategy in CLP(FD) since he has access to the constraint variables. Also, the $\mathcal{FD}$-solver offers global constraints, which are optimized algorithms for specific constraint networks. For example, the *pigeon hole* problem constructs a constraint network of disequalities in which each variable must be different from any other variable. The constraint network is covered by the `all_different(Vars)` global constraint. This global constraint finds out the unsatisfiability of the *pigeon hole* problem before the labeling of the constraint store starts. The use of the `all_different/1` constraint allows to solve very large instances of the *pigeon hole* problem, which are impossible to solve by the simple approach.

Despite these 'computational' drawbacks, the $\mathcal{A}$system can be viewed as an excellent platform for experimenting with CSP problems. It offers the modeling support which is lacking in CLP(FD) systems: a flexible declarative modeling language that allows the expert to verify its problem specification without the need for designing data structures and writing program code.

There also is a third system in this evaluation: DLV. The results for this system are worse than those of the $\mathcal{A}$system and the $\mathcal{FD}$-solver. A combination of several factors explain this result. First, DLV reasons on propositional theories. To solve the problems it grounds the specifications. In this process, each variable is replaced by its domain values, generating in this way all possible assignments. Although the grounder is able to remove a large number of inconsistent ground formulas immediately, the DLV model generator must handle still big propositional theories. On the contrary, the constraint based approach keeps the internal representation very compact. Secondly, the selected problems are typical problems where constraint reasoning is very efficient. Each other method for solving such problems must be very intelligent before it can meet or do better than constraint reasoning. Thirdly, DLV applies one single search strategy to all problems. Constraint solvers offer the ability to adapt the search strategy to the problem. For example, if another than the ffc strategy, e.g. left-to-right, is used for solving the N-queens problem, then the $\mathcal{FD}$-solver needs much more time to find a solution.

3.9.2 Comparison of the problem specifications

Before discussing the results for the planning problems, we discuss in short the differences in the problem specifications. The example problem is the N-queens problem.

The $\mathcal{A}$system and DLV have a similar problem specification. We also have included the representation of sModels [208, 240], another ASP system, because it has an alternative representation that is of interest. Let the predicate $position(Q, P)$ denote the position of the queen Q on column Q and row P. All three specifications share the constraint that two queens cannot be in a position so that one threatens the other.

$$\begin{array}{l} \forall \ \leftarrow position(Q_1, P_1) \wedge position(Q_1, P_2) \wedge Q_1 \neq Q_2 \wedge \neg notattacked(Q_1, P_1, Q_2, P_2). \\ notattacked(Q_1, P_1, Q_2, P_2) \leftarrow P_1 \neq P_2 \wedge \\ \qquad P_1 + Q_1 \neq P_2 + Q_2 \wedge P_1 - Q_1 \neq P_2 - Q_2. \end{array}$$

The difference are in the requirement that there must be a queen at each column. Let $queen/1$ denote the queens and $position/1$ denote the positions a queen can take.

- The $\mathcal{A}$system formulates the above assertion directly (using the Lloyd-Topor transformation to obtain the next denial and program rule).

$$\begin{array}{l} \forall\ Q. \leftarrow queen(Q) \wedge \neg exists\_queen(Q). \\ exists\_queen(Q) \leftarrow position(Q, P) \wedge position(P). \end{array}$$

- DLV uses a disjunctive program rule that expresses that a queen can be either on the first row, or the second, etc. For a 4 by 4 board the disjunctive rule is

$$position(Q, 1) \vee position(Q, 2) \vee position(Q, 3) \vee position(Q, 4) \leftarrow queen(Q).$$

- sModels uses two range restricted rules. The first expresses that a queen Q can be on row P if it is not believed that the queen is on another position (denoted with $position^*(Q, P)$). The second rule expresses the inverse believe.

$$\begin{array}{l} position(Q, P) \leftarrow queen(Q) \wedge position(P) \wedge \neg position^*(Q, P). \\ position^*(Q, P) \leftarrow queen(Q) \wedge position(P) \wedge \neg position(Q, P). \end{array}$$

  Note that this rule can be used also in DLV.

The differences between those three specifications are small, but important. All three express that there is incomplete knowledge for the predicate $position(Q, P)$. The $\mathcal{A}$system uses a declaration (abducible), DLV a disjunction and sModels negation. In my opinion, the $\mathcal{A}$system approach is the most attractive one, and the use of negation the least.

The most complex formulation and less declarative encoding of the N-queens problem is that of CLP(FD). As the code below shows, the encoding is much longer and it requires the understanding of the different data structures and constraints. This again argues in favor of our statement that the $\mathcal{A}$system can be used as a declarative modeling layer for CLP(FD).

```
nqueens(N,Vars) :-
   % construct a list of variables in which
   % the ith variable denotes the position
   % of the ith queen on the board
   length(Vars,N),
   % the postions of all queens lie in 1..N
   domain(Vars,1,N),
   % set up the constraints
   post_constraints(Vars,N),
   % find a solution
   labeling([ffc],Vars).

post_constraints([],N).
post_constraints([QX|Qs],N):-
        N1 is N-1,
        not_attack(Qs,QX,N,N1),
        post_constraints(Qs,N1).

not_attack([],QX,N,N1).
not_attack([QY|Qs],QX,NX,NY):-
        QX#\=QY,
        NX+QX#\=NY+QY,
        QX-NX#\=QY-NY,
        NY1 is NY-1,
        not_attack(Qs,QX,NX,NY1).
```

3.9.3 Planning problems

The AI-planning problems show a different picture than the CSP experiments. Here, the behavior of the $\mathcal{A}$system as complete system is measured. In contrast to the CSP problems, the different components in the $\mathcal{A}$system have to co-operate in a non-trivial manner to solve the planning problems. We will present the results for two classical planning problems: the blocks world domain and the logistics domain. Due to the different characteristics of the two planning domains, the behavior of the $\mathcal{A}$system is completely different. As the tables will show, the blocks world domain is easier for the $\mathcal{A}$system than the logistics domain.

The other systems in this experiments are the strips planner FF and DLV. (The

$\mathcal{FD}$-solver is not included for the obvious reason that it cannot solve AI planning problems.) For these experiments, we have used the special planning front-end $\mathcal{K}$ of DLV [113]. This defines a special language that simplifies the specification of planning problems. Internally the system translates a $\mathcal{K}$ program to a disjunctive logic program that contains the Event Calculus.

Blocks world

This problem domain describes an artificial world in which there is a table, a number of blocks and a single robot. The blocks are located either on the table or on top of another block. They can be moved from one spot to another by the robot. The robot can only handle one block at the time, i.e. its hand must be free before picking up a new block.

For the experiments, we have used the classical problem setting with an infinite table, i.e. a block can be always placed on the table. The used representation consists of a single action $move(A, B)$, meaning move the block A on location B (the table or a free block). The benchmark consists of the problem instances generated for the AIPS planning competition of 2000 [5, 6]. The problem number refers to the number of blocks in the problem instance, and it corresponds also to the number of blocks in the query (the goal state that must be reached). The plan size is the number of actions that are needed to achieve the goal. Since we compute sequential plans, i.e. the actions are ordered sequentially, the number of time points that is needed to reach the goal state is equal to the plan size.

problem	$\mathcal{A}$system		FF	
	time	plan size	time	plan size
17	700ms	23	0.03s	46
30	3s 520ms	51	1.12s	102
50	11s 920ms	86	1.73s	166
60	21s 150ms	104	1.52s	208

The $\mathcal{A}$system results are obtained by the following settings: select the choice points according to their amount of instantiation (default), no intermediate check of the constraint store and the strategy for the labeling of the finite domain constraint store is *min*, i.e. select the variable with the least domain lower bound. DLV has not been included in this table since it needs already 18 minutes to solve the 10 blocks problem. The plan size of FF differs from the $\mathcal{A}$system's; the specification for FF has two actions: picking up a block so that the robot holds a block, and putting a taken block on a free location.

Logistics

In this domain, the world of a package shipment enterprise is modeled. The goal is to bring the packages from the sender to its destination. The packages are picked up at the sender's location and transported by using trucks and airplanes. In this domain, the locations are grouped in cities. Each location is part of one single city. Some of these locations are airports, and each city has at least one airport. The cities are separated from each other and the only way to get packages from one city to another is by using an airplane. Airplanes are only allowed to land on airports. Every truck only drives within one single city.

The classical strips encoding consists of actions that describe the loading, unloading of a package in a vehicle and movement of the vehicles. This encoding is used by FF in our experiments. The $\mathcal{A}$system and DLV encoding uses a simplification, consisting of two actions: the movement of an object with a vehicle and the movement of a (eventually unloaded) vehicle. A plan consisting of these two actions can be easily expanded to a plan in terms of the classical encoding. By this, an important reduction of the number of actions is obtained (and consequently also a large number of choices).

The main difference with the blocks world domain is that the logistics domain allows parallel actions. The plans in the logistics domain require very few time points to reach the goal state.

The problem instances also originate from the AIPS planning competition 2000 [5, 6]. All these instances ensure that all locations are reachable, i.e. at least one airplane exists and for each city at least one truck exists. Each problem is identified by a pair $P - I$, where P is the number of packages that must be delivered and I is an identifier to distinguish between problem instances of the same size. The problem instances forms four groups: the first three contain one airplane, only the last has two airplanes. The first group contains two cities, the second consists of three cities. The third and fourth group have four cities.

problem	$\mathcal{A}$system				FF		DLV
	time	time (w. check)	actions	time-points	time	actions	time
4-1	250ms	1s 520ms	12	4	0s	19	888ms
5-0	330ms	990ms	16	4	1ms	27	3s 406ms
5-2	150ms	170ms	5	2	1ms	8	940ms
6-0	320ms	830ms	15	4	1ms	25	1s 544ms
6-1	200ms	350ms	9	4	1ms	14	1s 035ms
7-0	-	7s 690ms	25	8	2ms	36	2s 112ms
7-1	-	9s 920m	29	8	1ms	44	3s 054ms
8-0	-	15s 270ms	29	9	1ms	31	1s 765ms
8-1	-	15s 940ms	29	9	1ms	44	2s 268ms
9-0	-	4s 500ms	24	8	1ms	36	2s 208ms
9-1	-	9s 650ms	20	7	1ms	30	1s 642ms
10-0	-	8s 200ms	29	7	2ms	42	6s 365ms
11-0	-	-	-	-	2ms	60	9s 97ms
12-0	-	12s 630ms	29	9	2ms	42	16s 45ms
20-0	-	-	-	-	330ms	118	-

The $\mathcal{A}$system has been tested twice. The first column presents the results when no intermediate consistency checks of the constraint store are done. The second column shows the results when those checks are used. The third column contains the number of actions in the plan computed by the $\mathcal{A}$system. (For both parameters settings these are the same.) The fourth column shows the time point when the goal state is reached. That value is lower than the number of actions in the plan, indicating that there are many parallel actions. The results for DLV are those computed for 10 time points, which is in all problems up to 12-0 sufficient. For problem 20-0 a larger number has been tried, but DLV was unable to find a solution in time. The dash represents that the system was unable to find a solution within a reasonable amount of time (some minutes). The results of FF contains the time to construct the plan and the size of the plan. The differences in plan length between the $\mathcal{A}$system and FF are due to the differences in encodings, where the one of FF is much more detailed (and complex) than that of the $\mathcal{A}$system. We did not include the plan length generated by DLV since all plans are of length 10 (i.e. the number of time points).

Discussion of the results

The results of the $\mathcal{A}$system are excellent for the blocks world domain. It is able to solve very large problem instances (60 blocks) in a reasonable amount of time. When analyzing the $\mathcal{A}$system behavior, one sees that each step the $\mathcal{A}$system takes is towards a solution. The system backtracks very rarely. This can be observed in the debugging mode of the system, which only shows backtracking when the $\mathcal{A}$system is trying to prove if the fluent `on(A,B)` holds in the case that A is initially on B. It turns out that the $\mathcal{A}$system is able to decide immediately that this initial situation persists until the time point of the goal. If not, the $\mathcal{A}$system backtracks and selects the next branch, which will insert a new move action to achieve the goal. The debugging mode shows also that there is another choice point, namely putting a block on the table or on another block. Since putting on the table is the first clause, the $\mathcal{A}$system selects that option. According to the problem definition, this is always possible, and hence this choice will never lead to inconsistency. These properties explain why the $\mathcal{A}$system performs so well for the blocks world domain. Note that the computed plans are rather long since they come close to the simple strategy of putting first all blocks on the table and afterwards stacking them to the desired towers.

Such greedy behavior is absent when solving the logistics problem. For this problem, the $\mathcal{A}$system generates constraint stores which are unsatisfiable but which is not detected by the finite domain constraint solver. This problem cannot be avoided by manipulating the search strategy. Hence, without intermediate checks (see Section 3.6) the $\mathcal{A}$system will enter unsatisfiable branches. The applied check to obtain these results does not check the complete constraint store for satisfiability. That takes too long, even in small instances. The constraint store check happens after each newly added constraint, checking only the variables that are involved in the added constraint. For small problem instances, this yields good results. However, the same problem reoccurs in larger problem instances (e.g. 20-0).

The origin for this behavior is found in the combination of two facts. In contrast to the blocks world domain, the logistics domain generates a lot of similar abducibles (in particular of the action `moveVehicle(Vec,From,To)`). Also the constructed actions have more undetermined arguments. Both make the reuse of abducibles more complicated.

EXAMPLE 3.31 Suppose a problem with three airports a,b and c, and the airplane is initially in c. For a given query, the $\mathcal{A}$system derives that the airplane must fly from a to b and from a to c. These actions must satisfy their preconditions, which require that the airplane must be in a before flying to their destinations. Intuitively, two flights to a are needed, since the two flights starting from a cannot be served by the same incoming flight to a.

Since the airplane is in c initially, the airplane must fly to a. This is derived by the $\mathcal{A}$system. Suppose that first the precondition for the flight to b is evaluated, the $\mathcal{A}$system is able to detect that the incoming flight is needed. This results in the abduction of an action which represents a flight from some location X to a. Next the $\mathcal{A}$system evaluates the precondition of the second flight to c. Again, the $\mathcal{A}$system detects that an action is needed, i.e. a flight from some location Y to a. But now, since there is already a flight (from X to a) in the set of abducibles, the $\mathcal{A}$system tries to reuse this action. This requires that X must be equal to Y.

As mentioned, this is not possible. But the $\mathcal{A}$system, or more precisely the finite domain constraint solver, is not able to detect this. Only if the consistency of the finite domain constraint store is explicitly checked, then an inconsistency of the finite domain constraint store will be detected, and backtracking will happen. The presented situation is still easily detected by the current checks in the $\mathcal{A}$system, but when the abducible may reuse more than one already abduced atom, the checks become more complicated. This happens in the larger problem instances of the logistics domain.

The second factor why the $\mathcal{A}$system has difficulties with the logistics problem, is the big distance between the important choices that are responsible for the inconsistencies. Between two important choices there may be several choices with a first branch which is a good and consistent choice (lets call these easy choices). In order to resolve the inconsistency, backtracking will also occur over these easy choices. One reason for this distance is the evaluation of the choices according to their type. Since the reuse choice points are evaluated after the positive definition choices, there will be a lot of easy reuse choices between an important positive choice and an important reuse choice point. Hence when these are far away from each other, a long search process is initiated until the system backtracks to this positive definition choice point. None of the search heuristics that we have presented in Section 3.4 is able to solve this problem, since they cannot distinguish between easy and important choice points.

As the results of FF show, there is still a lot of space for improvement. FF is a heuristic search strips planner [52, 51]. Such planners derive a domain specific heuristic from the strips specification, which is based on a simplified version of the strips specification. E.g. by removing all delete-effects of the actions, a planning specification is obtained that forms a under-estimate for the original problem. A solution for the simplified version determines the minimum number of actions that must be applied in order to reach the goal state. For a strips specification all heuristic values are computed in advance (using backward reasoning), so that forward search can be applied in order to find a solution. Since the key of the success of FF and other heuristics planners lays in the computation of the domain specific heuristic, it is of interest if the $\mathcal{A}$system can be augmented with a similar kind of heuristics.

The results of DLV show that it does not scale well. The growing size of the propositional theory that must be created is a bottleneck for the system. For the logistics problem the size is relatively small because the grounder can determine a lot of false atoms and not many time points are needed. On the contrary, the blocks world needs a lot of time points. Moreover from an early time point on all fluent atoms are possibly true. This results in very large grounded theories and hence a large search space must be explored.

The conclusions that can be drawn from these AI planning experiments, are that the $\mathcal{A}$system performs reasonable to excellent for problems where the reuse of abducibles is not really necessary or not too complex to check for consistency. For the more complex problems, e.g. the logistics problem, the $\mathcal{A}$system must be tuned in order to solve the problems. That is a non-trivial task which requires a lot of experimentation. DLV handles those problems better than the $\mathcal{A}$system, but its application is severely restricted by the grounding. Both systems are still far away from the results that FF produces. (Recall that FF uses a more detailed domain description which is more complex than those of the $\mathcal{A}$system and DLV.) This system shows that domain specific heuristics can reduce the search time drastically.

3.9.4 Application area

The experiments show that the $\mathcal{A}$system is able to deal with two completely different problems for which specialized solvers exists. The same holds for the ASP solvers. It is maybe not so surprising that they are able to formalize these problems, given the high expressiveness of the logics. More surprising is that these solvers handle such different problems with an acceptable efficiency (as shown before).

Between the $\mathcal{A}$system and ASP solvers one important difference may potentially lead to a different application area: ASP solvers reason on propositional (finite) theories, in contrast to the $\mathcal{A}$system that allows infinite domains. For a large class of problems, this matches exactly the problem domain. This is not always the case. For example, in planning problems the size of a plan is unknown in advance. Therefore, the number of time points (time is supposed to be discrete) cannot be limited to a finite number. Consequently, the problem specification is not groundable and hence an ASP solver cannot be used. ASP solvers tackle this problem by solving the planning problem incrementally. One fixes the number of time points. In this way, the problem is groundable and suited for ASP solvers. If for this number of time points no plan is found, the number is increased and the planning problem is solved again. This incremental approach is a general method that can be applied when the domain size has no fixed upper limit. In principle, the infiniteness of time is not an issue for the $\mathcal{A}$system, since it generates the necessary time points on the fly. However, for reasons of efficiency the time is represented by a finite domain variable, having thus a limited domain. This limited domain prohibits the $\mathcal{A}$system to generate an infinite number of time points, but it still allows to generate less. Hence, when the upper-bound is large enough, the $\mathcal{A}$system will reason as if there was no limitation on the number of time points.

The $\mathcal{A}$system also differs from ASP solvers in the fact that it is not a model generator. It determines the truth of only those atoms that are relevant to prove the query. As shown in Section 3.3.4, the $\mathcal{A}$system achieves only local consistency w.r.t. the query. (For that reason, the $\mathcal{A}$system is sound w.r.t. the three-valued completion semantics.) If global consistency is required, ASP solvers are more suited since they compute the complete model.

In the introductory chapter we also discussed Description Logics. These logics have the expressiveness to model scheduling and planning problems (eventually by lifting some typical Description Logics restrictions). However, the developed solvers such as RACER [139], only compute the entailment of a formula, (called subsumption in DL-terminology). This limits their application area mainly to the development of (hierarchical) knowledge bases.

3.9.5 Limitations of the $\mathcal{A}$system

The $\mathcal{A}$system has some limitations which are worthwhile to be mentioned here.

Infinite loops

The $\mathcal{A}$system may get trapped in a loop. This may happen when a problem specifies a cyclic dependency. We illustrate this with the search for a Hamiltonian path in a given graph. The graph is given by a set of facts for the predicates $edge/2$ and $vertex/1$. A Hamiltonian path (denoted by $hamiltonian\_path/2$) is the transitive closure of a subset of edges ($hamiltonian\_edge/2$) from the graph.

$$hamiltonian\_path(X,Y) \leftarrow hamiltonian\_edge(X,Y).$$
$$hamiltonian\_path(X,Y) \leftarrow hamiltonian\_edge(X,Z) \wedge hamiltonian\_path(Z,Y).$$
$$\forall\ hamiltonian\_edge(X,Y) \rightarrow edge(X,Y).$$

Moreover, a Hamiltonian path visits every vertex *just once*. This is ensured by the constraints on the Hamiltonian edges. An edge from vertex X to vertex Y is a Hamiltonian edge if there is no other Hamiltonian edge that already starts from X or ends in Y.

$$\forall\ hamiltonian\_edge(X,Y_1) \wedge hamiltonian\_edge(X,Y_2) \rightarrow Y_1 = Y_2$$
$$\forall\ hamiltonian\_edge(X_1,Y) \wedge hamiltonian\_edge(X_2,Y) \rightarrow X_1 = X_2$$

A Hamiltonian path visits every vertex: in all vertices except the start vertex (denoted by $start/1$) a Hamiltonian edge ends; analogously, in all vertices except the final vertex ($end/1$) a Hamiltonian edge starts.

$$\forall\ vertex(X) \wedge \neg end(X) \rightarrow \exists\ Y.hamiltonian\_edge(X,Y).$$
$$\forall\ vertex(Y) \wedge \neg start(Y) \rightarrow \exists\ X.hamiltonian\_edge(X,Y).$$

For a Hamiltonian cycle the start and final vertex are the same.

For this problem the $\mathcal{A}$system will loop when the graph contains a cycle. E.g. consider a undirected linear graph $v_1 - v_2 - v_3$:

$$edge(X,Y) \leftarrow edged(X,Y).$$
$$edge(X,Y) \leftarrow edged(Y,X).$$
$$edged(v_1,v_2).$$
$$edged(v_2,v_3).$$

The search for a Hamiltonian path from vertex v_1 to v_3 (which exists) may go in the following loop (we show a simplified $\mathcal{A}$system-derivation):

$$(\mathcal{G}_0, \mathcal{ST}_0) = (\{\mathbf{hamiltonian\_path(v_1,v_3)}\}, \emptyset)$$
$$\downarrow (D.1)\ unfold$$
$$(\{\mathbf{hamiltonian\_edge(v_1,Z_1)} \wedge hamiltonian\_path(Z_1,v_3)\}, \emptyset)$$
$$\downarrow (A.1,\ldots)\ select\ edge\ v_1 - v_2$$
$$(\{\mathbf{hamiltonian\_path(v_2,v_3)}\}, \{Z_1 = v_2, hamiltonian\_edge(v_1,v_2)\})$$
$$\downarrow (D.1)\ unfold$$
$$(\{\mathbf{hamiltonian\_edge(v_2,Z_2)} \wedge hamiltonian\_path(Z_2,v_3)\}, \mathcal{ST}_3)$$
$$\downarrow (A.1,\ldots)\ select\ edge\ v_2 - v_3$$
$$(\mathcal{G}_4, \mathcal{ST}_4) = (\{hamiltonian\_path(v_1,v_3)\}, \mathcal{ST}_3 \cup \{Z_2 = v_1, hamiltonian\_edge(v_2,v_1)\})$$

In the last goal stack $\mathcal{G}_4$, the same goal $hamiltonian\_path(v_1,v_3)$ reoccurs as in the initial goal stack $\mathcal{G}_0$. If the same strategy is followed from the last state, the $\mathcal{A}$system will never terminate. This trace illustrates in case of recursive relations the $\mathcal{A}$system may enter a non-terminating derivation.

For this problem (and others having similar cyclic relations) the ASP solvers do not loop. They construct the different Hamiltonian paths without any problem.

Model generation

The $\mathcal{A}$system is not a model generator, but an abductive query answering system. Consider the propositional abductive normal program for the abducibles a and b

$$\leftarrow a \wedge b.$$

Applying the $\mathcal{A}$system for the query **true** on this program results in the answer:

$$\Delta = \emptyset \qquad \Delta^* = \{\leftarrow a \wedge b\}$$

For many problems, this inference returns the expected answers. However, for some problems such as diagnosis problems this is unsatisfactory. One would like that the $\mathcal{A}$system generates the solutions

$$\begin{array}{ll} \Delta_1 = \{a\} & \Delta_1^* = \{\leftarrow b\} \text{ and} \\ \Delta_2 = \{b\} & \Delta_2^* = \{\leftarrow a\} \end{array}$$

The $\mathcal{A}$system will not generate these solutions for the query **true** because there is no inference rule that abduces an atom from a denial. A denial only checks the consistency of an atom, and does not generate abducibles.

A solution that results in the computation of the two answers is to add an auxiliary hypothesis generator predicate. For example,

$$\begin{array}{l} gen \leftarrow a. \\ gen \leftarrow b. \end{array}$$

By adding this generator to the query, i.e. $\mathbf{true} \wedge gen$, the $\mathcal{A}$system will compute the above two answers: The $\mathcal{A}$system creates for the generator gen a backtrack point. In the first branch, the first solution is computed, the second computes the second.

As mentioned before, for most problems this is not an issue, since the domain knowledge often includes implicitely a 'hypothesis generator'. When this is not the case, the expert has to help the system by adding auxiliary generators. In the case of the mentioned diagnosis problems, the generator takes the form of a plan generator.

3.9.6 Conclusion

In this section, we have experimentally verified the current status of the $\mathcal{A}$system. It shows excellent and encouraging results. In comparison to other related systems, the $\mathcal{A}$system is computationally close to less declarative systems (e.g. the $\mathcal{FD}$-solver) and often more efficient than closely related reasoning systems such as DLV. The presented results affirm the mutual positions as concluded in [213]. For this reason, we can state that the $\mathcal{A}$system is a state-of-the-art solver for Declarative Problem Solving.

The current implementation is clearly an improvement of the prototypes of SLDNFA and ACLP. The least-commitment strategy results in a more robust system: e.g. the system is less sensitive to the order of the literals in clauses. Another improvement is the more efficient equality reasoning ($\mathcal{E}$-solver). SLDNFA treated equalities by unification which formed a major bottleneck in the first prototypes. In ACLP the equalities were treated by the finite domain solver, but this limits the equality reasoning to finite domains. Arbitrary (infinite) domains could not be handled.

When comparing the $\mathcal{A}$system with the integrated $\mathcal{FD}$-solver, we show that the $\mathcal{A}$system problem specifications are more concise than the logic programs of CLP($\mathcal{FD}$). The conciseness reflects itself in the larger constraint setup times for the constraint store. The constructed constraint store by the $\mathcal{A}$system is almost as good as the one constructed by the $\mathcal{FD}$-solver (for CSP's). But because the $\mathcal{A}$system does not provide direct access to the constraint variables, the human expert cannot specify optimized search strategies. Such strategies are of great importance in order to find (very) efficient a solution. $\mathcal{FD}$-solvers have another advantage over the $\mathcal{A}$system: they have access to global constraints. In this way their specification also becomes more concise, but, more importantly, the solution construction is much faster.

The results for the planning problems are encouraging, although they are still far from the results of the strips planner FF. For the blocks world problem, the $\mathcal{A}$system execution is excellent, while for the logistics problem, the $\mathcal{A}$system performance is reasonable. Our analysis of the $\mathcal{A}$system's behavior has indicated the main sources of efficiency and inefficiency that explain the presented results. We did not focus on variants of planning problems: for example problems with conditional effects, incomplete initial states, extending partial plans,... These variants can be represented as an Abductive Logic theory and handled by the $\mathcal{A}$system. That is an important advantage w.r.t. strips planning. Most strips-planners are only suited for a limited number of such extensions. E.g. most strips-planners will be able to handle conditional effects, but not problems with an incomplete initial state.

Answer Set Programming systems are the closest to the $\mathcal{A}$system. Our limited set of experiments show that for some problems the $\mathcal{A}$system is performing better than ASP systems. In the first place, it holds for the (finite domain) CSP problems. These problems are very efficiently handled by constraint reasoning. ASP solvers are less efficient for these problems because they ground the problem specification. Especially, when the domains are large and the constraints do not allow to prune much of the search space when grounding, the performance of the ASP solvers will be poor. Another reason for the computational results of the ASP solvers is that they present themselves as blackbox solvers, in contrast to the $\mathcal{A}$system which is a glassbox solver. This means that the $\mathcal{A}$system can be tuned for a particular problem by specifying special search heuristics. The current ASP solvers do not offer this. The experimental results indicate that it may be worthwhile to consider this.

ASP solvers are in one respect superior to the $\mathcal{A}$system: robustness. Since they reason on proposition theories, the search space is finite. This ensures that they always terminate. Consequently, recursive definitions are not troublesome. This general robustness is a very attractive property which makes ASP solvers, from user's point of view, more reliable than the $\mathcal{A}$system.

Taking everything into account, the current status of the $\mathcal{A}$system shows that abductive inference, when carefully implemented, can be used as the basis for an efficient solver. Of course, there is a lot of space for improvement. An overview of possible topics is given in the next Section 3.10.

3.10 Future work

A complex system as the $\mathcal{A}$system offers of course a variety of possibilities to extend and improve the system. This section gives a (short) overview of several topics that can be tackled. Note that it does not form a complete list.

3.10.1 User support

Computational efficiency and expressiveness of the modeling language are not the only criteria that make a tool being used in practice. An important element is the given user support. At this moment, the user support is rather limited because our efforts have been focussed on the computational efficiency of the $\mathcal{A}$system. We have mentioned that the preprocessor is able to detect some errors in problem specifications. The system also provides a special command prompt which gives the user some extra guidance. A last piece of user support is the integrated debugging mode that allows to follow the $\mathcal{A}$system derivation from backtrack point to backtrack point.

These elements are sufficient for somebody that knows the system well, however for new users more support must be provided. Basically, a specification development tool must be provided with features as syntax highlighting, error detections, editing support, extended documentation, etc...

3.10.2 Dealing with loops

One of the $\mathcal{A}$system's most limiting elements, is that it does not deal well with recursive definitions that contain cycles. The evaluation of such definitions often makes the $\mathcal{A}$system loop.

In Logic Programming, one has developed an extension of SLDNF-resolution, called SLG-resolution, that deals with this issue. During a SLG-derivation, a separated derivation tree is initiated for each goal. The answers of this derivation are stored in a table. When a goal is selected for which there exists already a derivation tree (i.e. it is a variant of a tabled goal), the already computed answers that are stored in that table are reused. Only if these tabled answers do not satisfy, the corresponding derivation tree is further expanded in order to find new solutions. The procedure, commonly called *tabling*, ensures that each solution is computed only once. Because goals that have already a derivation tree are not recomputed, but reuse the work that has been done previously, cycles can be detected and broken.

The extension of the $\mathcal{A}$system with this mechanism is non-trivial due to the abducibles. There are two possible ways to integrate abduction and tabling. The first is to extend the inference procedure of the $\mathcal{A}$system with a memoization technique, in similar way as SLG does it for SLD(NF). A second possibility is to extend SLG-resolution with abduction. The latter has been explored by the

AbDual-system [11]. This system studies the possibility to use the tables also for the construction of the table of abduced atoms. By a smart transformation of the abductive program, this is possible for *propositional* abductive programs. The first order case, in which we have interest, is much more complex.

This complexity can be illustrated by the next example. Let $p(\cdot)$ be a defined recursive predicate with a single clause that contains an atom of the abducible $a(\cdot,\cdot)$: $p(X) \leftarrow a(X,Y) \wedge p(Y)$.

- The selection of an atom $p(\overline{t})$.
  The $\mathcal{A}$system will unfold this atom with its definition, which yields $a(\overline{t},Y) \wedge p(Y)$. This conjunction is evaluated further by e.g. abducing $a(\overline{t},Y)$ which results again in the selection of an atom $p(Y)$ of the predicate $p(\cdot)$. Without tabling, the $\mathcal{A}$system continues the evaluation of $p(\overline{t})$ forever.

- The selection of the denial $\forall\, \overline{X} \leftarrow p(\overline{t}) \wedge Q$.
  This denial may also invoke an infinite derivation. The $\mathcal{A}$system unfolds $p(\cdot)$ to $\forall\, \overline{X}, Y \leftarrow a(\overline{t},Y) \wedge p(Y) \wedge Q$, which will be stored in Δ^* waiting for abduced atoms of $a(\cdot,\cdot)$ that must be checked for consistency. Suppose that the $\mathcal{A}$system abduces the atom $a(\overline{t},\overline{t})$. Then this denial is further evaluated as $\forall\, \overline{X}, Y \leftarrow a(\overline{t},Y) = a(\overline{t},\overline{t}) \wedge p(Y) \wedge Q$ which leads to $\forall\, \overline{X}. \leftarrow p(\overline{t}) \wedge Q$. Now $p(\overline{t})$ is again unfolded and since $a(\overline{t},\overline{t}) \in \Delta$ an infinite evaluation happens.

Both problems require adequate extensions of the tabling mechanism so that the loops are detected and can be interrupted.

3.10.3 Domain specific search strategies

The results for the AI planning problems in Section 3.9 show excellent computation times for the strips planning system FF [145]. FF's efficiency is due to (1) its specialization for this problem task and (2) the ability to derive a search heuristic from the problem specification. The importance of the right heuristic is also known in Constraint Logic Programming. CLP approaches this issue from a different perspective than heuristic planners: one provides the user with tools to develop a domain specific search. The few standard search strategies the Sicstus solver provide take little domain information into account. Both are valuable and complementary techniques to deal with a search problem. Currently, the $\mathcal{A}$system follows more the line of CLP.

However, as we have experienced, it is often difficult to find the right heuristic for a problem. Therefore, it would be of great help if the system can derive a heuristic from the problem specification. The general idea behind the AI-planning heuristics is to simplify the problem to a known easy problem and to use the solutions of this simple problem as heuristic values for the real problem. For strips planning problems this is relatively easy, but for general theories that form the input for the $\mathcal{A}$system (and ASP solvers), this simplification is not straightforward. But as the strips planners show, the reward of such heuristic may be large.

3.10.4 Redundant computations

When reasoning with high level specifications one comes across the problem of the generation of the same or similar information. We have observed this phenomenon for these situations:

- The symmetry in constraints: e.g. the constraint that linearizes the actions of a plan

  $$\forall\ A_1, A_2, T.\ \leftarrow act(A_1, T) \wedge act(A_2, T) \wedge A_1 \neq A_2.$$

  is symmetric. For two abduced atoms $\{act(X, Y), act(R, S)\}$ the denials $\leftarrow Y = S \wedge X \neq R$ and $\leftarrow S = Y \wedge R \neq X$ are generated, which are variants of each other.

  Our suggested solution is based on the internal identifiers of the abduced atoms. The symmetric constraint can be transformed to

  $$\begin{array}{l} \forall\ A_1, A_2, T. \\ \leftarrow ID1 - act(A_1, T) \wedge ID2 - act(A_2, T) \wedge ID1 < ID2 \wedge A_1 \neq A_2. \end{array}$$

  where $ID1$ and $ID2$ represent the identity of the abduced atom. The identities are *non-logical* information in the form of positive integers. Internally, the evaluation of a denial starting with an abducible must be extended to handle the transformed reference to the abducible in the integrity constraint. Instead of evaluating the denial literal-by-literal, the system has to evaluate this denial up to the asymmetric condition $ID1 < ID2$ at once. Indeed, the condition $ID1 < ID2$ allows to halve the to be evaluated combinations of $act/2$ atoms. For the set $\{1 - act(X, Y), 2 - act(R, S)\}$ only the denial $\leftarrow Y = S \wedge X \neq R$ is generated.

  Since this encoding uses solver directions that pollute the pure logical meaning of the assertion, the use of such directions should be handled with care. Note that it is sometimes possible to get approximately the same effect by adding a logical asymmetric condition in the assertion. Consider for example the N-queens problem, in which the main assertion is often stated as

  $$\begin{array}{l} \forall\ Q_1, P_1, Q_2, P_2. \\ \leftarrow position(Q_1, P_1) \wedge position(Q_2, P_2) \wedge Q_1 < Q_2 \wedge \neg notattack(Q_1, P_1, Q_2, P_2). \end{array}$$

  The $\mathcal{A}$system evaluation ensures that no symmetric $notattack/4$ conditions are added to the constraint store, but it will still evaluate the assertion for all possible combinations of $position/2$ atoms. The latter is still a large overhead that is avoided by using the above method.

  In general, if an assertion is known to be symmetric then the above transformation can be applied. But the application is limited since the automated detection for a sufficiently large class is a non-trivial task.

- Another source of redundancy is found in the equality and finite domain constraint choice points. It may happen that two choice points are generated with the same condition: e.g. a choice $\forall\ \overline{X} \leftarrow A \wedge B$ and a choice $\forall\ \overline{Y} \leftarrow A \wedge C$, where A, B and C denote formulas. Recall that for each choice point a reified expression is constructed for the suspension of the choice points. In this situation, the condition A is reified twice, introducing twice the same constraint in the constraint store. If the constraint solvers would be able to recognize the similarity between expressions then the duplication could be avoided by reusing existing expressions.

  In the current $\mathcal{A}$system implementation constraints are immediately handed to the corresponding constraint solver. Constraint solvers consider each constraint as a unique and independent piece of knowledge and do not check for equal expressions. Moreover they usually do not store the original expression so that a redundancy check is difficult to implement. Therefore an intermediate layer is required which mediates between the abductive reasoner and the constraint solver.

  Note that a more generalized version of this redundancy problem occurs when one condition is a sub-expression of another condition.

- The $\mathcal{A}$system can compute the same answer more than once. It does not guarantee that each solution is presented just once to the user.

  Recall that Δ, which is the main element of an $\mathcal{A}$system solution, is a set, and thus the order of the atoms is irrelevant. For example, the $\mathcal{A}$system solutions[15] $\Delta_1 = \{1 - a(a), 2 - a(b)\}$ and $\Delta_2 = \{1 - a(b), 2 - a(a)\}$ are equivalent. One could expect that the system presents only one of those as an answer. However both solutions are presented. The explanation for this behavior is that from the reasoner's point of view they are different variable assignments (e.g. $\{X/a, Y/b\}$ and $\{X/b, Y/a\}$, where X is the free variable of the first atom, and Y the one of the second). In that view the solutions are different.

  In order to compute each solution just once the variable assignment construction must be aware of all previous solutions. This clearly leads to a larger computation time for the next solution.

  When this uniqueness is a must, one can apply the following approach. One computes the first solution of the problem. With this solution, a new problem from the original is derived, by adding the integrity constraint that any solution should be different than the computed solution. The second solution is then the abductive solution of the updated problem specification. This solution is surely different from the first one (by the integrity constraint) and thus has not been presented as an answer previously. The third and following

[15] The N-A elements in the solutions denotes the atom A with identifier N.

solutions can be obtained by repeating the procedure, until all solutions are found.

Since almost none of the problems we have faced did require the computation of all (unique) solutions, the above procedure (or other reasoning in order to ensure the uniqueness of the solution) is not supported.

3.10.5 Improving the quality of the finite domain constraint store

The experiments with the logistics planning problem showed that the quality of the finite domain store constructed by the $\mathcal{A}$system has to be improved in order to allow the $\mathcal{A}$system to find a solution. This issue has its foundations in the difference between the intended use of the constraint solver and how the $\mathcal{A}$system uses the solver. The constraint solver expects a hand-engineered constraint store, while the $\mathcal{A}$system generates the store from a specification automatically.

We already mentioned in Section 3.6 the issue of redundant constraints in the store. (We mean with redundant here constraints that really do not contribute to the search: copies of the same constraint, entailed disjunctions, etc...) The current version of the $\mathcal{A}$system has a limited detection mechanism for entailed disjunction. A more improved version is needed, for example to detect copies of the same constraint.

Another improvement is at the level of the constraint reasoning itself. We use the finite domain constraint solver for many problems. In planning problems, the finite domain solver is used to reason on the time points. This involves reasoning with inequalities and disequalities. However the finite domain solver is not able to detect that the constraint store $\{T_1 \leq T_2, T_1 \neq T_2\}$ implies that $T_1 < T_2$. Since this information may be vital when solving planning problems, we either have to improve the finite domain solver so that this information is derived, or we have to implement a specialized constraint solver for these relations. Both options are sensible.

A last improvement we see, is the integration of global constraint in the constraint store. A possible use of global constraints is via pattern recognition of constraint networks that can be replaced by a global constraint. The constraint store is searched for a combination of constraints for which an equivalent global constraint exists. If such a constraint network is found, the selected constraints are replaced by the corresponding global constraint. Note that the complete optimization happens transparent for the $\mathcal{A}$system user.

Related work is the dynamisation of global constraints [37, 38, 39]. Currently all global constraints are static in the sense that they require that all variables that are taken into account are known prior to the posting; this assumption does not hold for the $\mathcal{A}$system. Later generated variables cannot be included in the constraint. This is sufficient for many constraint problems, but in situations where constraints come from an external process or are generated during the solving of a dynamic problem as planning, the use of static global constraints is restricted. Barták [37, 38] has recognized this when solving combined planning and scheduling applications. He has defined a general dynamisation scheme for global constraints and applied it to the `all_different` constraint [39]. This work is useful since as long the $\mathcal{A}$system has not reached a final state, new variables can be added to the constraint store.

3.10.6 Extending the expressive power

It has been mentioned several times that the $\mathcal{A}$system is a modular and extendable system. This has been illustrated with the *open function* extension (see Section 3.8). We have also experimented with other extensions which are not included in the thesis. One is our work on aggregate expressions [263]. These are a very useful class of second order expressions such as the sum, cardinality and maximum. This work forms an important track of future work since these expressions occur very frequently in domain knowledge. Moreover aggregates from a necessary component in order to handle AI planning and scheduling problems at the same time.

The addition of other constraint domains in the $\mathcal{A}$system is another possibility.For example, a very early version of the $\mathcal{A}$system included the CLP(R) solver, allowing in this way to reason on real numbers. This line of research is strongly case-based motivated. Which brings us to another, yet important, line of future work: solving more problems by the $\mathcal{A}$system. In the next chapter we present contributions to one single application area: the intelligent integration of databases. By applying the $\mathcal{A}$system to more problem areas, the understanding on how to optimize the inference in the system will grow. Also such experiments allows us to identify key extensions and future directions of development.

One such experiment we did was in the context of the departmental duty system [42, 41]. In this we related an object-oriented conceptual language ERŒS [253, 243, 242] with ID-Logic. Via this relations we were able to simulate a prototype system, that transforms one state of the world into another by executing events (tasks). This work raises two important questions for future research: *how do we integrate the $\mathcal{A}$system (and in general the Declarative Problem Solving) in classical programming languages such as JAVA?*, and *is it possible to build an online knowledge representation system which purely applies DPS to solve problems?* The latter question requires a model (or models) describing the evolution of the real world (and its relation to a computer system). In our opinion, both questions must be sufficiently addressed in order to give DPS (and thus also the $\mathcal{A}$system) a more widespread use than the current position it holds in software development practice.

3.11 Final conclusions

The $\mathcal{A}$system is a new abductive constraint logic solver. To our knowledge, the system is at this moment the best elaborated first order abductive solver. It is more robust and efficient than the earlier abductive systems mainly due to a much improved implementation of the search. As the experimental results show the implementation effort has resulted in a good computational behavior.

The proof procedure of the $\mathcal{A}$system is based on the abductive systems SLDNFA (the inference rules), the IFF-procedure (the presentation as state rewrite system) and ACLP (the integration with CLP). An important feature of this representation is the explicit formalization of multiple constraint domains. By this structure, it is easy and clear how to add new constraint solvers to the $\mathcal{A}$system.

Procedurally, the system takes as input a first order abductive normal logic program $\mathcal{P}$, and a conjunction of literals $\mathcal{Q}$ as query. An abductive solution is found as the consistent final state of a derivation process in which the query augmented with the denials of $\mathcal{P}$ are step by step reduced to basic formulas which are stored in (constraint) stores. Our main contributions are at the implementation of this derivation process. We made improvements at different levels of the system:

1. The *search process* is made more efficient by the implementation of a least-commitment strategy with forward propagation rules. The system suspends the execution of choice points until they must be evaluated to proceed with the search. During their suspension, the choice points are informed about the state of the system by a forward propagation mechanism. In this way, suspended choice points can be entailed, disentailed or become deterministic. In the latter case, the choice point can be evaluated without the creation of a backtrack point.

   We have identified five types of choice points in the $\mathcal{A}$system. For each of them, we have presented forward propagation rules and some additional optimizations. The implementation of the forward propagation is based on *reification*, a technique that we made available for more expressions and extended with a possibility to retract reified expressions from the constraint store.

2. We have designed new *data structures* for storing the abduced atoms and the denials which start with an abducible atom, that are derived during execution of the system. These data structures allow a fast retrieval of a superset of the elements that are relevant for the query. Computationally, the improvement is due to the fact that the returned set of elements often is a small portion of all elements stored in the data structure. Another advantage it yields, is that the $\mathcal{A}$system becomes more scalable. The main drawback is the increased space requirement.

3. A third improvement has been at the level of the evaluation of the inference rules. We observed that the classical non-ground meta-interpretation requires copying in order to deal correctly with the quantification of the variables in the denials. This overhead can be removed by the design of a ground meta-interpreter. That method is more complex to implement than the classical one, but it yields a large reduction of computation time for some problems. This ground meta-interpretation can be seen as a first step towards the design of a compiler for abductive normal programs.

4. The development of the (dis)equality constraint solver, $\mathcal{E}$-solver, is a fourth element in the improvement of the performance. It has a key role in the $\mathcal{A}$system since equality and disequality reasoning is the most applied reasoning in the system. Also it provides the means to develop an efficient forward propagation.

5. A final contribution is the integration of the finite domain solver in the $\mathcal{A}$system. In principle, the integration is simple. However, in practice it turns out that the finite domain solver is not strong enough to provide in most cases a good behavior. We have provided several solutions in order to improve the relation between the finite domain solver and the $\mathcal{A}$system.

We like to point out that the above improvements are the major technical contributions. The current implementation contains also many small decisions we did not mention. They include the design of less important data structures, the formulation of the Prolog code, small optimizations in order to avoid redundant computations, etc ...

The implementation of the $\mathcal{A}$system has been (and still is) a difficult task. Finding out if a certain new technique improves or decreases the performance of the system is tedious. The main factors in this process are the quality of the by-myself written code and the interaction with external components e.g. the finite domain solver. The latter aspect makes it non-trivial to judge if my code or the external code is responsible for a certain behavior. The development is further complicated by the fact that some effects only happen in large problem instances (after long and complex execution traces). Detecting the sources of the inefficiency is then time-consuming.

Probably the most important weakness of the $\mathcal{A}$system is its unpredictable behavior. For each problem, the $\mathcal{A}$system has to be tuned in order to get (fast) solutions. This aspect has the $\mathcal{A}$system inherited from Constraint Logic Programming. In that framework, good behavior is only achieved after experimentation with different parameter settings. The $\mathcal{A}$system requires additional configuration: not only the finite domain constraint solver must be tuned, but also the abductive reasoning. Therefore, the user must understand the execution of the $\mathcal{A}$system and its interaction with the constraint solvers, which is a serious effort.

Another, related, issue is that the $\mathcal{A}$system is not guaranteed to terminate. This is ensured for model generators such as sModels and DLV which act on a finite search space (due to the grounding). Like SLD(NF), the $\mathcal{A}$system can get trapped in loops, e.g. invoked by the program rule $p \leftarrow p$. Extending the $\mathcal{A}$system with techniques such as tabling may remove such causes of non-termination. This is a complex and still open problem for the first order case. For propositional abductive programs, the AbDual [11] system can be used.

The most frequent occurring 'non-termination' behavior, however, happens when the $\mathcal{A}$system enters a branch that is unsatisfiable and which has a large proof-tree. Although this is correct behavior, it looks as if the $\mathcal{A}$system is trapped and will never find a solution. Such behavior can happen when the system is able to construct a long sequence of hypotheses that potentially support the goal. For example, in AI planning if a plan with N actions does not suffice to reach a goal state, there might be a plan with N+1 actions. A connected issue is the lack of good domain dependent search heuristics (that can be derived automatically). By the current domain independent heuristics, the $\mathcal{A}$system may do inappropriate decisions leading to very long derivations. Although many of the presented improvements have as effect that the $\mathcal{A}$system will enter less often such branches than the earlier abductive systems, the logistics experiment has shown the $\mathcal{A}$system still may suffer from this behavior.

Our experiments with the $\mathcal{A}$system have been mainly on classical artificial AI problems. These allow to study the system in well-controlled situations and to compare it with other systems. We also have done some experiments with two (small) real world problems: the Tractebel problem [238] which is a CSP problem with aggregates and the DAF problem [194] which is a temporal production planning problem. Both show that modeling the problems with ID-Logic is relatively easy and straightforward. Computationally, the $\mathcal{A}$system is able to handle only small problem instances in the DAF case, and a realistic size in the case of Tractebel. In the next chapter, we will discuss the use of the $\mathcal{A}$system in the context of the integration of databases. The experiments that have been executed in that context were successful, but the problem instances were small. Experiments, both with artificial as real world problems, are important for the future development of the $\mathcal{A}$system. They validate design decisions and indicate new directions for development.

Taking everything into account, the current state of the $\mathcal{A}$system is at the level of being useful for experimenting with small size real world problems. When the generation of choice points is kept under control and the intermediate constraint stores become not to hard to check for consistency, the $\mathcal{A}$system will probably be able to find a solution. The system is free for academic use and available at `http://www.cs.kuleuven.ac.be/~dtai/kt/`.

Chapter 4

Coherent integration of databases

The last decade our society has become characterized by the increasing accessibility to information. Communication networks link information sources with each-other and provide worldwide access to the information stored in them. This evolution has shifted the view on information systems from stand-alone knowledge bases that centralize the knowledge about one domain towards a co-operating network of smaller, specialized and independently maintained information sources. The classical example is the Internet, where information about a subject is distributed over many places, often duplicated and available in many different formats.

Retrieving information in such a heterogenous context poses new challenges. Instead of querying a single source, the information has to be combined from several sources. An automated solution needs, therefore, a structure in which the information stored in the sources is correctly linked with each other. In addition, the combining process should avoid the materializing of the complete composed knowledge because the sources are normally not under the control of the person who is retrieving information. Another important issue concerns the coherence of the retrieved information. It might occur that two sources have contradictory information. In that case, the integrating process should not collapse or return unfounded information.

Here, we will consider the integration problem in a uniform setting, namely the *information integration problem for relational databases*. Given a set of independent databases, called the sources, the goal is to construct a system that provides global access to the information residing in the sources and that returns only coherent information. In practice, information is stored in many formats. By assuming that all information sources are (relational) databases, we avoid the problem of extracting the relevant information out of the thousands of formats.

Usually one supposes that there exists for each data format a translator system that does the job. These are called *wrappers*.

The interest in systems for (coherent) integration of databases has been continuously growing in the last years (see e.g., [32, 33, 62, 124, 135, 138, 137, 175, 177, 182, 186, 199, 209, 225, 246, 252]). A recent survey is found in [72]. The interest in information integration techniques is not only due to the growing importance of the Internet, but also due to the many organizations that face the problem of data integration. When two companies merge, e.g. two financial institutions, their information systems have to be integrated. Partially due to lack of good integration techniques, a complete integration may take several years. The integration issue is also faced by companies that build Data Warehouses or Enterprise Resource Planning systems, e.g. SAP. These examples indicate that almost every information system faces the issue of integration in one or the other form. Hence, information integration is an important problem for which good techniques and methods must be provided.

In this chapter we present several contributions to the integration of relational databases. First we will precise the integration problem in the preliminaries. Afterwards, each section is devoted to one particular (sub)problem and solution that we have elaborated for that problem.

4.1 Preliminaries

What's in a database

The basic component in our information integration problem are databases [114]. Databases store in a compact way information about the world. Basically, there exists two common ways to regard a database. According to the first, a database is a structure consisting of a domain and corresponding relations. The other describes the meaning of the database by an associated logic theory [73]. We adopt the latter because it is more suited to express incomplete knowledge. The former supposes that the world is precisely reflected in the state of the database. In our problem situation, where the source databases normally contain only partial information about the global domain, this assumption cannot be maintained.

In our (logic) view, a database is composed of three elements:

1. An *alphabet* $\mathcal{A}$ to formalize the domain knowledge. An alphabet consists of symbols (constant, function and predicate symbols) to denote the objects and their relations in the real world domain. Some terminology: the *vocabulary* is the set of predicate symbols. The *domain* is the collection of objects in the real world.

2. A *database instance* $\mathcal{D}$ that stores the *extensional knowledge* of the database, i.e. the information that is known by the databases. This information is expressed by a logic theory based on the alphabet $\mathcal{A}$.

We adopt the simplest representation (which corresponds most closely to relational databases), namely a database instance $\mathcal{D}$ consists of (a finite number of) atoms of the alphabet, representing the **true** facts. More precisely, we apply the Closed World Assumption [223] to the database instance, i.e. only the facts that are explicitly mentioned in $\mathcal{D}$ are considered **true**. It is supposed that the Unique Names Axioms (UNA) hold, i.e two syntactically different terms denote a different object of the domain. The underlying semantics corresponds, therefore, to minimal Herbrand interpretations.

DEFINITION 4.1 The *minimal Herbrand model* $\mathcal{H}^{\mathcal{D}}$ of a database instance $\mathcal{D}$ is the model of $\mathcal{D}$ that assigns true to all the ground instances of atomic formulas in $\mathcal{D}$, and false to all the other atoms.

3. A finite set of first order formulas $\mathcal{IC}$ based on the alphabet $\mathcal{A}$ that describe the *intensional knowledge* of the database, i.e. the outline of the information that may be known by the database. The collection of all classical models of $\mathcal{IC}$ form the acceptable instances of the database. These formulas are often called *integrity constraints* since if the extensional knowledge violates one of these formulas, the integrity of the database is violated.

Formally, a database is defined as follows:

DEFINITION 4.2 An *database* $\mathcal{DB}$ is a structure $\langle \mathcal{A}, \mathcal{D}, \mathcal{IC} \rangle$ composed of an alphabet $\mathcal{A}$, a database instance $\mathcal{D}$ and a finite set of integrity constraints $\mathcal{IC}$. The database instance $\mathcal{D}$ is a set of atoms, which are constructible from the alphabet $\mathcal{A}$. The integrity constraints are first order sentences based on the alphabet $\mathcal{A}$.

NOTE 4.1.1
In the (database) literature the alphabet often is called the *(database) schema*. Also, we will use (first order) language or ontology as synonym for the alphabet.

EXAMPLE 4.1 A database $\mathcal{DB}$ about the Belgian royal family is e.g. $\langle \mathcal{A}, \mathcal{D}, \mathcal{IC} \rangle$ where

- $\mathcal{A} = \{father(\cdot,\cdot), mother(\cdot,\cdot)\}$,
- $\mathcal{D} = \left\{ \begin{array}{l} father(albert, filip), mother(paola, filip), \\ father(filip, elisabeth) \\ mother(mathilde, elisabeth) \end{array} \right\}$ and
- $\mathcal{IC} = \{\forall\ X, Y. \neg(father(X,Y) \wedge mother(X,Y))\}$

The database of example 4.1 stores information about the Belgian royal family. Obviously, this information is partial, e.g. there is no information about Gabriel, the second child of Filip and Mathilde. Hence the scope of the database is limited, e.g. this database stores only information about the persons who may become king or queen of Belgium. It is common to neglect the limitation of the scope, by assuming that the domain is given by the objects that are mentioned (by their symbol) in the database. When one concerns one database in isolation, usually this is a sensible assumption. However in our case, where we will consider more than one database storing information about the same problem domain, this assumption is unrealistic. Therefore, we distinguish between the *active domain* and the *domain* of a database. The active domain the set of objects mentioned by the database instance, and it is a subset of the domain. For example, the active domain of the database in example 4.1 is $\{albert, filip, paola, elisabeth, mathilde\}$. It is a subset of the full domain of all royal family members, which includes also e.g. *gabriel*, *laurent* and *astrid*. Note that the above definitions still hold for the case that the full domain is not precisely known, since all true information according to the database is stored in the database instance.

An important notion is the consistency of a database.

DEFINITION 4.3 A formula ψ *follows* from a database instance $\mathcal{D}$ (alternatively, $\mathcal{D}$ *entails* ψ; notation: $\mathcal{D} \models \psi$) if the minimal Herbrand model $\mathcal{H}^{\mathcal{D}}$ of $\mathcal{D}$ is also a model of ψ. A database $\mathcal{DB} = (\mathcal{A}, \mathcal{D}, \mathcal{IC})$ is *consistent* if every formula in $\mathcal{IC}$ follows from $\mathcal{D}$. (Denoted as $\mathcal{D} \models \mathcal{IC}$.) Otherwise $\mathcal{DB}$ is *inconsistent*.

In the example 4.1, the database is consistent. When e.g. $father(paola, filip)$ is added to the database instance, the database would be inconsistent.

Issues in the integration of databases

The problem, we consider here, is the integration of databases. The goal is to combine n independent databases $\mathcal{DB}_1 = \langle \mathcal{A}_1, \mathcal{D}_1, \mathcal{IC}_1 \rangle, \ldots, \mathcal{DB}_n = \langle \mathcal{A}_n, \mathcal{D}_n, \mathcal{IC}_n \rangle$, called *sources*, in one new global database $\langle \mathcal{A}, \mathcal{D}, \mathcal{IC} \rangle$. We assume that all databases store related information, i.e. there exists a common information domain (formalized by $\langle \mathcal{A}, \mathcal{D}, \mathcal{IC} \rangle$). This integration process faces three (consecutive) problems[1]:

1. The integration of the alphabets.
   Since each database $\langle \mathcal{A}_i, \mathcal{D}_i, \mathcal{IC}_i \rangle$, $1 \leq i \leq n$ may have its own alphabet, the first step is to express formally the relationships between the different alphabets. This phase is called the *integration of ontologies* because here different views (alphabets) on the same information domain are related. These relations exists because of our assumption of related information domains.

[1] In the literature all these phases are often denoted with the term *data integration*. As each phase makes sometimes opposite assumptions, the term is often misleading and must be used with care. Also, the problems are presented in the usual order of application; it is, however, possible to combine these phases in one large integration process.

However, these relations often are not exact because either one alphabet describes a subdomain of that of the other alphabet, or there is a difference in abstraction level: one alphabet describes the information domain in more detail than the other.

2. The integration of the integrity constraints.
The next phase is to build a classical consistent set of integrity constraints for the global database. The global integrity constraints may either originate from the integrity constraints of the sources, or they are new formulas which describe new requirements which is not present in the (union of the) sources. The latter are a natural consequence from the fact that in general the domain of the global database is larger that each of the sources.

3. The integration of the database instances.
The last phase concerns the construction of a consistent global database instance. Ideally, the union of the database instances of the sources satisfies. However, often this is not the case (it happens even if each source is individually consistent). Because inconsistency makes logical inference trivial (anything follows from an inconsistent theory), adapted techniques are needed: either the use of logics that do not degrade in case of inconsistency (*para-consistent logics*), or the application of restoration methods that allow to remove the inconsistencies from an inconsistent database instance (*coherent integration methods*).

On each of these topics one finds a vast amount of literature. Indeed, the integration of databases is a complex task, which demands solutions to questions from different disciplines, such as belief revision, merging and updating, reasoning with inconsistent information, constraint enforcement, query processing and many aspects of knowledge representation.

In this chapter we consider the first and the last phase, mostly from a knowledge representation perspective. The first section, considers the first phase in a specific problem context: namely the construction of mediator based systems. These systems construct the global database as a virtual database. This problem context will not consider integrity constraints and hence inconsistencies will not arise. The problems that do arise are due to the differences in the alphabets and the differences in the actual stored information in the sources.

The integration of database instances is the subject of the other sections in this chapter. Here it will be assumed that all database instances are in the same alphabet and that a consistent set of global integrity constraints are given. The second section will present an ID-Logic solution to the problem, together with an elaboration on how to use the $\mathcal{A}$system to compute the restored database instances. The last section discusses a novel method to compute the restored databases based on a special encoding, called signed formulas.

Our contributions

This chapter presents solutions to the information integration problem from a knowledge representation perspective. In our approach, the problems are formalized using ID-Logic as the modeling language. Based on this representation, we define and study abductive reasoning to solve the above problem tasks.

In the first section the construction of a mediator-based system is considered for non-conflicting sources. The resulting framework unifies and generalizes many existing approaches in the literature. (This work is published as [262].) The problem of consistency is tackled in the consequent sections. Section 4.3 presents an abductive method for the construction of coherent data out of inconsistent data, based on the notion of a repair for an inconsistent database. We show that the $\mathcal{A}$system enhanced with an appropriate optimizer, is able to compute the right (preferred) repairs. (Our contributions are found in [26, 23, 24].) However, computationally the abductive approach is not necessarily the best one. Therefore an alternative method for finite databases is considered in the last section. (The related publication is [25].) The problem is encoded as a propositional theory via a simple transformation. With this encoding, we experimentally verify the computational feasibility of the different inference systems for this problem. Although limited, the experiments show that this transformation together with the best performing systems may be used to build automated database repairing systems.

We like to mention here that the work on coherent consistency restoration has been the result of intensive collaboration between all authors. Especially Ofer Arieli has contributed many of the presented ideas.

4.2 Mediator based systems

This section concerns the integration problem for a set of independent databases from which information is wanted. Querying each database individually is tedious and time-consuming: because each database has its own alphabet, the query must be translated for each data source. Also, often not all information for an answer is available in one source, which forces to divide the query in many subparts, one for every relevant source. A solution to this problem is a system that provides a uniform interface to the databases in which the above query transformation tasks are automated.

A naive solution is to merge all information stored in the databases into a new large database. Often this is not an option. The control of the sources often is outside the power of the designers of the integrating system. Also, materializing all composed information would result only in a snap shot integration because a source can update its knowledge after the integration. That leads to differences in the integrated knowledge base and the actual information stored in the sources. Moreover if the source database is very large, the materializing of the merged knowledge often is physically impossible.

One solution is the use of *mediator-based systems* [179, 252, 178]. These systems consist of an alphabet, called the *global schema* (representing the global information), and a structure, the *mediator*, that links the information of the sources with the global schema, such that a virtual knowledge-base in terms of the global schema is obtained. Such systems confront the user only with the global schema as if there is a single information source, hiding all internal complexity. To get information, the user expresses his queries in terms of the global ontology. The mediator-based system answers the query by exploiting the links between the global schema and sources in order to retrieve the necessary information.

This section tackles the problem of the construction of a mediator-based system where the global schema is given.[2] More precisely, there is an *intended* database, which is described by using the global schema and which has to be constructed from the sources. Note that the intended global database is not necessarily the union of the information residing in the sources.

A first issue concerns the *ontological relationships*. Each source alphabet, including the global alphabet, reflects its own view on the information domain. Therefore, the different alphabets are related with each-other and these relations can be expressed in a logic theory. This theory often consists of definitions defining the predicates of one ontology in terms of the other(s). However, the differences in views between the alphabets seldom allow a perfect match, and so the used specification logic has to provide support to express the correct relation.

[2]A related, yet different, problem is the (automated) construction of a global schema out of the ontologies of the source databases. It is a form a knowledge discovery, which is outside the scope of this thesis. At this moment, most approaches to this problem, e.g. [43, 106], are premature.

EXAMPLE 4.2 Consider the relations *student*$(\cdot)$ and *girlstudent*$(\cdot)$, each belonging to a different ontology. Both are related, describing students, however there is no ontologically perfect match possible. The first relation describes all students, male and female, while in the second all students that are female.

There are two common methods to define ontological relations: one, called *Global-as-View* (GAV) [252], expresses the relations of the global schema in terms of those of the sources. The other, called *Local-as-View* (LAV) [179], defines each source relation in terms of the relations of the global schema.

The second issue concerns the actually stored information by a source. A source may have complete or incomplete knowledge about its relations. More of interest, the source may have complete or incomplete knowledge about a relation in the global ontology. (This is called *relative (in)completeness* w.r.t. a relation.) In the case of complete knowledge, the source is able to describe the knowledge of the global relation in terms of its own relations. Otherwise the source can only specify partial information about the global relation. This incompleteness might be complemented by other sources, so that together the sources posses complete knowledge about (part of) the intended global database. Note that even if all sources have complete knowledge about their own relations there might be incomplete knowledge about the global relations.

EXAMPLE 4.3 (EXAMPLE 4.2 CONTINUED) Suppose that the *student*$(\cdot)$ relation is given by a source, and *girlstudent*$(\cdot)$ is part of the global schema. Ontologically there is not perfect match possible because the source lacks the knowledge about the sex of the students. A weaker statement is, however, possible: each girl student is a student. When we add an extra source that stores the sex of a student, the relation *girlstudent* is perfectly definable.

One method to represent this information is by labels as e.g. in [40]. There, the possible relations between the knowledge a source has about a relation of the global database are designated by different labels: a source that contains a proper subset [respectively, superset] of the intended knowledge of the corresponding global relation is labeled *open* [respectively, *closed*]. A source whose information is the same as that of the global one is labeled *clopen*.

Sometimes when the sources have incomplete knowledge about their relations, the sources might provide extra meta-information about the actual stored information. This may be exploited to define more precise relations.

EXAMPLE 4.4 (EXAMPLE 4.3 CONTINUED) Suppose it is known that the source that stores information about the students has incomplete knowledge about the student relation. Although there is ontologically no perfect match possible with the global relation *girlstudent*, it might be that the source stores only the female students. Then there is a de facto perfect match.

Note that this kind of information is very sensitive to changes, and hence not very reliable.

In the following we assume that the sources have complete knowledge about their relations. Hence the incompleteness of the global database is purely due to the ontological differences.

Commonly, the above discussed issues are tackled by the definition of a logic theory that expresses the inter-relation (ontological) relationships. The incompleteness of the global database is formalized by the existence of multiple models for the logic theory. When this is settled, a last issue which has to be resolved, is the definition of an appropriate inference procedure in order to answer queries. Different semantics [3] have been used for query answers (in terms of the global schema):

- According to the *certain answer* semantics a tuple $[t]$ is an answer to a global query $\mathcal{Q}$ if its true in all models of the logic theory representing the global database.
- According to the *possible answer* semantics a tuple $[t]$ is an answer of a query $\mathcal{Q}$ if $[t]$ holds in at least one model of the logic theory representing the global database.

Because the information of the sources is not materialized at the global level, the answers are typically computed in two phases. First the (global) query is transformed into an equivalent query in terms of the source ontologies by using the logical structure and then, secondly, this query is evaluated w.r.t. the sources. Among query answering algorithms are Bucket [179], Minicon [217] and Inverse-rule [109].

In the following, we present an ID-Logic framework for the logical structure of a mediator-based system. With each source database are associated: an ID-Logic theory to represent the source and an ID-Logic theory encoding the relationships with the global intended database. All these ID-Logic theories are composed together such that a knowledge base is formed describing the intended global database in terms of the sources. Contrary to most approaches in the literature the completeness or incompleteness of knowledge is addressed explicitly. This leads to a uniform treatment of the GAV and LAV approaches and to a formalization (and generalized use) of the labels of [40].

The query answering procedure is provided by the $\mathcal{A}$system. In general a mediator-based system contains a lot of incomplete knowledge, often enforced by the dynamics of the set of the sources e.g. when a source is temporary unavailable. It turns out that if the GAV approach is taken, an abductive problem is obtained; while the use of LAV results in a model generation problem. In the first case, the application of abductive inference has two advantages. On the level of computations, the rewritten queries can simply be extracted from the abductive answers w.r.t. a specific ID-Logic theory. This theory is obtained by the merging of only the theories that encode the relationships between the sources and the intended global database. The second advantage is the construction of more informative answers when sources are (temporary) lacking. The abductive solution provides a supporting trace in that case, which can be used to inform the user about the availability of the information (or a degree of confidence). We show this by an example.

4.2.1 Mediator-based systems

In the ontological integration problem, the basic elements are source databases. A *source database* $\mathcal{DB}$, or in short a source, is a database $\mathcal{DB} = \langle \mathcal{A}, \mathcal{D}, \mathcal{IC} \rangle$ as defined previously. Because consistency is not relevant in this case, the integrity constraints have no role in the reasoning. Thus, for the sake of a simpler presentation, a source database is denoted shortened as $\langle \mathcal{A}, \mathcal{D} \rangle$.

EXAMPLE 4.5 A source describing a set of students may have the following structure: $\mathcal{S}_1 = \langle \{student(\cdot)\}, \{student(john), student(mary)\} \rangle$

Given a set of source databases $\mathcal{S}_1, \ldots, \mathcal{S}_n$, the goal is to setup a mediator-based system which combines the information stored in the sources according to the ontological relations.

DEFINITION 4.4 A *mediator-based system* $\mathfrak{G}$ is a triple $\langle \mathcal{L}, S, M \rangle$, where $\mathcal{L}$ is the first-order language of the integrated (or global) database, $S = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ is a set of source databases, and M is a logic theory that formalizes the relationships between the ontologies of the sources and the intended global database, based on $\mathcal{L}$.

To simplify the presentation, we assume that each predicate is uniquely defined by either a source or the global schema[3]. We also accept the unique domain assumption: all involved database languages share the same domain, i.e. all constant and function symbols are shared and have everywhere the same interpretation. Hence, as in the above example, the language of a database is completely determined by its set of predicates. Furthermore, in all examples, the amount of information in a source about a predicate will be exactly the intended meaning.

EXAMPLE 4.6 Consider the following two data-sources:

$\mathcal{S}_1 = \langle \{student(\cdot)\}, \{student(john), student(mary)\} \rangle,$
$\mathcal{S}_2 = \langle \{enrolled(\cdot, \cdot)\}, \{enrolled(john, 1999), enrolled(mary, 2000)\} \rangle.$

A possible mediator-based system $\mathfrak{G}$ for these sources is $\langle \mathcal{L}, S, M \rangle$, where

$\mathcal{L} = \{st99(\cdot)\}, \; S = \{\mathcal{S}_1, \mathcal{S}_2\}$, and

[3]If needed, a simple renaming of predicates in sources can establish this property.

$$M = (\{\{\forall\, X.st99(X) \leftarrow student(X) \wedge enrolled(X, 1999)\}\}, \emptyset)^4$$

Queries w.r.t. $\mathfrak{G}$ will be first-order formulas over $\mathcal{L}$ (*e.g.* $\exists\, X.st99(X)$).

4.2.2 Denoting partial knowledge in ID-Logic

Our approach will represent the mediator as an ID-Logic theory. Because in mediator based systems incomplete knowledge has a central role and we like to have an explicit grip on the incompleteness, we will introduces auxiliary open predicates to pinpoint the incomplete knowledge. In this subsection we explain how auxiliary open predicates can be used to complete the available partial knowledge so that a correct definition is specified.

An incomplete set of rules.

Suppose that the set of rules $\{p(\overline{t}) \leftarrow B_i | i = 1..k\}$ only partially defines p. A complete definition can be obtained by adding a rule $p(\overline{s}) \leftarrow p^*(\overline{s})$, in which the auxiliary open predicate p^* represents all the tuples in p that are not defined by any of the bodies B_i. To ensure that the tuples in p^* do not overlap with the other tuples, the integrity constraint $\forall\ (p^*(\overline{s}) \rightarrow \neg(B_1 \vee \ldots \vee B_k))$ can be added.

An over-estimating rule

Another type of incompleteness occurs when the body of a rule $p(\overline{t}) \leftarrow B$ is overly general, *i.e.*, includes tuples not intended to be in the relation p. This can be repaired by adding to the body an auxiliary open predicate p^s that filters the extraneous tuples. The completed rule in this case is $p(\overline{t}) \leftarrow B \wedge p^s(\overline{t})$.

Example 4.7 Assume that $st99(\cdot)$ has to be defined in terms of the predicate $student(\cdot)$. The rule $st99(X) \leftarrow student(X)$ is overly general since not all students did enroll in 1999. By adding an auxiliary predicate $st99^s(\cdot)$, denoting all persons enrolled during 1999, the revised rule $st99(X) \leftarrow student(X) \wedge st99^s(X)$ correctly defines $st99(\cdot)$.

4.2.3 An ID-Logic mediator-based system

We now have the necessary ingredients to formalize mediator-based systems in ID-Logic.

Definition 4.5 (An ID-Logic mediator-based system) Consider a set of sources $\{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ and a global schema $\mathcal{L}_G$. Suppose that all source languages $\mathcal{L}_{\mathcal{S}_i}, 1 \leq i \leq n$ and $\mathcal{L}_G$ are mutually distinct, i.e. no predicate symbol is shared by two languages, and share the same domain, i.e. the constant and function symbols.

[4] M is here an ID-Logic theory

$\mathcal{L}$ is the common first order language obtained as the union of all source languages $\mathcal{L}_{\mathcal{S}_i}, 1 \leq i \leq n$ and the global schema $\mathcal{L}_G$.

An ID-Logic mediator-based system for the sources $\{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ and the global language $\mathcal{L}_G$ is $\mathfrak{G} = \langle \mathcal{L}_G, S, M \rangle$ where

- S is a set of ID-Logic theories $\{S_1, \ldots, S_n\}$, each based on the language $\mathcal{L}$, encoding the content of the source databases.
- M is a set of ID-Logic theories $\{\mathcal{W}_1, \ldots, \mathcal{W}_n, \mathcal{K}\}$, each based on the language $\mathcal{L}$, encoding the relationships between the sources and the intended global database:
  - $\mathcal{W}_i$, i=1..n, are ID-Logic theories that relate the global language $\mathcal{L}_G$ with the source $\mathcal{S}_i$, and
  - $\mathcal{K}$ is an ID-Logic theory containing the knowledge about how the information in the different sources complement each other.

and the knowledge of the ID-Logic mediator-based system $\mathfrak{G}$ is represented by the ID-Logic theory[5]

$$\mathcal{T} = S_1 \circ \cdots \circ S_n \circ \mathcal{W}_1 \circ \cdots \circ \mathcal{W}_n \circ \mathcal{K}$$

based on the language $\mathcal{L}$.

We are now going to describe how to construct the different ID-Logic theories in a mediator based system.

The sources

Consider the source $\mathcal{S} = \langle \mathcal{L}_{\mathcal{S}}, I \rangle$. According to our definition of a database, a database instance I is a set of atoms which are true, all other atoms of $\mathcal{L}_{\mathcal{S}}$ are false. This view can be modeled as an ID-Logic theory $S = (\{I'\}, \emptyset)$ having a single definition I' in which all predicates of $\mathcal{L}_{\mathcal{S}}$ are defined as an enumeration of facts. Each fact is constructed from an atom of I. In case that there are no facts available for a predicate, this predicate has an empty definition.

Example 4.8 (Example 4.5 revisited) The source

$$\mathcal{S} = \langle \{student(\cdot)\}, \{student(john), student(mary)\} \rangle$$

is interpreted as the ID-Logic theory

$$S = \left(\left\{ \left\{ \begin{array}{l} student(john). \\ student(mary). \end{array} \right\} \right\}, \emptyset \right)$$

[5] $\circ$ represents the composition operation defined in 2.2.2 as the pairwise union of the theories.

Relating one source with the global schema: $\mathcal{W}$

This part defines the relationships between the relations in a source and those in the global database. These relationships are expressed in the form of (inductive) definitions, taking into account the ontological relationships between the predicates and the actual knowledge of the source. The techniques described in Section 4.2.2 are used when there is a mismatch between the information in the source and in the global database.

DEFINITION 4.6 (A SOURCE MAPPING) A source mapping from the vocabulary of a language $\mathcal{L}_2$ to the vocabulary of a language $\mathcal{L}_1$ is an ID-Logic theory $\mathcal{W}$ defining the predicates of $\mathcal{L}_1$ in terms of the predicates of $\mathcal{L}_2$ and of the necessary auxiliary open predicates.

Local-as-View (LAV) and Global-as-View (GAV) are particular instances of source mappings. For a source with vocabulary $\mathcal{L}_S$ and a global database with vocabulary $\mathcal{L}_G$, LAV (GAV) defines the predicates of $\mathcal{L}_S$ ($\mathcal{L}_G$) in terms of the predicates of $\mathcal{L}_G$ ($\mathcal{L}_S$).

EXAMPLE 4.9 For the languages $\mathcal{L}_1 = \{st99(\cdot)\}$ and $\mathcal{L}_2 = \{student(\cdot)\}$, where $st99(\cdot)$ means the students enrolled in 1999 and $student(\cdot)$ all students, the possible source mappings are

$$\mathcal{W}_{1\to 2} = (\{\{st99(X) \leftarrow student(X) \wedge student^s(X).\}\}, \emptyset) \quad (1)$$

$$\mathcal{W}_{2\to 1} = (\{\{student(X) \leftarrow st99(X) \vee st99^*(X).\}\}, \{\forall X.st99^*(X) \rightarrow \neg st99(X).\}) \quad (2)$$

The meaning of the predicates allows only these two representations. When $\mathcal{L}_1$ is the source predicate, the first mapping is LAV and the second is GAV. In a concrete situation, the auxiliary predicates get a precise domain specific meaning. The auxiliary predicate $st99^*(\cdot)$ represents the students that are not enrolled in 1999, while $student^s(\cdot)$ represents the students enrolled in 1999.

REMARK 4.2.1 (GAV OR LAV?) According to our definition GAV and LAV are equivalent in the sense that they can model the same ontological relations.[6] However, as it is more natural to define abstract concepts in terms of more detailed notions, differences in abstraction levels of the source and the global languages imply that in practice one approach could be more appropriate than the other. Moreover, since the abstraction levels of the sources' languages may also be different (some of which may be more abstract than the global language and some may be less abstract), it makes sense to combine both approaches, i.e. to use

[6] In the literature one finds arguments in favor of one or the other. For our representational point of view there is no difference. However, in the section on query answering, we give an argument in favor of GAV.

GAV for the source mappings between certain sources and the global schema, and LAV for the mappings between the other sources and the global schema. The fact that our framework supports such a combination may serve, therefore, as one of its advantages over other formalisms.

As mentioned in the introduction, [40] defined special labels to denote the relative (in)completeness of a source w.r.t. a global relation. This was done in the context of LAV mappings between one source relation and one global relation. The following example illustrates that our use of open auxiliary predicates exactly captures the meaning of those labels. It is a variant on the world cup example in [40].

1. *closed source*: The source can only provide an **over-estimate** of the global relation.
   Consider the languages $\mathcal{L}_G = \{st99(\cdot)\}$ and $\mathcal{L}_S = \{student(\cdot)\}$. The mapping is given by

$$\left(\begin{array}{l} \{\{ \ student(X) \leftarrow st99(X) \vee student^*(X). \ \}\} \, , \\ \\ \{ \ \forall \, X.student^*(X) \rightarrow \neg st99(X). \ \} \end{array} \right)$$

   In this case $student^*(\cdot)$ represents the students that are not enrolled in $st99(\cdot)$.

2. *open source*: the source provides a **under-estimate** of the global relation.
   Suppose that $\mathcal{L}_G = \{st99(\cdot)\}$ and $\mathcal{L}_S = \{st99male(\cdot)\}$. Then:

$$(\{\{ \ st99male(X) \leftarrow st99(X) \wedge st99male^s(X). \ \}\} \, , \emptyset)$$

   Here the auxiliary predicate $st99male^s(X)$ represents a unknown selection on the elements of $st99(X)$, i.e. the male students enrolled in 1999.

3. *clopen source*: the source has **exact** information for the mediator.
   Suppose that $\mathcal{L}_G = \{st99(\cdot)\}$ and $\mathcal{L}_S = \{studentsOf1999(\cdot)\}$.

$$(\{\{ \ studentsOf1999(X) \leftarrow st99(X). \ \}\} \, , \emptyset)$$

The knowledge of multiple sources: $\mathcal{K}$

The last component, denoted by $\mathcal{K}$, in the composition of the ID-Logic theories, introduced in Definition 4.5 for representing a mediator-based system, allows the mediator designer to formulate additional meta-knowledge about how partial information of one source (regarding a certain predicate of the global schema) is completed by data of other sources. As shown below, its information is vital for a proper schema integration.

EXAMPLE 4.10 Consider the global schema $\{student(\cdot)\}$ and the sources $\mathcal{S}_1 = \langle\{st99(\cdot)\}, \{st99(john)\}\rangle$ and $\mathcal{S}_2 = \langle\{st00(\cdot)\}, \{st00(mary)\}\rangle$ having the source mappings

$$\begin{aligned}\mathcal{W}_{G\to 1} &= (\{\{student(X) \leftarrow st99(X) \vee st99^*(X).\}\}, \{\forall\, X.st99^*(X) \rightarrow \neg st99(X).\})\\ \mathcal{W}_{G\to 2} &= (\{\{student(X) \leftarrow st00(X) \vee st00^*(X).\}\}, \{\forall\, X.st00^*(X) \rightarrow \neg st00(X).\})\end{aligned}$$

Note that $\mathcal{W}_{1\to G} \circ \mathcal{W}_{2\to G}$ contains two (alternative and equivalent) definitions for the student relation. The statement that the relation $student(\cdot)$ is complete w.r.t. the set of sources $\{\mathcal{S}_1, \mathcal{S}_2\}$ can be formalized by the first-order expression

$$\mathcal{K} = (\emptyset, \{\forall\, X.\neg(st99^*(X) \wedge st00^*(X)).\})$$

Of course, there is no general rule for expressing such meta-knowledge. It depends on the representation choices of the source mappings, the actual information content of the sources and the intended information content of the global database.

An elaborated example

We conclude this section with an elaboration of Example 4.6. It shows, in particular, that a certain data integration problem can be described by many different mediator-based systems.

EXAMPLE 4.11 Consider the two sources, each one has complete knowledge about its relations. Source $\mathcal{S}_1$ stores all full-time students, and source $\mathcal{S}_2$ contains data about the year of enrollment of all students (including part-time and full-time ones):

$$\begin{aligned}\mathcal{S}_1 &= \langle\{student(\cdot)\}, && \{student(john), student(mary), student(bob)\}\rangle,\\ \mathcal{S}_2 &= \langle\{enrolled(\cdot,\cdot)\}, && \{enrolled(john, 1999), enrolled(eve, 1999),\\ & && \;enrolled(mary, 2000), enrolled(alice, 2003)\}\rangle\end{aligned}$$

A mediator-based system that extracts lists of full-time students enrolled in the years 1999 and 2000 looks as follows: $\mathfrak{G} = \langle \mathcal{L}_G, \{S_1, S_2\}, M\rangle$, where:

- $\mathcal{L}_G = \{st99(\cdot), st00(\cdot)\}$
- $S_1 = \left(\left\{\left\{\begin{array}{l} student(john).\\ student(mary).\\ student(bob).\end{array}\right\}\right\}, \emptyset\right)$
- $S_2 = \left(\left\{\left\{\begin{array}{l} enrolled(john, 1999).\\ enrolled(eve, 1999).\\ enrolled(mary, 2000).\\ enrolled(alice, 2003)\end{array}\right\}\right\}, \emptyset\right)$

- The GAV approach where each source is individually related with the global schema.

$$\mathcal{W}_{G\to 1} = \left(\left\{ \left\{ \begin{array}{l} st99(X) \leftarrow student(X) \wedge st99^{s}_{\mathcal{S}_1}(X). \\ st00(X) \leftarrow student(X) \wedge st00^{s}_{\mathcal{S}_1}(X). \end{array} \right\} \right\}, \emptyset \right)$$

$$\mathcal{W}_{G\to 2} = \left(\left\{ \left\{ \begin{array}{l} st99(X) \leftarrow enrolled(X,1999) \wedge st99^{s}_{\mathcal{S}_2}(X). \\ st00(X) \leftarrow enrolled(X,2000) \wedge st00^{s}_{\mathcal{S}_2}(X). \end{array} \right\} \right\}, \emptyset \right)$$

$$\mathcal{K} = \left(\emptyset, \left\{ \begin{array}{l} \forall\, X.st99^{s}_{\mathcal{S}_1}(X) \leftrightarrow enrolled(X,1999). \\ \forall\, X.st00^{s}_{\mathcal{S}_1}(X) \leftrightarrow enrolled(X,2000). \\ \forall\, X.st99^{s}_{\mathcal{S}_2}(X) \leftrightarrow student(X). \\ \forall\, X.st00^{s}_{\mathcal{S}_2}(X) \leftrightarrow student(X). \end{array} \right\} \right)$$

- An alternative GAV approach that treats the two sources as if they were one.

$$\mathcal{W}_{G\to\{1,2\}} = \left(\left\{ \left\{ \begin{array}{l} st99(X) \leftarrow student(X) \wedge enrolled(X,1999). \\ st00(X) \leftarrow student(X) \wedge enrolled(X,2000). \end{array} \right\} \right\}, \emptyset \right)$$

- The LAV approach

$$\mathcal{W}_{1\to G} = \left(\begin{array}{l} \{\{\ student(X) \leftarrow st99(X) \vee st00(X) \vee student^{*}(X).\ \}\}, \\ \{\forall\, X.student^{*}(X) \rightarrow \neg(st99(X) \vee st00(X)).\} \end{array} \right)$$

$$\mathcal{W}_{2\to G} = \left(\begin{array}{l} \left\{ \left\{ \begin{array}{ll} enrolled(X,Y) \leftarrow & st99(X) \wedge Y = 1999. \\ enrolled(X,Y) \leftarrow & st00(X) \wedge Y = 2000. \\ enrolled(X,Y) \leftarrow & enrolled^{*}(X,Y) \wedge \\ & (Y \neq 1999 \vee Y \neq 2000). \end{array} \right\} \right\}, \\ \\ \left\{ \begin{array}{l} \forall\, X,Y.enrolled^{*}(X,Y) \rightarrow \neg(st99(X) \wedge Y = 1999). \\ \forall\, X,Y.enrolled^{*}(X,Y) \rightarrow \neg(st00(X) \wedge Y = 2000). \end{array} \right\} \end{array} \right)$$

$$\mathcal{K} = \left(\emptyset, \left\{ \begin{array}{l} \forall\, X.st99(X) \rightarrow student(X). \\ \forall\, X.st00(X) \rightarrow student(X). \\ \forall\, X.st99(X) \rightarrow enrolled(X,1999). \\ \forall\, X.st00(X) \rightarrow enrolled(X,2000). \end{array} \right\} \right)$$

According to any one of the representations above, the unique model of $\mathfrak{G}$ (restricted to $\mathcal{L}_G$) is $\{st99(john), st00(mary)\}$.

To show the versatility of our ID-Logic framework, we reverse in the next example the vocabularies.

EXAMPLE 4.12 Again consider two sources, but now source $\mathcal{S}_1$ stores all students enrolled in 1999 en source $\mathcal{S}_2$ those that enrolled in 2000.

$$\mathcal{S}_1 = \langle\{st99(\cdot)\}, \{st99(john), st99(eve)\}\rangle,$$
$$\mathcal{S}_2 = \langle\{st00(\cdot)\}, \{st00(mary)\}\rangle$$

The goal is to build a mediator-based system $\mathfrak{G}$ that expresses if a person is a student ($student(\cdot)$) and his enrollment year ($enrolled(\cdot,\cdot)$). Thus, $\mathfrak{G} = \langle\mathcal{L}_G, \{S_1, S_2\}, M\rangle$, where:

- $\mathcal{L}_G = \{student(\cdot), enrolled(\cdot,\cdot)\}$
- $S_1 = \left(\left\{\left\{\begin{array}{l} st99(john). \\ st99(eve). \end{array}\right\}\right\}, \emptyset\right\}$
- $S_2 = \left(\left\{\left\{\ st00(mary).\ \right\}\right\}, \emptyset\right\}$

The Global-As-View approach yields the following mediator structure.

$$\mathcal{W}_{G\to 1} = \left(\begin{array}{l} \left\{\begin{array}{l} \left\{\ student(X) \leftarrow st99(X) \vee student^*_{\mathcal{S}_1}(X).\ \right\}, \\ \left\{\begin{array}{ll} enrolled(X,Y) \leftarrow & (st99(X) \wedge Y = 1999) \\ enrolled(X,Y) \leftarrow & enrolled^*_{\mathcal{S}_1}(X,Y). \end{array}\right\} \end{array}\right\}, \\ \\ \left\{\begin{array}{l} \forall\ X.student^*_{\mathcal{S}_1}(X) \rightarrow \neg st99(X). \\ \forall\ X,Y.enrolled^*_{\mathcal{S}_1}(X,Y) \rightarrow \neg(st99(X) \wedge Y = 1999). \end{array}\right\} \end{array}\right)$$

$$\mathcal{W}_{G\to 2} = \left(\begin{array}{l} \left\{\begin{array}{l} \left\{\ student(X) \leftarrow st00(X) \vee student^*_{\mathcal{S}_2}(X).\ \right\}, \\ \left\{\begin{array}{ll} enrolled(X,Y) \leftarrow & (st00(X) \wedge Y = 2000). \\ enrolled(X,Y) \leftarrow & enrolled^*_{\mathcal{S}_2}(X,Y). \end{array}\right\} \end{array}\right\}, \\ \\ \left\{\begin{array}{l} \forall\ X.student^*_{\mathcal{S}_2}(X) \rightarrow \neg st00(X). \\ \forall\ X,Y.enrolled^*_{\mathcal{S}_2}(X,Y) \rightarrow \neg(st00(X) \wedge Y = 2000). \end{array}\right\} \end{array}\right)$$

$$\mathcal{K} = \left(\emptyset, \left\{\begin{array}{l} \forall\ X.\neg(student^*_{\mathcal{S}_1}(X) \wedge student^*_{\mathcal{S}_2}(X)). \\ \forall\ X,Y.\neg(enrolled^*_{\mathcal{S}_1}(X,Y) \wedge enrolled^*_{\mathcal{S}_2}(X,Y)). \end{array}\right\}\right)$$

The Local-As-View is as follows.

$$\mathcal{W}_{1\to G} = (\ \{\{\ st99(X) \leftarrow student(X) \wedge enrolled(X, 1999).\ \}\}, \emptyset\ )$$

$$\mathcal{W}_{2\to G} = (\ \{\{\ st00(X) \leftarrow student(X) \wedge enrolled(X, 2000).\ \}\}, \emptyset\ )$$

$$\mathcal{K} = (\emptyset, \emptyset)$$

The unique model of $\mathfrak{G}$ is here {student(john), student(eve), student(mary), enrolled(john,1999), enrolled(eve,1999), enrolled(mary,2000)}.

Both examples show that irrespective of the chosen representation style the relationships between the languages can be correctly specified. However, in each example one of the approaches leads to a more concise representation. In example 4.11 this is the (simplified) GAV version and in example 4.12 it is the LAV version. Both concise encodings share that they follow closely the natural definitional direction: the defined (head) predicate is the more abstract concept which is defined in terms of more fine-grained relations (the body). This indicates that the favorable approach depends strongly on the involved vocabularies.

4.2.4 Query Answering

In the previous sections we have shown how to set up an ID-Logic mediator-based system. This section discusses how queries can be answered with respect to such system. First, we will recall the general context of ID-Logic theories and then concentrate on abductive inference as a general technique of computing answers posed to mediator-based systems. We will also argue that the answers generated by the abductive process are more informative than those produced by other techniques.

DEFINITION 4.7 (TYPES OF QUERIES) Let $\mathcal{T}$ be an ID-Logic theory and $\mathcal{Q}$ a query.

a) $\mathcal{Q}$ is *skeptically true* iff it is entailed by $\mathcal{T}$. I.e.,

$$\mathcal{T} \models_{skep} \mathcal{Q} \text{ iff } \forall\, M \in Mod(\mathcal{T}) : M \models \mathcal{Q}.$$

b) $\mathcal{Q}$ is *credulously true* iff it is satisfied by $\mathcal{T}$. I.e.,

$$\mathcal{T} \models_{cred} \mathcal{Q} \text{ iff } \exists\, M \in Mod(T) : M \models \mathcal{Q}.$$

In a mediator-based system, the sources contain the extensional knowledge. Thus, answers to queries that are supported by the sources will always be skeptically true, while answers to queries for which the sources have no complete information might be either skeptically false or credulously true.

As the mediator-based system does not materialize the knowledge of the sources in its own schema, the process of answering a global query $\mathcal{Q}$ is two-phased:

a) Computing for $\mathcal{Q}$ an equivalent (if possible) query $\mathcal{Q}_s$ expressed in terms of source languages. Note that $\mathcal{Q}_s$ can be a disjunction.

b) Querying the sources with $\mathcal{Q}_s$.

In what follows we show how to compute $\mathcal{Q}_s$ by abductive reasoning.

Answers for Queries

Consider a mediator-based system $\mathfrak{G} = \langle \mathcal{L}_G, \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}, \{\mathcal{W}_1, \ldots, \mathcal{W}_l, \mathcal{K}\}\rangle$, and the derived ID-Logic theory $\mathcal{T}_q = \mathcal{W}_1 \circ \cdots \circ \mathcal{W}_l \circ \mathcal{K}$. The information of this theory contains only the relationships between the languages. Then a new query $\mathcal{Q}_s$ expressed in terms of $\mathcal{L}_G$ is *derivable* from an abductive solution for the query $\mathcal{Q}$ w.r.t. $\mathcal{T}_q$.

According to the mapping style, the abductive solution forms the new query $\mathcal{Q}_s$ or the basis to compute it. In GAV the open predicates are the source predicates, and so, an abductive solution *Sol* is in terms of the source predicates. *Sol* forms a representation of $\mathcal{Q}_s$. The LAV case is different: the open predicates are those of the global schema $\mathcal{L}_G$. An abductive solution *Sol* does not encode directly $\mathcal{Q}_s$. However, all models satisfying *Sol* correspond to answers for $\mathcal{Q}_s$. Hence one has to design an extra procedure to compute answers form $\mathcal{Q}_s$ out of the abductive solution for LAV mappings. For example, the Inverse-rule for datalog [109] is such a procedure.

Example 4.13 (Computation of the transformed query) Consider a set of two sources

$$S = \{\mathcal{S}_1 = \langle\{st99(\cdot)\}, \{st99(john)\}\rangle, \mathcal{S}_2 = \langle\{st00(\cdot)\}, \{st00(mary)\}\rangle\}$$

We build for these sources two, a GAV and a LAV based, mediator-based systems for the global schema $student(\cdot)$. It is supposed that the set of sources have *incomplete* knowledge about the $student(\cdot)$ relation, i.e. there exists other (not known yet) sources which complete the information.

- $\mathfrak{G}^{GAV} = \langle\{student(\cdot)\}, S, \{\mathcal{W}_{G\to\{1,2\}}\}\rangle$, where

$$\mathcal{W}_{G\to\{1,2\}} = (\{student(X) \leftarrow st99(X) \lor st00(X) \lor student^*(X)\}, \emptyset).$$

- $\mathfrak{G}^{LAV} = \langle\{student(\cdot)\}, S, \{\mathcal{W}_{1\to G}, \mathcal{W}_{1\to G}, \mathcal{K}\}\rangle$, where

$$\begin{aligned}
\mathcal{W}_{1\to G} &= (\{st99(X) \leftarrow student(X) \land student_1^s(X).\}, \emptyset).\\
\mathcal{W}_{2\to G} &= (\{st00(X) \leftarrow student(X) \land student_2^s(X).\}, \emptyset).\\
\mathcal{K} &= (\emptyset, \{\forall\, X.\neg(student_1^s(X) \land student_2^s(X).\})
\end{aligned}$$

Now suppose we want to know if *john* is a student. Therefore we pose the global query $\mathcal{Q} : \exists\ student(john)$ to the mediator-based systems. According our explanation above, the abductive solver is applicable for $\mathfrak{G}^{GAV}$. The $\mathcal{A}$system computes for $\mathcal{Q}$ with respect to $\mathfrak{G}^{GAV}$ three abductive solutions

$$\Delta_1 = \{st99(john)\}\ \Delta_2 = \{st00(john)\}\ \Delta_3 = \{student^*(john)\}$$

which can be compacted together in one disjunction:

$$\mathcal{Q}_s : st99(john) \vee st00(john) \vee student^*(john).$$

where each disjunct is an abductive solution. This formula is the rewritten query $\mathcal{Q}_s$ that was looked for.

The application of the $\mathcal{A}$system to $\mathfrak{G}^{LAV}$ is less useful. The computed abductive answer is the query itself $\Delta = \{student(john)\}$. This illustrates our arguments above that in case of a LAV representation abductive inference is not the right method for query answering. For instance, model generation is more appropriate. Such a system would generate three models

$$\begin{aligned} M_1 &= \{student(john), st99(john), student^s_1(john)\} \\ M_2 &= \{student(john), st00(john), student^s_2(john)\} \\ M_3 &= \{student(john)\} \end{aligned}$$

These correspond exactly to the three abductive solutions computed for the GAV mediator based system. Based on them one could correctly query the sources and answer the query $\mathcal{Q}$.

No matter which method is used, the sources $\mathcal{S}_1$ and $\mathcal{S}_2$ will be queried for information about *john*. In our example, source $\mathcal{S}_1$ contains the information that John is enrolled in 1999 and so its follows that John is a student. This is a skeptically true answer. When none of the sources has supporting information, e.g. about Eve, the query about the being-a-student will be supported by the assumption that this information is stored in some unknown source. The answer is then credulously true. In the GAV-case this is expressed by the third abductive solution, in which $student^*(eve)$ is true, while in the LAV case it is expressed by the fact that $student^s_1(eve)$ and $student^s_2(eve)$ are false.

In the GAV-example only the generated set of abducibles (Δ) is used to compose the rewritten query. In general, the constructed constraint stores, i.e. Δ^*, $\mathcal{E}$ and $\mathcal{FD}$, are also relevant and must be included. There are two ways for their use: 1) the sources can be queried with the conjunction of the atoms in Δ and the retrieved answer is verified afterwards by an extra constraint check w.r.t. the generated constraint stores Δ^*, $\mathcal{E}$ and $\mathcal{FD}$. 2) The constraint information is used to produce a more detailed query for the sources, and then the sources have to verify the added constraints themselves. Both have advantages. Because the former needs no extra machinery, it is immediately applicable. The drawback of this approach is that all computations have to be done by the mediator-based system. The latter is able to return less tuples because a part of the computational effort is shifted to the sources; however it requires an extra process to prepare the correct query.

Remark 4.2.2 (GAV or LAV? revisited) Example 4.13 illustrates that the design of a query answering procedure for mediator-based systems in the LAV case requires different, maybe more complex inference procedures. In case of a GAV mediator-based system, a top-down abductive solver is clearly well-suited. The abductive reasoning is goal-directed: from the abductive answers it is easy to select the information that must be retrieved from the sources.

LAV is more connected with bottom-up computational approaches, as illustrated in the example by the model-generation. In the literature, however, one finds mostly top-down procedures for LAV, e.g. the Inverse-rule and the Bucket algorithm [180]. The idea of these procedures is to reverse the LAV ontological relations so that a representation similar to the GAV one, is obtained. These methods have typically strong restrictions to ensure that the inverse relation exists. For some complex ontological relationships, the inverse relation might even not be definable, e.g. for recursive relationships.

For these reasons, GAV is the more preferred encoding. The query answering can rely on a general purpose abductive reasoner instead of limited special purpose reasoners. Since all methods have to deal with the inherent incompleteness, they will have inference steps in common with the abductive inference. Therefore, it would be interesting future work to compare the different systems described in the literature and study how much they differ from an abductive solver.

Supporting dynamics of a mediator-based system

Abductive inference is particularly useful when the mediator-based system is dynamic, i.e source databases are dynamically added or/and removed. In such a situation the produced abductive answer contains certain information that justifies the result and helps to understand it.

Consider, e.g., the following scenario: in the university restaurant one gets only a student reduction if he or she is registered in the university database as a student. When the source that contains all part-time students falls out, none of these students can get its reduction. Knowing that a data-source is not available wouldn't solve the problem, since there is no way to find out what kind of information that source contained. Only when the restaurant is informed that the list of part-time students is unavailable, it can question every person that is not recognized as a student if he or she is a part-time student. Such detailed information is represented by the computed explanation formula produced by the abductive reasoning. The intended behavior is obtained by a source removal operation. Given a mediator-based system $\langle \mathcal{L}_G, \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}, \{\mathcal{W}_1, \ldots, \mathcal{W}_l, \mathcal{K}\}\rangle$ and a corresponding ID-Logic theory

$$\mathcal{T} = \mathcal{S}_1 \circ \cdots \circ \mathcal{S}_n \circ \mathcal{W}_1 \circ \cdots \circ \mathcal{W}_l \circ \mathcal{K},$$

a removal of a source $\mathcal{S}_k$ $(1 \leq k \leq n)$ yields the following theory:

$$\mathcal{T}' = \mathcal{S}_1 \circ \cdots \circ \mathcal{S}_{k-1} \circ \mathcal{S}_{k+1} \circ \cdots \circ \mathcal{S}_n \circ \mathcal{W}_1 \circ \cdots \circ \mathcal{W}_l \circ \mathcal{K},$$

in which all predicates of $\mathcal{S}_k$ are open predicates.[7]

EXAMPLE 4.14 (EXAMPLE 4.13 CONTINUED) If source $\mathcal{S}_1$ drops out, the abductive answer $\{st99(john)\}$ will have no source to be queried. The system can report to the user that in order to answer skeptically the query it is necessary to wait until $\mathcal{S}_1$ is available again, and the information $st99(john)$ can be verified.

Adding a new source requires slightly more work. It requires the construction of a new source mapping and the addition of assertions that express how this new source complements the knowledge of the other sources (and vice versa). For the latter, it may happen that all sources are related with the new source, which leads to many changes. Fortunately, in most practical cases this is unlikely to happen.

4.2.5 Related Work

Classical methods for mediator-based systems

Mediator-based systems [178] are a solution for the problem of composing the information in data sources without materializing the composed knowledge. To build such systems, two approaches have been proposed:

- the Global-As-View [252]:
  This approach specifies a global predicate p as a view on the source database S.
  $$p(\overline{X}) \leftarrow \phi_S$$
  In GAV, this formula is interpreted commonly as a definition. Often it is assumed that the information of the global database is complete w.r.t. the sources.

- the Local-As-View [179, 180]:
  This approach is in principle the inverse of the GAV. Here a source predicate q is expressed as a view on the global intended database G.
  $$\psi_G \rightarrow q(\overline{X})$$
  Commonly, this formula is interpreted as an implication, assuming that the information of the global database is incomplete w.r.t sources.

People have argued in favor of both, however some of these arguments just refer to different assumptions on the problem. Their main conclusions are: GAV has a more natural reading and leads to less complex inference procedures for the query answering, while LAV is more flexible to changes in the mediator logic, and is able

[7] Many other mediator-based systems just remove all knowledge of the dropped source. This can be simulated here by replacing the source by the empty source, in which all predicates are false.

to deal more naturally with the incompleteness at the level of the global intended database.

Only recently [63], Calí et al. have argued that the incompleteness argument is just based on a difference in assumption on the problem domain. Having tackled the problem from a Knowledge Representation perspective, we are not surprised by this result. Incompleteness of knowledge is inherent to the mediator-based information integration problem and so irrespective of the chosen formulation the proof-procedure has to deal with incomplete knowledge.

Generalized methods for mediator-based systems

In most mediator-based systems one modeling approach (LAV or GAV) is exclusively applied. [125] shows that simple problem situations exist which cannot be dealt by only applying GAV or LAV. At least two extensions have been proposed to increase the expressive power of LAV and GAV paradigms.

> *GLAV approach ([126]).* This is an extension of LAV that allows to map a conjunctive query expression ϕ_G over the global schema into a conjunctive query expression ϕ_S over the sources [8]. This variant can be simulated in our framework by the introduction of an auxiliary predicate, say p, which has the view definition $\{p(\overline{t}) \leftarrow \phi_S\}$ w.r.t. the sources (ϕ_S). Using p, a LAV mapping can be constructed by $\{p(\overline{t}) \leftarrow \phi_G\}$.
>
> *Both-As-View (BAV) ([196]).* McBrien and Poulovassilis present a novel method that combines the advantages LAV and GAV in the presence of dynamic schemas. They show how LAV and GAV view definitions can be fully derived from BAV transformation sequences, and that BAV transformation sequences can be partially derived for LAV or GAV view definitions. We believe that these transformation sequences (or extensions of them) could be applied to translate BAV mapping into ID-Logic mappings.

Our framework can handle any combination of GAV and LAV, since it just means that in one source mapping, some predicates of the global schema are defined in terms of the source language and some predicates of the source language are defined in terms of the global schema.

Using Description Logics ([65])

Calvanese et al. present a general framework for data integration. It turns out that our framework encodes in a nice way this framework. In particular, they define labels (similar to [40]) to denote the amount of knowledge a mapping rule contributes to the global database. As shown before, this can be captured by the use of (auxiliary) open predicates.

[8] An equivalent extension for the GAV approach is straightforward.

In the literature one finds many approaches using Description logics [178, 28]. A remarkable statement in [65] concerns the appropriateness of Description Logics for data integration: due to the limitations on the use of variables, Description Logics are poor as query languages. Calvanese et al. argue that for the data integration problem the modeling language has to be good in that respect. Since ID-Logic imposes no restriction on the use of variables, and can be regarded as a very expressive Description Logic, it is not surprising that the use of ID-Logic leads to a general approach, enclosing many of the existing approaches.

Implementations

The two decades of active research on this topic has resulted in several information integration systems: two influential systems are the *Information Manifold* system [166, 179, 181] which applies the LAV approach and TSIMMIS (the Stanford-IBM Manager of Multiple Information Sources) [250, 130, 211, 210] which applies GAV. There are many more systems, but few apply abductive reasoning. We have been able to trace the following.

The COntext INterchange (COIN) [60, 136] project[9] used like us abductive logic programs to represent the mappings (using GAV). Their procedure is a Prolog implementation of the classic ALP framework [153] extended with CHR rules to encode integrity constraints. Similarly to our method that uses the $\mathcal{A}$system, they compute the rewritten query with their abductive procedure. Like other early GAV approaches the COIN system assumed that the global database is completely determined by the sources. In our opinion, as our method shows, this assumption may be lifted (provided the implementation does not rely on it). Like the $\mathcal{A}$system, this data integration system implements a first order top down abductive reasoner, hence in order to ensure termination recursive relations are not allowed.

Another abductive approach is by Xanthakos [277, 232]. He has studied the use of abductive reasoning for a variant of the mediator-based problem, namely where the sources are not databases but "intelligent" cognitive agents. Like in COIN, only the GAV approach is used, but here more complex ontological relationships are studied e.g. where the global relation is the transitive closure of a source predicate. His implementation applies and improves the IFF-procedure. Although the tackled problem is slightly different from ours, Xanthakos' work is closer in spirit to ours than COIN. In COIN, abductive reasoning is regarded as a handy and efficient way to compute the desired information, but it ignores the potential to deal with more expressive knowledge sources than relational databases.

Compositionality and mediator-based systems

As mentioned, we tackled the mediator-based integration from a Knowledge Representation perspective. In this field the merging of knowledge bases have been

[9] The term "Context" corresponds to our notion of ontology.

studied, usually called the composition of knowledge bases [34, 270, 269]. Closely related work is the work by Verbaeten et al. [270, 269] on the composition of OLP-FOL theories[10]. [269] defines a special merging operator for theories that represent the knowledge of an agent which is only a partial view on the world.

EXAMPLE 4.15 Consider two agents with a different definition for the same notion, a student. $\mathcal{T}_1$ exploits therefore the age of a person, while according to $\mathcal{T}_2$ a student holds a special identification card.

$$\begin{aligned} \mathcal{T}_1 = & \quad (\{\{student(X) \leftarrow person(X) \wedge age(X,Y) \wedge Y < 26.\}\}, \emptyset) \\ \mathcal{T}_2 = & \quad (\{\{student(X) \leftarrow has\_studentcard(X).\}\}, \emptyset) \end{aligned}$$

Simply merging the two definitions together into one is unsound in general w.r.t. the compositionality criterion

$$Mod(\mathcal{T}) = Mod(\mathcal{T}_1) \cap Mod(\mathcal{T}_2)$$

where $\mathcal{T}_1$ and $\mathcal{T}_2$ are the agents' theories and $\mathcal{T}$ is the theory representing the composed knowledge of both agents. Therefore [269] proposes a composition which consists of two phases. In the first, the predicates which represent shared and possibly incomplete information by the two agents, are annotated. Their definitions, e.g. for the predicate p, are *opened*: a new auxiliary predicate (e.g. p^*) is introduced for the selected defined predicates. The opening of the definition is the addition of an extra clause to the definition $p(\overline{X}) \leftarrow p^*(\overline{X})$ plus all occurrences of the opened predicate in the bodies of the other program rules are replaced by the new auxiliary predicate. This is formalized by an opening operator Θ.

EXAMPLE 4.16 The opening of the theories of example 4.15 is

$$\begin{aligned} \Theta(\mathcal{T}_1) = & \left(\left\{ \left\{ \begin{array}{l} student(X) \leftarrow person(X) \wedge age(X,Y) \wedge Y < 26. \\ student(X) \leftarrow student_1^*(X) \end{array} \right\} \right\}, \emptyset \right) \\ \Theta(\mathcal{T}_2) = & \left(\left\{ \left\{ \begin{array}{l} student(X) \leftarrow has\_studentcard(X) \\ student(X) \leftarrow student_2^*(X) \end{array} \right\} \right\}, \emptyset \right) \end{aligned}$$

The opening operator ensures that the composing conditions [270] are satisfied and thus that the composed theory is obtained as the pairwise union.

EXAMPLE 4.17

$$\begin{aligned} \mathcal{T} & = \Theta(\mathcal{T}_1) \cup \Theta(\mathcal{T}_2) \\ & = \left(\left\{ \left\{ \begin{array}{l} student(X) \leftarrow person(X) \wedge age(X,Y) \wedge Y < 26. \\ student(X) \leftarrow has\_studentcard(X) \\ student(X) \leftarrow student_1^*(X) \\ student(X) \leftarrow student_2^*(X) \end{array} \right\} \right\}, \emptyset \right) \end{aligned}$$

[10] Recall that OLP-FOL theories are ID-Logic theories with only one definition.

The opening has weakened the composed knowledge, therefore a closing operator $\mathcal{C}$ can be applied. This removes the clauses with the auxiliary predicates.

EXAMPLE 4.18

$$\begin{array}{rl} \mathcal{T} = & \mathcal{C}(\Theta(\mathcal{T}_1) \cup \Theta(\mathcal{T}_2)) \\ = & \left(\left\{ \left\{ \begin{array}{l} student(X) \leftarrow person(X) \wedge age(X,Y) \wedge Y < 26. \\ student(X) \leftarrow has\_studentcard(X) \end{array} \right\} \right\}, \emptyset \right) \end{array}$$

This composition is not always satisfactory because it ignores the origin of the knowledge. Therefore [269] defines a conditional opening operator which includes the origin of the clauses in the composed knowledge.

This composition shows many correspondences with our mediator-based approach. For instance, the use of auxiliary predicates in order to get a correct merging and the completion afterwards which expresses that the two sources have complete information about a definition. But there is difference in perspective, which can be best explained as that the work of [269] is an *one-time merging* and ours an *anytime merging*. In [269], the two agents will materialize their current knowledge into a 'super agent'. This 'super agent' keeps no connection with the original agents. Hence, if an original agent changes his knowledge afterwards, the 'super agent' is not adapted. In our case, the 'super agent' is virtual and its knowledge is completely dependent on the underlying independent sources. When a source changes its knowledge, it is immediately reflected at the global level.

4.2.6 Extending the ID-Logic framework

In this section we briefly sketch some possible extensions of the ID-Logic framework.

Lifting the unique domain assumption

Often two sources (alphabets) use different domain elements to denote the same object in the world (e.g., a client number and a social security number for the same person). Therefore, in general the Unique Names Axioms will not hold when two alphabets are merged into one. A specialized equality theory must be composed. It consists of the Unique Names Axioms for the terms belonging to one alphabet and a special equality theory that expresses the relationships between the terms of the alphabets. The latter specifies equality axioms for terms that denote the same object in the real world and disequality axioms for terms that denote different objects. In practice, this equality theory can be handled by the introduction of an auxiliary mapping, that translates the names at global database into those of the source.

Towards more expressive databases and query languages

We have limited the sources to simple classical databases. Although these are the majority of the information carriers in practice, it is of interest to extend the work towards more complex information sources.

In particular, it is important to study more complex relationships between ontologies e.g. where a global predicate is the transitive closure of the information in the sources. This is studied for example in [277].

Another direction for future extensions is the handling of aggregates. Aggregates are expressions such as summation and cardinality. Databases are probably one of the earliest fields in which aggregates have been introduced and still have a prominent place. Therefore, a mediator-based system should deal with them correctly. This extension seems feasible since ID-Logic has been extended with aggregates (see [212, 101]) and there exists a prototype implementation of the $\mathcal{A}$system that handles aggregates (see [263]).

Reasoning with inconsistent knowledge

Up to now, we have assumed that all information in the sources was consistent w.r.t. the intended global database, and so integrity constraints at the global level were not considered. In case that the global schema does contain such constraints, inconsistencies can arise. For example, in [63, 64], Calì et al. consider mediator-based systems with a restricted set of integrity constraints: key constraints and functional dependencies. For this reason, they have to extend their proof procedure for the GAV approach.

In the next section, we will tackle inconsistencies and sketch how both approaches can be merged into one system. The advantage of our approach is that no new proof procedure must be designed for which the soundness and completeness has to be proven. Due to the flexibility of our framework, the integration is easier; moreover we can reuse general results from the $\mathcal{A}$system and ID-Logic. For example, no new inference procedure must be designed.

4.2.7 Conclusion

Our ID-Logic mediator-based framework unifies and generalizes the existing approaches in the literature. The key-features are the separated specification of the ontological relationships and of the amount of knowledge relationships. This is enabled by the explicit denotation of incomplete and complete knowledge. The framework flattens the distinction between GAV and LAV. There is no strong reason to prefer the one or the other, from representational point of view. Therefore, it does not matter whether GAV or LAV is used.

ID-Logic is probably the most expressive logic that has been applied to this problem. Most applied logics have rather restricted expressiveness because either the databases languages have also a restricted expressiveness, or a corresponding proof-procedure with "good" computational behavior has to be built. From complexity point of view, if coping with incompleteness of knowledge cannot be avoided (which is the case in general) the proof-procedures will certainly be less effective than normal single database query answering procedures (otherwise a limited problem class is considered). If integrity constraints are also allowed, the complexity class will probably further increase, and so the (potentially) extra (worst-case) complexity introduced by ID-Logic is probably negligible. On the contrary, the expressiveness of ID-Logic can be regarded as an advantage, because more complex data sources and more complex relationships between the ontologies can be allowed.

We have pointed out the use of abductive inference for query answering, when the GAV approach is used. Especially, in the case when credulous answers must be computed, this approach is valuable. The implementation of a mediator based system, using the $\mathcal{A}$system for reasoning is for future work.

4.3 Coherent data integration

This section concerns the issue of merging database instances. We will study this problem for a simplified problem context: given are a finite number of databases $\mathcal{DB}_1, \ldots, \mathcal{DB}_n$ (called sources) sharing the same alphabet. Each database stores partial knowledge about the same domain. Also given is a classically consistent set of integrity constraints $\mathcal{IC}$, which is constructed from the integrity constraints of the given databases plus some additional integrity constraints. The goal is to construct a consistent database instance from the n database instances $\mathcal{D}_1, \ldots, \mathcal{D}_n$ which expresses the composed knowledge of the databases.

Ideally the union of the database instances suffices. In general, the union may be inconsistent w.r.t. the integrity constraints. That is a unacceptable situation since no sensible query answering can be performed on an inconsistent database. Note that even if all sources are consistent, the union of their database instances may be inconsistent. In the following, we will often ignore the origin of the data facts; our problem context is then restricted to an inconsistent database and how to deal with that.

Now, one cannot just disallow the integration of database instances when inconsistencies arise because the ultimate goal is to answer queries with the composed data. Therefore methods are developed in order to deal with the inconsistencies in a sensible way. Roughly, there are two approaches to handle this problem:

- *Paraconsistent* formalisms, in which the amalgamated data may remain inconsistent, but the set of conclusions implied by it is not explosive, i.e.: not every fact follows from an inconsistent database, and so query answering does not become trivial in the presence of contradictions. Paraconsistent procedures for integrating data (e.g., [88, 246]) are often based on a paraconsistent reasoning process, such as LFI [67], annotated logics [15, 165, 245], or other non-classical proof procedures [19, 27, 68, 218][11].

- *Coherent* (consistency-based) methods, in which the amalgamated data are revised in order to restore consistency (see, e.g., [16, 20, 32, 33, 45, 47, 138, 182]). In many cases the underlying formalisms of these approaches are closely related to the theory of belief revision [8, 131]. In the context of databases the idea is to consider consistent databases that are 'as close as possible' to the original database. These 'repaired' instances of the inconsistent database correspond to plausible and compact ways of restoring consistency.

Our goal is to develop a coherent method to 'repair' a database so that its consistency is restored. The presented approach develops an ID-Logic representation of the repaired database with respect to the integrity constraints. The obtained

[11]See also [89] for a historical perspective and some computational remarks on this kind of formalisms.

theory is a meta-theory, that has to be instantiated for each particular problem. For the computation of the repairs, a corresponding abductive task is derived and solved by the $\mathcal{A}$system. Because the basic $\mathcal{A}$system-solver is unable to compute the most preferred solutions, the solver is extended with an optimizing module. Additionally we prove that the $\mathcal{A}$system is terminating for consistency restoring problems that consist of only key and foreign key constraints. Furthermore the approach is applied to several extended problem settings. We also sketch how to integrate this consistency restoration module with the mediator-based systems, developed in the previous section.

4.3.1 Coherent integration of databases

We begin with a formal definition of our goal. It is supposed that all databases share the same language $\mathcal{L}$. This assumption allows now to drop the explicit mention of the ontology. Thus, a database $\mathcal{DB}$ is in this section regarded as a pair $(\mathcal{D}, \mathcal{IC})$.

The meaning of a database $\mathcal{DB}$ is still given by its minimal Herbrand model. Recall that $\mathcal{DB}$ is consistent if every integrity constraint follows from its minimal Herbrand model ($\mathcal{D} \models \mathcal{IC}$). Our goal is to integrate n consistent databases, $\mathcal{DB}_i = (\mathcal{D}_i, \mathcal{IC}_i)$, $i = 1, \ldots n$ to one *consistent* database such that it reflects as closely as possible the composed information of the source databases. The idea is to consider the union of all source databases and to restore its consistency in such a way that as much as possible information is preserved.

NOTATION 4.3.1 Let $\mathcal{DB}_i = (\mathcal{D}_i, \mathcal{IC}_i)$, $i = 1, \ldots, n$, and $\mathcal{IC}$ be a classically consistent set of integrity constraints. We denote the union of the databases $\mathcal{UDB}$ as

$$\mathcal{UDB} = (\bigcup_{i=1}^{n} \mathcal{D}_i \,, \mathcal{IC})$$

$\mathcal{IC}$ is the result of an integration process in which integrity constraints $\mathcal{IC}_i$ from the sources $\mathcal{DB}_i$,$1 \leq i \leq$, and new additional constraints. This process is out of the scope of this section; we assume that it is finished successfully, resulting in $\mathcal{IC}$.

A key notion in coherent database integration is a *repair*:

DEFINITION 4.8 A *repair* of a database $\mathcal{DB} = (\mathcal{D}, \mathcal{IC})$ is a pair $(\mathsf{Insert}, \mathsf{Retract})$, such that:

1. $\mathsf{Insert} \cap \mathcal{D} = \emptyset$,
2. $\mathsf{Retract} \subseteq \mathcal{D}$[12],
3. $(\mathcal{D} \cup \mathsf{Insert} \setminus \mathsf{Retract},\ \mathcal{IC})$ is a consistent database.

[12] Note that by conditions (1) and (2) it follows that $\mathsf{Insert} \cap \mathsf{Retract} = \emptyset$.

Intuitively, Insert is a set of elements that should be inserted into $\mathcal{D}$ and $\mathsf{Retract}$ is a set of elements that should be removed from $\mathcal{D}$ in order to have a consistent database.

The notion of the repair of a database underlies many formalisms for coherent database integration. In the context of databases, it was first introduced in [16], and later exploited by [138, 124, 46, 47, 88, 17]. Earlier versions of this notion can be traced back to [87, 274].

DEFINITION 4.9 A *repaired database* of $\mathcal{DB} = (\mathcal{D}, \mathcal{IC})$ is a consistent database of the form $(\mathcal{D} \cup \mathsf{Insert} \setminus \mathsf{Retract}\,, \mathcal{IC})$, where $(\mathsf{Insert}, \mathsf{Retract})$ is a repair of $\mathcal{DB}$.

As there may be many ways to repair an inconsistent database[13], it is often convenient to make preferences among the possible repairs, and consider only the most preferred ones. Below are two common preference criteria for preferring a repair $(\mathsf{Insert}, \mathsf{Retract})$ over a repair $(\mathsf{Insert}', \mathsf{Retract}')$:

DEFINITION 4.10 Let $(\mathsf{Insert}, \mathsf{Retract})$ and $(\mathsf{Insert}', \mathsf{Retract}')$ be two repairs of a given database.

- *set inclusion preference criterion* : $(\mathsf{Insert}', \mathsf{Retract}') \leq_i (\mathsf{Insert}, \mathsf{Retract})$, if $\mathsf{Insert} \subseteq \mathsf{Insert}'$ and $\mathsf{Retract} \subseteq \mathsf{Retract}'$.
- *minimal cardinality preference criterion*: $(\mathsf{Insert}', \mathsf{Retract}') \leq_c (\mathsf{Insert}, \mathsf{Retract})$, if
$|\mathsf{Insert}| + |\mathsf{Retract}| \leq |\mathsf{Insert}'| + |\mathsf{Retract}'|$.

Set inclusion has been considered by e.g. [16, 46, 47, 88, 138]; minimal cardinality is used by e.g. [87, 182, 137, 17].

In what follows we assume that the preference relation $\leq$ is a fixed pre-order that represents some preference criterion on the set of repairs (and we shall omit subscripted notations in it whenever possible). We shall also assume that if $(\emptyset, \emptyset)$ is a valid repair, it is the $\leq$-least (i.e., the "best") one. This corresponds to the intuition that a database should not be repaired unless it is inconsistent.

DEFINITION 4.11 A *$\leq$-preferred repair* of $\mathcal{DB}$ is a repair $(\mathsf{Insert}, \mathsf{Retract})$ of $\mathcal{DB}$, s.t. for every repair $(\mathsf{Insert}', \mathsf{Retract}')$ of $\mathcal{DB}$, if $(\mathsf{Insert}, \mathsf{Retract}) \leq (\mathsf{Insert}', \mathsf{Retract}')$ then $(\mathsf{Insert}', \mathsf{Retract}') \leq (\mathsf{Insert}, \mathsf{Retract})$. The set of all the $\leq$-preferred repairs of $\mathcal{DB}$ is denoted by $!(\mathcal{DB}, \leq)$.

DEFINITION 4.12 A *$\leq$-repaired database* of $\mathcal{DB}$ is a repaired database of $\mathcal{DB}$, constructed from a $\leq$-preferred repair of $\mathcal{DB}$. The set of all the $\leq$-repaired databases

[13] Some of them may be trivial and/or useless. For instance, one way to eliminate the inconsistency in $(\mathcal{D}, \mathcal{IC}) = (\{p, q, r\}, \{\neg p\})$ is by deleting every element in $\mathcal{D}$, but this is certainly not the optimal way of restoring consistency in this case.

is denoted by

$$\mathcal{R}(\mathcal{DB}, \leq) = \{ (\mathcal{D} \cup \mathsf{Insert} \setminus \mathsf{Retract}\,,\, \mathcal{IC}) \mid (\mathsf{Insert}, \mathsf{Retract}) \in\, !(\mathcal{DB}, \leq) \}.$$

Note that if $\mathcal{DB}$ is consistent and $\leq$ is a preference relation, then $\mathcal{DB}$ is the only $\leq$-repaired database of itself (there is nothing to repair in this case, as expected).

NOTE 4.3.1
It is usual to refer to the $\leq$-preferred databases of $\mathcal{DB}$ as the consistent databases that are 'as close as possible' to $\mathcal{DB}$ itself (see, e.g., [16, 88, 182]). Indeed, let

$$\mathtt{dist}(\mathcal{D}_1, \mathcal{D}_2) = (\mathcal{D}_1 \setminus \mathcal{D}_2) \cup (\mathcal{D}_2 \setminus \mathcal{D}_1)$$

It is easy to see that $\mathcal{DB}' = (\mathcal{D}', \mathcal{IC})$ is a $\leq_i$-repaired database of $\mathcal{DB} = (\mathcal{D}, \mathcal{IC})$, if the set $\mathtt{dist}(\mathcal{D}', \mathcal{D})$ is minimal (w.r.t. set inclusion) among all the sets of the form $\mathtt{dist}(\mathcal{D}'', \mathcal{D})$, where $\mathcal{D}'' \models \mathcal{IC}$. Similarly, if $|S|$ denotes the size of S, then $\mathcal{DB}' = (\mathcal{D}', \mathcal{IC})$ is a $\leq_c$-repaired database of $\mathcal{DB} = (\mathcal{D}, \mathcal{IC})$, if $|\mathtt{dist}(\mathcal{D}', \mathcal{D})| = \min\{|\mathtt{dist}(\mathcal{D}'', \mathcal{D})| \mid \mathcal{D}'' \models \mathcal{IC}\}$.

Given n databases and a preference criterion $\leq$, our goal is therefore to compute the set $\mathcal{R}(\mathcal{UDB}, \leq)$ of the $\leq$-repaired databases of $\mathcal{UDB}$. The reasoner may use different strategies to determine the consequences of this set. Among the common approaches are the skeptical (conservative) one, that it is based on a "consensus" among all the elements of $\mathcal{R}(\mathcal{UDB}, \leq)$ (see [16, 138]), a "credulous" approach, in which entailments are determined by any element in $\mathcal{R}(\mathcal{UDB}, \leq)$, an approach that is based on a "majority vote", etc. In cases that processing time is a major consideration, one may want to speed-up the computations by considering *any* repaired database. In such cases it is sufficient to find an arbitrary element in the set $\mathcal{R}(\mathcal{UDB}, \leq)$. We give now some examples[14] of the integration process[15].

EXAMPLE 4.19 Consider a relation *teaches* of the schema ($\mathtt{course}, \mathtt{teacher}$), and an integrity constraint, stating that the same course cannot be taught by two different teachers:

$$\mathcal{IC} = \{ \forall X \forall Y \forall Z\, (teaches(X, Y) \wedge teaches(X, Z) \rightarrow Y = Z) \}.$$

Consider now the following two databases:

$$\mathcal{DB}_1 = (\, \{teaches(c_1, n_1),\ teaches(c_2, n_2)\},\, \mathcal{IC}\,),$$
$$\mathcal{DB}_2 = (\, \{teaches(c_2, n_3)\},\, \mathcal{IC})$$

Clearly, the unified database $\mathcal{DB}_1 \cup \mathcal{DB}_2$ is inconsistent. It has two preferred repairs, which are $(\emptyset,\, \{teaches(c_2, n_3)\})$ and $(\emptyset,\, \{teaches(c_2, n_2)\})$. The corresponding repaired databases are the following:

$$\mathcal{R}_1 = (\, \{teaches(c_1, n_1),\ teaches(c_2, n_2)\},\, \mathcal{IC}\,),$$
$$\mathcal{R}_2 = (\, \{teaches(c_1, n_1),\ teaches(c_2, n_3)\},\, \mathcal{IC}\,).$$

[14] See, e.g., [16, 47, 138] for further discussions on these examples.
[15] In all the following examples we use set inclusion as the preference criterion.

EXAMPLE 4.20 Consider databases with relations *class* and *supply*, of schemas (`item`, `type`) and (`supplier`, `department`, `item`), respectively. Let

$\mathcal{DB}_1 = (\{supply(c_1, d_1, i_1),\ class(i_1, t_1)\},\ \mathcal{IC}),$
$\mathcal{DB}_2 = (\{supply(c_2, d_2, i_2),\ class(i_2, t_1)\},\ \emptyset),$

where $\mathcal{IC} = \{\, \forall X \forall Y \forall Z\, (supply(X, Y, Z) \wedge class(Z, t_1) \rightarrow X = c_1)\,\}$ states that only supplier c_1 can supply items of type t_1. Again, $\mathcal{DB}_1 \cup \mathcal{DB}_2$ is inconsistent, and has two preferred repairs: $(\emptyset,\ \{supply(c_2, d_2, i_2)\})$ and $(\emptyset,\ \{class(i_2, t_1)\})$. It follows that there are two ways to repair this database:

$\mathcal{R}_1 = (\ \{supply(c_1, d_1, i_1),\ class(i_1, t_1),\ class(i_2, t_1)\},\ \mathcal{IC}\ ),$
$\mathcal{R}_2 = (\ \{supply(c_1, d_1, i_1),\ supply(c_2, d_2, i_2),\ class(i_1, t_1)\},\ \mathcal{IC}\ ).$

EXAMPLE 4.21 Let $\mathcal{D}_1 = \{p(a),\ p(b)\}, \mathcal{D}_2 = \{q(a),\ q(c)\}$, and $\mathcal{IC} = \{\forall X\, (p(X) \rightarrow q(X))\}$. Again, $(\mathcal{D}_1, \emptyset) \cup (\mathcal{D}_2, \mathcal{IC})$ is inconsistent. The corresponding preferred repairs are $(\{q(b)\},\ \emptyset)$ and $(\emptyset,\ \{p(b)\})$. Thus, the repaired databases are the following:

$\mathcal{R}_1 = (\ \{p(a),\ p(b),\ q(a),\ q(b),\ q(c)\},\ \mathcal{IC}\ ),$
$\mathcal{R}_2 = (\ \{p(a),\ q(a),\ q(c)\},\ \mathcal{IC}\ ).$

In this case, then, both the 'consensus approach' and the 'credulous approach' allow to infer that, e.g., $p(a)$, $q(a)$, and $q(c)$ hold.

4.3.2 The ID-Logic repair composer

Here, we will model the repaired database instance in terms of a repair and the original database instance. This is done by a meta-theory in ID-Logic. By translating this ID-Logic theory into an Abductive Logic theory, the repairs can be computed by the $\mathcal{A}$system as an abductive solution in ID-Logic.

Consider a (possibly inconsistent) database $\mathcal{DB} = (\mathcal{D},\ \mathcal{IC})$ for which we like to represent the repaired database. The first step is to represent the database instance $\mathcal{D}$: a natural choice is by a definition. Here $\mathcal{D}$ is represented by a definition (by enumeration) of the predicate $db(\cdot)$.

EXAMPLE 4.22 Consider the database instance (the union of the database instances of example 4.19)

$$\mathcal{D} = \{teaches(c_1, n_1), teaches(c_2, n_2), teaches(c_2, n_3)\}$$

The corresponding definition in ID-Logic is

$$\left\{ \begin{array}{l} db(teaches(c_1, n_1)). \\ db(teaches(c_2, n_2)). \\ db(teaches(c_2, n_3)). \end{array} \right\}$$

According to definition 4.8, a repair of a database $\mathcal{DB}$ defines two sets of atoms: Insert and $\mathsf{Retract}$. We represent Insert (resp. $\mathsf{Retract}$) with the predicate $insert(\cdot)$ (resp. $retract(\cdot)$). For a given repair, the repaired database instance is $(\mathcal{D} \setminus \mathsf{Retract}) \cup \mathsf{Insert}$. This is expressed by the definition

$$\left\{ \begin{array}{l} fact(X) \leftarrow db(X) \land \neg retract(X). \\ fact(X) \leftarrow insert(X). \end{array} \right\}$$

in which $fact(\cdot)$ denotes the elements which belong to the repaired database. The first rule states that every element of the original database instance that is not retracted belongs to the repaired database instance. The second states that every inserted atom belongs to the repaired database instance.

Definition 4.8 also imposes two conditions on the repair, which are expressed as FOL statements:

- An inserted element should not belong to the given database instance:

$$\forall\, X.\neg(insert(X) \land db(X))$$

- A retracted element should belong to the given database instance:

$$\forall\, X.retract(X) \rightarrow db(X)$$

Summarized, the ID-Logic meta-theory that describes the repaired database instance is

Definition 4.13 The repaired database of a database is given by the ID-Logic meta-theory

$$\mathcal{T}_{composer} = \left(\begin{array}{l} \left\{ \left\{ \begin{array}{l} fact(X) \leftarrow db(X) \land \neg retract(X). \\ fact(X) \leftarrow insert(X). \end{array} \right\} \right\}, \\ \\ \left\{ \begin{array}{l} \forall\, X.\neg(insert(X) \land db(X)) \\ \forall\, X.retract(X) \rightarrow db(X) \end{array} \right\} \end{array} \right)$$

In the following, this theory is called the *composer*.

Note the similarity with the Event Calculus (EC) [170]. Like the EC, the composer defines a meta-property of some objects, namely the coherent elements of a database instance. Both leave some part of the specification undefined. In the EC, these are the *terminates*, *initiates*, *initially* and *act* predicates. Here, *db*, *retract* and *insert* are open. Depending on the problem which one wants to solve, one or more of these open predicates are defined expressing domain specific information. For a classical planning task, *terminates*, *initiates* and *initially* are defined by the user. Likewise, the classical task for the composer is to find a repair such that the repaired database instance is consistent. Therefore db is defined by the database instance $\mathcal{D}$ of the to-be-repaired database $\mathcal{DB}$ and the integrity constraints of $\mathcal{DB}$ must be expressed in terms of the repaired database. The latter operation adds a new aspect to the use of meta-theories (one that is not encountered in the context of the Event Calculus), namely that problem dependent integrity constraints must be transformed in terms of the concepts defined by the meta-theory.

EXAMPLE 4.23 Consider the union of the databases of example 4.19

$$\mathcal{UDB} = \left(\begin{array}{l} \{teaches(c_1, n_1), teaches(c_2, n_2), teaches(c_2, n_3)\}, \\ \{\forall\ X, Y, Z.teaches(X, Y) \wedge teaches(X, Z) \rightarrow Y = Z\} \end{array} \right)$$

The transformed integrity constraint is then

$$\forall\ X, Y, Z.fact(teaches(X, Y)) \wedge fact(teaches(X, Z)) \rightarrow Y = Z$$

by replacing each atom $p(\overline{t})$ which belongs to the database instance with $fact(p(\overline{t}))$. Together with the corresponding definition of the database instance, the ID-Logic theory for a coherent integration of $\mathcal{DB}$ is

$$\mathcal{T}_{teacher} =$$

$$\left(\begin{array}{l} \left\{ \left\{ \begin{array}{l} db(teaches(c_1, n_1)). \\ db(teaches(c_2, n_2)). \\ db(teaches(c_2, n_3)). \end{array} \right\}, \left\{ \begin{array}{l} fact(X) \leftarrow db(X) \wedge \neg retract(X). \\ fact(X) \leftarrow insert(X). \end{array} \right\} \right\}, \\ \left\{ \begin{array}{l} \forall\ X.\neg(insert(X) \wedge db(X)) \\ \forall\ X.retract(X) \rightarrow db(X) \\ \forall\ X, Y, Z.fact(teaches(X, Y)) \wedge fact(teaches(X, Z)) \rightarrow Y = Z \end{array} \right\} \end{array} \right)$$

This completes the first phase of our approach. In all the models of the ID-Logic theories constructed according to the above method, the interpretation of $fact(\cdot)$ corresponds to a consistently repaired database instance. The repair itself is given by the interpretation of $insert(\cdot)$ and $retract(\cdot)$.

EXAMPLE 4.24 Some models of $\mathcal{T}_{teacher}$ are (restricted to the composer-predicates):

1 $\{fact(teaches(c_1, n_1)), fact(teaches(c_2, n_2)), retract(teaches(c_2, n_3))\}$

2 $\{fact(teaches(c_1, n_1)), fact(teaches(c_2, n_3)), retract(teaches(c_2, n_2))\}$

$\vdots$

6 $\{retract(teaches(c_1, n_1)), retract(teaches(c_2, n_3)), retract(teaches(c_2, n_2))\}$

In total, there are 6 different models corresponding to the 6 different repaired databases of $\mathcal{DB}$.

These models can be constructed by the $\mathcal{A}$system because the task can be defined as finding the interpretations for $retract(\cdot)$ and $insert(\cdot)$, which are the open predicates in the ID-Logic theory. It is important to note that the used query is simply **true**, because the goal is to find all conditions in which the integrity constraints are satisfied.

As explained in the previous section, not all of these repairs are of interest. Therefore a preference criterion is defined and added to the search process of the $\mathcal{A}$system in order to select the really important ones. This is discussed in the next section.

4.3.3 Computing optimal abductive solutions

A normal $\mathcal{A}$system execution is able to compute the repaired databases defined by the above constructed ID-Logic theory. The computation of the $\leq$-preferred repair requires, however, an extension of the $\mathcal{A}$system. More precisely, the $\leq$-preference problem is formulated as finding an optimal abductive solution for an objective function. For example, $\leq_c$-preference criterion can be expressed as finding an abductive solution for which the sum

$$|\{retract(A)|retract(A) \in \Delta\}| + |\{insert(A)|insert(A) \in \Delta\}|$$

is minimal.

The incorporated optimization module uses the branch-and-bound principle. After finding the first solution, which is stored in an auxiliary data structure, the system continues by backtracking to find a new solution. When this solution is better than the stored one, the current optimal solution is replaced by the new found solution, otherwise it is discarded. After the exploration of the whole search space, the optimal solutions are presented. Depending on the objective function, the potential of a branch to lead to a better solution can be evaluated. When it is sure that the new branch does not lead to a better solution, it is immediately discarded. In that way, less search space must be explored.

The objective function associated with the cardinality preference criterion can be evaluated in each $\mathcal{A}$system state. In an intermediate state this evaluation yields a under-estimate of the final value: as the number of abducibles in Δ only increases towards a solution node, the objective value also increases. Let s_c be the current optimal cardinality. An intermediate state for which the object value is s_i and $s_c < s_i$, will never be part of a derivation to a better solution than the one which is associated with the value s_c. Because further computation based on this state is useless, it can be safely discarded.

This property holds also for the set inclusion preference criterion. But it has a far less effective pruning power. For set inclusion, the abductive solutions themselves are compared with each-other. At an intermediate state $\mathcal{S}_i$, the abduced atoms in Δ_i are often non-ground, which weakens the comparison. For example, a non-ground atom $retract(r(X))$ can be equal to $retract(r(a))$ or not, and unless this information is available, the branch has to be continued.

The optimizing module has been introduced in the $\mathcal{A}$system for the purpose of computing preferred repairs. As it is a general module, it is applicable for other problems as well, e.g. for finding minimal plans in AI-planning.

4.3.4 Computing preferred repairs through abduction

Using the ID-Logic composer and the $\mathcal{A}$system with the optimizing extension, we have developed a method to repair an inconsistent database. We already showed an example. The following extension of it, shows some other interesting properties of our approach. Especially, the ability of the $\mathcal{A}$system to produce non-ground answers is an advantage.

EXAMPLE 4.25 Consider the inconsistent database $\mathcal{DB}$, that describes a number of teachers and the courses they teach. The integrity constraints specify that each teacher should teach at least one course, and that each course is only taught by one teacher.

$$\mathcal{DB} = \left(\begin{array}{l} \left\{ \begin{array}{l} teaches(c_1, n_1), teaches(c_2, n_2), teaches(c_2, n_3) \\ teacher(n_1), teacher(n_2), teacher(n_3) \end{array} \right\}, \\ \left\{ \begin{array}{ll} \forall\, X, Y, Z. teaches(X, Y) \wedge teaches(X, Z) \rightarrow Y = Z & (I1) \\ \forall\, X. teacher(X) \leftarrow \exists\, Y. teaches(Y, X) & (I2) \end{array} \right\} \end{array} \right)$$

The ID-Logic encoding of the repaired database is the theory

$$\mathcal{T}_{teacher2} =$$

$$\left(\begin{array}{l} \left\{ \left\{ \begin{array}{l} db(teaches(c_1, n_1)). \\ db(teaches(c_2, n_2)). \\ db(teaches(c_2, n_3)). \\ db(teacher(n_1)). \\ db(teacher(n_2)). \\ db(teacher(n_3)). \end{array} \right\}, \left\{ \begin{array}{l} fact(X) \leftarrow db(X) \wedge \neg retract(X). \\ fact(X) \leftarrow insert(X). \end{array} \right\} \right\}, \\ \\ \left\{ \begin{array}{l} \forall\, X. \neg(insert(X) \wedge db(X)) \\ \forall\, X. retract(X) \rightarrow db(X) \\ \forall\, X, Y, Z. fact(teaches(X, Y)) \wedge fact(teaches(X, Z)) \rightarrow Y = Z \\ \forall\, X. fact(teacher(X)) \rightarrow \exists\, Y. fact(teaches(Y, X)) \end{array} \right\} \end{array} \right)$$

One of the answers computed by the $\mathcal{A}$system is

$$\begin{array}{l} \Delta = \{retract(teaches(c_2, n_3)), insert(teaches(C, n_3))\}, \\ \mathcal{E} = \{C \neq c_1, C \neq c_2\} \end{array}$$

It is non-ground, as one of the atoms in Δ contains the free variable C. This is exactly the expected result: retracting the $teaches(c_2, n_3)$ fact restores the consistency of integrity constraint $I1$. But at the same time it results in the violation of the second constraint $I2$, because n_3 is now not teaching a course. The new violation is resolved by the introduction of a unknown course that is taught by the teacher n_3. The second part of the abductive solution, $\mathcal{E}$, provides more precise information about this 'new' course. It is different from c_1 and c_2.

Note that this repair is not derivable when the active domain of the database coincides with the domain of the database. In above case[16], C should then be assigned to either c_1 or c_2, which makes that the above proposed solution is not a solution anymore. The restriction to consider only the active domain has even stronger effects: it restricts the repairs for our example to only consist of retracted data facts.

The above ID-Logic theory $\mathcal{T}_{teacher2}$ treats every database relation equivalent. Sometimes not every database relation is suspected and must be subject of a repair. For example, it is reasonable to suppose that the relation $teacher(\cdot)$ is more reliable than the $teaches(\cdot,\cdot)$. Such information can be reflected in the ID-Logic encoding by not encapsulating the $teacher(X)$ with $fact(\cdot)$ but by $db(\cdot)$ in the integrity constraint $I2$:

$$\forall\ X.db(teacher(X)) \rightarrow \exists\ Y.fact(teaches(Y,X)) \qquad (I2')$$

Because $db(\cdot)$ does not depend on $retract(\cdot)$ or $insert(\cdot)$, the relation $teacher(\cdot)$ will not be affected by a repair. This meta-information on how the database relations must be treated by the composer is expressed via the transformation of the integrity constraints.

The end-product of our approach, presented to the user, are the repairs. Although the preference criterion eliminates many of the less interesting repairs, usually more than one repair exists. The repairs reflect in some way the sources of the inconsistencies. Hence, they form useful information for the database administrator who likes to restore the consistency of the database. For other tasks the actual repairs are less valuable. Some tasks only consider the retrieval of consistent information from the database. This task is known as *Consistent Query Answering* (CQA) [62, 16, 46, 17]. The idea is that an answer to a query is certain (consistent) iff it is entailed by all repaired databases. For CQA, the computation of each repaired database individually and verifying the entailment of the query is computationally unacceptable. Therefore, CQA-techniques are based on query rewriting: the original query is transformed into a new one for which all answers are certain.

[16] Under the assumption that the implicit typing of the arguments is followed.

4.3.5 Soundness and completeness

In this section we give some soundness and completeness results for the $\mathcal{A}$system with respect to the computation of preferred repairs. The completeness result is however based on rather unpractical conditions. To reinforce the result, we prove for two frequent occurring types of integrity constraints that the $\mathcal{A}$system is terminating and consequently is complete.

In what follows we denote by $\mathcal{T}$ an abductive theory (constructed as described in Section 4.3.2) for composing n given databases $\mathcal{DB}_1, \ldots, \mathcal{DB}_n$. Because $\mathcal{T}$ is non-recursive, all main semantics (two-valued completion, three-valued completion, stable models and well-founded models) coincide for it (see e.g. [96, 93]). The following property shows that the Herbrand models of such $\mathcal{T}$ correspond with the abductive solutions for the query `true`.

PROPOSITION 4.1 Let $\mathcal{T}$ =($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$) be an abductive logic theory based on the language $\mathcal{L}$, such that $\mathcal{P}$ is a non-recursive program. Let S be any of the following semantics: two-valued completion, three-valued completion, stable model or well-founded semantics. If $\mathcal{H}$ is a Herbrand model of $\mathcal{T}$, then the set Δ of abducible atoms in $\mathcal{H}$ is an abductive solution for the query `true`. Conversely, for every abductive solution for `true`, there exists a unique S-model $\mathcal{H}$ of $\mathcal{T}$ based on the language $\mathcal{L}$' that extends $\mathcal{L}$, such that Δ is the set of true abducible atoms in $\mathcal{H}$.

PROOF 24
When $\mathcal{H}$ is a Herbrand model of $\mathcal{T}$, there exists a set Δ of abducible atoms such that $\mathcal{H}$ is a S-model of $\mathcal{P} \cup \Delta$. (All S semantics coincide for non-recursive programs.) Obviously, $\mathcal{P} \cup \Delta$ is consistent and entails all integrity constraints $\mathcal{IC}$, which also entails `true`. Hence, Δ is an abductive solution for the query `true`. In the other direction, consider a set Δ of abducible atoms where the set SK contains all Skolem constants that are not a part of $\mathcal{L}$. Let the language $\mathcal{L}$' be the extension of $\mathcal{L}$ with SK. Then, $\mathcal{P} \cup \Delta$ based on $\mathcal{L}$' has a unique S-model $\mathcal{H}$ and the true abducible atoms of $\mathcal{H}$ are Δ. If $\mathcal{P} \cup \Delta$ satisfies the integrity constraints $\mathcal{IC}$, it is an abductive solution for `true`. Consequently, $\mathcal{H}$ is the unique model of $\mathcal{T}$ for the abductive solution Δ.

Let $\texttt{Proc}_{\texttt{ALP}}$ be some sound abductive proof procedure for $\mathcal{T}$[17]. The following proposition shows that $\texttt{Proc}_{\texttt{ALP}}$ provides a coherent method for integrating the databases that are represented by $\mathcal{T}$.

PROPOSITION 4.2 Every abductive solution that is obtained by $\texttt{Proc}_{\texttt{ALP}}$ for the query '$\leftarrow$ `true`' on a theory $\mathcal{T}$, is a repair of $\mathcal{UDB}$.

[17] That is, $\texttt{Proc}_{\texttt{ALP}}$ is a process for computing only the abductive solutions of $\mathcal{T}$ in the sense of Definition 2.31.

Proof: By the construction of $\mathcal{T}$ it is easy to see that all the conditions that are listed in Definition 4.8 are satisfied. Indeed, the first two conditions are assured by the integrity constraints of the composer. The last condition is also met since by the soundness of $\texttt{Proc}_{\texttt{ALP}}$ it produces abductive solutions Δ_i for query '$\leftarrow$ `true`' on $\mathcal{T}$. Thus, by the second property in Definition 2.31, for every such solution $\Delta_i = (\mathsf{Insert}_i, \mathsf{Retract}_i)$ we have that $\mathcal{P} \cup \Delta_i \models \mathcal{IC}$. Since $\mathcal{P}$ contains a data section with all the facts, it follows that $\mathcal{D} \cup \Delta_i \models \mathcal{IC}$, i.e. every integrity constraint follows from $\mathcal{D} \cup \mathsf{Insert}_i \setminus \mathsf{Retract}_i$. □

As SLDNFA is a sound abductive proof procedure ([96]), it can be taken as the procedure $\texttt{Proc}_{\texttt{ALP}}$, and Proposition 4.2 provides a soundness theorem for the current implementation of the $\mathcal{A}$system. When an optimizer is incorporated in the $\mathcal{A}$system, we have the following soundness result for the extended system:

THEOREM 25
(Soundness) *Every output that is obtained by the query '$\leftarrow$ `true`' on $\mathcal{T}$, where the $\mathcal{A}$system is executed with a $\leq_c$-optimizer [respectively, with an $\leq_i$-optimizer], is a $\leq_c$-preferred repair [respectively, a $\leq_i$-preferred repair] of $\mathcal{UDB}$.*

Proof: Follows from Proposition 4.2 (since the $\mathcal{A}$system is based on SLDNFA that is a sound abductive proof procedure), and the fact that the $\leq_c$-optimizer prunes paths that lead to solutions that are not $\leq_c$-preferable. Similar arguments hold for systems with an $\leq_i$-optimizer. □

PROPOSITION 4.3 Suppose that the query '$\leftarrow$ `true`' has a finite SLDNFA-tree w.r.t. $\mathcal{T}$. Then every $\leq_c$-preferred repair and every $\leq_i$-preferred repair of $\mathcal{UDB}$ is obtained by running $\mathcal{T}$ in the $\mathcal{A}$system.

Outline of proof: The proof that all the abductive solutions with minimal cardinality are obtained by the system is based on [96, Theorem 10.1], where it is shown that SLDNFA^o, which is an extension of SLDNFA, aimed for computing solutions with minimal cardinality, is complete (see [96, Section 10.1] for further details). Similarly, the proof that all the abductive solutions which are minimal w.r.t. set inclusion are obtained by the system is based on [96, Theorem 10.2] that shows that SLDNFA$_+$, which is another extension of SLDNFA, aimed for computing minimal solutions w.r.t. set inclusion, is also complete (see [96, Section 10.2] for further details).

Now, the $\mathcal{A}$system is based on the combination of SLDNFA^o and SLDNFA$_+$. Moreover, as this system does not change the refutation tree (but only controls the way rules are selected), Theorems 10.1 and 10.2 in [96] are applicable in our case as well. Thus, all the $\leq_c$- and the $\leq_i$-minimal solutions are produced. This means in particular that every $\leq_c$-preferred repair as well as every $\leq_i$-preferred repair of $\mathcal{UDB}$ is produced by our system. □

It should be noted that the last proposition does not guarantee that non-preferred repairs will not be produced (as this is not true in general). However, the use of an optimizer excludes this possibility.

THEOREM 26
(Completeness) *In the notations of Proposition 4.3 and under its assumptions, the output of the execution of $\mathcal{T}$ in the $\mathcal{A}$system together with a $\leq_c$-optimizer [respectively, together with an $\leq_i$-optimizer] is exactly $!(\mathcal{UDB}, \leq_c)$ [respectively, $!(\mathcal{UDB}, \leq_i)$].*

Proof: We shall show the claim for the case of $\leq_c$; the proof w.r.t. $\leq_i$ is similar.

Let $(\mathsf{Insert}, \mathsf{Retract}) \in\, !(\mathcal{UDB}, \leq_c)$. By Proposition 4.3, $\Delta = (\mathsf{Insert}, \mathsf{Retract})$ is one of the solutions produced by the $\mathcal{A}$system for $\mathcal{T}$. Now, during the execution of the system together with the $\leq_c$-optimizer, the path that corresponds to Δ cannot be pruned from the refutation tree, since by our assumption $(\mathsf{Insert}, \mathsf{Retract})$ has a minimal cardinality among the possible solutions, so the pruning condition is not satisfied. Thus Δ will be produced by the $\leq_c$-optimized system. For the converse, suppose that $(\mathsf{Insert}, \mathsf{Retract})$ is some repair of $\mathcal{UDB}$ that is produced by the $\leq_c$-optimized system. Suppose for a contradiction that $(\mathsf{Insert}, \mathsf{Retract}) \not\in\, !(\mathcal{UDB}, \leq_c)$. By the proof of Proposition 4.3, there is some $\Delta' = (\mathsf{Insert}', \mathsf{Retract}') \in\, !(\mathcal{UDB}, \leq_c)$ that is constructed by the $\mathcal{A}$system for $\mathcal{T}$, and $(\mathsf{Insert}', \mathsf{Retract}') <_c (\mathsf{Insert}, \mathsf{Retract})$. But $|\Delta'| < |\Delta|$, and so the $\leq_c$-optimizer would prune the path of the Δ solution once its cardinality becomes bigger than $|\Delta'|$. This contradicts our assumption that $(\mathsf{Insert}, \mathsf{Retract})$ is produced by the $\leq_c$-optimized system. □

Termination and completeness for special classes of integrity constraints

The main weakness of the completeness theorem is, of course, that it assumes that the refutation tree is finite, without providing any means to verify this. The common way to assure this or, more generally, to guarantee the convergence of the proof system, is to pose syntactical restrictions on the form of the integrity constraints (see e.g., [16, 46, 88, 138, 137, 63, 64]). In this section, we prove for two frequently occurring integrity constraints that the $\mathcal{A}$system is terminating. Hence the $\mathcal{A}$system is complete for database repair problems w.r.t. a set of integrity constraints of these form.

The considered integrity constraints are

- *Key constraints*: for a relation $r(\bar{t})$, the group of arguments $key(r)$ is denoted as the key. $key(r(\bar{t}))$ denotes the projection on the key arguments for the tuple $r(\bar{t})$. The key constraint states then for all tuples $r(\bar{t})$ and $r(\bar{s})$, if $key(r(\bar{t})) = key(r(\bar{s}))$ then $r(\bar{t}) = r(\bar{s})$.
- *Foreign Key constraints*: for two relations r and p, let $fk(r)$ be a group of arguments of relation r, for which the key of p $key(p)$ is called the foreign

key. The foreign key constraint states that for each tuple $r(\overline{t})$ there exists a tuple $p(\overline{s})$ such that $fk(r(\overline{t})) = key(p(\overline{s}))$.

To prove our goals, we rely on the work of Verbaeten [268] that gives a general method to detect the termination of SLDNFA for an abductive normal logic program. This method has been recalled in Section 3.3.4.

Recall the composer of 4.13:

$\mathcal{T}_{composer} =$

$$\left(\left\{ \left\{ \begin{array}{l} fact(X) \leftarrow db(X) \wedge \neg retract(X). \\ fact(X) \leftarrow insert(X). \end{array} \right\} \right\}, \left\{ \begin{array}{l} \forall\, X.\neg(insert(X) \wedge db(X)) \\ \forall\, X.retract(X) \rightarrow db(X) \end{array} \right\} \right)$$

The corresponding abductive normal logic program (in $\mathcal{A}$system notation) for this ID-Logic theory is

$$P = \left\{ \begin{array}{l} \texttt{fact(X)} \leftarrow \texttt{db(X)} \wedge \neg\texttt{retract(X)}. \\ \texttt{fact(X)} \leftarrow \texttt{insert(X)}. \\ \texttt{ic} \leftarrow \texttt{insert(X)} \wedge \texttt{db(X)}. \\ \texttt{ic} \leftarrow \texttt{retract(X)} \wedge \neg\texttt{db(X)}. \end{array} \right\}$$
$$\mathcal{A} = \{\texttt{insert}(\cdot), \texttt{retract}(\cdot)\}$$

In the program ic is an auxiliary predicate that results from the Lloyd-Topor transformation of the integrity constraints. (See Section 2.3.4.) It is assumed that $db(\cdot)$ is defined by an enumeration of facts.

The termination of the pure composer The termination theorem [268] imposes two conditions that have to be verified: the abductive normal program P has to be (1) semi-acyclic and (2) abductive non-recursive. The first one is clearly satisfied: as the composer (and the database definition) is hierarchical and this property is a subclass of the semi-acyclic programs. Which leaves us to prove the second condition.

To verify if the resulted abductive normal program is abductive non-recursive, a dependency graph has to be constructed. Let us start with the composer. The relevant part of the graph is given by the following table:

`fact`	$\leadsto$	`-retract`	`fact`	$\leadsto$	`+insert`
`ic`	$\leadsto$	`+insert`	`ic`	$\leadsto$	`+retract`

The entries involving db are excluded since this relation is a defined predicate having a fixed interpretation. Furthermore from the assumption that it definition is a finite enumeration of facts it follows that any SLDNF-derivation surely terminates for it. This can be exploited in our analysis: since both integrity constraints, i.e. the program rules for `ic` only consume the abducibles and do not generate new abducible atoms, both rules can be excluded from the analysis.[18]

[18] `ic` is only evaluated in the negative context because it is added as $\neg$`ic` to each query.

Key constraints Without loss of generality, we can assume for the first integrity constraint a relation $r(X, Y)$ having two arguments, of which the first one forms the key. Thus, the key constraint is expressed as the FOL statement

$$\forall\ X, Y_1, Y_2 . r(X, Y_1) \wedge r(X, Y_2) \rightarrow Y_1 = Y_2$$

which is transformed according to our methodology to

$$\forall\ X, Y_1, Y_2 . fact(r(X, Y_1)) \wedge fact(r(X, Y_2)) \rightarrow Y_1 = Y_2$$

Normalized this is the program rule

$$\texttt{ic} \leftarrow \texttt{fact}(\texttt{r}(\texttt{X}, \texttt{Y}_1)) \wedge \texttt{fact}(\texttt{r}(\texttt{X}, \texttt{Y}_2)) \wedge \neg \texttt{Y}_1 = \texttt{Y}_2$$

This constraint adds a dependency between `ic` and `fact`, which is further reduced to

$$\texttt{ic} \quad \rightsquigarrow \quad \texttt{-retract} \quad \texttt{ic} \quad \rightsquigarrow \quad \texttt{+insert}$$

The complete dependency graph is

$$\begin{array}{llllll} \texttt{fact} & \rightsquigarrow & \texttt{-retract} & \texttt{fact} & \rightsquigarrow & \texttt{+insert} \\ \texttt{ic} & \rightsquigarrow & \texttt{+insert} & \texttt{ic} & \rightsquigarrow & \texttt{-retract} \end{array}$$

Since there is no cycle, the composer extended with key constraints is abductive non-recursive. Hence, the program is terminating for every bounded query, according to [268].

Foreign key constraint Again, without loss of generality, suppose two relations $r(X, Y)$ and $p(Y)$. The foreign key integrity constraint is then

$$\forall\ X, Y . r(X, Y) \rightarrow p(Y)$$

and transformed to

$$\forall\ X, Y . fact(r(X, Y)) \rightarrow fact(p(Y))$$

In denial form this is

$$\texttt{ic} \leftarrow \texttt{fact}(\texttt{r}(\texttt{X}, \texttt{Y})) \wedge \neg \texttt{p}(\texttt{Y}).$$

Again, this constraint adds a dependency between `ic` and `fact`, which is further reduced to

$$\begin{array}{llllll} \texttt{ic} & \rightsquigarrow & \texttt{+insert} & \texttt{ic} & \rightsquigarrow & \texttt{-retract} \\ \texttt{ic} & \rightsquigarrow & \texttt{-insert} & \texttt{ic} & \rightsquigarrow & \texttt{+retract} \end{array}$$

In contrast to the key constraints, the two occurrences of fact-atoms yield different dependencies of the abducibles.

Observe the cyclic dependency that makes the program abductive recursive. The cycle is partially broken by making the database relations in the constraint explicit in the dependency graph. Let `insert-r` be the representation of $insert(r(X,Y))$, etc... Then the dependency graph of the composer extended with this foreign key constraint will be

```
fact-r  ⇝  -retract-r  fact-r  ⇝  +insert-r
fact-p  ⇝  -retract-p  fact-p  ⇝  +insert-p
ic      ⇝  -insert-r   ic      ⇝  +retract-r
ic      ⇝  +insert-p   ic      ⇝  -retract-p
```

As one can see, there is no cycle for the relation $r/2$. The cycle still exists for the relation $p/1$. Based on this analysis we cannot prove general termination for the composer extended with one foreign key constraint. A weaker form a termination can, however, be proven: the cycle happens only when both $fact - p$ and ic are independently in the same mode of reasoning, i.e. as s denial, are executed. Since this is only possible via a query where the atom $fact(p(\cdot,\cdot))$ occurs in the scope of a negation, the exclusion of negation in the query will ensure that this does not happen. Hence the composer extended with a foreign key constraint is abductive non recursive for queries that do not contain negation.

This condition is, however, too weak when there are more than one foreign key constraints. If these define a cycle, e.g.

$$\forall\ X,Y.r(X,Y) \rightarrow \exists\ Z.p(Y,Z)$$
$$\forall\ Y,Z.p(Y,Z) \rightarrow \exists\ X.r(X,Y)$$

the resulting abductive normal program will be abductive recursive (via `ic`). These cycles can be detected by verifying if the set of foreign key constraints is abductive recursive when regarding the database relations as abducibles.

Termination of the $\mathcal{A}$system for the composer extended with key and foreign key constraints Based on a similar reasoning as above, we can prove that the combination of key and foreign key constraints yields no extra non-termination sources. Summarized, we can prove using the above case-based analysis and the theorem of [268] the following termination proposition about our composer:

PROPOSITION 4.4 (TERMINATION OF THE COHERENT COMPOSER) The computation of repairs based on the composer theory 4.13 by the $\mathcal{A}$system for the query **true** is terminating for an inconsistent database having the following properties:

- The database instance is given by a semi-acyclic definition.

- Having an integrity constraints set consisting of
  - key constraints
  - foreign key constraints which are abductive non-recursive

PROOF 27
The above analysis proves that for any set of integrity constraints composed of only key and foreign key constraints the abductive logic program is abductive non-recursive. If *db* is defined by a semi-acyclic definition, the whole abductive program is semi-acyclic. The last condition is that the query has to be bounded, which is clearly the case for **true**. Also in the query **true** negated fact atoms do not occur, hence, the termination condition for the foreign key constraints is fulfilled. Thus, all conditions of theorem 23 ([268], see Section 3.3.4) are satisfied, and the $\mathcal{A}$system is terminating.

COROLLARY 4.1 Since termination is ensured, the $\mathcal{A}$system derivation for the extended composer has a finite refutation tree. Consequently, the $\mathcal{A}$system is complete for this database repair computation problem.

REMARK 4.3.1 Despite its restrictions, this proposition suffices for most common database situations. Other approaches such as [16] are complete for only universally quantified integrity constraints (this includes the above constraints).

With respect to some other approaches, such as [63, 64], our method to tackle to problem yields a major advantage. They present a consistent query answering procedure for a mediator-based system that only handles the above two types of integrity constraints. We have obtained a similar system (later we sketch how the composer is integrated in a mediator-based structure, see Section 4.3.6) but with less effort. Our work completely relies on the existing general soundness, completeness and termination theorems, which enables us to get more easily the important results.

4.3.6 Handling specialized information

The purpose of this section is to demonstrate the potential use of our system in more complex scenarios, where various kinds of specialized data are incorporated in the system. In particular, we will briefly consider time information and source tracing, and will give some guidelines on how to extend the system with capabilities of handling these kinds of information. As a last topic, we will sketch the integration with the mediator-based system, defined in the previous section.

Timestamped information

Many database applications contain temporal information. This kind of data may be divided into two types: time information that is part of the data itself, and time information that is related to database operations (e.g., records on database update time). For instance, consider `birth_day(John,15/05/2001)`$_{16/05/2001}$. Here, John's date of birth is an instance of the former type of time information, and the subscripted data that describes the time in which this fact was added to the database, is an instance of the latter type of time information.

In our approach, timestamp information, i.e. the update time of the database, can be integrated by adding a temporal theory describing the state of the database at any particular time point. One way of doing so is by using *situation calculus*. In this approach a database is described by some initial information and a history of events performed during the database's lifetime (see [222]). Here we use a different approach, which is based on *event calculus* [167]. The idea is to make a distinction between two kinds of events, `add_db` and `del_db`, that describe the database modifications, and the composer-driven events `insert` and `retract` that are used for constructing database repairs. In this view, the extended composer has the following form:

$$\mathcal{T}_{temporal-composer} =$$

$$\left(\begin{array}{l} \left\{ \left\{ \begin{array}{l} fact\_at(X,T) \leftarrow initially\_db(X) \wedge \neg removed(0,X,T). \\ fact\_at(X,T) \leftarrow add(X,E) \wedge E < T \wedge \neg removed(E,X,T). \\ removed(E,X,T) \leftarrow del(X,C) \wedge E \leq C \wedge C < T. \\ add(X,T) \leftarrow add\_db(X,T). \\ add(X,T) \leftarrow insert(X,T). \\ del(X,T) \leftarrow del\_db(X,T). \\ del(X,T) \leftarrow retract(X,T). \end{array} \right\} \right\}, \\ \left\{ \begin{array}{l} \forall\ X,T.\neg(insert(X,T) \wedge retract(X,T)) \\ \forall\ X,T.\neg(insert(X,T) \wedge add\_db(X,T)) \\ \forall\ X,T.\neg(retract(X,T) \wedge del\_db(X,T)) \\ \forall\ X,T.insert(X,T) \rightarrow \neg fact\_at(X,T). \\ \forall\ X,T.retract(X,T) \rightarrow fact\_at(X,T). \end{array} \right\} \end{array} \right)$$

Note that in the above extended representation, the integrity constraints must be carefully specified. Consider, e.g. the statement that a person can be born only on one date:

$$\forall\ P,D_1,D_2,T.fact\_at(birth\_day(P,D_1),T) \wedge fact\_at(birth\_day(P,D_2),T) \rightarrow D_1 = D_2$$

A computational problem here is that to ensure consistency this constraint must be checked at every point in time. This may be avoided by a simple rewriting that ensures that the constraint will be verified only when an event for that person occurs:

$$\begin{array}{l} \forall\ P,D,T.add(birth\_day(P,D),T) \rightarrow check\_birth(P,D,T+1). \\ \forall\ P,D_1,D_2,T.check\_birth(P,D_1,T) \wedge \\ \qquad fact\_at(birth\_day(P,D_1),T) \wedge fact\_at(birth\_day(P,D_2),T) \rightarrow D_1 = D_2 \\ \forall\ P,D_1,D_2.fact\_at(birth\_day(P,D_1),0) \wedge fact\_at(birth\_day(P,D_2),0) \rightarrow D_1 = D_2 \end{array}$$

REMARK 4.3.2 In the last example we have used *temporal integrity constraints* in order to resolve contradicting update events. Clearly, contradicting events do not necessarily yield a classically inconsistent database, and so the role of such integrity constraints is to express possible events in terms of time and causation, and – if necessary – to describe their consequence as a violation of consistency.

Instead of using temporal integrity constraints and event calculus, one could repair a database with timestamps by using some time-driven preference criterion on its repairs. For instance, denote by $\mathtt{db}(x_1,\ldots,x_n)_{\mathtt{t}}$ that the data-fact $\mathtt{db}(x_1,\ldots,x_n)$ has a timestamp $\mathtt{t}$, and suppose that $(\mathsf{Insert},\mathsf{Retract})$ and $(\mathsf{Insert}',\mathsf{Retract}')$ are two repairs of a database $(\mathcal{D},\mathcal{IC})$. A time-driven criterion for preferring $(\mathsf{Insert},\mathsf{Retract})$ over $(\mathsf{Insert}',\mathsf{Retract}')$ could state, e.g., that for every data-fact $\mathtt{db}(x_1,\ldots,x_n)$ and timestamps $\mathtt{t}_1,\mathtt{t}_2$ s.t. $\mathtt{db}(x_1,\ldots,x_n)_{\mathtt{t}_1}$ follows from $\mathcal{D}\cup\mathsf{Insert}\setminus\mathsf{Retract}$ and $\mathtt{db}(x_1,\ldots,x_n)_{\mathtt{t}_2}$ follows from $\mathcal{D}\cup\mathsf{Insert}'\setminus\mathsf{Retract}'$, necessarily $\mathtt{t}_1\geq\mathtt{t}_2$.

The last example and the note that proceeds it give, so we hope, some intuitive feeling on how the $\mathcal{A}$system can be used to handle properly timestamped information and temporal databases. The interested reader is referred to e.g. [190, 222, 241] for a detailed discussion on related topics.

Keeping track of source identities

There are cases in which it is important to preserve the identity of the database from which a specific piece of information has originated. This is useful, for instance, when one wants to make preferences among different sources, or when some specific source should be filtered out (e.g, when the corresponding database is not available or becomes unreliable). This kind of information may be decoded by adding another argument to every fact, which denotes the identity of its origin. This requires minor modifications in the basic composer, since the composer controls the way in which the data is integrated. As such, it is the only component that can keep track of the source of the information.

Suppose, then, that for every database fact we add another argument that identifies its source. I.e., `db(X,S)` denotes that `X` is a fact originated from a database `S`. Because it is valuable information, the source is also added as an extra argument to the predicate $retract(X,S)$. In this way, the computed repair provides direct information for possible updates. The composer then has the following form:

$$\mathcal{T}_{source\_composer}=\left(\begin{array}{l}\left\{\left\{\begin{array}{l}fact(X,S)\leftarrow db(X,S)\wedge\neg retract(X,S).\\ fact(X,composer)\leftarrow insert(X).\end{array}\right\}\right\},\\ \left\{\begin{array}{l}\forall\, X.\neg(insert(X)\wedge db(X,S))\\ \forall\, X,S.retract(X,S)\rightarrow db(X,S).\end{array}\right\}\end{array}\right)$$

Note that the composer considers itself as an extra source that inserts brand new data facts. Now it is possible, e.g. to trace information that comes from a specific source, to make preferences among different sources (by specifying appropriate integrity constraints), and to filter data that comes from certain sources. The last property is demonstrated by the next definition:

$$\{validFact(X) \leftarrow fact(X,S) \wedge trusted\_source(S)\}$$

where *trusted_source* enumerates all reliable sources of the data.

The last example of 'source tracing' can be further extended in order to make preferences among different sources (and not only ignoring some unreliable sources). By introducing a new predicate, *trust(Source,Amount)*, that attaches a certain level of reliability (*Amount*) to each source, it is possible, in case of conflicts, to prefer sources with higher reliability as follows:

$$\{\forall\, X, Y, S_1, S_2 . fact(X, S_1) \wedge db(Y, S_2) \wedge X \approx Y \wedge S_1 \neq S_2 \rightarrow more\_trusted(S_1, S_2).\}$$

where *more_trusted* is defined as

$$\{more\_trusted(S_1, S_2) \leftarrow trust(S_1, A_1) \wedge trust(S_2, A_2), A_1 > A_2\}$$

and $X \approx Y$ denotes a correspondence between fact X and fact Y. This method is particularly useful when the integrity constraint above acts as a functional dependency (as instance of $X \approx Y$) on specific facts. The following example (originally introduced in [246]) demonstrates this.

Example 4.26 Consider the following simple scenario of 'target recognition', where three sensors of an autonomous vehicle, which have different degrees of reliability, should identify objects in the vehicle's neighborhood:

$$\left(\begin{array}{l} \left\{ \left\{ \begin{array}{l} trust(radar), 10). \\ trust(gunchar, 8). \\ trust(speedometer, 5). \\ db(observe(object1, t72), radar). \\ db(observe(object1, t60), gunchar). \\ db(observe(object1, t80), speedometer). \end{array} \right\} \right\}, \\ \left\{ \begin{array}{l} \forall\, X, S_1, S_2, T_1, T_2 . fact(observe(O, T_1), S_1) \wedge \\ db(observe(O, T_2), S_2) \wedge S_1 \neq S_2 \rightarrow more\_trusted(S_1, S_2) \end{array} \right\} \end{array} \right)$$

As the radar has the highest reliability, its indication will be the (only) one that is accepted in this case.

Integration in a mediator-based system

At the start of this chapter we expressed the goal to build a system for coherent integration of multiple databases. The previous Section 4.2 constructed a mediator-based system which provides a uniform view for a group of databases.

This section has developed a composer that restores the consistency of a database instances. We sketch by an example how both approaches for two independent problems can be handled together in one setting.

EXAMPLE 4.27 Consider a university students domain, in which information is stored about the first enrollment of a student at the university. Suppose that there are two database sources

$$\mathcal{DB}_1 = \langle \{st99(\cdot)\}, \{st99(john)\} \rangle$$
$$\mathcal{DB}_2 = \langle \{st00(\cdot)\}, \{st00(mary), st00(john)\} \rangle$$

and a mediator-based system for these

$$\mathfrak{G} = \langle \{enrolled(\cdot,\cdot)\}, \{\mathcal{DB}_1, \mathcal{DB}_2\}, \{\mathcal{W}_1, \mathcal{W}_2\} \rangle$$

where

$$\begin{array}{lcl} \mathcal{W}_1 & = & (\{enrolled(X,Y) \leftarrow (st99(X) \wedge Y = 1999) \vee enrolled_1^*(X,Y)\}, \emptyset) \\ \mathcal{W}_2 & = & (\{enrolled(X,Y) \leftarrow (st00(X) \wedge Y = 2000) \vee enrolled_2^*(X,Y)\}, \emptyset) \\ \mathcal{K} & = & (\emptyset, \{\forall\, X,Y. \neg(enrolled_1^*(X,Y) \wedge enrolled_2^*(X,Y))\}) \end{array}$$

The associated ID-Logic theory is

$$\mathcal{T}_{\mathfrak{G}} = \mathcal{DB}_1 \circ \mathcal{DB}_2 \circ \mathcal{W}_1 \circ \mathcal{W}_2 \circ \mathcal{K}$$

Suppose that the single global integrity constraint is

$$\mathcal{IC} = \{\forall\, X, Y, Z. enrolled(X,Y) \wedge enrolled(X,Z) \rightarrow Y = Z.\}$$

It states that there is just one student's first enrollment year. Intuitively, one can see that for the student $John$ this integrity constraint is violated.

Then, the repaired global database is given by the ID-Logic theory

$$\left(\begin{array}{l} \left\{ \begin{array}{l} \left\{ \begin{array}{l} fact(X) \leftarrow db(X) \wedge \neg retract(X). \\ fact(X) \leftarrow insert(X). \end{array} \right\}, \\ \left\{ \; db(enrolled(X,Y)) \leftarrow enrolled(X,Y). \; \right\} \end{array} \right\}, \\ \left\{ \begin{array}{l} \forall\, X. \neg(insert(X) \wedge db(X)) \\ \forall\, X. retract(X) \rightarrow db(X) \\ \forall\, X, Y, Z. fact(enrolled(X,Y)) \wedge fact(enrolled(X,Z)) \rightarrow Y = Z \end{array} \right\} \end{array} \right)$$

composed with the mediator-based system $\mathcal{T}_{\mathfrak{G}}$.

Evaluating this theory by the $\mathcal{A}$system the following repairs are computed:

$$\Delta_1 = \{retract(enrolled(john, 1999))\}$$
$$\Delta_2 = \{retract(enrolled(john, 2000))\}$$

These repairs are indeed the expected answers: either retract the enrollment n the year 1999 of John, either the one of 2000 is retracted.

The example shows that an intermediate definition suffices to link both solutions to the integration problem: $\{db(enrolled(X,Y)) \leftarrow enrolled(X,Y).\}$ represents that the database under consideration is given by the global database relation. To find out the precise origin of the inconsistency, this system can be enhanced with the source tracking of the previous extension. For this the global relations are extended with an extra argument denoting the source origin. E.g.

$$\mathcal{W}_1 = (\{enrolled(X,Y,source1) \leftarrow (st99(X) \wedge Y = 1999) \vee enrolled_1^*(X,Y)\}, \emptyset)$$

Then the source-tracking composer is usable:

$$\left(\begin{array}{l} \left\{ \begin{array}{l} \left\{ \begin{array}{l} fact(X,S) \leftarrow db(X,S) \wedge \neg retract(X,S). \\ fact(X,S) \leftarrow insert(X). \end{array} \right\}, \\ \left\{ \; db(enrolled(X,Y),S) \leftarrow enrolled(X,Y,S). \; \right\} \end{array} \right\}, \\ \left\{ \begin{array}{l} \forall\, X.\neg(insert(X) \wedge db(X,S)) \\ \forall\, X,S.retract(X,S) \rightarrow db(X,S) \\ \forall\, X,Y,Z,S_1,S_2.fact(enrolled(X,Y),S_1) \wedge \\ fact(enrolled(X,Z),S_2) \rightarrow Y = Z \end{array} \right\} \end{array} \right)$$

The computed repairs include, hence, their source:

$$\begin{array}{l} \Delta_1 = \{retract(enrolled(john,1999),source1)\} \\ \Delta_2 = \{retract(enrolled(john,2000),source2)\} \end{array}$$

Based on these repairs, the user knows that there is a clash between the information of $\mathcal{DB}_1$ and $\mathcal{DB}_2$.

NOTE 4.3.2

This example is simple in the sense that the mediator-based system contains no incomplete knowledge. When there is incomplete knowledge the interaction between the repairing module and the mediator-based system is more complex. That is a topic for future work.

4.3.7 Discussion and an overview of related works

The interest in systems for coherent integration of databases has been continuously growing in the last few years (see [74] for a recent overview). The literature provides many solutions originating from many different disciplines. In this section we discusss some of these.

A major issue that has to be addressed is the ability of data integration systems to cope properly with dynamically evolving worlds. In particular, the domain of discourse should not be fixed in advance, and information may be revised on a regular basis. This last demand is usually handled by methods of *belief revision*

[8, 131] and *non-monotonic reasoning*. In particular, [8] presents a set of postulates that formulate basic and sensible properties of a update. Therefore, any update is well-behaving when it follows these. When the types of changes are predictable, or can be characterized in some sense, temporal integrity constraints (in the context of *temporal databases*) can be used to specify how to treat new information. This method is also useful when the revision criteria are known in advance (e.g. 'in case of collisions, prefer the more recent data', cf. Section 4.3.6). See, e.g. [190, 241] for a detailed discussion on temporal integrity constraints and temporal databases.

In the context of belief revision it is common to make a distinction between revisions of integrity constraints and changes in the sets of the data-facts, since the two types of information have different nature and thus may require different approaches for handling dynamic changes. When the set of integrity constraints is given in a clause form, methods of *dynamic logic programming* [9, 10] may be useful for handling revisions. As noted in [10], assuming that each local database is consistent (as in our case), dynamic logic programming (together with a proper language for implementing it, like LUPS [10]) provide a way of avoiding contradictory information, and so this may be viewed as a method of updating an integrated database by a sequence of integrity constraints that arrive in different time points.

The second type of revisions (i.e., modifications of data-facts) is obtained here through the (preferred) repairs of the unified database, which induce corresponding modifications of data-facts. In general, a repair of a given database is a key notion in many formalisms for data integration. In the context of database systems, this notion was first introduced in [16], and later considered in [47, 46, 88, 124, 138, 182]. Earlier versions of repairs and inclusion-based consistency restoration may be traced back to [87, 274]. A repair is usually induced by a method of restoring (or assuring) consistency of the amalgamated database by a minimal amount of changes. As in our case, the minimization criterion is often determined by the aspiration to remain 'as close as possible' to the set of the collective information. This is a typical kind of a *repair goal*, and the standard ways of formally expressing it are by enumeration methods, such as the following[19]:

- Minimizing the Hamming distance between the (propositional) models of the unified database and its repairs [182], or minimizing the distance between the corresponding three-valued interpretations [88] (according to a suitable generalization of the Hamming distance).
- Minimizing the symmetric distance between the sets of consequences of the corresponding databases [16, 15, 46] (or, equivalently, minimizations in terms of set inclusion [138]).
- When the underlying data is prioritized, the corresponding quantitative information is also considered in the computations of distances (see, e.g. [182]).

[19]See also [135, Section 5] for a discussion on repair strategies.

Various ways of computing (preferred/minimal) repairs are described in the literature, among which are proof-theoretical (deductive) methods [88], abductive methods [147, 155, 233], and algorithmic approaches that are based on computations of maximal consistent subsets [32, 33], or that use techniques from model-based diagnosis [135]. A common approach is to view a database as a logic program, and to adopt standard techniques of giving semantics to logic programs in order to compute database repairs. For instance, stable-model semantics on disjunctive logic programs is used for computing repairs in [124, 138, 137, 25], and resolution-based procedures for integrating several annotated databases are introduced in [245, 246]. As it follows from Section 4.3.2, the application introduced here is also based on a resolution strategy, applied on abductive normal logic programs.

As repairing a database means in particular elimination of contradictions, reasoning with inconsistent information has been a major challenge for data integration systems. First, it is important to note in this respect that not every formalism for handling inconsistency is acceptable in the context of databases, even if the underlying criterion for handling inconsistency is the same as one of the repair goals mentioned above. The following example demonstrates such a case:

EXAMPLE 4.28 [16] Consider the following (inconsistent) database: $\mathcal{DB} = (\{p, q\}, \{\neg(p \wedge q)\})$. In the approach of [185], for instance, $p \vee q$ may be inferred as the repaired database, following a strategy of minimal change. However, in this approach none of p, q, and $\neg(p \wedge q)$ holds in the repaired database. In particular (since in [185] there is no distinction between data-facts and integrity constraints), the integrity constraint $\{\neg(p \wedge q)\}$ itself cannot be inferred, which violates the intended meaning of an integrity constraint in databases.

Many techniques for consistency enforcement and repairs of constraint violations have been suggested, among which are methods for resolving contradictions by quantitative considerations (such as 'majority vote' [186]) or qualitative ones (e.g., defining priorities on different sources of information or preferring certain data over another [18, 44]). Another common method of handling inconsistent (and incomplete) information is by turning to multi-valued semantics: three-valued formalisms such as the one considered in [23, 24] are used as a semantical basis of paraconsistent methods to construct database repairs [88] and are useful in general for pinpointing inconsistencies [218]. [23, 24] defines three-valued models of a database as the join of the minimal Herbrand model of the database instance and a two-valued model of the integrity constraints. Each three-valued model corresponds exactly to one repair. It is shown that for the set inclusion and cardinality preference relation a corresponding ordering relation exists on the set of three-valued models. Consequently, it follows that the repairs computed by the $\mathcal{A}$system (as presented here) correspond with the three-valued characterization of the repairs. Other approaches use lattice-based semantics to decode within the language itself some meta-information, such as confidence factors, amount of belief for or against a specific assertion, etc. These approaches combine corresponding formalisms of knowledge representation (such as annotated logic programs [15, 245, 246] or bilattice-based logics [19, 120, 199]), together with non-classical refutation procedures [119, 165, 245] that allow to detect inconsistent parts of a database and to maintain these inconsistencies explicitly without making the logical inference trivial.

4.3.8 Summary and Future Work

In this section we have developed a formal declarative foundation for rendering coherent data provided by different databases, and presented an implementation of this approach based on abductive logic programming.

The underlying formalism for representing knowledge in our approach is ID-Logic. Coherent composition of several data sources is encoded by a meta-theory, and it is possible to extend this theory to capture meta-information of the data facts, such as timestamps and source identities. Moreover, since the reasoning process of the system is based on abductive inference, no syntactical embedding of first-order formulas into other languages, nor any extensions of two-valued semantics, is necessary. Our approach has the ability to reason with *any* set of first-order formulas as integrity constraints[20]. This makes our approach more expressive than most other data integration approaches. Most of the other approaches impose (syntactic) conditions on the integrity constraints. Due to this expressiveness, we are not able to provide a general and strong completeness and termination result. Therefore, we have proven for a particular, yet frequent occurring, class of integrity constraints the termination and completeness of the abductive inference.

Because of the inherent modularity of the system, each component is independent and can be modified to meet different needs. Thus, for instance, the underlying solver may be replaced with any other solver that is capable of dealing with the meta-theory[21], and any improvement of the optimizer will affect the whole system and its efficiency, regardless the nature of its input. Also, the way of keeping data coherent is encapsulated in the component that integrates the data (i.e., the composer). This implies, in particular, that no input from the reasoner nor any other external policy for making preferences among conflicting sources is compulsory in order to resolve contradictions.

There remain many issues that deserve further elaboration. In the first place, more complex data has to be considered. Nowadays many large database applications involve geo-spacial and temporal data. In this domain conflicts between

[20] Provided, of course, that the constraints do not lead to floundering.

[21] As we have shown, it might be any sound abductive solver even with different underlying semantics e.g. the IFF-procedure, or ASP solvers such as DLV and sModels.

data and lack of information happen a lot. For example[22], since 1950's the Belgian/Flemisch government stores maps on which the destination of each piece of land is colored. Example destinations are woodland, housing, heavy industry, etc. Over the years these destinations have been often changed in various ways. Currently these maps are digitalized and stored in large databases, which have as main task to trace the evolution of a piece of land, because based on this information juridical and political decisions are taken. According to [256], good support for reasoning with time is currently lacking in all main commercial systems. Moreover, the temporal reasoning is complicated due to the fact that often information is only partially available (e.g. lost maps) or inconsistent data is available (in reality the destination of some piece of land is different than is indicated by the map, etc.)

We have shown the modularity of our approach, by presenting a straightforward integration in a mediator-based system. This is important because in this way a complete solution to the information integration problem can be provided. The user who faces such a problem can then select the necessary pieces to build his domain dependent integration. In order to obtain a complete solution for the information integration problem, the integration of integrity constraints has to be studied. Other further elaborations involve the study of other merging policies, e.g. based on majority vote, and the improvement of the $\mathcal{A}$system to deal with real databases, and much more.

[22] Personal communication with Nico Van de Weghe [256]

4.4 Using Signed Formulas To Compute Database Repairs

In the previous sections, we have studied the data integration problem from a 'pure' knowledge representation point of view. The problem is represented by an ID-Logic theory and then different tasks are studied. It turned out that most tasks of interest need abduction to solve the problem, allowing us to apply the $\mathcal{A}$system.

However this approach is not always computationally the most efficient one. In this section, we introduce a novel representation of the database repair problem as a propositional theory that consists of what we call *signed formulas*. The models of these signed theories are the repairs of the inconsistent database. To find these, we apply and evaluate a variety of computational systems:

- We show that the problem of finding repairs with minimal cardinality for a given database can be converted to the problem of finding minimal Herbrand models for the corresponding 'signed theory'. Thus, once the process for consistency restoration of the database has been represented by a signed theory (using a polynomial transformation), tools for minimal model computations (such as the Sicstus Prolog constraint solver [66], or the answer set programming solver DLV [105, 112]) can be used to efficiently find the required repairs.

- For finding repairs that are minimal with respect to set inclusion, satisfiability solvers on appropriate quantified Boolean formulas (QBF) can be utilized. Again, we provide a polynomial-time transformation to (signed) QBF theories, and show how QBF solvers [1, 195, 251, 226, 111, 144, 151] can be used to restore the database consistency.

4.4.1 Representation of Repairs by Signed Formulas

Since we aim to solve the same problem as in the previous section, our notion of a database $\mathcal{DB}$ as the pair $(\mathcal{D},\mathcal{IC})$ is not changed. The common global language is denoted by $\mathcal{L}^{\mathcal{DB}}$.

In contrast to the previous approaches, this technique requires propositional theories. Therefore, the database has to be grounded. Since the database instance is already ground, only the integrity constraints must be grounded. Let $\mathcal{DA}$ be a finite set of domain elements. Then $\mathcal{IC}^{\mathcal{DA}}$ is the grounding of the integrity constraints w.r.t. $\mathcal{DA}$:

$$\mathcal{IC}^{\mathcal{DA}} = \{\rho^{\mathcal{DA}}(\phi) | \phi \in \mathcal{IC}\}$$

where $\rho^{\mathcal{DA}}$ is a ground substitution[23] of the variables to the individuals of $\mathcal{DA}$.

[23]Thus, e.g., $\rho^{\mathcal{DA}}(\forall x\ \psi(x)) = \psi(p_1) \wedge \ldots \wedge \psi(p_n)$ and $\rho^{\mathcal{DA}}(\exists x\ \psi(x)) = \psi(p_1) \vee \ldots \vee \psi(p_n)$, where $p_1, \ldots, p_n$ are the elements of $\mathcal{A}$; the transformation for other formulas is standard.

A sensible selection for the set $\mathcal{DA}$ is the active domain of $\mathcal{DB}$, i.e. all domain elements that occur in the database instance $\mathcal{D}$. (Other choices are possible and are discussed at the end of the section.) Since $\mathcal{D}$ is finite, $\mathcal{DA}$ will also be finite and hence the grounding $\mathcal{IC}^{\mathcal{DA}}$ is also finite. To simplify the presentation, from now on it is assumed that $\mathcal{IC}$ is a set of ground integrity constraints.

For every (ground) atom $p \in \mathcal{L}^{\mathcal{DB}}$ we introduce a new atom, s_p, intuitively understood as 'switch p', or 'change the status of p', that is, s_p holds iff $p \in \mathsf{Insert} \cup \mathsf{Retract}$. For every integrity constraint $\psi \in \mathcal{IC}$ we define a new formula, $\overline{\psi}$, obtained from ψ by simultaneously substituting every appearance of an atom p by a corresponding expression τ_p that is defined as follows:

$$\tau_p = \begin{cases} \neg s_p & \text{if } p \in \mathcal{D}, \\ s_p & \text{otherwise.} \end{cases}$$

The formula $\overline{\psi} = \psi\,[\,\tau_{p_1}/p_1\,,\,\dots\,,\,\tau_{p_m}/p_m\,]$ (i.e., the simultaneous substitution in ψ of all the atomic formulas p_i, $1 \leq i \leq m$, by their 'signed expressions' τ_{p_i}) is called the *signed formula* that is obtained from ψ.

Given a repair $\mathcal{R} = (\mathsf{Insert}, \mathsf{Retract})$ of a database $\mathcal{DB}$, define a valuation $\nu^{\mathcal{R}}$ on $\{s_p \mid p \in \mathcal{L}^{\mathcal{DB}}\}$ as follows:

$$\nu^{\mathcal{R}}(s_p) = t \text{ iff } p \in \mathsf{Insert} \cup \mathsf{Retract}.$$

$\nu^{\mathcal{R}}$ is called the valuation that is *associated with* $\mathcal{R}$. Conversely, a valuation ν on $\{s_p \mid p \in \mathcal{L}^{\mathcal{DB}}\}$ *induces* a database update $\mathcal{R}^{\nu} = (\mathsf{Insert}, \mathsf{Retract})$, where $\mathsf{Insert} = \{p \notin \mathcal{D} \mid \nu(s_p) = t\}$ and $\mathsf{Retract} = \{p \in \mathcal{D} \mid \nu(s_p) = t\}$.[24] Obviously, these mappings are the inverse of each other.

Example 4.29 Consider the (ground) database $\mathcal{DB} = (\{p\}, \{p \to q\})$. In this case, the sign formula of $\psi = p \to q$ is $\overline{\psi} = \neg s_p \to s_q$, or, equivalently, $s_p \vee s_q$. Intuitively, this formula indicates that in order to restore the consistency of $\mathcal{DB}$, at least one of p or q should be 'switched', i.e., either p should be removed from the database or q should be inserted to it. Indeed, the three classical models of $\overline{\psi}$ are exactly the three valuations of $\{s_p, s_q\}$ that are corresponding to the three repairs of $\mathcal{DB}$: $\{retract(p)\}$, $\{insert(q)\}$ and $\{retract(p), insert(q)\}$. The next theorem shows that this is not a coincidence.

Theorem 28
Let $\mathcal{DB} = (\mathcal{D}, \mathcal{IC})$ be a database. Denote: $\overline{\mathcal{IC}} = \{\overline{\psi} \mid \psi \in \mathcal{IC}\}$.

a) *if $\mathcal{R}$ is a repair of $\mathcal{DB}$ then $\nu^{\mathcal{R}}$ is a model of $\overline{\mathcal{IC}}$,*

b) *if ν is a model of $\overline{\mathcal{IC}}$ then $\mathcal{R}^{\nu}$ is a repair of $\mathcal{DB}$.*

[24] Clearly, $\mathcal{R}^{\nu}$ is an update of $\mathcal{DB}$, but it is not necessarily a repair of $\mathcal{DB}$ (see Definition 4.8).

Proof. For (a), suppose that $\mathcal{R}$ is a repair of $\mathcal{DB} = (\mathcal{D}, \mathcal{IC})$. Then, in particular, $\mathcal{D}^{\mathcal{R}} \models \mathcal{IC}$, where $\mathcal{D}^{\mathcal{R}} = \mathcal{D} \cup \mathsf{Insert} \setminus \mathsf{Retract}$. Let $\mathcal{H}^{\mathcal{D}^{\mathcal{R}}}$ be the least Herbrand model of $\mathcal{D}^{\mathcal{R}}$, and let $\psi \in \mathcal{IC}$. Then $\mathcal{H}^{\mathcal{D}^{\mathcal{R}}}(\psi) = t$, and so it remains to show that $\nu^{\mathcal{R}}(\overline{\psi}) = \mathcal{H}^{\mathcal{D}^{\mathcal{R}}}(\psi)$. The proof of this is by induction on the structure of ψ, and we show only the base step (the rest is trivial), i.e., for every $p \in \mathcal{L}^{\mathcal{DB}}$, $\nu^{\mathcal{R}}(\overline{p}) = \mathcal{H}^{\mathcal{D}^{\mathcal{R}}}(p)$. Indeed,

– $p \in \mathcal{D} \setminus \mathsf{Retract} \Rightarrow p \in \mathcal{D}^{\mathcal{R}} \Rightarrow \nu^{\mathcal{R}}(\overline{p}) = \nu^{\mathcal{R}}(\neg s_p) = \neg\nu^{\mathcal{R}}(s_p) = \neg f = t = \mathcal{H}^{\mathcal{D}^{\mathcal{R}}}(p)$.
– $p \in \mathsf{Retract} \Rightarrow p \in \mathcal{D} \setminus \mathcal{D}^{\mathcal{R}} \Rightarrow \nu^{\mathcal{R}}(\overline{p}) = \nu^{\mathcal{R}}(\neg s_p) = \neg\nu^{\mathcal{R}}(s_p) = \neg t = f = \mathcal{H}^{\mathcal{D}^{\mathcal{R}}}(p)$.
– $p \in \mathsf{Insert} \Rightarrow p \in \mathcal{D}^{\mathcal{R}} \setminus \mathcal{D} \Rightarrow \nu^{\mathcal{R}}(\overline{p}) = \nu^{\mathcal{R}}(s_p) = t = \mathcal{H}^{\mathcal{D}^{\mathcal{R}}}(p)$.
– $p \notin \mathcal{D} \cup \mathsf{Insert} \Rightarrow p \notin \mathcal{D}^{\mathcal{R}} \Rightarrow \nu^{\mathcal{R}}(\overline{p}) = \nu^{\mathcal{R}}(s_p) = f = \mathcal{H}^{\mathcal{D}^{\mathcal{R}}}(p)$.

For part (b), suppose that ν is a model of $\overline{\mathcal{IC}}$. Let

$$\mathcal{R}^{\nu} = (\mathsf{Insert}, \mathsf{Retract}) = (\{p \notin \mathcal{D} \mid \nu(s_p) = t\}, \{p \in \mathcal{D} \mid \nu(s_p) = t\}).$$

We shall show that $\mathcal{R}^{\nu}$ is a repair of $\mathcal{DB}$. According to Definition 4.8, it is obviously an update. It remains to show that every $\psi \in \mathcal{IC}$ follows from $\mathcal{D}^{\mathcal{R}} = \mathcal{D} \cup \mathsf{Insert} \setminus \mathsf{Retract}$, i.e., that $\mathcal{H}^{\mathcal{D}^{\mathcal{R}}}(\psi) = t$, where $\mathcal{H}^{\mathcal{D}^{\mathcal{R}}}$ is the least Herbrand model of $\mathcal{D}^{\mathcal{R}}$. Since ν is a model of $\overline{\mathcal{IC}}$, $\nu(\overline{\psi}) = t$, and so it remains to show that $\mathcal{H}^{\mathcal{D}^{\mathcal{R}}}(\psi) = \nu(\overline{\psi})$. Again, the proof is by induction on the structure of ψ, and we show only the base step, that is: for every $p \in \mathcal{L}^{\mathcal{DB}}$, $\mathcal{H}^{\mathcal{D}^{\mathcal{R}}}(p) = \nu(\overline{p})$:

– $p \in \mathcal{D} \setminus \mathsf{Retract} \Rightarrow p \in \mathcal{D}^{\mathcal{R}}$, $\nu(s_p) = f \Rightarrow \mathcal{H}^{\mathcal{D}^{\mathcal{R}}}(p) = t = \neg\nu(s_p) = \nu(\neg s_p) = \nu(\overline{p})$.
– $p \in \mathsf{Retract} \Rightarrow p \in \mathcal{D} \setminus \mathcal{D}^{\mathcal{R}}$, $\nu(s_p) = t, \Rightarrow \mathcal{H}^{\mathcal{D}^{\mathcal{R}}}(p) = f = \neg\nu(s_p) = \nu(\neg s_p) = \nu(\overline{p})$.
– $p \in \mathsf{Insert} \Rightarrow p \in \mathcal{D}^{\mathcal{R}} \setminus \mathcal{D}$, $\nu(s_p) = t, \Rightarrow \mathcal{H}^{\mathcal{D}^{\mathcal{R}}}(p) = t = \nu(s_p) = \nu(\overline{p})$.
– $p \notin \mathcal{D} \cup \mathsf{Insert} \Rightarrow p \notin \mathcal{D}^{\mathcal{R}}$, $\nu(s_p) = f, \Rightarrow \mathcal{H}^{\mathcal{D}^{\mathcal{R}}}(p) = f = \nu(s_p) = \nu(\overline{p})$. □

The last theorem implies, in particular, that in order to compute repairs for a given database $\mathcal{DB}$, it is sufficient to find the models of the signed formulas that are induced by the integrity constraints of $\mathcal{DB}$; the pairs that are induced by these models are the repairs of $\mathcal{DB}$.

EXAMPLE 4.30 Consider again the (grounded) database of Examples 4.29. The corresponding signed formula $\overline{\psi} = s_p \vee s_q$ has three models $\{s_p : t, s_q : f\}$, $\{s_p : f, s_q : t\}$, and $\{s_p : t, s_q : t\}$.[25] These models induce, respectively, three pairs, $(\{\}, \{p\})$, $(\{q\}, \{\})$, and $(\{q\}, \{p\})$, which are the repairs of $\mathcal{DB}$ (cf. Example 4.29).

4.4.2 Computing Preferred Repairs by Model Generation

In this section we show how solvers for constraint logic programs (CLPs), answer-set programming (ASP) and SAT solvers can be used for computing $\leq_c$-preferred

[25] $p : x$ denotes here the fact that the atom p is assigned the value x by the corresponding valuation.

repairs and $\leq_i$-preferred repairs. The experimental results are presented in Section 4.4.4.

Computing $\leq_c$-Preferred Repairs

By Theorem 28, the repairs of a database correspond exactly to the models of the signed theory. It is straightforward to see that $\leq_c$-preferred repairs of $\mathcal{DB}$ (i.e., those with minimal cardinality) correspond to models of $\overline{\mathcal{IC}}$ that minimize the number of t-assignments of the atoms s_p. Hence, the problem is to find Herbrand models for $\overline{\mathcal{IC}}$ with minimal cardinality (called $\leq_c$-minimal Herbrand models).

THEOREM 29
Let $\mathcal{DB} = (\mathcal{D}, \mathcal{IC})$ be a database and $\overline{\mathcal{IC}} = \{\overline{\psi} \mid \psi \in \mathcal{IC}\}$. Then:

a) *if $\mathcal{R}$ is a $\leq_c$-preferred repair of $\mathcal{DB}$, then $\nu^{\mathcal{R}}$ is a $\leq_c$-minimal Herbrand model of $\overline{\mathcal{IC}}$.*

b) *if ν is a $\leq_c$-minimal Herbrand model of $\overline{\mathcal{IC}}$, then $\mathcal{R}^{\nu}$ is a $\leq_c$-preferred repair of $\mathcal{DB}$.*

We discuss two techniques to compute $\leq_c$-minimal Herbrand models. The first approach is to use a finite domain CLP solver. Encoding the computation of $\leq_c$-preferred repair using a finite domain constraint solver is a straightforward process. The 'switch atoms' s_p are encoded as finite domain variables with domain $\{0, 1\}$. A typical encoding specifies the relevant constraints (i.e., the encoding of $\overline{\mathcal{IC}}$), assigns a special variable, `Sum`, for summing-up all the signed variables that are assigned the value '1', and asks for a solution with a minimal value for `Sum`.

EXAMPLE 4.31 Below is a code for repairing the database of Example 4.30 with Sicstus Prolog finite domain constraint solver CLP(FD) [66][26].

```
domain([Sp,Sq],0,1),                  % domain of the signed atoms
Sp #\/ Sq,                            % the signed theory
sum([Sp,Sq],#=,Sum),                  % Sum = num of vars with val 1
minimize(labeling([],[Sp,Sq]),Sum). % find a solution with min sum
```

The solutions computed here are $[1, 0]$ and $[0, 1]$, and the value of `Sum` is 1. This means that the cardinality of the $\leq_c$-preferred repairs of $\mathcal{DB}$ should be 1, and that these repairs are induced by the valuations $\nu_1 = \{s_p : t, s_q : f\}$ and $\nu_2 = \{s_p : f, s_q : t\}$. Thus, the two $\leq_c$-minimal repairs here are $(\{\}, \{p\})$ and $(\{q\}, \{\})$, which indeed insert or retract exactly one atomic formula.

[26] A Boolean constraint solver would also be appropriate here. As Sicstus Prolog Boolean constraint solver has no minimization capabilities, we prefer to use here the finite domain constraint solver.

A second approach is to use the disjunctive logic programming system DLV [112]. To compute $\leq_c$-minimal repairs using DLV, the signed theory $\overline{\mathcal{IC}}$ is transformed into a propositional clausal form. A clausal theory is a special case of a disjunctive logic program without negation in the body of the clauses. The stable models of a disjunctive logic program without negation as failure in the body of rules coincide exactly with the $\leq_i$-minimal models of such a program. Hence, by transforming the signed theory $\overline{\mathcal{IC}}$ to clausal form, DLV can be used to compute $\leq_i$-minimal Herbrand models. To eliminate models with non-minimal cardinality, *weak constraints* are used. A weak constraint is a formula for which a cost value is defined. With each model computed by DLV, a cost is defined as the sum of the cost values of all weak constraints violated in the model. The DLV system can be asked to generate models with minimal total cost. The set of weak constraints used to compute $\leq_c$-minimal repairs is exactly the set of all atoms s_p; each atom has cost 1. Clearly, $\leq_i$-minimal models of a theory with minimal total cost are exactly the models with least cardinality.

EXAMPLE 4.32 Below is a code for repairing the database of Example 4.30 with DLV.

```
Sp v Sq.        % the clause
:~ Sp.          % the weak constraints (their cost is 1 by default)
:~ Sq.
```

Clearly, the solutions here are $\{s_p : t, s_q : f\}$ and $\{s_p : f, s_q : t\}$. These valuations induce the two $\leq_c$-minimal repairs of $\mathcal{DB}$, $\mathcal{R}_1 = (\{\}, \{p\})$ and $\mathcal{R}_2 = (\{q\}, \{\})$.

Computing $\leq_i$-Preferred Repairs

The $\leq_i$-preferred repairs of a database correspond to minimal Herbrand models with respect to set inclusion of the signed theory $\overline{\mathcal{IC}}$. We focus on the computation of one minimal model. The reason is simply that in most sizable applications, the computation of all minimal models is not feasible (there are too many of them). We consider here three simple techniques to compute a $\leq_i$-preferred repair. In the next section we consider another more complex method.

I. One technique, mentioned already in the previous section, is to transform $\overline{\mathcal{IC}}$ to clausal form and use the DLV system. In this case the weak constraints are not needed.

II. Another possibility is to adapt CLP-techniques to compute $\leq_i$-minimal models of Boolean constraints. The idea is simply to make sure that whenever a Boolean variable (or a finite domain variable with domain $\{0, 1\}$) is selected for being assigned a value, one first assigns the value 0 before trying to assign the value 1.

PROPOSITION 4.5 If the above strategy for value selection is used, then the first computed model is provably a $\leq_i$-minimal model.

Proof. Consider the search tree of the CLP-problem. Each path in this tree represents a value assignment to a subset of the constraint variables. Internal nodes, corresponding to partial solutions, are labeled with the variable selected by the labeling function of the solver and have two children: the left child assigns value 0 to the selected variable and the right child assigns value 1. We say that node n_2 is on the right of a node n_1 in this tree if n_2 appears in the right subtree, and n_1 appears in the left subtree of the deepest common ancestor node of n_1 and n_2. It is then easy to see that in such a tree, each node n_2 to the right of a node n_1 assigns the value 1 to the variable selected in this ancestor node, whereas n_1 assigns value 0 to this variable. Consequently, the left-most node in the search tree which is a model of the Boolean constraints, is $\leq_i$-minimal. □

In CLP-systems such as Sicstus Prolog, one can control the order in which values are assigned to variables. We have implemented the above strategy and discuss the results in Section 4.4.4.

III. A third technique considered here uses SAT-solvers. SAT-solvers, such as zChaff [203], do not compute directly minimal models, but can be easily extended to do so. The algorithm uses the SAT-solver to generate models of the theory $\mathcal{T}$, until it finds a minimal model. Minimality of a model M of $\mathcal{T}$ can be verified by checking the unsatisfiability of $\mathcal{T}$, augmented with the axioms $\bigvee_{p \in M} \neg p$ and $\bigwedge_{p \notin M} \neg p$. The model M is minimal exactly when these axioms are inconsistent with $\mathcal{T}$. This approach has been tested using the SAT solver zChaff [203]; the results are discussed in Section 4.4.4.

4.4.3 Computing $\leq_i$-Preferred Repairs by QBF Solvers

In this section we show how solvers for quantified Boolean formulas (QBFs) can be used for computing the $\leq_i$-preferred repairs of a given database. In this case it is necessary to add to the signed formulas of $\overline{\mathcal{IC}}$ an axiom (represented by a quantified Boolean formula) that expresses $\leq_i$-minimality, i.e., that a $\leq_i$-preferred repair is not included in any other database repair. Then, QBF solvers such as QUBOS [1], EVALUATE [195], QUIP [251], QSOLVE [226], QuBE [111], QKN [144], SEMPROP [177], and DECIDE [151], can be applied to the signed quantified Boolean theory that is obtained, in order to compute the $\leq_i$-preferred repairs of the database. Below we give a formal description of this process.

Quantified Boolean Formulas

Quantified Boolean formulas (QBFs) are propositional formulas extended with quantifiers $\forall, \exists$ over propositional variables. In what follows we shall denote propositional formulas by Greek lower-case letters (usually ψ, ϕ) and QBFs by Greek upper-case letters (e.g., Ψ, Φ). Intuitively, the meaning of a QBF of the form $\exists p\, \forall q\, \psi$ is that there exists a truth assignment of p such that ψ is true for every truth assignment of q. Next we formalize this intuition.

As usual, we say that an occurrence of an atomic formula p is *free* if it is not in the scope of a quantifier $\mathsf{Q}p$, for $\mathsf{Q} \in \{\forall, \exists\}$, and we denote by $\Psi[\phi_1/p_1, \ldots, \phi_m/p_m]$ the uniform substitution of each free occurrence of a variable p_i in Ψ by a formula ϕ_i, for $i = 1, \ldots, m$. The notion of a *valuation* is extended to QBFs as follows: Given a function $\nu_{\rm at} : \mathcal{L}^{\mathcal{DB}} \cup \{\mathsf{t}, \mathsf{f}\} \to \{t, f\}$ s.t. $\nu(\mathsf{t}) = t$ and $\nu(\mathsf{f}) = f$, a valuation ν on QBFs is recursively defined as follows:

$\nu(p) = \nu_{\rm at}(p)$ for every $p \in \mathcal{L}^{\mathcal{DB}} \cup \{\mathsf{t}, \mathsf{f}\}$,

$\nu(\neg\psi) = \neg\nu(\psi)$,

$\nu(\psi \circ \phi) = \nu(\psi) \circ \nu(\phi)$, where $\circ \in \{\wedge, \vee, \to, \leftrightarrow\}$,

$\nu(\forall p\, \psi) = \nu(\psi[\mathsf{t}/p]) \wedge \nu(\psi[\mathsf{f}/p])$,

$\nu(\exists p\, \psi) = \nu(\psi[\mathsf{t}/p]) \vee \nu(\psi[\mathsf{f}/p])$.

A valuation ν *satisfies* a QBF Ψ if $\nu(\Psi) = t$; ν is a *model* of a set Γ of QBFs if it satisfies every element of Γ. A QBF Ψ is *entailed by* a set Γ of QBFs (notation: $\Gamma \vdash \Psi$) if every model of Γ is also a model of Ψ. In what follows we shall use the following notations: for two valuations ν_1 and ν_2 we denote by $\nu_1 \leq \nu_2$ that for every atomic formula p, $\nu_1(p) \to \nu_2(p)$ is true. We shall also write $\nu_1 < \nu_2$ to denote that $\nu_1 \leq \nu_2$ and $\nu_2 \not\leq \nu_1$.

Representing $\leq_i$-Preferred Repairs by Signed QBFs

It is well-known that quantified Boolean formulas can be used for representing circumscription [197], thus they properly express logical minimization [49, 50]. In our case we use this property for expressing minimization of repairs w.r.t. set inclusion.

Given a database $\mathcal{DB} = (\mathcal{D}, \mathcal{IC})$, denote by $\overline{\mathcal{IC}}_\wedge$ the conjunction of all the elements in $\overline{\mathcal{IC}}$ (i.e., the conjunction of all the signed formulas that are obtained from the integrity constraints of $\mathcal{DB}$). Consider the following QBF, denoted $\Psi_{\mathcal{DB}}$:

$$\forall s'_{p1}, \ldots, s'_{p_n} \Big(\overline{\mathcal{IC}}_\wedge \left[s'_{p_1}/s_{p_1}, \ldots, s'_{p_n}/s_{p_n} \right] \to \big(\bigwedge_{i=1}^{n} (s'_{p_i} \to s_{p_i}) \to \bigwedge_{i=1}^{n} (s_{p_i} \to s'_{p_i}) \big) \Big).$$

Consider a model ν of $\overline{\mathcal{IC}}_\wedge$, i.e., a valuation for $s_{p_1}, \ldots, s_{p_n}$ that makes $\overline{\mathcal{IC}}_\wedge$ true. The QBF $\Psi_{\mathcal{DB}}$ expresses that every interpretation μ (valuation for $s'_{p_1}, \ldots, s'_{p_n}$) that is a model of $\overline{\mathcal{IC}}_\wedge$, has the property that $\mu \leq \nu$ implies $\nu \leq \mu$, i.e., there is no model μ of $\overline{\mathcal{IC}}_\wedge$, s.t. the set $\{s_p \mid \nu(s_p) = t\}$ properly contains the set $\{s_p \mid \mu(s_p) = t\}$. In terms of database repairs, this means that if $\mathcal{R}^\nu = (\mathsf{Insert}, \mathsf{Retract})$ and $\mathcal{R}^\mu = (\mathsf{Insert}', \mathsf{Retract}')$ are the database repairs that are associated, respectively, with ν and μ, then $\mathsf{Insert}' \cup \mathsf{Retract}' \not\subset \mathsf{Insert} \cup \mathsf{Retract}$. It follows, therefore, that in this case $\mathcal{R}^\nu$ is a $\leq_i$-preferred repair of $\mathcal{DB}$, and in general $\Psi_{\mathcal{DB}}$ represents $\leq_i$-minimality.

Example 4.33 With the database $\mathcal{DB}$ of Example 4.29, $\overline{\mathcal{IC}} \cup \Psi_{\mathcal{DB}}$ is the following theory, Γ:

$$\left\{ s_p \vee s_q \ ,\ \forall s'_p \forall s'_q \Big((s'_p \vee s'_q) \to \big((s'_p \to s_p) \wedge (s'_q \to s_q) \to (s_p \to s'_p) \wedge (s_q \to s'_q) \big) \Big) \right\}.$$

The models of Γ are those that assign t either to s_p or to s_q, but not to both of them, i.e., $\nu_1 = (s_p : t, s_q : f)$ and $\nu_2 = (s_p : f, s_q : t)$. The database updates that are induced by these valuations are, respectively, $\mathcal{R}^{\nu_1} = (\{\}, \{p\})$ and $\mathcal{R}^{\nu_2} = (\{q\}, \{\})$. By Theorem 30 below, these are the only $\leq_i$-preferred repairs of $\mathcal{DB}$.

Theorem 30
Let $\mathcal{DB} = (\mathcal{D}, \mathcal{IC})$ be a database and $\overline{\mathcal{IC}} = \{\overline{\psi} \mid \psi \in \mathcal{IC}\}$. Then:

a) *if $\mathcal{R}$ is a $\leq_i$-preferred repair of $\mathcal{DB}$ then $\nu^{\mathcal{R}}$ is a model of $\overline{\mathcal{IC}} \cup \Psi_{\mathcal{DB}}$,*

b) *if ν is a model of $\overline{\mathcal{IC}} \cup \Psi_{\mathcal{DB}}$ then $\mathcal{R}^\nu$ is a $\leq_i$-preferred repair of $\mathcal{DB}$.*

Proof. Suppose that $\mathcal{R} = (\mathsf{Insert}, \mathsf{Retract})$ is an $\leq_i$-preferred repair of $\mathcal{DB}$. In particular, it is a repair of $\mathcal{DB}$ and so, by Theorem 28, $\nu^{\mathcal{R}}$ is a model of $\overline{\mathcal{IC}}$. Since Theorem 28 also assures that a database update that is induced by a model of $\overline{\mathcal{IC}}$ is a repair of $\mathcal{DB}$, in order to prove both parts of the theorem, it remains to show that the fact that $\nu^{\mathcal{R}}$ satisfies $\Psi_{\mathcal{DB}}$ is a necessary and sufficient condition for assuring that $\mathcal{R}$ is $\leq_i$-minimal among the repairs of $\mathcal{DB}$. Indeed, $\nu^{\mathcal{R}}$ satisfies $\Psi_{\mathcal{DB}}$ iff for every valuation μ that satisfies $\overline{\mathcal{IC}}_\wedge$ and for which $\mu \leq \nu^{\mathcal{R}}$, it is also true that $\nu^{\mathcal{R}} \leq \mu$. Thus, $\nu^{\mathcal{R}}$ satisfies $\Psi_{\mathcal{DB}}$ iff there is no model μ of $\overline{\mathcal{IC}}$ s.t. $\mu < \nu^{\mathcal{R}}$, iff (by Theorem 28 again) there is no repair $\mathcal{R}'$ of $\mathcal{DB}$ s.t. $\nu^{\mathcal{R}'} < \nu^{\mathcal{R}}$, iff there is no repair $\mathcal{R}' = (\mathsf{Insert}', \mathsf{Retract}')$ s.t. $\mathsf{Insert}' \cup \mathsf{Retract}' \subset \mathsf{Insert} \cup \mathsf{Retract}$, iff $\mathcal{R}$ is a $\leq_i$-minimal repairs of $\mathcal{DB}$. $\square$

Note 4.4.1
(*Complexity results*) A skeptical (also called conservative) approach to query answering is considered, e.g., in [16, 138], where an answer to a query $\mathcal{Q}$ and a database $\mathcal{DB}$ is evaluated with respect to (the databases that are obtained from) *all* the $\leq_i$-preferred repairs of $\mathcal{DB}$. A credulous approach to the same problem

evaluates queries with respect to *some* $\leq_i$-preferred repair of $\mathcal{DB}$. Theorem 30 implies the following upper complexity bounds for these approaches:

COROLLARY 4.2 Credulous query answering lies in $\mathcal{L}_2^P$, and skeptical query answering is in Π_2^P.

Proof. By Theorem 30, credulous query answering is equivalent to satisfiability checking for $\overline{\mathcal{IC}} \cup \Psi_{\mathcal{DB}}$, and conservative query answering is equivalent to entailment checking for the same theory (see also Corollary 4.3 below). Thus, these decision problems can be encoded by QBFs in prenex normal form with exactly one quantifier alternation. The corollary is obtained, now, by the following well-known result:

PROPOSITION 4.6 [86] Given a propositional formula ψ, whose atoms are partitioned into $i \geq 1$ sets $\{p_1^1, \ldots, p_{m_1}^1\}, \ldots, \{p_1^i, \ldots, p_{m_i}^i\}$, deciding whether

$$\exists p_1^1, \ldots, \exists p_{m_1}^1, \forall p_1^2, \ldots, \forall p_{m_2}^2, \ldots, \mathsf{Q} p_1^i, \ldots, \mathsf{Q} p_{m_i}^i \psi$$

is true, is $\mathcal{L}_i^P$-complete (where $\mathsf{Q} = \exists$ if i is odd and $\mathsf{Q} = \forall$ if i is even). Also, deciding if

$$\forall p_1^1, \ldots, \forall p_{m_1}^1, \exists p_1^2, \ldots, \exists p_{m_2}^2, \ldots, \mathsf{Q} p_1^i, \ldots, \mathsf{Q} p_{m_i}^i \psi$$

is true, is Π_i^P-complete (where $\mathsf{Q} = \forall$ if i is odd and $\mathsf{Q} = \exists$ if i is even). □

As shown, e.g., in [138], the complexity bounds specified in the last corollary are strict, i.e., these decision problems are hard for the respective complexity classes.

NOTE 4.4.2
(*Consistent query answering*) Another consequence of Theorem 30 is that the conservative approach to query answering [16, 138] may be represented in our context in terms of a consequence relation as follows:

COROLLARY 4.3 $\mathcal{Q}$ is a consistent query answer of a database $\mathcal{DB} = (\mathcal{D}, \mathcal{IC})$ in the sense of [16, 138] iff $\overline{\mathcal{IC}} \cup \Psi_{\mathcal{DB}} \vdash \mathcal{Q}$.

The last corollary and Section 4.4.2 provide, therefore, some additional methods for consistent query answering, all of them are based on signed theories.

4.4.4 Experiments and Comparative Study

The idea of using formulas that introduce new ('signed') variables aimed at designating the truth assignments of other related variables is used, for different purposes, e.g. in [21, 22, 48, 49]. In the area of database integration, signed variables are used in [138], and have a similar intended meaning as in our case. In [138], however, only $\leq_i$-preferred repairs are considered, and a rewriting process for converting relational queries over a database with constraints to extended disjunctive queries (with two kinds of negations) over database without constraints, must be employed. As a result, only solvers that are able to process disjunctive Datalog programs and compute their stable models (e.g., DLV), can be applied. In contrast, as we have already noted above, motivated by the need to find *practical* and *effective* methods for repairing inconsistent databases, signed formulas serve here as a representative platform that can be directly used by a variety of off-the-shelf applications for computing (either $\leq_i$-preferred or $\leq_c$-preferred) repairs. In what follows we examine some of these applications and compare their appropriateness to the kind of problems that we are dealing with.

We have randomly generated instances of a database, consisting of three relations: *teacher* of the schema (`teacher_name`), *course* of the schema (`course_name`), and *teaches* of the schema (`teacher_name`, `course_name`). Also, the following two integrity constraints were specified:

ic1 A course is given by one teacher:

$$\begin{aligned}\forall X, Y, Z.\quad & teacher(X) \wedge teacher(Y) \wedge course(Z) \wedge teaches(X, Z) \wedge \\ & teaches(Y, Z) \rightarrow X = Y\end{aligned}$$

ic2 Each teacher gives at least one course:

$$\forall X.teacher(X) \rightarrow \exists Y.course(Y) \wedge teaches(X, Y)$$

The next four test cases (identified by the enumeration below) were considered:

1. Small database instances with **ic1** as the only constraint.

2. Larger database instances with **ic1** as the only constraint.

3. Databases with $\mathcal{IC} = \{\mathbf{ic1}, \mathbf{ic2}\}$, where the number of courses equals the number of teachers.

4. Databases with $\mathcal{IC} = \{\mathbf{ic1}, \mathbf{ic2}\}$ and with fewer courses than teachers.

Note that in the first two test cases, only retractions of database facts are needed in order to restore consistency, in the third test case both insertion and retractions may be needed, and the last test case is unsolvable, as the theory is not satisfiable.

For each benchmark we generated a sequence of instances with an increasing number of database facts, and tested them w.r.t. the following applications:

- **ASP/CLP-solvers:**
  DLV [112] (release 2003-05-16), CLP(FD) [66] (version 3.10.1).

Table 4.1: Results for test case 1

Test info.			$\leq_i$-repairs						$\leq_c$-repairs	
No.	vars	IC	size	DLV	CLP	zChaff	SEMPROP	QuBE	DLV	CLP
1	20	12	8	0.005	0.010	0.024	0.088	14.857	0.011	0.020
2	25	16	7	0.013	0.010	0.018	0.015		0.038	0.020
3	30	28	12	0.009	0.020	0.039	0.100		0.611	0.300
4	35	40	15	0.023	0.020	0.008	0.510		2.490	1.270
5	40	48	16	0.016	0.020	0.012	0.208		3.588	3.220
6	45	42	17	0.021	0.030	0.008	0.673		12.460	10.350
7	50	38	15	0.013	0.020	0.009	0.216		23.146	20.760
8	55	50	20	0.008	0.030	0.018	1.521		29.573	65.530
9	60	58	21	0.014	0.030	0.036	3.412		92.187	136.590
10	65	64	22	0.023	0.030	0.009	10.460		122.399	171.390
11	70	50	22	0.014	0.030	0.019	69.925			
12	75	76	27	0.021	0.030	0.010	75.671			
13	80	86	29	0.021	0.030	0.009	270.180			
14	85	76	30	0.022	0.030	0.010				
15	90	78	32	0.024	0.040	0.020				
16	95	98	35	0.027	0.040	0.047				
17	100	102	40	0.017	0.040	0.016				
18	105	102	37	0.018	0.040	0.033				
19	110	124	43	0.030	0.040	0.022				
20	115	116	44	0.027	0.040	0.041				

- **QBF-solvers:**
  SEMPROP [177] (release 24.02.02), QuBE-BJ [111] (release 1.3), DECIDE [151].
- **SAT-solvers:**
  A minimal-model generator based on zChaff [203].

The goal was to construct $\leq_i$-preferred repairs within a time limit of five minutes. The systems DLV and CLP(FD) were tested also for constructing $\leq_c$-preferred repairs. All the experiments were done on a Linux machine, 800MHz, with 512MB memory. Tables 4.1–4.4 show the results for providing the first answer.[27] No timings are given for computing all repairs; there are so many of them (of which a lot are 'symmetric') so that printing them is soon infeasible.

The results of the first benchmark (Table 4.1) already indicate that DLV, CLP, and zChaff perform much better than the QBF-solvers. In fact, among the QBF-

[27]Times are in given in seconds, empty cells mean that timeout is reached without an answer, vars is the number of variables, IC is the number of grounded integrity constraints, and size is the size of the repairs.

Table 4.2: Results for test case 2

Test info.			$\leq_i$-repairs			
No.	vars	IC	size	DLV	CLP	zChaff
1	480	470	171	0.232	0.330	0.155
2	580	544	214	0.366	0.440	0.051
3	690	750	265	0.422	0.610	0.062
4	810	796	300	0.639	0.860	0.079
5	940	946	349	0.815	1.190	0.094
6	1080	1108	410	1.107	1.560	0.123
7	1230	1112	428	1.334	2.220	0.107
8	1390	1362	509	1.742	2.580	0.135
9	1560	1562	575	2.254	3.400	0.194
10	1740	1782	675	2.901	4.140	0.182
11	1930	2042	719	3.592	5.260	0.253

Table 4.3: Results for test case 3

Test info.		$\leq_i$-repairs				$\leq_c$-repairs	
No.	vars	size	DLV	CLP	zChaff	DLV	CLP
1	25	4	0.008	0.030	0.066	0.010	0.05
2	36	9	0.008	0.030	0.087	0.070	0.42
3	49	15	0.027	0.250	0.050	0.347	9.48
4	64	23	0.019	0.770	0.013	2.942	58.09
5	81	30	0.012	4.660	0.102	26.884	
6	100	34	0.021		0.058	244.910	
7	121	38	0.626		1.561		
8	144	47	0.907		2.192		
9	169	51	0.161		0.349		
10	196	68	1.877		4.204		
11	225	70	8.496		16.941		

Table 4.4: Results for test case 4

Test info.			$\leq_i$-repairs			$\leq_c$-repairs	
No.	teachers	courses	DLV	CLP	zChaff	DLV	CLP
1	5	4	0.001	0.01	0.001	0.001	0.001
2	7	5	0.005	0.13	0.010	0.005	0.120
3	9	6	0.040	1.41	0.020	0.042	1.400
4	11	7	0.396	17.18	0.120	3.785	17.170
5	13	8	3.789		1.050	44.605	
6	15	9	44.573		13.370		
7	17	10					

solvers that were tested, only SEMPROP could repair within the time limit most of the database instances of benchmark 1, and none of them could successfully repair (within the time restriction) the larger database instances, tested in benchmark 2. Also, we encountered some space limitation problems and a bug[28] in DECIDE, and this discouraged us from using it in our experiments.

Another observation from Tables 4.1–4.4 is that DLV, CLP, and the zChaff-based system, perform very good for minimal inclusion greedy algorithms. However, when using DLV and CLP for cardinality minimization, their performance is much worse. This is due to an exhaustive search for a $\leq_c$-minimal solution.

While in benchmark 1 the time differences among DLV, CLP, and zChaff, for computing $\leq_i$-repairs are marginal, in the other benchmarks the differences become more evident. Thus, for instance, zChaff performs better than the other solvers w.r.t. bigger database instances with many simple constraints (see benchmark 2), while DLV performs better when the problem has bigger and more complicated sets of constraints (see benchmark 3). The SAT approach with zChaff was the fastest in detecting unsatisfiable situations (see benchmark 4). As shown in Table 4.4, detecting unsatisfiability requires a considerable amount of time, even for small instances.

Some of the conclusions from the experiments may be summarized as follows:

1. In principle, QBF-solvers, CLP-solvers, ASP-solvers, and SAT-solvers are all adequate tools for computing database repairs.

2. All the QBF-solvers, as well as DLV and zChaff, are 'black-boxes' that accept the problem specification in a certain format. In contrast, CLP(FD) provides a more 'open' environment, in which it is possible to incorporate problem-specific search algorithms, such as the greedy algorithm for finding $\leq_i$-minimal repairs (see Section 4.4.2).

3. Currently, the performance of the QBF-solvers is considerably below that of the other solvers. Moreover, most of the QBF-solvers require that the formulas are represented in prenex CNF, and specified in Dimacs or Rintanen format. These requirements are usually space-demanding. In our context, the fact that many QBF-solvers (e.g., SEMPROP and QuBE-BJ) return only yes/no answers (according to the satisfiability of the input theory), is another problem, since it is impossible to construct repairs only by these answers. One needs to be able to extract the assignments to the outmost existentially quantified variables (as done, e.g., by DECIDE).

4. The solvers provide three different ways to express the minimality criteria. The minimality criterion can be expressed as a QBF formula, allowing to

[28] For the unsatisfiable QBF $\exists xy \forall uv((x \vee y) \wedge (u \vee v))$, the answer $x = 1$ and $y = 0$ is returned. The system developers were notified about this and the bug is being fixed.

find the minimal models without special solver support. The second way is provided by the formal semantics of the logic theory: the stable model semantics are the minimal models of a disjunctive logic program without negation as failure. Hence, DLV naturally supports minimality. The CLP(FD) and SAT approach has no language support for the minimality criteria. Minimality is supported by solver extensions.

4.4.5 Concluding Remarks

This work provides further evidence for the well-known fact that in many cases a proper representation of a given problem is a major step in finding robust solutions to it. In our case, a uniform method for encoding the restoration of database consistency by signed formulas allows us to use off-the-shelf solvers for efficiently computing the desired repairs.

The signed formulas technique has one main drawback: it is based on *propositional theories*. Therefore, the integrity constraints must be grounded. This grounding determines the possible repairs that can be found. For example, consider $\mathcal{DB} = (\{p(a)\}, \{\forall\ X.p(X) \rightarrow (\exists\ Y.q(X,Y)), \forall\ X.\neg q(X,a).\})$. The grounding of $\mathcal{IC}$ w.r.t. the active domain $\mathcal{DA} = \{a\}$ yields $\mathcal{IC}^{\mathcal{DA}} = \{p(a) \rightarrow q(a,a), \neg q(a,a)\}$. The only repair is $(\mathsf{Insert}, \mathsf{Retract}) = (\emptyset, \{p(a)\})$. However, intuitively it is expected that there is another repair in which an atom $q(a,b)$ is inserted, where b is a unknown constant (i.e. Skolem constant) that is different from a. This repair is found when the repairs are computed for a grounding w.r.t. an extended set of domain elements, e.g. $\mathcal{DA}_2 = \{a, b\}$. Unfortunately, there is no way to determine in advance the number of extra Skolem constants are needed. A simple, yet costly, approach to find all best repairs is by stepwise increasing the size of the grounding set $\mathcal{DA}$ with one new Skolem constant and computing for each grounding the best repairs. This costly procedure is not needed for the abductive approach since it is able to determine the need for a Skolem constant at run-time.

As shown in Corollary 4.2, the task of repairing a database is on the second level of the polynomial hierarchy, hence it is not tractable. However, despite the high computational complexity of the problem, the experimental results of Section 4.4.4 show that our method of repairing databases by signed theories is *practically appealing*.

In practice, most of the data in a large database is consistent. Also, an integrity constraint describes usually the consistency of a small number of relations. It should be possible to construct a clever division in small pieces of the database of which the consistency can be independently checked. In the experiments we performed, this division could be applied. For example, the database instances in the experiment 1 and 2 can be divided in subsets, in which each subset stores the knowledge about one course. Each subset can independently be checked for consistency. The global repair of the database instance is formed by the accumulation of all repairs of the subsets. The application of such a reduction strategy is probably able to produce problem instances for large databases that are small enough to be handled by systems such as zChaff and DLV.

4.5 Conclusions on data integration

The integration of databases is an important problem within information systems, which requires adequate and effective solutions. As shown throughout this chapter, the problem raises interest from both practitioners and theorists, making it an ideal field to apply our declarative problem solving methodology.

We have tackled two subproblems in this area: the integration of alphabets and the integration of databases instances. The first problem has been considered in the context of mediator-based systems. These build a virtual global database for a set of databases, called the sources. We have presented an ID-Logic framework for building mediator-based systems. Due to the expressiveness and modularity of ID-Logic, the framework yields a compact representation of a mediator based system, that can deal easily with the retraction and insertion of database sources. Compared to the existing approaches, the main advantage of our framework is the natural way it copes with the ontological differences between the alphabets. We also have explored the usefulness of abductive inference for query answering. It turned out that for a particular style of representing the ontological relations between the global alphabet and the source alphabets, namely the Global-as-View approach, abductive reasoning is applicable.

The second integration problem has been subject to two contributions. Both concern the problem of restoring the consistency of an inconsistent database instance w.r.t. a given set of integrity constraints. The first contribution models the problem in ID-Logic. We have designed an ID-Logic composer which describes the repaired database instance in terms of the original database instance and a repair. The composer is a meta-theory which is instantiated for each particular problem. In case we are looking for the repairs, the ID-Logic theory can be transformed into an Abductive Logic theory in which the abducibles represent the repair. The repairs can thus be computed by an abductive solver, in casu the $\mathcal{A}$system. Since there are many repairs possible, often not all relevant, the notion of preferred repairs is introduced. In order to compute the most preferred repairs, we have extended the $\mathcal{A}$system with an optimization module. We have shown that by this extension the $\mathcal{A}$system is able to find all most preferred repairs.

Our approach allows any first order formula as integrity constraint. To our knowledge, our approach is in this respect the most general one. This expressiveness implies that we cannot provide a general sound and complete solver for our ID-Logic composer. In particular, the abductive solver, the $\mathcal{A}$system that has been applied is complete in a very restricted way. However, we have shown that this suffices for integrity constraints that frequently occur in practice: namely key and foreign key constraints. Furthermore we have discussed the flexibility of our framework by presenting extensions that consider time stamps and the origin of the sources. We also sketched how this composer can be integrated in a mediator-based system.

Our last contribution is the design of a novel method to encode the repairs of a database instance as the models of a propositional theory, called a signed theory. We have proven that the $\leq$-preferred repairs correspond to $\leq$-minimal models of the signed theory. These models can be computed by a wide variety of solvers. We have experimented with several systems, including finite domain constraint solvers, satisfiability solvers, model generators and quantified boolean formula solvers. The experimental results show that when it is possible to reduce in an intelligent way the problem size (which may be possible), some of these systems have the computational efficiency to be used in practice.

For the future, the presented work should be combined in a database integration framework based on ID-Logic using the $\mathcal{A}$system and possibly other reasoning tools to perform the computational tasks. The here presented work has studied different aspects in isolation and from a rather theoretical point of view. It is important to bring these ideas in practice and evaluate their behavior on real problems, rather than on small artificial and well-controlled problems.

The work can be extended in many different ways. One direction is to generalize some of ideas that we have presented. For example, we have considered the ontological integration of two alphabets in the context of mediator-based systems. The need to express these relationships also occurs outside this specific problem context which may require generalizations of our approach. Another direction is the extension of the different approaches to deal with more complex and expressive databases. To name a few: databases with view definitions, temporal and spatial databases, etc. A last possible direction is to develop an approach for the integration of integrity constraints. There exists already frameworks that handle this problem, e.g. LUPS [10]. However it is of interest if we can model this issue in ID-Logic so that the complete database integration process can be handled in the same logical context.

Chapter 5

Conclusion

The last two decades, the majority of the research in the context of Declarative Problem Solving (in particular in non-monotonic reasoning) has focussed on how to represent domain knowledge of a problem as a logic theory in such a way that the formal semantics of this theory corresponds to the real world. This has resulted in a vast amount of formal semantics, and different logics and proof procedures. Until recently, very little attention was paid to the development of efficient solvers for those logics. Nowadays the focus has moved more towards the latter kind of research.

In this thesis, we have presented a Declarative Problem Solving method based on the knowledge representation language ID-Logic and the abductive solver the $\mathcal{A}$system [1]. In this method, the domain knowledge of a problem is modeled using ID-Logic. When the problem requires abductive reasoning, the $\mathcal{A}$system can be used (provided that some conditions are satisfied) to compute the answers by transforming the ID-Logic specification into an abductive normal logic program. Our contribution to this method is the implementation of the $\mathcal{A}$system, and the demonstration of the power of this method in the context of the integration of databases.

The $\mathcal{A}$system proof procedure is a mixture of three existing abductive proof procedures, i.e. SLDNFA [95, 96, 90], IFF [129] and ACLP [152, 161]. It has been shown sound and complete w.r.t. the three-valued completion semantics by extending the soundness and completeness results of SLDNFA. Our main contribution has been the implementation of the $\mathcal{A}$system. In short, we have presented the following design decisions in the implementation.

- We have designed an efficient search process by a *least-commitment strategy*

[1]The $\mathcal{A}$system is implemented on top of Sicstus Prolog [239] and can be downloaded from `http://www.cs.kuleuven.ac.be/~dtai/kt/`. It can be freely used for academic and research purposes.

with forward propagation rules. This leads to a drastic reduction in the number of backtrack points and makes the system more robust, e.g. the system's execution is less affected by syntactic changes.

- The basic technique that is used in the implementation of the search process is *reification*. Because this technique was only available for a limited number of finite domain constraint expressions, we have reimplemented the reification in order to handle more expressions. We have also extended the reification inference so that a reified expression can be removed from the constraint store.

- We have designed appropriate *data structures* for the storage of abduced atoms (Δ) and residual denials (Δ^*). At the cost of a larger space requirement, the data structures are able to return a significant smaller set of elements so that the number of inference steps is reduced.

- We have studied how the inference rules can be evaluated efficiently. In particular for the evaluation of denials, we have shown that *ground meta-interpretation* is a more appropriate choice than the classical non-ground meta-interpretation.

- We have implemented a new *(dis)equality constraint solver*, called the $\mathcal{E}$-solver. This solver can deal with conjunction and disjunction of equations and disequations. It also supports our extended reification. The $\mathcal{E}$-solver eliminates a large part of the inefficiency of the early SLDNFA prototypes. Moreover, by its reification reasoning, it forms an essential part of the $\mathcal{A}$system search process and hence it is responsible for an important part of the current computational efficiency of the $\mathcal{A}$system.

- We have studied the *integration of the finite domain solver*. We have mainly focused on the problems that arise due to the incompleteness of the finite domain solver and due to the automated generation of the constraint store. For each of these issues, we have presented some (partial) solutions.

All those elements make the $\mathcal{A}$system a state-of-the-art knowledge representation reasoning system. It is more robust than any abductive reasoning system before and it is modular and easily extendable as illustrated by the open function extension. We have validated the computational efficiency of the $\mathcal{A}$system experimentally using classical AI problems. These problems are useful to explain the interesting properties and the limitations of the system. Also, it eases us to position the $\mathcal{A}$system w.r.t. other related reasoning systems. In particular, the experiments show that the $\mathcal{A}$system has a comparable or better computational behavior than the Answer Set Programming DLV. DLV can be regarded as a nephew of the $\mathcal{A}$system: it has been designed for the same application area as the $\mathcal{A}$system, but it uses a completely different reasoning mechanism. DLV is a model generator for propositional theories, while the $\mathcal{A}$system is based on resolution for first order theories. Each approach has it advantages: DLV is a very robust system, dealing well with inductive definitions (e.g. transitive closure). The $\mathcal{A}$system is more scalable since it does not require to ground the theories to a propositional theory. Also the $\mathcal{A}$system's search strategy is adaptable by the user, so that the system's behavior can be more tailored to the problem. Comparing the $\mathcal{A}$system to the task specific solvers, we observe that the overhead of the $\mathcal{A}$system w.r.t. a pure Constraint Logic Programming solution is reasonable. For increasing problem sizes this is a constant factor. This indicates that it is worthwhile for a user first to model its problem in the high level language of ID-Logic/$\mathcal{A}$system. When the specification has been fixed, a more efficient constraint logic program can be designed. The results for the A.I. planning problems are encouraging, but the strips planners are still a magnitude more efficient than the $\mathcal{A}$system. One of the reasons for this large difference is the use of a domain dependent search heuristic, that the planner derives from the problem specification. Such guiding is totally absent in the $\mathcal{A}$system, hence it often occurs that the $\mathcal{A}$system makes bad choices.

These artificial problems are still far away from real world problems. These have no well-designed characteristics and tend to hit the limitations of any reasoning system at unexpected places. We have experimented with some real world problems, which are not included in the dissertation due to the several reasons:

- The *Tractebel problem* [238, 104, 265] is a scheduling problem which requires aggregates. Since we have omitted this extension from the thesis, this experiment has not been included. This experiment was successful [263]: we integrated support for simple aggregates in an ancestor of the $\mathcal{A}$system which reduced the setup times of the constraint store from about 24 hours [104] to several minutes [263]. The drawback of our aggregation support was the generation of more complex constraint stores. Our system needed, thus, more time to solve the constraint store.

- The *BTW problem* [41, 42] aimed at the (re)design of our departmental duty system, called BTW [224, 107]. BTW maintains the duties (e.g. teaching sessions, exercise sessions, webpages maintenance, ...) that have to be done in the department. This system is used to plan and organize these duties in order to obtain a fair distribution of the work load over all staff members. The work on this problem was a collaboration with the SOM research group and it has resulted in the definition of a transformation of the conceptual modeling language ERŒS [253, 243] to ID-Logic [42, 41, 123]. Computationally, the $\mathcal{A}$system has been used for a limited number of experiments to test the principles.

- We also have not been able to complete a study on abductive support for requirement specification of embedded systems. This study uses as case study domain a realistic *lighting system* for a computer science department. The goal is to find errors in the (partial) specification, by checking system invariants. These errors can be computed as abductive answers of a problem specification, consisting of the system description composed with a formula encoding that the invariant hold initially and not at a later time point. At the time we tackled this problem, the $\mathcal{A}$system was not equipped with the new equality solver $\mathcal{E}$ that has been described in the thesis. Due to this, some experiments failed; hence we hope that the new version is able to offer better results.

- The *DAF problem* [194] is a complex production planning problem. This experiment was done during the last phase of the dissertation in the context of a master thesis. For this reason it is not included. The $\mathcal{A}$system was able to solve only very simple problems, the more complex problems failed because of a bad goal ordering.

These experiences indicate that there is still a lot of work needed before the $\mathcal{A}$system is at a stage that it can handle real world problems *easily*. Some important items for future work are:

- Like SLD and SLDNF, the $\mathcal{A}$system may loop when there are recursive definitions in the program. This behavior limits the application of the $\mathcal{A}$system. We expect that the integration of tabling in the proof procedure will solve this problem. This integration is, however, a complex problem since the existing tabling techniques cannot be used due to the existence of the free variables in the abductive proof procedure.

- As the planning experiments show, the used search heuristic has an enormous effect on the time to find a solution. We see two potential ways for improvements: (1) Because a good search heuristic is hard to find for a user, it is necessary that the $\mathcal{A}$system is able to derive a heuristic from the problem specification. This idea is applied by strips planners with great success. For the $\mathcal{A}$system the construction of such heuristic is more difficult, but in our opinion the $\mathcal{A}$system must provide such a support in order to be more robust. (2) Another way to improve the search is to improve the inconsistency detection. As shown by the logistics planning problem, if the $\mathcal{A}$system does not recognize early that it has entered an inconsistent part of the search tree, it needs a long time before it backtracks to the right branch.

- The computational efficiency of the $\mathcal{A}$system depends much on the integrated constraint solvers. This provides several ways for improvement of the system.

  One direction is the improvement of the integration of the finite domain solver in the $\mathcal{A}$system. This constraint solver is a main factor for the current computational results in the $\mathcal{A}$system, but the experiments indicate that the complex constraint stores that are generated are limiting better results.

Another direction is the extension of the $\mathcal{A}$system with new constraint solvers. Extending the $\mathcal{A}$system is fairly simple, as illustrated with the open functions. These constraint solvers may optimize the inference for certain problems, or offer new concepts so that the $\mathcal{A}$system becomes more expressive.

In the latter category, we like to mention aggregate expressions. Aggregates are expressions such as the summation, the average or the cardinality of a set. They are among the few second order expressions that occur often in problem specifications. For this reason, a DPS method should provide support for them. For semantical foundations of aggregate expressions we refer to the work of Pelov (et al.) [101, 212]. As part of this work ID-Logic has been extended with aggregates. We did a preliminary study on the integration of aggregates in an abductive solver [263]. This implementation is restricted to aggregates which depend on sets that are described as a subset of an open function. For scheduling problems this suffices, nevertheless it is a severe restriction that limits the application. One problem domain that requires more expressive set descriptions is *planning with resources.*

Those problems combine planning with scheduling. Because we focused our attention on the improvement of the basic inference in the $\mathcal{A}$system, the work on this aggregate solver has not been continued.

The second contribution of the thesis is the application of ID-Logic and the $\mathcal{A}$system to the problem of the integration of databases. We have been able to provide solutions to two subproblems in the integration of (relational) databases.

Firstly, we have presented an ID-Logic mediator-based system which constructs a uniform global view for a set of independent database sources. The main issue is to formulate the ontological relations between different views (alphabets) on the same domain. For this problem, we have to express the ontological relations between the global alphabet and the source alphabets. Since in general there is no perfect match possible, it means that there is incomplete knowledge about the global (intended) database from a single source point of view. However, other sources may complement this incompleteness. In our approach, a mediator-based system is materialized as the ID-Logic theory which is the composition of ID-Logic theories that formulate the ontological relations between the global alphabet and one source alphabet, and ID-Logic theories that encode the information stored in the sources. Compared to existing approaches, our approach uses the most expressive logic of all. By this, our approach is able to incorporate the different existing formalizations (GAV, LAV, GLAV, BAV). Most work on this problem has been at the level of the problem formalization. W.r.t. query answering of the global database (the reason why the mediator-based system is developed), we have illustrated that for some scenarios the expected answers to a query are based on explanations, e.g. when a source is temporary unavailable. This is exactly the major advantage of abductive reasoning.

Secondly, the coherent integration of database instances is tackled. This problem aims at restoring the consistency of a database instance, that is obtained as the union of the database instances of the sources, w.r.t. a set of integrity constraints. The consistency restoration is done by updating the database instance with a repair: some atoms of the database instance will be retracted, other (new) atoms will be inserted. Our approach uses a ID-Logic meta-theory, called the composer, which expresses the repaired database instance in terms of the repair and the original database instance. For this problem, we have shown that the $\mathcal{A}$system is able to compute the repairs. Since there are many repairs which are often not sensible, the goal is refined to finding the repairs according to a preference criterion. We have shown that an extension of the $\mathcal{A}$system with an optimizer is able to compute the preferred repairs. Moreover, we have proven that the $\mathcal{A}$system is able to find all optimal repairs for an important class of integrity constraints: key and foreign key constraints. This solution has several advantages. The approach is flexible: it is relatively easy to extend the composer to take into account other information such as temporal and source origin, or to integrate it with the mediator-based systems from above. Also, our approach makes no assumptions on the domain (i.e. there is no domain closure). In case of insertions, it allows us to provide very compact representations of the repairs which are computed as non-ground abductive solutions by the $\mathcal{A}$system.

For the coherent integration problem we have presented an alternative, yet elegant, method, in which the repairs are encoded as the models of a (propositional) theory, called the signed theory. This last work does not use ID-Logic, nor the $\mathcal{A}$system. The idea of the method is to represent the switch of an atom's status when the database is repaired. The signed theory is obtained by replacing in the grounded integrity constraints (grounded w.r.t. a finite domain) each atom with its status of the negation of its status. We have shown that the (minimal) models of the signed theory correspond with a (preferred) repair of the database instance w.r.t. the integrity constraints. Furthermore, we have experimented with several computational paradigms. The results of this experiment indicate that when the size of the database is not too large, some of the systems can be used as tool for repairing databases in practice.

Our work has focused on relational databases. The integration problem also occurs in other contexts: in multi-agent systems where agents want to share information; in hierarchical planning where different abstraction levels are connected; in advanced information systems, e.g. Enterprise Resource Management systems, where information is stored, manipulated and analyzed, ... Because of the generality of the applied methods (ID-Logic and the $\mathcal{A}$system), we hope that the obtained insights in the context of relational databases can be applied in these new contexts.

Another direction for future work, it the implementation of an integration system for realistic experimentations. In this system, one can study the computational feasibility of the basic approach for relational databases. This will require, for instance, to extend the $\mathcal{A}$system with access to a database system, the design of methods to reduce the amount of information that must be processed, the implementation of a procedure to compute the entailment of a query, ...

To conclude, the pair of ID-Logic and the $\mathcal{A}$system forms a very powerful Declarative Problem Solving method. Based on our experience, we expect that this DPS-method will be able to provide solutions in some problem areas which are poorly tackled by classical software. We have demonstrated this for the information integration domain, but it holds also for domains such as billing, tariff calculations and administrative applications that deal with legal matters (e.g. VDAB-software). In these problems domains, the domain knowledge is complex and often subject to many changes during the lifetime of the software. The expressive and flexible environment offered by our DPS-method is exactly what is needed to handle these problems.

However, DPS is not a silver bullet for all problems. Therefore, it must be integrated in common programming environments. This is still an open problem. One starting point may be the continuation of the work on ERŒS [42, 41, 123] where (1) an object-oriented conceptual modeling language ERŒS is related with ID-Logic and (2) the idea is sketched of generating prototypes in which the $\mathcal{A}$system is the reasoning tool. Building a relation with object-oriented modeling is important since the majority of software engineers is trained to understand this view on the world. A good integration will be beneficial for both (now mostly separated) software development areas.

Bibliography

[1] A.Ayari and D.Basin. QUBOS: Deciding quantified Boolean logic using propositional satisfiability solvers. In M.D.Aagaard and J.W.O'Leary, editors, *Proceedings of the fourth International Conference on Formal Methods in Computer-Aided Design(FMCAD'02)*, volume 2517 of *LNCS*, pages 187–201. Springer, 2002.

[2] Slim Abdennadher and Henning Christiansen. An experimental clp platform for integrity constraints and abduction. In *Proceedings of FQAS2000*, Advances in Soft Computing series, pages 141–152. Springer, 2000.

[3] Serge Abiteboul and Oliver M. Duschka. Complexity of answering queries using materialized views. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS-98 . Seattle, WA, June 1998.*, volume 1, pages 591–596. Morgan Kaufmann Publishers, Inc, 1998.

[4] ACLP. Obtainable via `http://www.cs.ucy.ac.cy/aclp/`.

[5] AIPS-00 Planning Competition. The Fifth International Conference on Artificial Intelligence Planning and Scheduling Systems, 2000.

[6] Journal of Artificial Intelligence, Volume 20 , 2003. Special Issue on the 3rd International Planning Competition.

[7] H. Ait-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990.

[8] Carlos E. Alchourrón, Peter. Gärdenfors, and David Makinson. On the logic of theory change: Partial meet contraction and revision function. *Journal of Symbolic Logic*, 50(2):510–530, 1985.

[9] J. J. Alferes, J. A. Leite, L. M. Pereira, and P. Quaresma. Planning as abductive updating. In *Proceedings of the Symposium on AI Planning and Intelligent Agents (AISB'00)*, pages 1–8, 2000.

[10] J. J. Alferes, L. M. Pereira, H. Przymusinska, and T.C. Przymusinski. LUPS – a language for updating logic programs. *Artificial Intelligence*, 138(1-2):87–116, 2002.

[11] Jose Julio Alferes, Luis Moniz Pereira, and Terrance Swift. Well-founded abduction via tabled dual programs. In *International Conference on Logic Programming*, pages 426–440, 1999.

[12] ALIAS, an Abductive LogIc Agent System. Obtainable via `http://www.lia.deis.unibo.it/Research/ALIAS/`.

[13] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[14] Krzysztof R. Apt and Roland N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19/20:9–71, 1994.

[15] Marcelo Arenas, Leopoldo Bertossi, and Michael Kifer. Applications of annotated predicate calculus to querying inconsistent databases. In J. Lloyd, editor, *Proceedings of the 1st International Conference on Computational Logic (CL'2000)*, volume 1861 of *LNAI*, pages 926–941. Springer, 2000.

[16] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of 18th ACM Symposium on Principles of Database Systems (PODS'99)*, pages 68–79, 1999.

[17] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Answer sets for consistent query answers in inconsistent databases. *Theory and Practice of Logic Programming*, 3(4–5):393–424, 2003.

[18] Ofer Arieli. Four-valued logics for reasoning with uncertainty in prioritized data. In B.Bouchon-Meunier, R.R.Yager, and L.Zadeh, editors, *Information, Uncertainty, Fusion*, pages 263–309. Kluwer, 1999.

[19] Ofer Arieli and Arnon Avron. Reasoning with logical bilattices. *Journal of Logic, Language, and Information*, 5(1):25–63, 1996.

[20] Ofer Arieli and Arnon Avron. A model theoretic approach to recover consistent data from inconsistent knowledge-bases. *Journal of Automated Reasoning*, 22(3):263–309, 1999.

[21] Ofer Arieli and Marc Denecker. Modeling paraconsistent reasoning by classical logic. In T.Eiter and K.D.Schewe, editors, *Proceedings of the second Symposium on Foundations of Information and Knowledge Systems (FoIKS'02)*, volume 2284 of *LNCS*, pages 1–14. Springer, 2002.

[22] Ofer Arieli and Marc Denecker. Reducing preferential paraconsistent reasoning to classical entailment. *Journal of Logic and Computation*, 13(4):557–580, 2003.

[23] Ofer Arieli, Marc Denecker, Bert Van Nuffelen, and Maurice Bruynooghe. Repairing inconsistent databases: a model-theoretic approach and abductive reasoning. In *Proceedings ICLP'02 Workshop on Paraconsistent Computational Logic*, pages 51–65. ICLP, 2002.

[24] Ofer Arieli, Marc Denecker, Bert Van Nuffelen, and Maurice Bruynooghe. Coherent integration of databases by abductive logic programming. *Journal of Artificial Intelligence Research*, 21:245–286, 2004.

[25] Ofer Arieli, Marc Denecker, Bert Van Nuffelen, and Maurice Bruynooghe. Database repair by signed formulae. In *Foundations of Information and Knowledge Systems*, volume 2942 of *Lecture Notes in Computer Science*, pages 14–30. European Association for Theoretical Computer Science (EATCS), the Association for Symbolic Logic (ASL), and the Austrian Computer Society (OCG), Springer, 2004.

[26] Ofer Arieli, Bert Van Nuffelen, Marc Denecker, and Maurice Bruynooghe. Coherent composition of distributed knowledge-bases through abduction. In *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, December 3-7, 2001, Proceedings*, volume 2250 of *Lecture Notes in Computer Science*, pages 620–635. Springer, 2001.

[27] Arnon Avron. Classical gentzen-type methods in propositional many-valued logics. In M.Fitting and E.Orlowska, editors, *Theory and Applications in Multiple-Valued Logics*, pages 131–151. Springer, 2002.

[28] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook. Theory, Implementation and Applications.* Cambridge University Press, 2002.

[29] Franz Baader and Wayne Snyder. Unification Theory. In Alan Robinsion and Andrei Voronkov, editors, *Handbook of Automated Reasoning.* Elsevier Science Publishers, 1999.

[30] J. Balsa, V. Dahl, and J.G. Pereira Lopes. Datalog Grammars for Abductive Syntactic Error Diagnosis and Repair. In *Proceedings of the Natural Language Understanding and Logic Programming Workshop*, 1995.

[31] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2003.

[32] Chitta Baral, Sarit Kraus, and Jack Minker. Combining multiple knowledge bases. *IEEE Trans. on Knowledge and Data Engineering*, 3(2):208–220, 1991.

[33] Chitta Baral, Sarit Kraus, Jack Minker, and V.S. Subrahamanian. Combining multiple knowledge bases consisting of first order theories. *Computational Intelligence*, 8:45–71, 1992.

[34] Chitta Baral, Jack Minker, and S. Kraus. Combining multiple knowledge bases. *IEEE Transactions on Knowledge and Data Engineering*, 3(2):208–221, June 1991.

[35] Roberto Barbuti and Maurizio Martelli. Recognizing Non-floundering Logic Programs and Goals. *International Journal of Foundations of Computer Science*, 1:151–163, 1990.

[36] Roman Bartak. Constructive negation in clp(h). Technical Report 6, Department of Theoretical Computer Science, Charles University, Prague, 1998.

[37] Roman Barták. Dynamic global constraints: A first view. In *Proceedings of ERCIM Workshop on Constraints*, Prague, June 2001. (revised version of CP-AI-OR2001).

[38] Roman Barták. Dynamic global constraints in constraint logic programming. In *ITI Series*, volume 18, Prague, 2001.

[39] Roman Barták. Dynamic global constraints in backtracking based environments. *Annals of Operations Research*, 118:101–119, 2003.

[40] Catriel Beeri and Peter Buneman, editors. *Tableau Techniques for Querying Information Sources through Global Schemas*, volume 1540 of *Lecture Notes in Computer Science*, Jerusalem, Israel, January 1999. Springer.

[41] Pieter Bekaert and Bert Van Nuffelen. On the transformation of object oriented conceptual models to logical theories: from EROOS to ID-logic. 21th International Conference on Conceptual Modeling, ER2002, Tampere, Finland, October 7-11, 2002, oct 2002.

[42] Pieter Bekaert, Bert Van Nuffelen, Maurice Bruynooghe, David Gilis, and Marc Denecker. On the transformation of object-oriented conceptual models to logical theories. In *Conceptual Modeling - ER 2002, 21st International Conference on Conceptual Modeling, Proceedings*, volume 2503 of *LNCS*, pages 152–166. Springer Verlag, 2002.

[43] Domenico Beneventano, Sonia Bergamaschi, Silvana Castano, Valeria De Antonellis, Alfio Ferrara, Francesco Guerra, Federica Mandreoli, Giorgio Carlo Ornetti, and Maurizio Vincini. Semantic integration and query

optimization of heterogeneous data sources. In Jean-Michel Bruel and Zohra Bellahsene, editors, *Proceedings of the Workshop on Advances in Object-Oriented Information Systems, (OOIS 2002)*, volume 2426 of *Lecture Notes in Computer Science*, pages 154–165. Springer, 2002.

[44] Salem Benferhat, C. Cayrol, Didier Dubois, J. Lang, and Henri Prade. Inconsistency management and prioritized syntax-based entailment. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 640–645, 1993.

[45] Salem Benferhat, Didier Dubois, and Henri Prade. How to infer from inconsistent beliefs without revising? In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 1449–1455, 1995.

[46] Leopoldo Bertossi, Jan Chomicki, Alvaro Cortés, and Claudio Gutiérrez. Consistent answers from integrated data sources. In A.Andreasen et al., editor, *Flexible Query Answering Systems, 5th International Conference, (FQAS 2002)*, volume 1, pages 71–85, Copenhagen, Denmark, October 27-29, 2002. Springer.

[47] Leopoldo E. Bertossi and C. Schwind. Analytic tableau and database repairs: Foundations. In *Proceedings of the 2nd International Symposium on Foundations of Information and Knowledge Systems (FoIKS'02)*, volume 2284 of *LNCS*, pages 32–48. Springer, 2002.

[48] P. Besnard and T. Schaub. Signed systems for paraconsistent reasoning. *Journal of Automated Reasoning*, 20(1):191–213, 1998.

[49] P. Besnard, T. Schaub, H. Tompits, and S. Woltran. Paraconsistent reasoning via quantified Boolean formulas, part I: Axiomatizing signed systems. In S.Flesca et al., editor, *Proceedings of the eight European Conference on Logics in Artificial Intelligence(JELIA'02)*, volume 2424 of *LNAI*, pages 320–331. Springer, 2002.

[50] P. Besnard, T. Schaub, H. Tompits, and S. Woltran. Paraconsistent reasoning via quantified Boolean formulas, part II: Circumscribing inconsistent theories. In T.D.Nielsen and N.L.Zhang, editors, *Proceedings of the seventh European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU'03)*, volume 2711 of *LNAI*, pages 528–539. Springer, 2003.

[51] Blay Bonet and Hector Geffner. HSP2: Decription HSP Planner in AIPS-2000 Competition. *AI Magazine*, 22(3):77–80, 2001.

[52] Blay Bonet and Hector Geffner. Planning as Heuristic Search. *Artificial Intelligence*, 129(1–2), 2001. Special issue on Heuristic Search.

[53] Antony F. Bowers and Patricia M. Hill. Representing Gödel Object Programs in Gödel, 1995. in documentation of the Gödel system.

[54] Ronald J. Brachman, Richard E. Fikes, and Hector J. Levesque. KRYPTON: Integrating Terminology and Assertion. In *Proceedings of the National Conference on Artificial Intelligence (AAAI'83)*, pages 31–35. William Kaufman, 1983.

[55] Ronald J. Brachman, V. Gilbert, and Hector Levesque. An essential hybrid reasoning system: Knowledge and symbol level accounts of KRYPTON. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 532–539, 1985.

[56] Ronald J. Brachman and Hector Levesque, editors. *Readings in Knowledge Representation*. Morgan Kaufmann, 1985.

[57] Ronald. J. Brachman and Hector J. Levesque. Competence in Knowledge Representation. In *Proc. of the National Conference on Artificial Intelligence*, pages 189–192, 1982.

[58] Ronald J. Brachman and Hector J. Levesque. The Tractability of Subsumption in Frame-Based Description Languages. In Ronald J. Brachman, editor, *Proceedings of the National Conference on Artificial Intelligence*, Austin, TX, August 1984. AAAI Press.

[59] Ivan Bratko. *Prolog, Programming for Artificial Intelligence.* Addison-Wesley, third edition, 2001.

[60] Stéphane Bressan, Cheng Hian Goh, Kofi Fynn, Marta Jakobisiak, Karim Hussein, Henry B. Kon, Thomas Lee, Stuart E. Madnick, Tito Pena, Jessica Qu, Annick W. Shum, and Michael Siegel. The Context Interchange mediator prototype. In *Proceedings of ACM SIGMOD/PODS Joint Conference*, pages 525–527, 1997.

[61] Maurice Bruynooghe, Nikolay Pelov, and Marc Denecker. Towards a more declarative language for solving finite domain problems. In *Proceedings of the 1999 ERCIM/COMPULOG Workshop on Constraints*, pages 1–14. University of Cyprus, 1999.

[62] François Bry. Query answering in information systems with integrity constraints. In Sushil Jajodia, William List, Graeme W. McGregor, and Leon Strous, editors, *Proceedings of the International Conference on Integrity and Control in Information Systems (IICIS'97)*, volume 109 of *IFIP Conference*

Proceedings, pages 113–130, Zurich, Switzerland, December 1997. Chapman Hall.

[63] Andrea Calí, Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Data integration under integrity constraints. In *The Fourteenth International Conference on Advanced Information Systems Engineering*, volume 2348 of *Lecture Notes in Computer Science*. Springer, 2002.

[64] Andrea Calí, Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. On the expressive power of data integration systems. In Stefano Spaccapietra, Salvatore March, and Yahiko Kambayashi, editors, *21st International Conference on Conceptual Modeling*, volume 2503 of *Lecture Notes in Computer Science*, pages 338–350. Springer, 2002.

[65] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Ontology of integration and integration of ontologies. In Carole A. Goble, Deborah L. McGuinness, Ralf Möller, and Peter F. Patel-Schneider, editors, *Working Notes of the 2001 International Description Logics Workshop (DL-2001), Stanford, CA, USA, August 1-3, 2001*, volume 49 of *CEUR Workshop Proceedings*, 2001.

[66] Mats Carlsson, Greger Ottosson, and Björn Carlson. An Open-Ended Finite Domain Constraint Solver. In H. Glaser, P. Hartel, and H. Kucken, editors, *Programming Languages: Implementations, Logics, and Programming*, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206, Southampton, September 1997. Springer-Verlag.

[67] Walter A. Carnielli and Jo ao Marcos. Tableau systems for logics of formal inconsistency. In *Proceedings of the International Conference on Artificial Intelligence (IC-AI'01)*, volume 2, pages 848–852. CSREA Press, 2001.

[68] Walter A. Carnielli and Jo ao Marcos. A taxomony of c-systems. In Walter A. Carnielli, Marcelo E. Coniglio, and Itala M. Loffredo D'Ottaviano, editors, *in Paraconsistency - the Logical Way to the Inconsistent. Lecture Notes in Pure and Applied Mathematics*, volume 228, pages 1–94, 2002.

[69] David Chan. Constructive negation based on the completed database. In R. A. Kowalski and K. A. Bowen, editors, *Proc. International Conference on Logic Programming (ICLP)*, pages 111–115. ICLP, The MIT Press, 1988.

[70] David Chan. An extension of constructive negation and its application in coroutining. In E. Lusk and R. Overbeek, editors, *Proc. North-American Conference on Logic Programming*, pages 477–493. The MIT Press, 1989.

[71] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence*. Addison-Wesley, 1985.

[72] Jan Chomicki and Leopoldo Bertossi. Query answering in inconsistent databases. In Jan Chomicki, Gunter Saake, and Ron van der Meyden, editors, *Logics for Emerging Applications of Databases*, pages 43–83. Springer, 2003.

[73] Jan Chomicki and Gunter Saake, editors. *Logics for Databases and Information Systems*. Kluwer Academic Publishers, 1998.

[74] Jan Chomicki, Gunter Saake, and Ron van der Meyden, editors. *Logics for Emerging Applications of Databases*. Springer, 2003.

[75] A. Ciampolini, E. Lamma, P. Mello, and C. Stefanelli. A coordination protocol for abductive logic agents. In *IEEE ICIPS97*, Bejing, October 1997. IEEE Computer Society.

[76] Anna Ciampolini, Evelina Lamma, Paola Mello, Francesca Toni, and Paolo Torroni. Co-operation and competition in ALIAS: a logic framework for agents that negotiate. *Annals of Mathematics and Artificial Intelligence*, 37(1–2):65–91, 2003. Computational Logic in Multi-Agent Systems.

[77] The Ciao Prolog System. Obtainable via `http://clip.dia.fi.upm.es/Software/Ciao/`.

[78] Keith L. Clark. Negation as failure. In Hervé Gallaire and Jack Minker, editors, *Proceedings of the Logic and Databases symposium (Logic and Databases '78)*, pages 293–322. Plenum Press, 1978.

[79] Phillipe Codognet and Christian Codognet. Abduction and concurrent logic languages. In Antony G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, Amsterdam, The Netherlands, August 1994. John Wiley and Sons.

[80] Phillipe Codognet and Vijay Saraswat. Abduction in concurrent constraint languages. In Robert Kowalski, editor, *Proceedings of the first Compulog network meeting on Logic Programming and Artificial Intelligence*, London, U.K., 1992.

[81] Alain Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 85–99, Tokyo, Japan, November 1984. OHMSHA Ltd. Tokyo and North-Holland.

[82] Alain Colmerauer and Philippe Roussel. The birth of prolog. In Thomas J. Bergin and Richard G. Gibson, editors, *History of Programming Languages*. ACM Press/Addison-Wesley, 1996.

[83] Hubert Comon and Pierre Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7:371–425, 1989.

[84] L. Console, L. Portinale, and D. Theseider Dupré. Using compiles knowledge tot guide and focus abductive diagnosis. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):690–706, 1996.

[85] Luca Console, Daniele Theseider Dupré, and Pietro Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 5:661–690, 1991.

[86] C.Wrathall. Complete sets and the polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):23–33, 1976.

[87] Mukesh Dalal. Investigations into a theory of knowledge base revision. In *Proceedings of the National Conference on Artificial Intelligence (AAAI'88)*, pages 475–479. AAAI Press, 1988.

[88] S. de Amo, Walter Carnielli, and Jo ao Marcos. A logical framework for integrating inconsistent information in multiple databases. In *Proceedings of the second International Symposion on Foundations of Information and Knowledge Systems (FoIKS'02)*, volume 2284 of *LNCS*, pages 67–84, 2002.

[89] H. Decker. Historical and computational aspects of paraconsistency in view of the logic foundations of databases. In *Proceedings of the Semantics in Databases*, volume 2585 of *LNCS*, pages 63–81. Springer, 2003.

[90] Marc Denecker. *Knowledge representation and reasoning in incomplete logic programming*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, September 1993. Pages: 292.

[91] Marc Denecker. The Well-founded Semantics Is the Principle of Inductive Definition. In J. Dix, L. Fariñas del Cerro, and U. Furbach, editors, *Logics in Artificial Intelligence*, pages 1–16, Schloss Daghstull, October 12-15 1998. Springer-Verlag, Lecture notes in Artificial Intelligence 1489.

[92] Marc Denecker. Extending Classical Logic with Inductive Definitions. In C. Baral and M. Truszczynski, editors, *Proceedings of NMR'2000*. 8th Intl. Workshop on Non-Monotonic Reasoning, 2000.

[93] Marc Denecker. Extending classical logic with inductive definitions. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luis Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic - CL 2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 703–717. Springer, 2000.

[94] Marc Denecker, Maurice Bruynooghe, and Victor Marek. Logic programming revisited: logic programs as inductive definitions. *ACM Transactions on Computational Logic*, 2(4):623–654, October 2001.

[95] Marc Denecker and Danny De Schreye. SLDNFA; an abductive procedure for normal abductive programs. In K.R. Apt, editor, *Proc. of the International Joint Conference and Symposium on Logic Programming*, pages 686–700. MIT Press, 1992.

[96] Marc Denecker and Danny De Schreye. SLDNFA: an abductive procedure for normal abductive programs. *Journal of Logic Programming*, 34(2):111–167, February 1998.

[97] Marc Denecker and Antonis C. Kakas. *Abductive Logic Programming*, volume 44. Special issue of Journal of Logic Programming, 2000.

[98] Marc Denecker and Antonis C. Kakas. Abduction in logic programming. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, volume 2407 of *Lecture Notes in Computer Science*, pages 402–436. Springer Verlag, 2002.

[99] Marc Denecker, Victor Marek, and M. Truszczynski. Approximating operators, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. In *NFS-workshop on Logic-based Artificial Intelligence*, pages 1–26, 1999.

[100] Marc Denecker, Lode Missiaen, and Maurice Bruynooghe. Temporal reasoning with abductive event calculus. In *Proceedings of the 10th European Conference on Artificial Intelligence, ECAI92*, pages 384–388. John Wiley & Sons, 1992.

[101] Marc Denecker, Nikolay Pelov, and Maurice Bruynooghe. Ultimate Well-founded and stable semantics for logic programs with aggregates. In *Logic Programming, 17th International Conference, ICLP 2001, Paphos, Cyprus, Proceedings*, volume 2237 of *Lecture Notes in Computer Science*, pages 212–226. Springer, 2001.

[102] Marc Denecker and Kristof Van Belleghem. Temporal reasoning with abduction. In *Handbook of Time in AI*. 2002.

[103] Marc Denecker and Bert Van Nuffelen. Experiments for integration CLP and abduction. In *Proceedings of the 1999 ERCIM/COMPULOG workshop on Constraints*, pages 1–15, 1999.

[104] Marc Denecker, Henk Vandecasteele, Danny De Schreye, Gino Seghers, and Tom Baeyens. Scheduling by "Abductive Execution" of a Classical Logic Specification. Compulog Network Meeting on Constraint Programming, Linz, October 27-28 1997.

[105] The DLV project, A Disjunctive Datalog System (and more). Obtainable via `http://www.dlvsystem.com`.

[106] A. Doan, P. Domingos, and A. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *Proceedings of the ACM SIGMOD*, Santa Barbara, California, USA, 2001.

[107] Lenka Dulière. Ondersteuning voor de planning van lessen en oefenzittingen. Master's thesis, Department of Computing Science, K.U.Leuven, 1999.

[108] P.M. Dung. Negations as Hypotheses: an abductive foundation for Logic Programming. In *Proceedings of the International Conference on Logic Programming*, 1991.

[109] O. Duschka, M. Genesereth, and A. Levy. Recursive query plans for data integration. In *Journal of Logic Programming*, volume 1, pages 49–73, 2000.

[110] The ECL^{i}PS^e Constraint Logic Programming System. Obtainable via `http://www.icparc.ic.ac.uk/eclipse/`.

[111] E.Giunchiglia, M.Narizzano, and A.Tacchella. QuBE: A system for deciding quantified Boolean formulas satisfiability. In R.Gor, A.Leitsch, and T.Nipkow, editors, *Proceedings of the first International Conference on Automated Reasoning(IJCAR'01)*, volume 2083 of *LNCS*, pages 364–369. Springer, 2001.

[112] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR system DLV: Progress report, comparisons and benchmarks. In *Proceedings of the sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417. Morgan Kaufmann Publishers, 1998.

[113] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Planning under incomplete knowledge. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luis Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Proceedings of the First International Conference on Computational Logic (CL'2000)*, volume 1861 of *Lecture Notes in AI (LNAI)*, pages 807–821. Springer Verlag, 2000.

[114] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Data Systems.* Addison Wesley. Second edition.

[115] A. Eremin and Mark G. Wallace. Hybrid Benders Decomposition Algorithms in Constraint Logic Programming. In Toby Walsh, editor, *Constraint Logic Programming, in Proceedings of the Seventh International Conference on Principles and Practices of Constraint Programming - CP '01*, volume 2239 of *LNCS*, pages 1–15. Springer-Verlag, 2001.

[116] K. Eshghi. Abductive Planning with Event Calculus. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the International Conference on Logic Programming*. MIT press, 1988.

[117] K. Eshghi and Robert A. Kowalski. Abduction compared with negation as failure. In *Proceedings of the International Conference on Logic Programming*. MIT press, 1989.

[118] Hartshorn et al, editor. *Collected papers of Charles Sander Pierce*, volume 2. Harvard University Press, 1931-1958.

[119] Melvin Fitting. Negation as refutation. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS'89)*, pages 63–70, Asilomar Conference Center, Pacific Grove, California, 5-8 June, 1989. IEEE Computer Society Press.

[120] Melvin Fitting. Bilattices and the semantics of logic programming. *Journal of Logic and Computation*, 11(2):91–116, 1991.

[121] Peter A. Flach and Antonis C. Kakas, editors. *Abduction and Induction: Essays on their relation and integration*. Kluwer Academic Publishers, April 2000.

[122] Peter A. Flach and Antonis C. Kakas. On the relation between abduction and inductive learning. In Dov M. Gabbay and Rudolf Kruse, editors, *Handbook of defeasible reasoning and uncertainty management systems, Vol. 4: Abductive reasoning and learning*, pages 1–33. Kluwer Academic Publishers, October 2000.

[123] Bruno De Fraine. Uitwerking op metaniveau van een transformatie naar logische theorieën. Master's thesis, Department of Computing Science, K.U.Leuven, 2003.

[124] Enrico Franconi, Antonio Laureti Palma, Nicola Leone, Simona Perri, and Francesco Scarcellor. Census data repair: A challenging application of disjunctive logic programming. In R. Nieuwenhuis and A. Voronokov, editors, *Proceedings of the Eighth International Conference on Logic Programming, Artificial Intelligence and Reasoning (LPAR'01)*, volume 2250 of *LNCS*, pages 561–578. Springer, 2001.

[125] Marc Friedman, Alon Y. Levy, and Todd D. Millstein. Navigational plans for data integration. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on on Innovative Applications of Artificial Intelligence*, pages 67–73, Orlando, Florida, USA, July 1999. AAAI Press / The MIT Press.

[126] S. Friedman, A. Levy, and T. Millstein. Navigational plans for data integration. In *Proceedings of the Sixteenth National Conference on AI*, pages 67–73. AAAI Press, 1999.

[127] Thom W. Frühwirth. Constraint Handling Rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 90–107. Springer Verlag, March 1995.

[128] Thom W. Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998. Special Issue on Constraint Logic Programming.

[129] Tzee Ho Fung and Robert A. Kowalski. The IFF Proof Procedure for Abductive Logic Programming. *Journal of Logic Programming*, 33(2):151–165, November 1997.

[130] H. Garcia-Monia. The TSIMMIS approach to mediation: data models and languages. In *Proceedings of the second Workshop on Next-Generation Information Technologies and Systems*, Naharia, Israel, June, 1995.

[131] P. Gärdenfors and H. Rott. Belief revision. In D.M.Gabbay, J.Hogger, and J.A.Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 4, pages 35–132. Oxford University Press, 1995.

[132] Marco Gavanelli, Evelina Lamma, Paola Mello, Michela Milano, and Paolo Torroni. Interpreting abduction in CLP. In Francesco Buccafurri, editor, *APPIA-GULP-PRODE Joint Conference on Declarative Programming*, pages 25–35, Reggio Calabria, Italy, September 3 - 5 2003. Università Mediterranea di Reggio Calabria.

[133] Allen Van Gelder. The Alternating Fixpoint of Logic Programs with Negation. *Journal of Computer and System Sciences*, 47:185–221, 1993.

[134] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[135] M. Gertz and U.W. Lipeck. An extensible framework for repairing constraint violations. In *Proceedings of the International Conference on Integrity and Control in Information Systems (IICIS'97)*, pages 89–111. Kluwer Academic Publishers, 1997.

[136] Cheng Hian Goh, Stéphane Bressan, Thomas Lee, Stuart E. Madnick, and Michael Siegel. A procedure for the context mediation of queries to disparate sources. In *Proceedings of the International Logic Programming Symposium (ILPS-97)*, pages 213–227, 1997.

[137] Gianluigi Greco, Sergio Greco, and Ester Zumpano. A logic programming approach to the integration, repairing and querying of inconsistent databases. In Philippe Codognet, editor, *Proceedings of the International Conference on Logic Programming(ICLP'01)*, volume 2237 of *LNCS*, 2001.

[138] Sergio Greco and Ester Zumpano. Querying inconsistent databases. In M.Parigot and A.Voronokov, editors, *Proceedings of the International Conference on Logic Programming and Automated Reasoning (LPAR'2000)*, volume 1955 of *LNAI*, pages 308–325. Springer, 2000.

[139] Volker Haarslev and Ralf Möller. Description of the racer system and its applications. In Carole A. Goble, Deborah L. McGuinness, Ralf Möller, and Peter F. Patel-Schneider, editors, *Working Notes of the 2001 International Description Logics Workshop (DL-2001)*, volume 49 of *CEUR Workshop Proceedings*, 2001.

[140] Hisashi Hayashi. Abductive Constraint Logic Programming with Constructive Negation. In *the Working Notes of the Third Workshop on Non-Monotonic Reasoning, Action, and Change of the International Joint Conference on Artificial Intelligence*, August 1999.

[141] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming. MIT Press, 1989.

[142] Patricia M. Hill and Joel Gallagher. Meta-programming in logic programming. In Dov M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming. Logic Programming*, volume 5, pages 421–497. Oxford Science Publications, 1998.

[143] Patricia M. Hill and John W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[144] H.Kleine-Büning, M.Karpinski, and A.Fögel. Resolution for quantified Boolean formulas. *Journal of Information and Computation*, 177(1):12–18, 1995.

[145] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[146] Christian Holzbaur. *Specification of Constraint Based Inference Mechanism through Extended Unification*. PhD thesis, Departement of Medical Cybernetics and AI, University of Vienna, 1990.

[147] K. Inoue and C. Sakama. Abductive framework for nonmonotonic theory change. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 204–210, 1995.

[148] Interfakultair seminarie over abductie, K.U.Leuven, 2000.

[149] Joxan Jaffar and Michael Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19–20:503–581, 1994.

[150] J.R. Josephson and S.G. Josephson, editors. *Abductive Inference:Computation, Philosophy, Technology.* Cambridge University Press, 1994.

[151] J.T.Rintanen. Improvements of the evaluation of quantified Boolean formulae. In *Proceedings of the sixteenth International Joint Conference on Artificial Intelligence(IJCAI'99)*, pages 1192–1197. Morgan Kaufmann Publishers, 1999.

[152] A. C. Kakas and A. Michael. Integrating abductive and constraint logic programming. In Leon Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 399–416, Cambridge, June 13–18 1995. MIT Press.

[153] Antonis Kakas, Robert Kowalski, and Francesca Toni. The role of abduction in logic programming. In Dov M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of logic in Artificial Intelligence and Logic Programming* , volume 5, pages 235–324. Oxford University Press, 1998.

[154] Antonis C. Kakas and Paolo Mancarella. Abductive Logic Programming. In W. Marek, A. Nerode, D. Pedreschi, and V.S. Subrahmanian, editors, *Proceedings of the International Workshop on Non-Monotonic Reasoning and Logic Programming (NACLP'90)*, pages 49–62, Austin,Texas, 1990.

[155] Antonis C. Kakas and Paolo Mancarella. Database updates through abduction. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB'90)*, pages 650–661, 1990.

[156] Antonis C. Kakas and Paolo Mancarella. Generalized Stable Models: a Semantics for Abduction. In *Proceedings of the 9th European Conference on AI (ECAI'90)*, pages 385–391. John Wiley and sons, 1990.

[157] Antonis C. Kakas and Paolo Mancarella. On the Relation between Truth Maintenance and Abduction. In T. Fukumura, editor, *Proceedings of the first Pacific Rim International Conference on Artificial Intelligence (PRICAI'90)*, pages 438–443, Japan, 1990. Ohmsha Ltd.

[158] Antonis C. Kakas and A. Michael. Timetabling in an Integrated Abductive and Constraint Logic Programming Framework. In *Proceedings of the First International Conference on the Practice and Theory of Automated Timetabling*, pages 167–177, 1995.

[159] Antonis C. Kakas and A. Michael. Air-Crew Scheduling through Abduction. In *Proceedings of the IEA*, pages 600–612, 1999.

[160] Antonis C. Kakas and A. Michael. An abductive-based scheduler for air-crew assignment. *Journal of applied Artificial Intelligence*, 15:333–360, 2001.

[161] Antonis C. Kakas, A. Michael, and Costas Mourlas. ACLP: Abductive constraint logic programming. *Journal of Logic Programming*, 44(1-3):129–177, July,August 2000.

[162] Antonis C. Kakas and Costas Mourlas. ACLP: Flexible solutions to complex problems. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programing and Nonmonotonic Reasoning*, volume 1265 of *LNAI*, pages 387–398, Berlin, July 28–31 1997. Springer.

[163] C. Kakas, Antonis, Bert Van Nuffelen, and Marc Denecker. A-system : Problem solving through abduction. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, volume 1, pages 591–596. IJCAI, inc and AAAI, Morgan Kaufmann Publishers, Inc, 2001.

[164] C. Kakas, Antonis, Bert Van Nuffelen, and Marc Denecker. A-system : Problem solving through abduction. 13th Belgium-Dutch Conference on Artificial Intelligence, BNAIC'01, Amsterdam, The Netherlands, October 25-26, 2001, 32993, 2001.

[165] M. Kifer and E.L. Lozinskii. A logic for reasoning with inconsistency. *Journal of Automated Reasoning*, 9(2):179–215, 1992.

[166] T. Kirk. The Information Manifold. In *Proceedings of the AAAI Spring Symposium on Information Gathering*, 1995.

[167] R.A. Kowalski and M.J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986. Reprinted in Foundations of Knowledge Base Management Systems, pp. 23–53, Springer Verlag, 1989.

[168] Robert A. Kowalski. Predicate logic as a programming language. In *Proceedings of the Information Processing (IFIP'74)*, pages 569–574. North-Holland, 1974.

[169] Robert A. Kowalski and Fariba Sadri. From Logic Programming towards Multi-agent Systems. *Annals of Mathemathics and Artificial Intelligence*, 25:391–419, 1999.

[170] Robert A. Kowalski and M. Sergot. A logic-based calculus of events. *New generation Computing*, 4(4):319–340, 1986.

[171] Robert A. Kowalski, Francesca Toni, and Gerhard Wetzel. Executing Suspended Logic Programs. *Fundamentae Informaticae*, 34:1–22, 1998.

[172] Robert A. Kowalski, Gerhard Wetzel, and Francesca Toni. A unifying framework for ALP, LCP and SQO. Technical report, Departement of Computing, Imperial College, London, U.K., 1996.

[173] Jean-Louis Lassez, Michael J. Maher, and Kim Marriott. Unification Revisited. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, 1988.

[174] Fritz Lehmann, editor. *Semantic Networks in Artificial Intelligence.* Pergamon Press, Oxford, UK, 1992.

[175] Maurizio Lenzerini. Data integration needs reasoning. In T. Eiter, W. Faber, and M. Truszczyński, editors, *Proceedings of the International Conference on Logic Programming and Non-Monotonic Reasoning(LPNMR'01)*, volume 2173 of *LNAI*, pages 54–61. Springe, 2001.

[176] Maurizio Lenzerini. Data integration needs reasoning. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczynski, editors, *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001*, volume 2173 of *Lecture Notes in Computer Science*, pages 54–61. Springer, September 2001.

[177] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS'02)*, pages 233–246, 2002.

[178] Maurizio Lenzerini. Data integration: A theoretical perspective. In Lucian Popa, editor, *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 233–246. ACM, jun 2002.

[179] A. Y. Levy, A. Rajaraman, and Ordille J.J. Querying heterogeneous information sources using source descriptions. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'96)*, volume 1, pages 251–262, 1996.

[180] Alon Y. Levy. Logic-based techniques in data integration. In Jack Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14–16, 1999*, College Park, Maryland, 1999. Computer Science Department, University of Maryland.

[181] A.Y. Levy. Query answering algorithms for information agents. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*

and Eighth Innovative Applications of Artificial Intelligence Conference (AAAI'96 and IAAI'96), volume 1, pages 40–47. AAAI Press / The MIT Press, 1996.

[182] P. Liberatore and M. Schaerf. BReLS: a system for the integration of knowledge bases. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *Proceedings of the Seventh International Conference on Principles of Knowledge, Representation and Reasoning (KR'2000)*, pages 145–152, Breckenridge, Colorado, USA, April 11-15, 2000. Morgan Kaufmann Publishers.

[183] Fangzhen Lin and Jia-Huai You. Abduction in logic programming: A new definition and an abductive procedure based on rewriting. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, volume 1, pages 655–666. IJCAI, inc and AAAI, Morgan Kaufmann Publishers, Inc, 2001.

[184] Fangzhen Lin and Jia-Huai You. Abduction in logic programming: A new definition and an abductive procedure based on rewriting. *Artificial Intelligence*, 140(1–2):175–205, 2002.

[185] J. Lin. A semantics for reasoning consistently in the presence of inconsistency. *Artificial Intelligence*, 86(1):75–95, 1996.

[186] J. Lin and A.O. Mendelzon. Merging databases under constraints. *International Journal of Cooperative Information Systems*, 7(1):55–76, 1998.

[187] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[188] John W. Lloyd and R.W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

[189] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1997.

[190] Carlos A. Mareco and Leopoldo E. Bertossi. Specification and implementation of temporal databases in a bitemporal event calculus. In Peter P. Chen, editor, *Proceedings of the Advances in Conceptual Modeling: ER '99 Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling*, volume 1727 of *Lecture Notes in Computer Science*, pages 74–85, Paris, France, November 15-18, 1999. Springer.

[191] Kim Marriot and Peter Stuckey. *Programming with Constraints: An Introduction.* MIT Press, 1998.

[192] Alberto Martelli and Ugo Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[193] David Martens. Een vergelijkende studie van oplossingsmethodes voor educationele timetabling problemen. Master's thesis, Department of Computing Science, K.U.Leuven, 2003.

[194] Birgit Matthé. Een geavanceerd planningssysteem voor vrachtwagenassenproductie. Master's thesis, Department of Computing Science, K.U.Leuven, 2004.

[195] M.Cadoli, M.Schaerf, A.Giovanardi, and M.Giovanardi. An Algorithm to evaluate quantified Boolean formulae and its experimental evaluation. *Journal of Automated Reasoning*, 28(2):101–142, 2002.

[196] P. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *International Conference on Data Engineering (ICDE' 03)*, pages 227–238. IEEE Computer Society, 2003.

[197] John McCarthy. Applications of circumscription to formalizing commonsense knowledge. *Artificial Intelligence*, 28:89–116, 1986.

[198] John McCarthy. Elaboration tolerance. In *Proceedings of the workshop on the Formalization of Commonsense Reasoning (FCS'98)*, pages 98–183, 1998.

[199] B. Messing. Combining knowledge with many-valued logics. *Data and Knowledge Engineering*, 23:297–315, 1997.

[200] A. Michael. *Abductive Logic Programming and its application to scheduling*. Phd, University of London, United Kingdom, dec 1998.

[201] Marvin Minsky. A framework for representing knowledge. In J Haugeland, editor, *Mind Design*. MIT Press, 1981.

[202] Lode Missiaen, Maurice Bruynooghe, and Marc Denecker. CHICA, an abductive planning system based on event calculus. *Journal of Logic and Computation*, 5(5):579–602, sep 1995.

[203] M.Moskewicz, C.Madigan, Y.Zhao, L.Zhang, and S.Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the thirty-nineth Design Automation Conference*, 2001.

[204] Juan José Moreno and Susana Muñoz Hernández. On the practical use of negation in a Prolog compiler. In *Proceedings of Joint Conference on Declarative Programming APPIA-GULP-PRODE'98*, pages 457–469, July 1998.

[205] Juan José Moreno and Susana Muñoz Hernández. How to incorporate negation in a Prolog compiler. In *The 2nd International Workshop, PADL'2000. Practical Aspects of Declarative Languages*, volume 1753 of *Lecture Notes in Computer Science*, pages 124–140. Springer, January 2000.

[206] Susana Muñoz Hernández and Juan José Moreno. Constructive negation for Prolog: A real implementation. In J.J. Moreno and J. Mariño, editors, *Joint Conference on Declarative Programming: APPIA-GULP-PRODE 200*, pages 39–52, september,16-20 2002.

[207] Susana Muñoz Hernández, Juan José Moreno, and Manuel V. Hermenegildo. Efficient negation using abstract interpretation. In *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, December 3-7, 2001, Proceedings*, volume 2250 of *Lecture Notes in Computer Science*, pages 485–494, 2001.

[208] Illka Niemelä and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the Fourth International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 1265 of *Lecture Notes in Artificial Intelligence*, pages 420–429, Dagstuhl, Germany, July 1997.

[209] A. Olivé. Integrity checking in deductive databases. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, September 3-6, 1991, Barcelona, Catalonia, Spain, 1991. Morgan Kaufmann.

[210] Y. Papakonstantinou. Object exchange across heterogeneous information sources. In *Proceedings of the International Conference on Data Engineering*, 1995.

[211] Y. Papakonstantinou. *Query processing in heterogeneous information sources.* Phd, Stanford University, 1996.

[212] Nikolay Pelov. *Semantics of Logic Programs with aggregates.* Phd, Department of Computer Science, K.U.Leuven, Leuven, Belgium, april 2004. 154 pages.

[213] Nikolay Pelov, Emmanuel De Mot, and Marc Denecker. Logic programming approaches for representing and solving constraint satisfaction problems : a comparison. In *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 2000, Proceedings*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 225–239. Springer, 2000.

[214] Nikolay Pelov, Emmanuel De Mot, and Marc Denecker. Logic programming approaches for representing and solving constraint satisfaction problems : a comparison. In Parigot M. and Voronkov A., editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 2000, Proceedings*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 225–239. Springer, 2000.

[215] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Nonmonotonic reasoning with well founded semantics. In K. Furukawa, editor, *8th International Conference on Logic Programming*, pages 475–489. MIT Press, 1991.

[216] D. Poole and R.G. Goebel. Theorist: a logical reasoning system for default and diagnosis. In Cercone and McCalla, editors, *The Knowledge Fronteer: Essays in the Representation of Knowledge*, Lecture Note in Computer Science, pages 331–352. Springer, 1987.

[217] R. Pottinger and A Levy. A scalable algorithm for answering queries using views. In *VLDB-2000*, volume 1, pages 484–495. Morgan Kaufmann Publishing Co., 2000.

[218] G. Priest. Minimally inconsistent lp. *Studia Logica*, 50:321–331, 1991.

[219] Stathis Psillos. Ampliatative Reasoning: Induction or Abduction. In *ECAI96, Workshop on Abductive and Inductive Reasoning*, 1996. paresis example: causation and entailment difference.

[220] M. Ross Quillian. Word concepts: A theory and simulation of some basic capabilities. *Behavioral Science*, 12:410–430, 1967.

[221] Ray Reiter. Equality and domain closure in first-order databases. *Journal of the ACM*, 27:235–249, 1980.

[222] Ray Reiter. On specifying database updates. *Journal of Logic Programming*, 25(1):53–91, 1995.

[223] Raymond Reiter. On Closed World Data Bases. In Hervé Gallaire and Jack Minker, editors, *Symposium on Logic and Data Bases*, pages 55–76. Plemum Press, 1978.

[224] Stephan Renkens. Een prologpakket voor de verificatie van taakverdelingen. Master's thesis, Department of Computing Science, K.U.Leuven, 1986.

[225] Peter Z. Revesz. On the semantics of theory change: Arbitration between old and new information. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'93)*, pages 71–82, May 25-28, Washington, DC, 1993. ACM Press.

[226] R.Feldmann, B.Monien, and S.Schamberger. A distributed algorithm to evaluate quantified Boolean formulae. In *Proceedings of the National Conference on Artificial Intelligence (AAAI'00)*, pages 285–290. AAAI Press, 2000.

[227] D. Riera and N. Yorke-Smith. An improved hybrid model for the generic hoist scheduling problem. *Annals of Operations Research*, 115:173–191, September 2002.

[228] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[229] Fariba Sadri and Francesca Toni. Abduction with negation as failure for active databases and agents. In *Proceedings AI*IA99, Sixth Congress of the Italian Association for Artificial Intelligence*, pages 353–362. Pitagora Editrice Bologna, 1999.

[230] Fariba Sadri and Francesca Toni. Abduction with Negation as Failure for Active and Reactive Rules. In E. Lamma and P. Mello, editors, *Proceedings AI*IA99, Sixth Congress of the Italian Association for Artificial Intelligence*, LNAI 1792, pages 49–60. Springer Verlag, 2000.

[231] Fariba Sadri, Francesca Toni, and P. Torroni. Dialogues for negotiation: agent varieties and dialogue sequences. In *(ATAL'01)*, Seattle, Washington, USA, 2001. special track on negotiation.

[232] Fariba Sadri, Francesca Toni, and Ioannis Xanthakos. A logic-agent based system for semantic integration. In *Proceedings of 17th International Data and Information for the Coming Knowledge Millennium Conference (CODATA 2000)*, 2000.

[233] Chiaki Sakama and Katsumi Inoue. Updating extended logic programs through abduction. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proceedings of the Fifth International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'99)*, pages 147–161, 1999.

[234] Vijay Saraswat and Pascal Van Hentenryck. *Principles and Practice of Constraint Programming.* MIT Press, May 1995.

[235] K. Satoh and N. Iwayama. A query evaluation method for Abductive Logic Programming. In Kryzstof R. Apt, editor, *Proceedings of the International Joint Conference and Symposium on Logic Programming*, 1992.

[236] Ken Satoh and Noboru Iwayama. Computing abduction by using the TMS. In *Proceedings of the International Conference on Logic Programming*, pages 505 – 518, 1991.

[237] N. Schelkens, Frank Van Eynde, and Sven Verdoolaege. The semantics of temporal adjuncts. In *Computational Linguistics in the Netherlands 1999 Selected Papers from the Tenth CLIN Meeting*, pages 169–179, 2000.

[238] Gino Seghers and Tom Baeyens. Solving a combinatorial maintenance problem for electrical power plants, developed in OLP-FOL and CLP. Master's thesis, Department of Computing Science, K.U.Leuven, 1996.

[239] Sicstus Prolog. Obtainable via `http://www.sics.se/isl/sicstus.html`.

[240] sModels. Obtainable via `http://www.tcs.hut.fi/Software/smodels/`.

[241] S.M. Sripada. Efficient implementation of the event calculus for temporal database applications. In Leon Sterling, editor, *Proceedings of the International Conference on Logic Programming (ICLP'95)*, pages 99–113, June 13-16, 1995, Tokyo, Japan, 1995.

[242] Eric Steegmans, Pieter Bekaert, Frank Devos, Jan Dockx, Bart Swennen, and Stefan Van Baelen. Object-Oriented Analysis. Course Notes, Department of Computer Science, K.U.Leuven, 2001.

[243] Eric Steegmans, Johan Lewi, M. D'Haese, Jan Dockx, D. Jehoul, Bart Swennen, Stefan Van Baelen, and P. Van Hirtum. EROOS Reference Manual. Version 1.0. Report CW 208, Department of Computer Science, K.U.Leuven, Leuven, Belgium, January 1995.

[244] Peter J. Stuckey. Negation and Constraint Logic Programming. *Information and Computation*, 118(1):12–33, 1995.

[245] V.S. Subrahmanian. Mechanical proof procedures for many valued lattice-based logic programming. *Journal of Non-Classical Logic*, 7:7–41, 1990.

[246] V.S. Subrahmanian. Amalgamating knowledge-bases. *ACM Transactions on Database Systems*, 19(2):291–331, 1994.

[247] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[248] Francesca Toni. A theorem-proving approach to job-shop scheduling. Technical report, Departement of Computing, Imperial College, London, U.K., 1994.

[249] Paolo Torroni. *Reasoning and interaction in logic-based multi-agent systems.* Phd, Department of Electronics, Computer Science, and Systems, University of Bologna, Italy, February 2002. DEIS Technical Report DEIS-LIA-02-005, LIA Series No 58.

[250] The Standord-IBM Manager of Multiple Information Sources (TSIMMIS). `http://www-db.stanford.edu/tsimmis`.

[251] U.Egly, T.Eiter, H.Tompits, and S.Woltran. Solving advanced reasoning tasks using quantified Boolean formulas. In *Proceedings of the National Conference on Artificial Intelligence(AAAI'00)*, pages 417–422. AAAI Press, 2000.

[252] Jeffrey D. Ullman. Information integration using logical views. *Theoretical Computer Science*, 239(2):189–210, may 2000.

[253] Stefan Van Baelen, Johan Lewi, Eric Steegmans, and H. Van Riel. EROOS : An Entity-Relationship based Object-Oriented Specification Method. In B. Magnusson G. Heeg and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems TOOLS 7*, pages 103–117. Prentice-Hall, Hertsfordshire, UK, 1992.

[254] Kristof Van Belleghem. *Open logic programming as a knowledge representation language for dynamic problem domains.* Phd, Department of Computer Science, K.U.Leuven, Leuven, Belgium, dec 1997.

[255] Kristof Van Belleghem, Marc Denecker, and Danny De Schreye. A strong correspondence between description logics and open logic programming. In L. Naish, editor, *Logic Programming, Proceedings of the Fourteenth International Conference on Logic Programming*, pages 346–360. MIT Press, 1997.

[256] Nico Van de Weghe. Development of a Conceptual Data Model for Digital Spatio-Temporal Geographical Information, with Application to Several Themes and GIS. In *Conceptual Modeling - ER 2002, 21st International Conference on Conceptual Modeling, Proceedings*, volume 2503 of *LNCS*, page 16. Springer Verlag, 2002.

[257] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

[258] Bert Van Nuffelen. SLDNFA-system. In *Proceedings of the 8th International Workshop on Nonmonotonic Reasoning NMR'2000*, pages 1–6, 2000. Special session "System demonstrations and descriptions" organised by J. Dix. Paper available in CoRR under number cs.AI/0003027.

[259] Bert Van Nuffelen. Problem solving using declarative specifications. Faculty of Engineering, PhD Symposium, Leuven, Belgium, December 11, 2002, 40446, 2002.

[260] Bert Van Nuffelen. Reasoning with preferences in ID-logic. In *Proceedings of BNAIC'02 - Belgian-Dutch Conference on Artificial Intelligence*, pages 323–330, oct 2002. Presented as poster.

[261] Bert Van Nuffelen. Knowledge engineering vs software engineering. Industry-Ready Innovative Research: 1st Flanders Engineering PhD Symposium, Brussels, Belgium, December 11, 2003, Agoria, KUL, RUG, VUB, 2003.

[262] Bert Van Nuffelen, Alvaro Cortés-Calabuig, Marc Denecker, Ofer Arieli, and Maurice Bruynooghe. Data Integration Using ID-logic. In *Proceedings of The 16th International Conference on Advanced Information Systems Engineering, Riga, Latvia*, volume 3084 of *Lecture Notes in Computer Science*, pages 67–81. Riga technical University, 2004.

[263] Bert Van Nuffelen and Marc Denecker. Problem solving in ID-logic with aggregates. In *Proceedings of the 8th International Workshop on Nonmonotonic Reasoning NMR'2000*, pages 1–10, 2000. Special session "Abduction" organised by M. Denecker, T. Kakas and F. Toni. Paper available in CoRR under number cs.AI/0003030.

[264] Bert Van Nuffelen and C. Kakas, Antonis. $\mathcal{A}$system : Programming with abduction. In *Logic Programming and Nonmonotonic Reasoning, LPNMR 2001, Proceedings*, volume 2173 of *LNAI*, pages 393–396. Springer Verlag, 2001.

[265] Henk Vandecasteele. *Constraint logic programming: applications and implementation*. Phd, Department of Computer Science, K.U.Leuven, Leuven, Belgium, may 1999.

[266] Henk Vandecasteele, Bart Demoen, and Gerda Janssens. A finite domain CLP solver on top of Mercury. In *New Trends in Constraints : Joint ERCIM/Compulog Workhsop, Paphos, Cyprus, October 1999, Selected Papers*, volume 1865 of *Lecture Notes in Artificial Intelligence*, pages 256–273. ERCIM/Compulog, Springer, 2000.

[267] Henk Vandecasteele, Bart Demoen, and Joachim Van Der Auwera. The use of Mercury for the implementation of a finite domain solver. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 299–312. Nova Science Publishers, 1999.

[268] Sofie Verbaeten. Termination analysis for abductive general logic programs. In *Logic Programming, Proceedings of the 1999 International Conference on Logic Programming*, pages 365–379. MIT Press, 1999.

[269] Sofie Verbaeten and Annalisa Bossi. Composing complete and partial knowledge. *Journal of Functional and Logic Programming*, 2000(6):1–25, mar 2000. Special Issue on Logical Formalisms for Program Composition, Guest Editors : A. Brogi and P.M. Hill.

[270] Sofie Verbaeten, Marc Denecker, and Danny De Schreye. Compositionality of normal open logic programs. *Journal of Logic Programming*, 41(3):151–183, March 2000.

[271] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI, 1983.

[272] Gerhard Wetzel. *Abductive and Constraint Logic Programming*. Phd, Imperial College, 1997.

[273] Gerhard Wetzel, Robert A. Kowalski, and Francesca Toni. PROCALOG — Programming with Constraints and Abducibles. In M. Maher, editor, *Joint International Conference and Symposium on Logic Programming*, page 535. MIT Press, 1996.

[274] M. Winslett. Reasoning about action using a possible models approach. In *Proceedings of the National Conference on Artificial Intelligence (AAAI'98)*, pages 89–93. AAAI press, 1988.

[275] Uwe Wirth. Homepage des Arbeitskreises für Abduktionsforschung. `http://www.rz.uni-frankfurt.de/~wirth/`.

[276] William Woods. What's in a link: Foundations for semantic networks. In D.G. Bobrow and A.M. Collins, editors, *Representation and Understanding: studies in cognitive science*. Academic Press, 1975. reprinted in [56].

[277] Ioannis Xanthakos. *Semantic integration of information by abduction*. Phd, University of London, United Kingdom, 2003.

Appendix A

A.1 The Event Calculus [170]

The Event Calculus [170] is a temporal theory that formalizes the law of persistence for properties (fluents). There exists many variants and extensions of the Event Calculus. The version that we usually use, is presented here.

A fluent P holds at a time T ($holds\_at(P, T)$) if either it was initially true and it has not been terminated between the initial timepoint and T ($clipped(0, P, T)$), or there has happened an action A at time E, before T, that initiated the fluent P ($initiates(A, P)$) and that is not clipped afterwards. A fluent is clipped between time E and time T if there has happened an action A at time C, between E and T, that terminates the fluent P ($terminates(A, P)$).

$$holds\_at(P, T) \leftarrow initially(P) \land \neg clipped(0, P, T).$$
$$holds\_at(P, T) \leftarrow initiates(A, P) \land act(A, E) \land E < T \land \neg clipped(E, P, T).$$

$$clipped(E, P, T) \leftarrow terminates(A, P) \land act(A, C) \land E \leq C \land C < T.$$

In a specific planning domain, the predicates $initiates(\cdot, \cdot)$ and $terminates(\cdot, \cdot)$ are defined, to encode the effects of the actions. The $act(\cdot, \cdot)$ will be an open predicate for a planning problem. Before an action is allowed to happen the preconditions must be satisfied. This is expressed by the assertion

$$\forall\ A, T.act(A, T) \rightarrow preconditions(A, T).$$

where $preconditions(A, T)$ defines the preconditions for an action A. Sometimes it is required that actions must be ordered in a strict sequential order. This is expressed by the assertion

$$\forall\ A_1, T_1, A_2, T_2.act(A_1, T_1) \land act(A_2, T_2) \land A_1 \neq A_2 \rightarrow T_1 \neq T_2.$$

A.2 An example $\mathcal{A}$system-specification

This is the full specification of the blocks world planning problem.

A.2.1 The pretty printed version

$$on(X,Y,T) \leftarrow init\_on(X,Y) \wedge \neg init\_moved(X,Y,T).$$
$$on(X,Y,T) \leftarrow move(X,Y,E) \wedge E < T \wedge \neg moved(X,Y,E,T).$$

$$moved(X,Y,E,T) \leftarrow move(X,Z,C) \wedge Z \neq Y \wedge E \leq C \wedge C < T.$$

$$init\_moved(X,Y,T) \leftarrow moved(X,Y,0,T).$$
$$\forall\ X,Y,E. \leftarrow move(X,Y,E) \wedge \neg succeeds\_move(X,Y,E).$$

$$succeeds\_move(X,Y,E) \leftarrow ablock(X) \wedge location(Y) \wedge time(E) \wedge$$
$$X \neq Y \wedge clear\_block(X,E) \wedge clear\_location(Y,E).$$

$$clear\_location(0,\_E).$$
$$clear\_location(Y,E) \leftarrow Y \neq 0 \wedge \neg something\_on(Y,E).$$

$$clear\_block(X,E) \leftarrow \neg something\_on(X,E).$$

$$something\_on(X,E) \leftarrow on(Y,X,E).$$

$$\forall\ X,Y,E. \leftarrow move(X1,Y1,T1) \wedge move(X2,Y2,T2) \wedge Y1 \neq Y2 \wedge T1 = T2.$$
$$\forall\ X,Y,E. \leftarrow move(X1,Y1,T1) \wedge move(X2,Y2,T2) \wedge X1 \neq X2 \wedge T1 = T2.$$

A.2.2 The $\mathcal{A}$system version

```
%----------------------------------------
% Declarations
%----------------------------------------

abducible(move(_,_,_)).

defined(init_on(_,_)).
defined(moved(_,_,_,_)).
defined(init_moved(_,_,_)).
defined(between(_,_,_)).
defined(location(_)).
defined(ablock(_)).
defined(succeeds_move(_,_,_)).
defined(something_on(_,_)).
```

```
defined(time(_)).
defined(clear_block(_,_)).
defined(clear_location(_,_)).
defined(on(_,_,_)).

%-----------------------------------------
% The Blocks World Event Calculus Encoding
%-----------------------------------------

% general rules
on(X,Y,T):- init_on(X,Y), not(init_moved(X,Y,T)).
on(X,Y,T):- move(X,Y,E), clp(E#<T),
            not(moved(X,Y,E,T)).

init_moved(X,Y,T):- moved(X,Y,0,T).

moved(X,Y,E,T)   :- move(X,Z,C), clp(Z#\=Y), between(C,E,T).

between(C,E,T)   :- clp(E#=<C), clp(C#<T).

% the precondition
ic:- move(X,Y,E), not(succeeds_move(X,Y,E)).

succeeds_move(X,Y,E):-
   ablock(X),
   location(Y),
   time(E),
   clp(X#\=Y),
   clear_block(X,E),
   clear_location(Y,E).

clear_location(Y,E):-clp(Y#=0).
clear_location(Y,E):-clp(Y#\=0),not(something_on(Y,E)).

clear_block(X,E) :- not(something_on(X,E)).

something_on(X,E):- on(Y,X,E).

% sequential ordering
ic:- move(X1,Y1,T1),move(X2,Y2,T2),clp(Y1#\=Y2),clp(T1#=T2).
ic:- move(X1,Y1,T1),move(X2,Y2,T2),clp(X1#\=X2),clp(T1#=T2).

%-----------------------------------------
```

```
% the problem instance
%-----------------------------------------

% load the problem instance.
loadfile('probBLOCKS-5-0.pl').
```

The `loadfile` declaration concatenates the file in its argument to the specification. This facilitates experimentations.

```
%-----------------------------------------
%  problem BLOCKS-5-0
%-----------------------------------------

% 5 blocks
ablock(X):- clp(X in 1..5).
% location = blocks + table (0)
location(X):- clp(X in 0..5).
% time from 1 to 8
time(T):-clp(T in 1..8).

% the initial position of the blocks
% element(X,L,Y) is a finite domein expression
init_on(X,Y):-clp(element(X,[3,1,0,2,0],Y)).

/*
 An less efficient variant is

init_on(5,0).
init_on(3,0).
init_on(4,2).
init_on(2,1).
init_on(1,3).
*/


% the goal position of the blocks
defined(query(T)).

query(T):-time(T),
   on(3,2,T),
   on(2,1,T),
   on(1,5,T),
   on(5,4,T).
```

A.3 Reification

In Section 3.7 we have discussed at a high level the extended reification reasoning. This appendix section presents the implementation of this reification for the finite domain solver. It has been implemented as a new (global) constraint using the extension interface of the solver. (In the $\mathcal{E}$-solver, it has been integrated in the structure of the solver.)

We show the finite domain extension for the disjunction, for the conjunction it is analogous. The procedure below encodes the reification of a single constraint. The `fd_global/3` predicate calls a user defined global constraint. Here, the constraint is called `constraint_entailed/2`. The constraint will be checked by the constraint solver when the domain of any involved variable is changed. That parameter is set by the `wakeup/3` predicate.

```
constraint_entailed(Constraint,Boolean):- % Constraint <=> Boolean
   term_variables(Constraint,Vars),
   wakeup(domval,[Boolean|Vars],WakeUp),
   Boolean in 0..2,
   fd_global(constraint_entailed(Vars,Boolean),s(Constraint),WakeUp).
```

The actual reasoning performed by the new global constraint `constraint_entailed/2` must be encoded as the extension of the hook predicate `dispatch_global` of the finite domain solver. When the constraint is evaluated, the associated Boolean is first checked. If it is ground, the reification is finished, and depending on its value, it just removes the constraint from the store, or has to ensure that it posts the constraint or its negation. When the associated Boolean is non-ground, the constraint is checked for entailment. If it is (dis)entailed, the global constraint can be removed, otherwise the global constraint is kept in the store.

```
clpfd:dispatch_global(constraint_entailed(Vars,Bool),
                      s(Constraint), s(Constraint), Actions):-
   (ground(Bool)
      -> (Bool = 2
            -> Actions = [exit]
         ;Bool = 1
            -> Actions = [exit,call(Constraint)]
         ;Bool = 0,
               Actions = [exit,call(user:clp_negation(Constraint))]
         )
      ;
         user:test_constraint(Constraint,Result),
         (Result = suc
            -> (ground(Constraint)
```

```
                -> Actions = [exit,Bool=1]
                ;  Actions = []
                )
            ;  Actions = [exit,Bool=0]
            )
   ).
```

When a disjunction is reified, each disjunct is reified first individually, using above reification. The associated Booleans of the disjuncts (below denoted by `BoolVars`) are then related with the global boolean (`Bool`) according to the above truth table of the disjunction. To support the evaluation of the constraint, the sum of associated Booleans is computed (`Sum`). Since the domain of `Sum` changes when the domain of an associated Boolean changes, it suffices to evaluate the global constraint `c_or_reify/2` when `Sum` and `Bool` update their domain.

```
or_reify(BoolVars,Bool):-
   or_reify_domains(BoolVars,Bool),
   sum(BoolVars,#=,Sum),
   fd_global(c_or_reify(BoolVars,Bool),s(Sum),[dom(Sum),dom(Bool)]).

or_reify_domains([],_).
or_reify_domains([B|Bs],Bool):-
   or_reify_domain(Bool,B),
   or_reify_domains(Bs,Bool).

or_reify_domain(Boolean,B):-
   relation(Boolean,[0-{0},1-{0,1,2},2-{0,2}],B).
```

The constraint `c_or_reify/2` maintains the consistency of the reified Boolean disjunction. When the global Boolean `Bool` becomes ground, the global constraint is removed from the store. Depending on the value it has, the other variables are grounded or a new constraint is invoked. A ground sum `Sum` which is equal to 0, infers that `Bool` is 0.

```
clpfd:dispatch_global(c_or_reify(Vars,Bool),s(Sum),s(Sum),Actions):-
   (intermediate_check(yes) -> Actions = [] ;
   (ground(Bool)
      -> (Bool = 2
            -> Actions = [exit,call(user:ground_all(Vars))]
         ;Bool = 1
            -> Actions = [exit,call(user:disjunction(Vars,Sum))]
         ;Bool = 0
            -> Actions = [exit,call(user:ground_all(Vars))]
         )
```

```
   ; ground(Sum)
      -> (Sum = 0
           -> Actions = [exit,Bool=0]
           ;  Actions = [exit,Bool=1]
           )
   ; Actions = []
   )
   ).

ground_all([]).
ground_all([D|Ds]):-
   fd_max(D,M),   % find the maximal value M of D
   D=M,           % assign the maximal value to D
   ground_all(Ds).
```

The final part ensures that a disjunction of Booleans is true. If the sum of the variables is larger than 1, then the constraint is satisfied and is removed from the store. When the maximal value of `Sum` is 2, just one undetermined disjunct is left. This is assigned true, by assigning `Sum` the value 1. The last way to satisfy the constraint is when one of the disjuncts is satisfied. Then all other reified disjuncts can be removed, since their value is not important.

```
disjunction(BoolVars,Sum):-
   fd_global(disjunction(BoolVars,Sum),s,[dom(Sum)]).

clpfd:dispatch_global(disjunction(Vars,Sum),s,s,Actions):-
   (ground(Sum)
     -> (Sum > 1
           -> Actions = [exit]  % disjunction satisfied
           ;  Actions = [fail]  % disjunction fails
         )
   ; fd_max(Sum,2)
     -> Actions = [exit,Sum=1]  % last disjunct
   ; user:exists_one(Vars)
     -> Actions = [exit,call(user:ground_all(Vars))]
                                % one disjunct is satisfied
   ; Actions = []               % wait for later
   ).

exists_one([V|Vs]):-
   V==1.
exists_one([_|Vs]):-
   exists_one(Vs).
```

Abductief Constraint Logisch Programmeren: Implementatie en Toepassingen

Nederlandse Samenvatting

Bert Van Nuffelen

Inhoud

1 Inleiding

Computers werden ontworpen om het oplossen van problemen te ondersteunen. Er zijn vele wijzen waarop dit kan gebeuren, maar wij spitsen ons toe op de volgende: de expert beschrijft eerst op een formele wijze de noodzakelijke domeinkennis van het probleem. Vervolgens kiest hij een computerprogramma (een redeneersysteem) waarmee een oplossing berekend kan worden a.d.h.v. de gespecifieerde kennis. Dit wordt het declaratief oplossen van een probleem genoemd.

Om problemen op die manier aan te pakken, hebben we enerzijds een goede kennisrepresentatietaal nodig en anderzijds goede redeneersystemen nodig. In deze thesis beschrijven we de implementatie van een abductief redeneersysteem, genoemd het $\mathcal{A}$system, voor de kennisrepresentatietaal ID-Logic. Dit vormt het eerste gedeelte van de thesis. In het tweede gedeelte behandelen we enkele aspecten van het integreren van gegevensbanken. Dit vormt een toepassing van de declaratieve methode bestaande uit ID-Logic en het $\mathcal{A}$system.

2 ID-Logic

ID-Logic [92, 94] is een uitbreiding van eerste orde logica met (niet-monotone) definities. Een definitie is een constructieve beschrijving van een concept (i.e. predikaat).

Formeel, een ID-Logic theorie $\mathcal{T}$ gebaseerd op de taal $\mathcal{L}$ is een paar $(\mathcal{D},\mathcal{F})$ waarbij $\mathcal{D}$ een verzameling van definities en $\mathcal{F}$ een verzameling van eerste orde formules van $\mathcal{L}$ is. Een definitie D is een verzameling van programmaregels

$$p(\bar{t}) \leftarrow B$$

waarbij B een eerste orde formule is.

De predicaten van een definitie D die voorkomen in een hoofd van een programmaregel worden de *gedefinieerde* predikaten genoemd ($Defined(D)$). Degene die niet voorkomen in het hoofd van een regel worden de *open* predikaten genoemd ($Open(D)$).

De formele betekenis van ID-Logic is als volgt: Een model van een definitie D is een interpretatie M die een interpretatie I voor de open predikaten $Open(D)$ uitbreidt tot een tweewaardig well-founded [257] model van D. Een model van een ID-Logic theorie $\mathcal{T} = (\mathcal{D},\mathcal{F})$ is een interpretatie M dat een model is van elk van de definities in $\mathcal{D}$ en alle formules van $\mathcal{F}$ vervult. $Mod(\mathcal{T})$ stelt de collectie van modellen van de theorie $\mathcal{T}$ voor.

Een ID-Logic theorie kan dus meerdere modellen hebben: in dat geval is er onvolledige kennis van het probleemdomein. Is er slechts één enkel model, dan is er volledige kennis. Wanneer er geen model bestaat, is de theorie inconsistent.

Een operatie die bij de integratie van gegevensbanken voorkomt, is het samenvoegen van ID-Logic theoriën. Veronderstel dat $\mathcal{T}_1$ en $\mathcal{T}_2$ twee ID-Logic theoriën zijn gebaseerd op dezelfde taal $\mathcal{L}$, dan geldt dat de samenstelling $\mathcal{T}_1 \circ \mathcal{T}_2 = (\mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{F}_1 \cup \mathcal{F}_2)$ voldoet aan de compositionaliteitsvoorwaarde: $Mod(\mathcal{T}_1 \circ \mathcal{T}_2) = Mod(\mathcal{T}_1) \cap Mod(\mathcal{T}_2)$. Dit betekent dat de samengevoegde theorie meer kennis (minder modellen) bezit dan elk van de theoriën apart.

We illustreren het declaratief modelleren m.b.v. ID-Logic a.d.h.v. een voorbeeld.

VOORBEELD 1 (HET KLEUREN VAN GRAAF) We beschouwen het probleem van het kleuren van een graaf. Een graaf bestaat uit knopen en verbindingen. Elke knoop heeft een (unieke) kleur. De doelstelling is om om een kleuring van de graaf te vinden zodanig dat elke knoop een andere kleur heeft als de knopen waarmee het verbonden is.

Een eerste stap in het proces is het opstellen van een goed alfabet om de kennis voor te stellen. De objecten in ons probleem zijn knopen en kleuren, beide stellen we voor met constanten $v_1, v_2, \ldots, red, blue, \ldots$. Om aan te duiden dat een constante een knoop (kleur) voorstelt, introduceren we de unaire predikaten $vertex(\cdot)$ ($color(\cdot)$). Het probleemdomein kent twee relaties: $edge(\cdot,\cdot)$ stelt de verbindingen tussen knopen voor en $colorOf(\cdot,\cdot)$ stelt de kleur van een knoop voor.

Met dit alfabet zullen we de domeinkennis uitdrukken als een ID-Logic theorie $\mathcal{T}$ =($\mathcal{D}$,$\mathcal{F}$). $\mathcal{F}$ bestaat uit de volgende asserties:

- beide relaties zijn goed getypeerd.

$$\forall x, y.\ edge(x,y) \rightarrow vertex(x) \wedge vertex(y).$$
$$\forall x, c.\ colorOf(x,c) \rightarrow vertex(x) \wedge colorOf(c).$$

- elke knoop heeft een kleur.

$$\forall\ x.\ vertex(x) \rightarrow \exists\ cx.colorOf(x,cx).$$

- de kleur van een knoop is uniek.

$$\forall\ x, cx, cy.\ colorOf(x,cx) \wedge colorOf(x,cy) \rightarrow cx = cy.$$

- de kleuren van verbonden knopen zijn verschillend.

$$\forall x, y, cx, cy.\ edge(x,y) \wedge colorOf(x,cx) \wedge colorOf(y,cy) \rightarrow cx \neq cy.$$

De kennis over de concrete graaf en de kleuren is definitieele kennis. Immers we kunnen de graaf (de predikaten $vertex(\cdot)$ en $edge$) en de beschikbare kleuren ($color(\cdot)$ door enumeratie precies voorstellen. Stel dat we een driehoekige graaf willen kleuren met 3 kleuren.

$$\mathcal{D} = \left\{ \left\{ \begin{array}{l} vertex(v_1). \\ vertex(v_2). \\ vertex(v_3). \end{array} \right\}, \left\{ \begin{array}{l} edge(v_1, v_2). \\ edge(v_1, v_2). \\ edge(v_2, v_3). \end{array} \right\}, \left\{ \begin{array}{l} color(blue). \\ color(red). \\ color(yellow). \end{array} \right\} \right\}$$

Zulke kennis is afwezig voor het predikaat $colorOf(\cdot,\cdot)$. Inderdaad, het zoeken van een kleuring van een graaf vereist het vinden van een interpretatie van $colorOf(\cdot,\cdot)$ zodanig dat aan al de bovenstaande beperkingen is voldaan. Deze redeneervorm is abductie.

3 Abductie

Naast deductie en inductie is abductie een derde vorm van logisch redeneren. Deductie is het afleiden van informatie die noodzakelijk volgt uit een logische theorie. Het opbouwen van veralgemeningen op basis van experimentele data die consistent zijn met de achtergrondkennis is inductie. Abductie is het opstellen van zinvolle hypothesen die een observatie verklaren in het licht van de achtergrondkennis.

De verschillende wijzen waarop mensen redeneren was een van de vele onderwerpen die de filosofen uit de Oudheid bestudeerden. Zo onderscheidde Aristoteles al abductie als een bijzondere vorm van redeneren. De moderne grondslagen van het abductieonderzoek werden gelegd door de filosoof C.S. Pierce [118], die o.a. de term abductie associeerde met deze vorm van redeneren. Vandaag is abductie het onderwerp van onderzoek in verscheidene disciplines: o.a. filosofie, psychologie, taalkunde en Artificiële Intelligentie.

Ons onderzoek valt binnen het laatst vernoemde domein en meerbepaald in het domein van Logisch Programmeren en Kennisrepresentatie. In deze context is abductief redeneren sterk verbonden met onderzoek naar niet-monotone logica's. Eind jaren 1980 werden de eerste stappen gezet in de integratie van abductief redeneren in Logisch Programmeren. Ruwweg kan men daarin twee richtingen onderscheiden: het gebruik van abductie (a) als een semantisch gereedschap om de logische eigenschappen van niet-monotone logica's te karakteriseren en (b) als een redeneerwijze om problemen op te lossen. Hoewel het eerste spoor min of meer doodlopend is, heeft het onderzoek ertoe geleid dat er meer inzicht is gekomen in de eigenschappen van niet-monotone logica's. O.a. ID-Logic kan beschouwd worden als een uitloper van dit onderzoek.

De andere richting heeft wel tot meer concrete resultaten geleid. Abductie (of meer bepaalde Abductief Logisch Programmeren (ALP) [153, 97, 98]) werd succesvol toegepast om een aantal problemen op te lossen in o.a. de volgende probleem domeinen: A.I. planning [116, 202], scheduling [159, 160, 238], multi-agent systemen [249, 169], natuurlijke taalanalyse [237, 30, 71], gegevensbankintegratie [156, 24] en diagnose [216, 84]. Een *abductief logische theorie* $\mathcal{T}$ is een structuur ($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$) voor een taal $\mathcal{L}$ waarbij $\mathcal{P}$ een logisch programma, $\mathcal{A}$ een verzameling van abductieve atomen en $\mathcal{IC}$ een verzameling van integriteitsbeperkingen (eerste order zinnen) is. Meestal wordt $\mathcal{A}$ vereenvoudigd genoteerd als een verzameling van abductieve predicaten: de abductieve atomen zijn dan alle atomen die met deze predikaten gebouwd kunnen worden ($\mathcal{A} \subseteq \mathcal{HU}(\mathcal{L})$). Een abductief atoom komt *nooit* voor als hoofd van een programmaregel. Stel Δ is een verzameling van atomische feiten, dan denoteert $\mathcal{P} \cup \Delta$ de uitbreiding van een logisch programma $\mathcal{P}$ met een enumeratie van atomische feiten Δ.

De betekenis van een abductief logische theorie is als volgt:

DEFINITIE 1 Een structuur M is een S-model van een abductief logische theorie $\mathcal{T} = (\mathcal{P}, \mathcal{A}, \mathcal{IC})$ asa er een $\Delta \subseteq \mathcal{A}$ bestaat zodanig dat

- M een model is van $\mathcal{P} \cup \Delta$ volgens de semantiek S
- M een klassiek model is van $\mathcal{IC}$: $M \models_S \mathcal{IC}$

Het logisch gevolg is gedefinieerd zoals gewoonlijk: stel F is een FOL formule, dan

$$\mathcal{T} \models_S F \text{ asa, voor elk S-model } M \text{ van } \mathcal{T}, M \models F$$

DEFINITIE 2 Gegeven een abductief logische theorie $\mathcal{T} = (\mathcal{P}, \mathcal{A}, \mathcal{IC})$, en een formule F, dan vormt het koppel (Δ, θ), bestaande uit een verzameling Δ van abductieve atomen $\Delta \subseteq \mathcal{A}$ en θ is een subsitutie, een abductief antwoord asa

- $\mathcal{P} \cup \Delta$ is consistent voor de semantiek S
- $\mathcal{P} \cup \Delta \models_S F\theta$
- $\mathcal{P} \cup \Delta \models_S \mathcal{IC}$

De bovenstaande definities hebben als parameter de formele semantiek S. Men vindt dan ook voor elke belangrijke semantiek in Logisch Programmeren een overeenkomstige abductieve procedure: completion b.v. [85], well-founded b.v. [215], en stable models b.v. [156].

Het onderzoek binnen ALP richtte zich in eerste instantie op het ontwerpen van correcte inferentiesystemen. De belangrijkste inferentiesystemen zijn Kakas-Mancarella [156, 157, 154], SLDNFA [95, 96, 90], IFF [129] en ACLP [4, 152, 162, 161] (in chronologische volgorde). Al deze systemen waren vanuit logisch perspectief 'volwassen'. Vanuit computationeel perspectief waren deze systemen echter te onbetrouwbaar. Hun onbetrouwbaarheid situeert zich zowel op het niveau van robuustheid, waarbij kleine veranderingen in de specificatie leidden tot grote computationele verschillen, als op het niveau van computationele efficiëntie, waarbij slechts kleine voorbeeldprobleempjes konden opgelost worden. Een eerste belangrijke stap in de richting van een beter computationeel gedrag was het ACLP systeem. Dat bouwde een abductief inferentie systeem op een eindig domein constraint solver. Deze constraint solver zorgde ervoor dat belangrijke onderdelen in het inferentieproces versneld werden: inconsistenties werden sneller ontdekt en een groot aantal keuzepunten kon worden doorgegeven naar de constraint solver, die een veel efficiëntere backtracking heeft. Deze situatie was de motivatie van ons onderzoekswerk: het zoeken naar nieuwe goede technieken om een abductief inferentiesysteem te implementeren zodanig dat (kleine) problemen uit de praktijk opgelost kunnen worden. Het systeem dat we ontworpen hebben om onze ideeën te toetsen, is het $\mathcal{A}$system.

3.1 ID-Logic en ALP

Het onderzoek rond de kennisrepresentatietaal ID-Logic kan beschouwd worden als een zoektocht naar de epistomologische grondslagen van (Abductief) Logisch Programmeren. Deze band laat toe om een overeenkomstige abductief logische theorie voor elke ID-Logic theorie te definiëren.

STELLING:
Elke ID-Logic theorie $\mathcal{T}$ =($\mathcal{D}$,$\mathcal{F}$) voor een taal $\mathcal{L}$ heeft een overeenkomstige abductief logische theorie $\mathcal{T}$'=($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$) voor een taal $\mathcal{L}$' ($\mathcal{L} \subseteq \mathcal{L}'$).

De transformatie die hiervoor nodig is, bestaat uit de volgende stappen:

1. Alle definities in $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ worden samengevoegd tot een enkele definitie $\mathcal{D}_t = \{D\}$.
   - Eerst worden de gemeenschappelijk gedefinieerde predikaten gedisambigueerd. Beschouw twee definities D_1 en D_2. Elk predikaat $p \in Pred(D_1) \cap Defined(D_2)$ wordt hernoemd als een hulppredikaat p^* in D_1. Voor elke hernoeming van een predikaat p wordt de assertie $\forall\, \overline{X}.p(\overline{X}) \leftrightarrow p^*(\overline{X})$ aan $\mathcal{F}$ toegevoegd.
   - Als alle gedefinieerde predikaten uniek voorkomen in een enkele definitie dan kunnen de definities samengevoegd worden tot een definitie $D = D_1 \cup D_2 \cup \cdots \cup D_n$
2. De omvorming van de bekomen theorie $\mathcal{T}_t$ tot $\mathcal{T}$'. De open predikaten van $\mathcal{D}$' $Open(\mathcal{D}')$ zijn de abductieve predikaten voor $\mathcal{A}$. Dan geldt $\mathcal{T}' = (\mathcal{P}, \mathcal{A}, \mathcal{IC}) = (D, Open(\mathcal{D}_t), \mathcal{F}_t)$

We hebben aangetoond dat deze transformatie leidt tot equivalente theoriën onder de twee-waardige well-founded semantiek.

Een laatste transformatiestap die we moeten maken voordat we onze bijdrage tot de implementatie van een abductief logisch systeem bespreken, is de omvorming van een abductief logische theorie naar een normaal abductief constraint logisch programma. De procedure die de kern van het systeem vormt, neemt deze programma's als invoer.

Een normaal abductief constraint logisch programma bestaat uit logische programmaregels van de vorm $p(\overline{t}) \leftarrow l_1 \wedge \ldots \wedge l_n$ waarbij elke $l_i, 1 \leq i \leq n$, een atoom of de negatie van een atoom is. Sommige atomen (van een aantal predikaten) zijn abductieve atomen. Deze komen niet voor als hoofd van een regel. Regels van de vorm **false** $\leftarrow l_1 \wedge \ldots \wedge l_n$ worden ontkenningen (*denials*) genoemd.

Een normaal abductief constraint logisch programma P voor een abductief logische theorie ($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$) is het resultaat van een normalisatie transformatie (de Lloyd-Topor transformatie [188]). De normalisatie bestaat erin door universele kwantoren te vervangen door dubbele negatie en dan de binnenste negatie naar de elementaire formules te propageren. Sommige formules (disjuncties of resterende universele kwantoren) worden genormaliseerd door het invoeren van hulppredikaten. Deze transformatie bewaart in het algemeen niet de twee-waardige well-founded semantiek. De oorzaak is een welbepaalde transformatieregel i.v.m. het wegwerken van een universele kwantor.

4 Het $\mathcal{A}$system

Het $\mathcal{A}$system is een nieuwe implementatie van een abductieve inferentie procedure. (Vanaf hier refereren we hiernaar als de $\mathcal{A}$system-procedure.) De basis van deze procedure is SLDNFA [96, 90], een abductieve uitbreiding van SLDNF. Deze hebben we uitgebreid met ideeën van twee andere abductieve procedures: de voorstellingswijze van de $\mathcal{A}$system-procedure komt van de IFF-procedure [129] en de integratie van constraint solvers in de procedure komt van de ACLP-procedure [152, 161].

Het $\mathcal{A}$system is een procedure voor abductief normaal logisch programma's voor de driewaardige completion semantiek. De verzwakking van de semantiek t.v.v. ID-Logic (tweewaardige well-founded) is noodzakelijk omwille van twee redenen: (1) omdat de procedure geen controle doet op oneindige afhankelijkheden (lussen) is er een verzwakking naar completion semantiek nodig, en (2) omdat de procedure enkel lokale consistentie van het antwoord garandeert, is er de verzwakking naar driewaardige semantiek nodig. Echter voor een grote groep van theorieën (b.v. hierarchische theorieën) zijn beide identiek.

De $\mathcal{A}$system procedure beschrijven we als een toestandherschrijfsysteem. Een $\mathcal{A}$system-toestand $\mathcal{S}$ is een paar ($\mathcal{G}$,$\mathcal{ST}$). $\mathcal{G}$ is een verzameling van doelformules. Doelformules zijn ofwel een conjunctie van literals $l_1 \wedge \ldots \wedge l_n$ ofwel een ontkenning $\forall\, \overline{X}. \leftarrow l_1 \wedge \ldots \wedge l_n$. De inferentieregels transformeren deze doelformules tot (eenvoudigere) basisformules die in de structuur $\mathcal{ST}$ worden bewaard. De structuur $\mathcal{ST}$ is een tuppel (Δ,Δ^*,$\mathcal{E}$,$\mathcal{FD}$) bestaande uit verzamelingen die elk een bepaald type van basisformules bewaren.

- Δ: abductieve atomen $a(\overline{t})$
- Δ^*: ontkenningen beginnende met een abductief atoom $\forall\, \overline{X}. \leftarrow a(\overline{t}) \wedge Q$.
- $\mathcal{E}$: (on)gelijkheidsatomen $s = t$ en $s \neq t$

- $\mathcal{FD}$: eindig domein contraint atomen e.g. $x < y$

Het onderhouden van de consistentie van deze substructuren valt ofwel onder het beheer van het $\mathcal{A}$system zelf (Δen Δ^*), ofwel onder het beheer van constraint solvers ($\mathcal{E}$ en $\mathcal{FD}$). Deze constraint solvers zijn onafhankelijke redeneersystemen die op deze wijze een gedeelte van het redeneerwerk overnemen van de abductieve procedure.

De betekenis van een $\mathcal{A}$system-toestand $\mathcal{S}$ is de conjunctie van alle formules in de verschillende (sub)verzamelingen. Een $\mathcal{A}$system-derivatie voor een query $\mathcal{Q}$, d.i. een conjunctie van literals, t.o.v. een abductief logisch programma P is een sequentie van toestanden,

$$\mathcal{S}_0 \rightarrow \mathcal{S}_1 \rightarrow \ldots$$

beginnend van de initiële toestand $\mathcal{S}_0 = (\{\mathcal{Q}\} \cup \mathcal{IC}, \mathcal{ST}^{\emptyset})$. De overgang van de ene toestand naar de andere gebeurt door toepassing van een inferentieregel. Dat betekent dat een van de doelformules wordt geselecteerd en de overeenkomstige inferentieregel wordt uitgevoerd. Indien een toestand inconsistent is, backtrackt het $\mathcal{A}$system. Een eindtoestand $\mathcal{S}_=(\mathcal{G}$,$\mathcal{ST})$ is een toestand zonder doelformules ($\mathcal{G} = \emptyset$) en waarvan alle substructuren in $\mathcal{ST}$consistent zijn. Als een $\mathcal{A}$system-derivatie een eindtoestand $(\emptyset, (\Delta, \Delta^*, \mathcal{E}, \mathcal{FD}))$ bereikt dan vormt Δtesamen met de substitutie σ voor de variabelen van $\mathcal{Q}$ die afgeleid kan worden uit $\mathcal{E}$, een abductief antwoord voor de query $\mathcal{Q}$.

De $\mathcal{A}$system-procedure is een (eenvoudige) uitbreiding van SLDNFA. Dit laat ons toe om de correctheids- en volledigheidsresultaten van SLDNFA voor het $\mathcal{A}$system te hergebruiken.

STELLING: (CORRECTHEID VAN HET $\mathcal{A}$SYSTEM [90])
Het abductief logisch programma P is het abductief normaal logisch programma afgeleid van de $\mathcal{A}$system-theorie $\mathcal{T}$ =($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$). Indien $(\theta, \mathcal{M}(\mathcal{S}_{Sol}))$ een $\mathcal{A}$system-antwoord is voor een query $\mathcal{Q}$ dan geldt dat

$$comp_3(P + \Delta) \models \mathcal{Q}\theta$$

waarbij $\Delta$is de verzameling van geabduceerde atomen in $\mathcal{S}_{Sol}$. Verder geldt ook dat $comp_3(P + \Delta)$ consistent is.

STELLING: (VOLLEDIGHEID VAN HET $\mathcal{A}$SYSTEM [90])
Het abductief logisch programma P is het abductief normaal logisch programma afgeleid van de $\mathcal{A}$system-theorie $\mathcal{T}$ =($\mathcal{P}$,$\mathcal{A}$,$\mathcal{IC}$). Veronderstel dat de query $\mathcal{Q}$ een eindige derivatieboom W heeft, dan geldt

a) indien alle takken van W gefaald hebben, dat $comp_3(P + \Delta) \models \forall\ (\neg\mathcal{Q})$

b) indien $comp_3(P + \exists\ (\mathcal{Q}))$ is vervulbaar, dat W minstens één succesvolle tak heeft.

De voorwaarde dat een vraag $\mathcal{Q}$ een eindige derivatieboom moet hebben is een sterk beperkende voorwaarde die moeilijk te controleren is. Een mogelijkheid is gebruik te maken van de eindigheidsconditie *abductieve recursiviteit* die Verbaeten [268] heeft opgesteld.

5 De implementatie van het $\mathcal{A}$system

Het $\mathcal{A}$system is geïmplementeerd als een meta-programma boven op Prolog (Sicstus Prolog [239]). Deze implementatiewijze laat toe om op een vrij eenvoudige wijze een complex systeem te ontwikkelen met een redelijke computationele efficiëntie. Bovendien zijn nieuwe implementatietechnieken eenvoudig te integreren zodat die experimenteel geëvalueerd kunnen worden.

Conceptueel vereist de ontwikkeling van het $\mathcal{A}$system beslissingen op verschillende niveaus:

1. De gegevensstructuren
2. De organisatie van het zoekproces
3. De evaluatie van de inferentieregels
4. De implementatie van een efficiënte gelijkheid constraint solver
5. De integratie van een eindig domein constraint solver

In de thesis behandelen we elk van deze elementen. Hier zullen we slechts enkele van onze bijdragen bespreken, namelijk de organisatie van het zoekproces en de implementatie van de gelijkheid constraint solver. Deze sectie wordt afgesloten met een bespreking van de resultaten van een experimentele vergelijking met andere redeneersystemen op basis van enkele klassieke A.I. problemen.

5.1 Het zoekproces

Een $\mathcal{A}$system-derivatie kan beschouwd worden als een zoekproces. Elk zoekproces heeft er voordeel bij om zo weinig mogelijk keuzes te maken om een oplossing te bereiken. Bijkomend is het voordelig om zoveel mogelijk informatie ter beschikking te hebben wanneer een keuze gedaan moet worden. Beide principes werden uitgewerkt in het $\mathcal{A}$system.

Een eerste stap is het opdelen van de inferentieregels in twee groepen: de deterministische inferentieregels die één toestand omvormen naar exact één andere en de niet-deterministische inferentieregels die één toestand kunnen omvormen naar verschillende toestanden. Deze laatste inferentieregels vormen de keuzepunten in het zoekproces. In plaats van de inferentieregels willekeurig toe te passen, bestaat het $\mathcal{A}$system uit een opeenvolging van twee fasen:

1. Een *propagatiefase* waarin enkel de deterministische inferentieregels worden toegepast totdat enkel doelen overblijven die een niet-deterministische inferentieregel vereisen.

2. Een *keuzefase* waarin één enkele niet-deterministische inferentieregel wordt uitgevoerd.

Door deze propagatiefase worden sommige mogelijkheden van een niet-deterministische inferentieregel uitgesloten, zodat die niet geëxploreerd worden tijdens het zoeken. Een tweede stap verbetert dit effect. Daarvoor wordt met elke mogelijke tak in een keuzepunt een formule geassocieerd die observeert of de tak nog mogelijk is of niet. Hiervoor gebruiken we de *reïficatietechniek*: de reïficatie van een formule F associeert met F een Booleaanse variabele B, die aangeeft of F een logisch gevolg is van een theorie. In het $\mathcal{A}$system hebben we reïficatie geïmplementeerd voor gelijkheden en eindig domein formules. Indien er afgeleid wordt dat alle takken in een keuzepunt niet vervulbaar zijn (alle B's zijn 0), kan de huidige toestand niet leiden tot een eindtoestand en backtrackt het $\mathcal{A}$system. Echter het $\mathcal{A}$system kan ook detecteren of juist één tak overblijft die vervuld kan worden. In dat geval is het keuzepunt geen werkelijke keuze en zal dit keuzepunt ook geëvalueerd worden tijdens de propagatiefase. Dit algemeen schema hebben we gespecialiseerd voor elk van de niet-deterministische inferentieregels. Deze specialisatie resulteert in nog meer propagatie.

Zelfs met de verbeterde propagatie zullen niet alle keuzepunten verwijderd worden. Op bepaalde momenten in een $\mathcal{A}$system-uitvoering zullen dus keuzepunten geselecteerd worden en aanleiding geven tot backtrackpunten. De selectie van het keuzepunt is erg belangrijk voor het verdere verloop van de derivatie. Omdat deze keuze sterk probleemafhankelijk is, hebben we ervoor gekozen om het selectieproces the parametriseren zodat de gebruiker de 'beste' strategie kan kiezen. Uit onze ervaringen hebben we slechts vage algemene heuristische regels kunnen afleiden. Het vinden van de juiste strategie vereist (1) inzicht in het probleem en de inferentieregels van het $\mathcal{A}$system (2) en heel wat experimenten.

5.2 De gelijkheid constraint solver $\mathcal{E}$

Het evalueren van gelijkheden ($s = t$) en ongelijkheden ($s \neq t$) vormt een belangrijk deel van de inferentiestappen die tijdens een $\mathcal{A}$system-derivatie gemaakt worden. Dit komt omdat bijna elke inferentieregel van het $\mathcal{A}$system (on)gelijkheden produceert. Daarom is het belangrijk om de (on)gelijkheden efficiënt te evalueren.

In SLDNFA (en andere abductieve procedures) werden enkel gelijkheden geëvalueerd door middel van unificatie. Ongelijkheden werden indirect behandeld als residuele ontkenningen van de vorm $\forall\ \overline{Y}.X = t \wedge Q$ (where $X \notin vars(\overline{Y})$). Om een correcte evaluatie van deze ontkenningen te hebben, moet de unificatie uitgebreid worden. Door deze aanpak zijn ontkenningen (en de residuele ontkenningen) passieve elementen in het $\mathcal{A}$system. Slechts na een unificatie werden de aanwezige ontkenningen gecontroleerd, wat inefficiënt is. Deze ineffecientie werd nog versterkt doordat de eerste prototypes geen directe toegang hadden tot de relevante ontkenningen. Dat had als gevolg dat na elke unificatie alle ontkenningen gecontroleerd werden, dus ook degene die geen relatie hadden met de variabelen die betrokken waren in de unificatie.

Onze bijdrage is de ontwikkeling van een nieuwe (on)gelijkheid constraint solver $\mathcal{E}$. Deze constraint solver maakt het redeneren met (on)gelijkheden een dynamisch proces. Nieuwe expressies die aan de constraint store worden toegevoegd, worden onmiddellijk geëvalueerd en enkel t.v.v. de gerelateerde expressies. Dat zorgt voor een grote verbetering van de efficiëntie. Daarnaast biedt $\mathcal{E}$ een grote variëteit aan expressies: gelijkheden ($s = t$), ongelijkheden ($s \neg t$), disjuncties en conjuncties van (on)gelijkheden (b.v. $s_1 = t_1 \vee (s_2 \neq t_2 \wedge s_3 = t_3)$ en de reïficatie van deze expressies (b.v. $s = t \Leftrightarrow B$). Deze laatste is erg belangrijk omdat het de basiscomponent is in de implementatie van de propagatie voor de inferentieregels. Zover onze kennis reikt, is de constraint solver $\mathcal{E}$ de enige (on)gelijkheid solver die reïficatie van zijn expressies toelaat.

5.3 Experimentele resultaten

Om de impakt van de ontwerpbeslissingen te evalueren, hebben we het $\mathcal{A}$system geëvalueerd m.b.v. enkele klassieke A.I. problemen. Meerbepaald hebben we het gedrag van het $\mathcal{A}$system bestudeerd op voor Constraint Satisfaction Problems (het *N-queens* en *pigeon hole* probleem) , en voor A.I. planningsproblemen (het *blocks world* en *logistics* probleem).

Tegelijkertijd hebben we deze problemen ook opgelost met andere gerelateerde systemen, zodat we de performantie van het $\mathcal{A}$system kunnen plaatsen t.o.v. andere systemen. We hebben gekozen voor DLV [105, 112], een Answer Set Programming systeem dat voor gelijkaardige redeneertaken als het $\mathcal{A}$system is gebouwd maar dat niet op resolutie gebaseerd is en voor twee taakspecifieke systemen: de eindig domein constraint solver van Sicstus Prolog ($\mathcal{FD}$-solver [66]) voor de CSP problemen en de heuristische planner FF [145] voor de A.I. planningsproblemen.

Het $\mathcal{A}$system vertoont bijzonder gedrag voor Constraint Satisfaction Problems: deze worden namelijk zonder het creëren van een backtrackpunt op het abductieve niveau tot een constraint store herleid. De resultaten van het $\mathcal{A}$system voor beide beschouwde CSP problemen zijn goed. Er is een constante overhead t.v.v. een oplossing in de $\mathcal{FD}$-solver. Dit computationeel resultaat samen met de grotere expressiviteit maakt het $\mathcal{A}$system een goede modelleringsomgeving voor Constraint Satisfaction Problems.

De A.I. planningsproblemen zijn een grotere uitdaging voor het $\mathcal{A}$system. Het $\mathcal{A}$system moet voor deze problemen zelf keuzes maken om een oplossing te vinden. De resultaten voor het *blocks world* probleem zijn goed omdat (1) het propagatiemechanisme van het $\mathcal{A}$system snel inconsistente toestanden detecteert en (2) er een eenvoudige strategie is die het $\mathcal{A}$system direct naar een oplossing leidt. Die strategie is het gevolg van het uitbuiten van een veronderstelling in het probleemdomein, namelijk dat de tafel waarop de blokken gezet worden oneindig groot is. De resultaten voor het *logistics* probleem zijn minder indrukwekkend. Het $\mathcal{A}$system kan slechts kleine probleeminstanties oplossen. Voor grotere problemen komt het system terecht in delen van de zoekboom die geen oplossing bevatten maar die wel zeer veel tijd vragen om volledig te exploreren. Eén oorzaak hiervoor is de lange tijd die nodig is om complexe eindig domein constraint stores te controleren op consistentie. Een andere reden is de late detectie van onmogelijke keuzes omdat de keuzepunten die verband houden met elkaar ver uit elkaar worden geëvalueerd. De resultaten van het $\mathcal{A}$system zijn nog ver van die van FF, de stripsplanner. De performantie van FF wordt hoofdzakelijk bepaald door de zoekheuristiek die de planner afleidt van de probleemspecificatie. Het $\mathcal{A}$system biedt momenteel slechts algemene domein onafhankelijke zoekstrategiën aan, die wel door de gebruiker geselecteerd kunnen worden. Door middel van experimentatie kan men een efficiënte zoekstrategie zoeken. Deze werkwijze is gelijkaardig aan de werkwijze in Constraint Logic Programming. Hoewel deze methode toelaat een goede computationele performantie te bekomen, is het dikwijls ook een moeizaam proces dat een gebruiker zich ontziet.

Vergelijken we de resultaten van het $\mathcal{A}$system met die van DLV, een redeneersysteem dat dicht bij het $\mathcal{A}$system staat qua doelstelling, dan bemerken we dat het $\mathcal{A}$system duidelijk performanter is voor de CSP problemen. Dit is niet verwonderlijk omdat deze problemen juist de typische problemen zijn waarvoor (eindig domein) constraint solvers ontworpen zijn. De resultaten voor de A.I. planningsproblemen tonen aan dat DLV beperkt wordt door de noodzakelijke grounding. Als er zoals in het *blocks world* probleem een grote grounding nodig is, kan DLV die problemen niet binnen een redelijke tijd oplossen. Indien dit wel het geval is, zoals voor het *logistics* probleem, is DLV in staat om dikwijls sneller dan het $\mathcal{A}$system een oplossing te vinden.

Een ander voordeel van DLV is zijn robuustheid: het eindigt zeker, ook wanneer er recursieve definities zijn. Dit is een belangrijk nadeel van het $\mathcal{A}$system. Het $\mathcal{A}$system eindigt niet altijd, sterker nog, het kan in een oneindige lus terecht komen (iets wat het heeft overgeërfd van SLD(NF)). Een mogelijke oplossing is de integratie van 'tabulatie' in het $\mathcal{A}$system. Deze uitbreiding van SLD(NF) procedure verbetert het eindigheidsgedrag voor SLD(NF) door het onthouden van de al uitgevoerde doelen (en de bijhorende antwoorden) in een tabel. Als een geselecteerd doel al in de tabel voorkomt, wordt geen nieuwe evaluatie opgestart maar worden de antwoorden uit de tabel herbruikt. Op deze wijze zal elke doel hoogstens één keer uitgevoerd worden.

5.4 Conclusie

Het $\mathcal{A}$system is een belangrijke stap in de richting van performante abductieve redeneersystemen. Onze ontwerpbeslissingen op verschillende niveaus in het systeem hebben bijgedragen tot een systeem dat een aanvaardbaar tot goed computationeel gedrag vertoont. Het systeem is door deze beslissingen geëvolueerd van een eenvoudig systeem in de zin van Prolog (SLD) waarbij één enkele inferentiestap exhaustief wordt uitgevoerd tot een complex systeem bestaande uit verschillende subsystemen waarin de zoekruimte actief verkleind wordt.

De ontwikkeling van het $\mathcal{A}$system is natuurlijk niet ten einde: er zijn uitbreidingen en verbeteringen mogelijk op verschillende vlakken: o.a. verbeterde zoekstrategiën, integratie van nieuwe constraint solvers, enz... De richting van de verdere ontwikkeling zien we vooral aan de hand van het uitvoeren van experimenten met realistische problemen. In de loop van ons doctoraatswerk hebben we verscheidene realistische problemen onder de loep genomen. Het resultaat van deze experimenten was zeer wisselend: meestal konden (zeer) eenvoudige probleempjes wel opgelost worden maar grotere problemen waren dikwijls onoplosbaar binnen een redelijke tijd. Omwille van allerlei redenen zijn geen van deze problemen opgenomen in de thesistekst.

6 Het samenvoegen van gegevensbanken

6.1 Inleiding

Onze maatschappij wordt een *kennis-* of *informatiemaatschappij* genoemd, verwijzend naar het belang dat (het bezit van) kennis in het sociale en economische leven inneemt. Een belangrijke oorzaak hiervan is de vereenvoudigde toegang tot informatie: kranten, radio, televisie en Internet bieden een onafgebroken stroom van bereikbare informatie aan. Echter, wie specifieke informatie wil vinden moet uit het aanbod de juiste en relevante informatiebrokken selecteren. Dat is een niet-triviaal proces. Bovendien is de aangeboden informatie dikwijls ongestructureerd en verspreid over verscheidene bronnen. Deze evolutie brengt nieuwe uitdagingen met zich mee.

In deze thesis behandelen we het samenvoegen van relationele gegevensbanken. Relationele gegevensbanken zijn de meest voorkomende gestructureerde informatiesystemen. Meerbepaald beschouwen we de volgende situatie waarbij een aantal gegevensbanken die informatie over een gemeenschappelijk informatiedomein bewaren. Het doel is een overkoepelende gegevensbank te construeren die dit gemeenschappelijk domein materialiseert. Hierbij moeten we rekening houden met de verschillen in voorstelling van de informatie in de gegevensbanken en de inconsistenties die ontstaan wanneer gegevens van de gegevensbanken worden samengevoegd.

6.2 Relationele gegevensbanken

De basiscomponent in het probleemdomein is een relationele gegevensbank.

DEFINITIE 3 Een *gegevensbank* $\mathcal{DB}$ is een structuur $\langle \mathcal{L}, \mathcal{D}, \mathcal{IC} \rangle$ samengesteld uit een eerste orde taal $\mathcal{L}$, een gegevensbank instantie $\mathcal{D}$ en een eindige verzameling van integriteitsbeperkingen $\mathcal{IC}$. Een gegevensbankinstantie $\mathcal{D}$ is een verzameling van atomen gebaseerd op de taal $\mathcal{L}$. De integriteitsbeperkingen zijn eerste order zinnen gebaseerd op de taal $\mathcal{L}$.

De formele betekenis van een gegevensbank $\mathcal{DB}$ is het minimale Herbrand model $\mathcal{H}^{\mathcal{D}}$ dat wordt voorgesteld door de gegevensbankinstantie $\mathcal{D}$. Indien het minimale Herbrand model $\mathcal{H}^{\mathcal{D}}$ een model is van de integriteitsbeperkingen $\mathcal{IC}$ dan is de gegevensbank $\mathcal{DB}$ *consistent*, anders is ze *inconsistent*.

VOORBEELD 2 Beschouw de Belgische koninklijke familie en meerbepaald de huidige koning en de troonopvolgers. De volgende gegevensbank $\mathcal{DB}_{Belg}$ bevat de familiegegevens van deze laatste groep:

- $\mathcal{L} = \{vader(\cdot,\cdot), moeder(\cdot,\cdot)\}$,
- $\mathcal{D} = \left\{ \begin{array}{l} vader(albert, filip), moeder(paola, filip), \\ vader(filip, elisabeth) \\ moeder(mathilde, elisabeth) \end{array} \right\}$
- $\mathcal{IC} = \{\forall\ X, Y. \neg(vader(X,Y) \wedge moeder(X,Y))\}$

Deze gegevensbank is consistent. Indien het atoom $vader(paola, filip)$ wordt toegevoegd aan deze gegevensbank $\mathcal{DB}_{Belg}$, dan is de uitgebreide gegevensbank inconsistent.

6.3 Het samenvoegen van relationele gegevensbanken

Stel dat we n gegevensbanken $\mathcal{DB}_1 = \langle \mathcal{A}_1, \mathcal{D}_1, \mathcal{IC}_1 \rangle, \ldots, \mathcal{DB}_n = \langle \mathcal{A}_n, \mathcal{D}_n, \mathcal{IC}_n \rangle$ met informatie over een gemeenschappelijk kennisdomein ter beschikking hebben. We noemen deze gegevensbanken de *bronnen*. Het doel is een overkoepelende gegevensbank $\langle \mathcal{A}, \mathcal{D}, \mathcal{IC} \rangle$ te construeren zodanig dat er een uniforme toegang is tot de informatie in de bronnen. De bronnen zijn onafhankelijke entiteiten: m.a.w. ze worden door verschillende personen of groeperingen beheerd. Deze onafhankelijkheid uit zich in verschillen in de gebruikte talen en de aanwezige integriteitsbeperkingen.

In dit integratieproces kunnen drie uitdagingen onderscheiden worden, die gewoonlijk in de gepresenteerde volgorde uitgevoerd worden:

1. Het samenvoegen van de talen $\mathcal{L}_i$ tot $\mathcal{L}$.
   Omdat elke gegevensbank een onafhankelijke entiteit is, heeft elke gegevensbank een eigen alfabet om de (gemeenschappelijke) informatie voor te stellen. Een eerste stap in het samenvoegen van de gegevensbanken is het formeel uitdrukken van de verbanden tussen de talen van de bronnen en de taal van de overkoepelende gegevensbank.

2. Het samenvoegen van de integriteitsbeperkingen $\mathcal{IC}_i$ tot $\mathcal{IC}$.
   Integriteitsbeperkingen formuleren de intentionele kennis van een gegevensbank waaraan de gegevensbankinstantie moet voldoen. Het doel van deze fase is het construeren van een consistente verzameling van integriteitsbeperkingen, die ofwel komen van de bronnen ofwel nieuwe integriteitsbeperkingen zijn die enkel gelden voor de overkoepelende gegevensbank.

3. Het samenvoegen van de gegevensbankinstanties $\mathcal{D}_i$ tot $\mathcal{D}$.
   Een laatste fase is het samenvoegen van de gegevensbankinstanties. Door het samenvoegen van de instanties kunnen er integriteitsbeperkingen geschonden worden.

6.4 Mediator gebaseerde systemen

Wanneer er geen integriteitsbeperkingen zijn, herleidt het integratieprobleem zich tot het opstellen van een structuur die de verbanden tussen de verschillende talen uitdrukt. Via die structuur is de informatie in de bronnen op een uniforme wijze beschikbaar.

Een structuur die in de literatuur voorgesteld wordt als oplossing voor dit probleem is een *mediator gebaseerd systeem* [252]. Een mediator gebaseerd systeem $\langle \mathcal{L}_G, S, M \rangle$ is een structuur bestaande uit de overkoepelende taal (alfabet) $\mathcal{L}_G$, een eindige verzameling $S = \{S_1, \ldots, S_n\}$ van bronnen en een logische theorie M die de verbanden tussen de talen van de bronnen en de overkoepelende taal $\mathcal{L}_G$ uitdrukt. Het doel van een mediator gebaseerd systeem is het bouwen van een virtuele overkoepelende gegevensbank. Op deze wijze biedt een mediator gebaseerd systeem een natuurlijke ondersteuning voor het toevoegen en verwijderen van bronnen. De samengevoegde kennis wordt immers enkel op vraag berekend.

Voor deze structuur hebben we een ID-Logic invulling gedefinieerd. We veronderstellen dat alle talen hetzelfde domein (constanten en functiesymbolen) delen, en dat elk predikaat symbool slechts in één enkele taal voorkomt. Ook veronderstellen we de aanwezigheid van *unique names axioms*. $\mathcal{L}$ is de samengestelde taal opgebouwd uit de unie van de alle betrokken talen: $\mathcal{L} = \mathcal{L}_G \cup \mathcal{L}_1 \cup \cdot \cup \mathcal{L}_n$. Een ID-Logic mediator gebaseerd systeem bestaat uit een aantal ID-Logic theoriën:

- Voor elke bron $S_i \in S$ wordt een overeenkomstige ID-Logic theorie $\mathcal{S}_i$ gebouwd gebaseerd op de taal $\mathcal{L}$.

- Voor elke bron $S_i \in S$ worden de ontologische verbanden tussen de talen $\mathcal{L}_i$ en $\mathcal{L}_G$ als een ID-Logic theorie $\mathcal{W}_i$ uitgedrukt.
- Een gemeenschappelijke ID-Logic theorie $\mathcal{K}$ die de meta-kennis i.v.m. de volledigheid van de kennis van een groep van bronnen t.o.v. de overkoepelende gegevensbank $\langle \mathcal{A}, \mathcal{D}, \mathcal{IC} \rangle$ uitdrukt.

De kennis $\mathcal{T}$ van het ID-Logic mediator gebaseerd systeem is de samenstelling van alle ID-Logic theoriën:

$$\mathcal{T} = \mathcal{S}_1 \circ \cdots \circ \mathcal{S}_n \circ \mathcal{W}_1 \circ \cdots \circ \mathcal{W}_n \circ \mathcal{K}$$

VOORBEELD 3 Stel dat we twee bronnen ter beschikking hebben. Elke bron heeft volledige kennis over zijn eigen relaties. Bron S_1 bevat alle studenten die een volledige studie programma volgen en bron S_2 bevat het inschrijvingsjaar van alle studenten (zowel van degene die een volledig programma als een gedeeltelijk programma volgen).

$$\begin{array}{ll} S_1 = \langle \{student(\cdot)\}, & \{student(john), student(mary), student(bob)\}, \emptyset \rangle, \\ S_2 = \langle \{enrolled(\cdot,\cdot)\}, & \{enrolled(john, 1999), enrolled(eve, 1999), \\ & enrolled(mary, 2000), enrolled(alice, 2003)\}, \emptyset \rangle \end{array}$$

Een mediator gebaseerd systeem $\mathfrak{G}$ dat een lijst van alle studenten die een volledig programma volgen en die ingeschreven zijn in het jaar 1999 aanbiedt, ziet er mogelijk als volgt uit: $\mathfrak{G} = \langle \mathcal{L}_G, \{\mathcal{S}_1, \mathcal{S}_2\}, \{\mathcal{W}_{G\to 1}, \mathcal{W}_{G\to 2}, \mathcal{K}\} \rangle$, waarbij

- $\mathcal{L}_G = \{st99(\cdot)\}$
- $\mathcal{S}_1 = \left(\left\{ \left\{ \begin{array}{l} student(john). \\ student(mary). \\ student(bob). \end{array} \right\} \right\}, \emptyset \right\}$
- $\mathcal{S}_2 = \left(\left\{ \left\{ \begin{array}{l} enrolled(john, 1999). \\ enrolled(eve, 1999). \\ enrolled(mary, 2000). \\ enrolled(alice, 2003) \end{array} \right\} \right\}, \emptyset \right\}$

$$\begin{array}{lcl} \mathcal{W}_{G\to 1} & = & \left(\left\{ \left\{ \; st99(X) \leftarrow student(X) \wedge st99^s_{\mathcal{S}_1}(X). \; \right\} \right\}, \emptyset \right) \\ \mathcal{W}_{G\to 2} & = & \left(\left\{ \left\{ \; st99(X) \leftarrow enrolled(X, 1999) \wedge st99^s_{\mathcal{S}_2}(X). \; \right\} \right\}, \emptyset \right) \\ \mathcal{K} & = & \left(\emptyset, \left\{ \begin{array}{l} \forall\, X.st99^s_{\mathcal{S}_1}(X) \leftrightarrow enrolled(X, 1999). \\ \forall\, X.st99^s_{\mathcal{S}_2}(X) \leftrightarrow student(X). \end{array} \right\} \right) \end{array}$$

Een alternatieve formulering $M = \{\mathcal{W}_{G\to\{1,2\}}\}$ bestaat slechts uit één enkele ID-Logic theorie:

$$\mathcal{W}_{G\to\{1,2\}} \quad = \quad (\{\{\ st99(X) \leftarrow student(X) \wedge enrolled(X, 1999).\ \}\}, \emptyset)$$

Als we ons ID-Logic mediator gebaseerd systeem vergelijken met de mediator gebaseerde systemen in de literatuur [252, 179, 176], blijkt dat onze aanpak een veralgemening is van de bestaande systemen. De grote expressiviteit, de natuurlijke wijze om met onvolledige informatie om te gaan en de mogelijkheid om meerdere definities van het zelfde predikaat te specifiëren maken ID-Logic tot een uitstekende logica voor dit probleem. Zoals het bovenstaande voorbeeld illustreert, leidt het gebruik van ID-Logic tot een overzichtelijke en compacte voorstelling.

We willen ons ID-Logic mediator gebaseerd systeem ook gebruiken voor het beantwoorden van vragen. Omdat in het algemeen de opgebouwde theorie $\mathcal{T}$ onvolledig is, kan een vraag $\mathcal{Q}$ op twee manieren beantwoord worden:

1. op een *zekere* wijze: $\mathcal{Q}$ is een logisch gevolg van $\mathcal{T}$.
2. op een *mogelijke* wijze: $\mathcal{Q}$ is voldaan in $\mathcal{T}$.

In de literatuur gaat het meeste aandacht naar de eerste wijze. Volgens ons is de tweede wijze in sommige contexten het verwachte antwoord. B.v. wanneer een belangrijke bron tijdelijk onbeschikbaar is, verwacht de gebruiker dat het integratiesysteem dit meldt wanneer het antwoord op de vraag $\mathcal{Q}$ afhangt van deze bron. Echter als het integratiesysteem enkel vermeldt dat de vraag niet met zekerheid kan beantwoord worden omwille van een tijdelijke onbeschikbaarheid, dan is dit dikwijls niet erg informatief voor de gebruiker. Deze verwacht meer informatie, namelijk een overzicht van de ontbrekende informatie. Dit laatste vereist abductief redeneren. Deze argumentatie toont aan dat abductief redeneren noodzakelijk is om een mediator gebaseerde systeem in alle gevallen het verwachte antwoord te laten produceren.

6.5 Coherent samenvoegen van gegevensbankinstanties

Het tweede beschouwde probleem veronderstelt dat alle bronnen $\mathcal{DB}_i, 1 \leq i \leq n$, dezelfde taal $\mathcal{L}$ delen en dat er een consistente verzameling van integriteitsbeperkingen $\mathcal{IC}$ voor de overkoepelende gegevensbank $\langle \mathcal{A}, \mathcal{D}, \mathcal{IC} \rangle$ is. De unie van de gegevensbankinstanties $\bigcup_{1\leq 1 \leq n} \mathcal{D}_i$ kan de integriteitsbeperkingen $\mathcal{IC}$ schenden. In tegenstelling tot bij het updaten van een gegevensbank waar bij het ontdekken van een inconsistentie de update ongedaan wordt gemaakt door middel van een *roll-back* mechanisme, is het ongedaan maken van de samenvoeging geen optie. Het doel is immers om juist de samenvoeging te maken. Omdat voor inconsistente theoriën inferentie triviaal is (alles volgt uit een inconsistentie), moeten technieken ontworpen worden die toelaten om toch *zinvolle* informatie uit een inconsistente gegevensbank af te leiden. Ruwweg bestaan er twee mogelijke oplossingen:

1. *Para-consistente* logica's vermijden dat de aanwezigheid van inconsistenties leidt tot triviale logische inferentie door het invoeren van een extra waarheidswaarde (inconsistentie).

2. *Coherente* samenvoegingstechnieken zijn gebaseerd op het herstellen van de consistentie d.m.v. het toevoegen of verwijderen van atomen. Meestal zijn er meerdere mogelijkheden om een gegevensbank te herstellen. Daarom worden er preferentiecriteria gebruikt om de meest belangrijke herstellingen te selecteren. Deze herstellingen kunnen gebruikt worden door de gebruiker om de samengevoegde informatie consistent te maken. Wat logisch volgt uit een inconsistente gegevensbank is dan datgene wat logisch volgt uit alle mogelijke herstellingen van de gegevensbank.

In ons doctoraatswerk modelleren we twee coherente samenvoegingstechnieken. De ene is gebaseerd op ID-Logic en het $\mathcal{A}$system, een tweede encodeert de mogelijke herstelling als de modellen van een propositionele theorie. Omwille van onze veronderstellingen volstaat het om de techniek te beschrijven voor een inconsistente gegevensbank.

Het herstel van een inconsistente gegevensbank

Een *herstel* van een inconsistente gegevensbank $\langle \mathcal{A}, \mathcal{D}, \mathcal{IC} \rangle$ is een paar (Insert,$\mathsf{Retract}$) zodanig dat

- $\mathsf{Insert} \cup \mathcal{D} = \emptyset$
- $\mathsf{Retract} \subseteq \mathcal{D}$
- $\langle \mathcal{L}, (\mathcal{D} \cup \mathsf{Insert}) \setminus \mathsf{Retract}, \mathcal{IC} \rangle$ is een consistente gegevensbank.

Insert is dus de verzameling van de atomen die toegevoegd worden aan de gegevensbankinstantie $\mathcal{D}$ en $\mathsf{Retract}$ de verzameling van atomen die eruit verwijderd worden. Omdat een gegevensbank op verschillende wijzen hersteld kan worden, is het zinvol om een preferentiecriterium te definiëren en enkel de meest geprefereerde herstellingen te gebruiken. De meest gebruikte criteria waarbij een herstel (Insert,$\mathsf{Retract}$) geprefereerd wordt over (Insert',$\mathsf{Retract}$'), zijn

- de minimale cardinaliteit: $(\mathsf{Insert}, \mathsf{Retract}) \leq_c (\mathsf{Insert}', \mathsf{Retract}')$ asa $|\mathsf{Insert}| + |\mathsf{Retract}| \leq |\mathsf{Insert}'| + |\mathsf{Retract}'|$
- setinclusie: $(\mathsf{Insert}, \mathsf{Retract}) \leq_i (\mathsf{Insert}', \mathsf{Retract}')$ asa $\mathsf{Insert} \subseteq \mathsf{Insert}'$ en $\mathsf{Retract} \subseteq \mathsf{Retract}'$

Een $\leq$-geprefereerd herstel van een gegevensbank $\mathcal{DB}$ is een herstel (Insert,$\mathsf{Retract}$) zodanig dat voor elke ander herstel (Insert',$\mathsf{Retract}$') van $\mathcal{DB}$ geldt dat indien $(\mathsf{Insert}', \mathsf{Retract}') \leq (\mathsf{Insert}, \mathsf{Retract})$ dan geldt $(\mathsf{Insert}, \mathsf{Retract}) \leq (\mathsf{Insert}'\mathsf{Retract}')$.

ID-Logic/$\mathcal{A}$system oplossingsmethode

In een eerste aanpak specifiëren we de herstelde gegevensbank m.b.v. de volgende ID-Logic meta-theorie:

$$\mathcal{T}_{composer} = \left(\begin{array}{l} \left\{ \left\{ \begin{array}{l} fact(X) \leftarrow db(X) \wedge \neg retract(X). \\ fact(X) \leftarrow insert(X). \end{array} \right\} \right\}, \\ \\ \left\{ \begin{array}{l} \forall\, X.\neg(insert(X) \wedge db(X)) \\ \forall\, X.retract(X) \rightarrow db(X) \end{array} \right\} \end{array} \right)$$

De relatie $db(X)$ stelt voor dat het atoom X behoort tot de gegevensbankinstantie. Een atoom X dat verwijderd (toegevoegd) wordt, wordt voorgesteld door het predikaat $retract(X)$ ($insert(X)$). De atomen die in de herstelde gegevensbankinstantie voorkomen, worden voorgesteld door het predikaat $fact(X)$. Dit predikaat kan precies gedefinieerd worden als de unie van alle atomen X van de (inconsistente) gegevensbankinstantie $db(X)$ die niet verwijderd werden ($\neg retract(X)$) met alle nieuw toegevoegde atomen $insert(X)$. De twee eerste order formules drukken de noodzakelijke beperkingen op $insert(\cdot)$ en $retract(\cdot)$ uit.

We illusteren het gebruik van deze meta-theorie met een voorbeeld.

Voorbeeld 4 Beschouw de inconsistente gegevensbank dat informatie bevat over leerkrachten n_i die een vak c_j geven ($teaches(c_j, n_i)$).

$$\begin{array}{ll} \mathcal{L} & = \{teaches(\cdot,\cdot)\} \\ \mathcal{D} & = \{teaches(c_1,n_1), teaches(c_2,n_2), teaches(c_2,n_3)\} \\ \mathcal{IC} & = \{\forall\, X,Y,Z.teaches(X,Y) \wedge teaches(X,Z) \rightarrow Y = Z\} \end{array}$$

De integriteitsbeperking drukt uit dat elk vak slechts door een leerkracht kan gegeven worden. Dan is de ID-Logic encodering van de herstelde gegevensbank

$$\left(\begin{array}{l} \left\{ \begin{array}{l} \left\{ \begin{array}{l} fact(X) \leftarrow db(X) \wedge \neg retract(X). \\ fact(X) \leftarrow insert(X). \end{array} \right\}, \\ \left\{ \begin{array}{l} db(teaches(c_1,n_1)). \\ db(teaches(c_2,n_2)). \\ db(teaches(c_2,n_3)). \end{array} \right\} \end{array} \right\}, \\ \left\{ \begin{array}{l} \forall\, X.\neg(insert(X) \wedge db(X)) \\ \forall\, X.retract(X) \rightarrow db(X) \\ \forall\, X,Y,Z.fact(teaches(X,Y)) \wedge fact(teaches(Y,Z)) \rightarrow Y = Z \end{array} \right\} \end{array} \right)$$

Het gebruik van de meta-theorie vereist dat alle integriteitsbeperkingen van $\mathcal{IC}$ herschreven worden in termen van de herstelde gegevensbankinstantie. Enkele modellen van deze theorie zijn (beperkt tot de meta-theorie predikaten):

1 $\{fact(teaches(c_1,n_1)), fact(teaches(c_2,n_2)), retract(teaches(c_2,n_3))\}$

2 $\{fact(teaches(c_1, n_1)), fact(teaches(c_2, n_3)), retract(teaches(c_2, n_2))\}$

$\vdots$

6 $\{retract(teaches(c_1, n_1)), retract(teaches(c_2, n_3)), retract(teaches(c_2, n_2))\}$

Deze komen overeen met verschillende mogelijke herstellingen (b.v. het eerste model verwijdert enkel het atoom $teaches(c_2, n_3)$). Model 6 is duidelijk minder interessant dan het eerste omdat het alle atomen verwijdert.

Het berekenen van deze modellen is een abductieve taak. Inderdaad, we zoeken naar een consistente interpretatie voor de predikaten $insert(\cdot)$ en $retract(\cdot)$. Hiervoor kunnen we het $\mathcal{A}$system gebruiken. We hebben aangetoond dat de abductieve oplossingen die het $\mathcal{A}$system berekent, overeenkomen met de herstellingen van de gegevensbank. Om de meest geprefereerde herstellingen te berekenen, hebben we het $\mathcal{A}$system uitgebreid met een *branch-and-bound* optimisatiemodule. Hiervoor is het vereist dat het $\mathcal{A}$system alle mogelijke herstellingen berekent. Voor twee veel voorkomende integriteitsbeperkingen, namelijk *primery key* beperkingen en *foreign key* beperkingen, hebben we aangetoond dat het $\mathcal{A}$system eindigt. Daardoor is het verzekerd dat het $\mathcal{A}$system alle (meest geprefereerde) herstellingen kan berekenen.

Een belangrijke eigenschap van deze ID-Logic aanpak is de flexibiliteit. De ID-Logic meta-theorie is vrij eenvoudig aan te passen aan een gewijzigde probleemsituatie. Dit illustreren we met een aangepaste meta-theorie voor temporele gegevens en een aangepaste meta-theorie waarin een referentie naar de oorsprong van de atomen is opgenomen. Tenslotte geven we een eerste aanzet tot het integreren van het ID-Logic mediator gebaseerd systeem en de ID-Logic meta-theorie voor het herstellen van de inconsistenties.

Een oplossingsmethode

De tweede oplossingsmethode codeert het herstel van een gegevensbank als de modellen van een propositionele theorie.

Het idee van deze methode is gebaseerd op de observatie dat de status van een atoom slechts wijzigt wanneer het behoort tot het herstel. B.v. als een atoom tot de gegevensbankinstantie behoort en niet behoort tot het herstel, behoort het ook tot de herstelde gegevensbankinstantie. Wanneer het wel behoort tot het herstel dan is het geen element meer van de gegevensbankinstantie. De status van een atoom p stellen we voor met het propositioneel symbool s_p. s_p is waar als $p \in \mathsf{Insert} \cup \mathsf{Retract}$.

Deze methode bestaat uit twee stappen. In een eerste stap worden de integriteitsbeperkingen $\mathcal{IC}$ eerst geïnstantieerd met een domein $\mathcal{DA}$. Dit domein $\mathcal{DA}$ bevat minstens de domeinelementen in de gegevensbankinstantie $\mathcal{D}$. Als $\mathcal{DA}$ eindig is, kunnen de geïnstantieerde integriteitsbeperkingen $\mathcal{IC}^{\mathcal{DA}}$ geconstrueerd worden. De geïnstantieerde integriteitsbeperkingen $\mathcal{IC}^{\mathcal{DA}}$ worden in een volgende stap getransformeerd door elk atoom te vervangen door

- $\neg s_p$ als $p \in \mathcal{D}$,
- s_p in het andere geval

De bekomen theorie wordt de *signed* theorie van de gegevensbank $\mathcal{DB}$ t.o.v. het domein $\mathcal{DA}$ genoemd. De modellen van deze theorie zijn exact de gezochte herstellingen van de gegevensbank. We hebben ook aangetoond dat voor de preferentiecriteria van de herstellingen een overeenkomstig criterium kan worden gedefinieerd op de modellen van de signed theoriën.

VOORBEELD 5 Beschouw de inconsistente gegevensbank van voorbeeld 4. Neem als domein $\mathcal{DA} = \{c_1, n_1, c_2, n_2, c_3, n_3\}$, de domeinelementen die voorkomen in de gegevensbankinstantie.

1. De geïnstantieerde integriteitsbeperkingen $\mathcal{IC}^{\mathcal{DA}}$ zijn

$$\left\{ \begin{array}{c} teaches(c_1, n_1) \wedge teaches(c_1, n_1) \rightarrow n_1 = n_1 \\ teaches(c_1, n_1) \wedge teaches(c_1, n_2) \rightarrow n_1 = n_2 \\ teaches(c_1, n_1) \wedge teaches(c_1, n_3) \rightarrow n_1 = n_3 \\ \vdots \\ teaches(c_3, n_3) \wedge teaches(c_3, n_3) \rightarrow n_3 = n_3 \end{array} \right\}$$

2. De signed theorie is dan

$$\left\{ \begin{array}{c} \neg s_{teaches(c_1,n_1)} \wedge \neg s_{teaches(c_1,n_1)} \rightarrow n_1 = n_1 \\ \neg s_{teaches(c_1,n_1)} \wedge s_{teaches(c_1,n_2)} \rightarrow n_1 = n_2 \\ \vdots \\ \neg s_{teaches(c_3,n_3)} \wedge \neg s_{teaches(c_3,n_3)} \rightarrow n_3 = n3 \end{array} \right\}$$

   Deze theorie kan vereenvoudigd worden: b.v. de eerste formule is voldaan en kan weggelaten worden.

De modellen van een signed theorie kunnen efficiënt berekend worden met behulp van verschillende systemen. We hebben enkele experimenten uitgevoerd met systemen voor vier verschillende computationele paradigma's. De beschouwde paradigma's zijn eindig domein constraint solving, modelgeneratie voor disjunctieve logische programma's, SAT solving en Quantified Boolean solving. Allemaal kunnen ze de herstelling t.o.v. setinclusie berekenen. Het berekenen van de herstellingen met minimale cardinaliteit wordt enkel ondersteund door de eerste twee paradigma's.

6.6 Besluit

Het integreren van informatie is een vaak voorkomend probleem, dat door de recente ontwikkelingen in de maatschappij aan belang wint. Daarom is het noodzakelijk dat goede technieken ontwikkeld worden om de verschillende problemen die optreden tijdens het samenvoegen, op te lossen. We hebben voor twee subproblemen oplossingen uitgewerkt. Een belangrijke eigenschap van onze aanpak is zijn flexibiliteit. Door de hoogniveaubeschrijving (ID-Logic) van het probleem kunnen kleine veranderingen van de probleemcontext vrij eenvoudig geïntegreerd worden. Door de declaratieve aanpak waarbij de probleemspecificatie gescheiden is van het computationele systeem, vereisen zulke veranderingen niet snel de ontwikkeling van een nieuw redeneersysteem.

Hoewel we geen uitgebreide experimentele evaluatie van het $\mathcal{A}$system in dit probleem domein hebben gemaakt, geloven we dat het $\mathcal{A}$system een bruikbare solver is voor dit probleem. Het kan gebruikt worden om de (meest geprefereerde) herstellingen van een inconsistente gegevensbank te berekenen. Ook de eigenschap om een verklaring te construeren voor een vraag is bijvoorbeeld belangrijk in de context van mediator gebaseerde systemen. Om meer precieze uitspraken te doen zijn echter meer experimenten noodzakelijk, liefst met realistische probleemgroottes.

Verder is er ook in dit domein nog heel wat toekomstig werk mogelijk. We denken o.a. aan het uitbreiden van de gepresenteerde technieken naar temporele en geo-spatiale gegevensbanken en het ontwikkelen van een methode voor het integreren van integriteitsbeperkingen.