

CHR^{rp}: Constraint Handling Rules with Rule Priorities

Leslie De Koninck Tom Schrijvers
Bart Demoen

Report CW 479, March 2007



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

CHR^{rp}: Constraint Handling Rules with Rule Priorities

Leslie De Koninck Tom Schrijvers
Bart Demoen

Report CW 479, March 2007

Department of Computer Science, K.U.Leuven

Abstract

We extend the Constraint Handling Rules language (CHR) with user-defined rule priorities. This language extension reduces the level of non-determinism that is inherent to the theoretical operational semantics of CHR, and gives a more high-level form of execution control compared to the refined operational semantics. We suggest some application areas. A formal operational semantics for the extended language, called CHR^{rp}, is given and its theoretical properties are discussed. We look at some issues with CHR^{rp} and discuss alternatives for rule priorities.

Keywords : Constraint Handling Rules, rule priorities, execution control.

CR Subject Classification : D.1.6 [Programming Techniques] Logic Programming, D.3.2 [Programming Languages] Language Classifications — Constraint and logic languages, D.3.3 [Programming Languages] Language Constructs and Features — Control structures

CHR^{FP}: Constraint Handling Rules with Rule Priorities

Leslie De Koninck*, Tom Schrijvers**, Bart Demoen

Department of Computer Science, K.U.Leuven, Belgium
{leslie,toms,bmd}@cs.kuleuven.be

Abstract We extend the Constraint Handling Rules language (CHR) with user-defined rule priorities. This language extension reduces the level of non-determinism that is inherent to the theoretical operational semantics of CHR, and gives a more high-level form of execution control compared to the refined operational semantics. We suggest some application areas. A formal operational semantics for the extended language, called CHR^{FP}, is given and its theoretical properties are discussed. We look at some issues with CHR^{FP} and discuss alternatives for rule priorities.

1 Introduction

Constraint Handling Rules (CHR) [12] is a rule based language, originally designed for the implementation of constraint programming facilities, but also more and more used as a general purpose programming language [18,15].

Algorithms that are described using rules, often require that certain rules fire before others, either for efficiency or for correctness. This requirement can be represented by adding priorities to rules. Under the theoretical operational semantics of CHR [9], derivations exist that satisfy such a priority scheme. This however, is generally not true for the refined operational semantics of CHR [9], which is used by all current CHR implementations. In the refined operational semantics, the textual order of the rules in the program determines which rule is tried next for the current “active” constraint, but no *global* rule priorities are supported.

In this paper, we extend CHR with user-defined rule priorities. The extended language, called CHR^{FP}, supports a flexible, high-level and declarative way of specifying execution strategies. It forms an intermediate step between the high-level, but also highly non-deterministic, theoretical operational semantics of CHR, and the low-level, mostly deterministic, refined operational semantics. It allows programmers to separate the logic and the control of their programs. Moreover, it makes programs that need fine-grained execution control, either for efficiency or for correctness, more concise.

Related Work Rule priorities are found in many rule based languages. They are used by production rule systems as part of conflict resolution. They have been introduced in term rewriting systems (“Priority Rewrite Systems” [2]) to ensure confluence. Recently, this idea was applied to term-graphs [6]. A bottom-up logic programming language with prioritized rules is presented in [13]. In [14] we present a translation scheme from this language to CHR^{FP}. Priorities are often used by Constraint (Logic) Programming systems. The finite domain solver of SICStus Prolog

* Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen)

** Post-Doctoral Researcher of the Fund for Scientific Research - Flanders (Belgium) (F.W.O. - Vlaanderen)

[7] supports two priority levels. The ECLⁱPS^e CLP system uses 12 priority levels that are shared between its solvers. These priorities can amongst others be used as constraint priorities in the Extended Constraint Handling Rules library (`ech`).

Overview In this paper, we present rule priorities as a language extension of CHR. In Section 2, we review the syntax and semantics of CHR. Section 3 presents motivational examples for the proposed language extension and Section 4 formalizes the syntax and semantics of the extended language, called CHR^{FP}. Some issues with CHR^{FP} as well as two alternatives for rule priorities, are discussed in Section 5. We conclude in Section 6.

2 Constraint Handling Rules

This section reviews the syntax and semantics of Constraint Handling Rules (CHR). For a more thorough introduction, see [12] or [17].

2.1 Syntax and Declarative Semantics

A constraint $c(t_1, \dots, t_n)$ is an atom of predicate c/n with t_i a host language value (e.g. a Herbrand term in Prolog) for $1 \leq i \leq n$. There are two types of constraints: built-in constraints and CHR constraints (also called user-defined constraints). The CHR constraints are solved by the CHR program whereas the built-in constraints are solved by an underlying constraint solver (e.g. the Prolog unification algorithm).

There are three types of Constraint Handling Rules: *simplification* rules, *propagation* rules and *simpagation* rules. They have the following form:

$$\begin{array}{ll} \mathbf{Simplification} & r @ \quad H^r \iff g \mid B \\ \mathbf{Propagation} & r @ \quad H^k \implies g \mid B \\ \mathbf{Simpagation} & r @ H^k \setminus H^r \iff g \mid B \end{array}$$

where r is the rule *name*, H^k and H^r are non-empty sequences of CHR constraints and are called the *heads* of the rule, the rule *guard* g is a sequence of built-in constraints, and the rule *body* b is a sequence of both CHR and built-in constraints. A program P is a set of CHR rules.

Let $vars(A)$ be the variables occurring in A and let $\exists_A F$ and $\forall_A F$ denote respectively $\exists X_1, \dots, \exists X_n F$ and $\forall X_1, \dots, \forall X_n F$ where $\{X_1, \dots, X_n\} = vars(F) \setminus vars(A)$. Let $H = \langle H^k, H^r, g \rangle$. A CHR program P forms a constraint theory consisting of the constraint theory of the built-in constraint solver in conjunction with one of the following formulas for each rule r in P (depending on its type):

$$\begin{array}{ll} \mathbf{Simplification} & \forall_B((g) \rightarrow (H^r \leftrightarrow \exists_H B)) \\ \mathbf{Propagation} & \forall_B((H^k \wedge g) \rightarrow (\exists_H B)) \\ \mathbf{Simpagation} & \forall_B((H^k \wedge g) \rightarrow (H^r \leftrightarrow \exists_H B)) \end{array}$$

Operationally, CHR constraints have multi-set semantics. To distinguish between different occurrences of syntactically equal constraints, CHR constraints are extended with a unique identifier. An identified CHR constraint is denoted by $c\#i$ with c a CHR constraint and i the identifier. We write $chr(c\#i) = c$ and $id(c\#i) = i$. An alternative declarative semantics for CHR based on intuitionistic linear logic, that amongst others takes into account this multi-set semantics, is given in [3].

2.2 The Theoretical Operational Semantics

The theoretical operational semantics of CHR, denoted ω_t , is given in [9] as a state transition system. A CHR execution state σ is represented as a tuple $\langle G, S, B, T \rangle_n$ where G is the goal, a multiset of constraints that need to be solved; S is the CHR constraint store, a set of identified CHR constraints; B is the built-in constraint store, a conjunction of built-in constraints; T is the propagation history, a set of tuples denoting the rule instances that have already fired; and n is the next free identifier, used to identify new CHR constraints. The transitions of ω_t are shown below.

1. **Solve** $\langle \{c\} \uplus G, S, B, T \rangle_n \xrightarrow{\omega_t}_P \langle G, S, c \wedge B, T \rangle_n$ where c is a built-in constraint.
2. **Introduce** $\langle \{c\} \uplus G, S, B, T \rangle_n \xrightarrow{\omega_t}_P \langle G, \{c\#n\} \cup S, B, T \rangle_{n+1}$ where c is a CHR constraint.
3. **Apply** $\langle G, H_1 \cup H_2 \cup S, B, T \rangle_n \xrightarrow{\omega_t}_P \langle C \uplus G, H_1 \cup S, \theta \wedge B, T \cup \{t\} \rangle_n$ where P contains a (renamed apart) rule

$$r @ H'_1 \setminus H'_2 \iff g \mid C$$

and a matching substitution θ such that $chr(H_1) = \theta(H'_1)$, $chr(H_2) = \theta(H'_2)$, $\mathcal{D} \models B \rightarrow \exists_B(\theta \wedge g)$, and $t = id(H_1) ++ id(H_2) ++ [r] \notin T$.

The **Solve** transition solves a built-in constraint from the goal, the **Introduce** transition inserts a new CHR constraint from the goal into the CHR constraint store, and the **Apply** transition fires a rule instance.

2.3 The Refined Operational Semantics

The refined operational semantics of CHR, denoted by ω_r , was introduced in [9] as a formalization of the execution mechanism of the current CHR implementations. While its initial purpose was descriptive, it has become a normative description of how CHR implementations should work. This is because many CHR programs are written with the ω_r semantics in mind and often they do not work correctly under the ω_t semantics (i.e. they are not confluent). In the Section 4, we present a more high-level alternative to the refined semantics that still allows considerable execution control.

The main concept of the ω_r semantics is that of the *active* constraint. The active constraint is a CHR constraint that is used as a starting point for finding fireable rule instances. To ensure that all fireable rule instances are eventually tried, all new CHR constraints become active after they are asserted. CHR constraints that have been active before and whose variables are affected by a new built-in constraint, are reactivated.

The active constraint tries rules in textual order until either it finds a fireable rule instance or all rules have been tried. When a rule instance fires, its body is processed from left to right. Every new CHR constraint that is processed, is activated as soon as it is inserted into the constraint store. Every new built-in constraint is solved for, and all affected CHR constraints are activated one by one before processing the next constraint in the body. After processing the rule body, the active constraint searches for the next fireable rule instance if it is not removed (i.e. it is part of the kept heads H^k of the rule that fired). Otherwise, processing resumes where it left when the constraint was activated.

3 Motivation and Examples

In this section, we show the usefulness of our proposed language extension for some typical problems and give some illustrating examples. We use CHR on top of Prolog for these examples, but the ideas are relevant for other host languages as well.

3.1 Constraint Propagators

Constraint solvers generally make use of constraint propagators which filter out inconsistent values from the constraint variables' domains. Efficient solvers use a priority system to make sure that constraint propagators that are computationally cheaper or are expected to have a big impact, are scheduled earlier [16,19].

CHR rules can be seen as templates for constraint propagators which can be instantiated by actual constraints.

Example 1. We represent a binary constraint C between two variables X and Y as $c(C, X, Y)$. The domain DX of a variable X is represented as $d(X, DX)$. The following rules implement two constraint propagators:

$$\begin{aligned} ac_1 @ c(C, X, Y), d(Y, DY) \setminus d(X, DX_0) <=> filter(DX_0, [C], [Y-DY], DX_1), \\ & DX_0 \setminus = DX_1 \mid d(X, DX_1) \text{ pragma priority}(1). \\ ac_2 @ c(C, X, Y), d(X, DX) \setminus d(Y, DY_0) <=> filter(DY_0, [C], [X-DX], DY_1), \\ & DY_0 \setminus = DY_1 \mid d(Y, DY_1) \text{ pragma priority}(1). \\ \\ pc_1 @ c(C_1, X, Y), c(C_2, Y, Z), c(C_3, X, Z), d(Y, DY), d(Z, DZ) \setminus d(X, DX_0) <=> \\ & filter(DX_0, [C_1, C_2, C_3], [Y-DY, Z-DZ], DX_1), \\ & DX_0 \setminus = DX_1 \mid d(X, DX_1) \text{ pragma priority}(2). \\ pc_2 @ c(C_1, X, Y), c(C_2, Y, Z), c(C_3, X, Z), d(X, DX), d(Z, DZ) \setminus d(Y, DY_0) <=> \\ & filter(DY_0, [C_1, C_2, C_3], [X-DX, Z-DZ], DY_1), \\ & DY_0 \setminus = DY_1 \mid d(Y, DY_1) \text{ pragma priority}(2). \\ pc_3 @ c(C_1, X, Y), c(C_2, Y, Z), c(C_3, X, Z), d(X, DX), d(Y, DY) \setminus d(Z, DZ_0) <=> \\ & filter(DZ_0, [C_1, C_2, C_3], [X-DX, Y-DY], DZ_1), \\ & DZ_0 \setminus = DZ_1 \mid d(Z, DZ_1) \text{ pragma priority}(2). \end{aligned}$$

In this example, the rules ac_1 and ac_2 implement arc consistency, and rules pc_1 , pc_2 and pc_3 implement path consistency. Because the latter is more costly, we assign it a lower priority. We use the *pragma*¹ *priority/1* to indicate the rule priorities. Smaller numbers denote higher priorities.

The consistencies are implemented by using the `filter/4` predicate which filters out the inconsistent values in the domain of one of the constraint variables, given the constraints in which it appears and the domains of the other involved variables.

At first sight it might appear that, given the textual order of the rules, the refined operational semantics of CHR ensures that the arc consistency rules are tried before the path consistency rules. The following situation shows that this is not always the case. Let X be a variable whose domain DX_0 has changed to DX_1 . The constraint $d(X, DX_1)$ becomes active and rule ac is tried. Let Y be the first variable whose domain DY_0 is changed into DY_1 after filtering using rule ac . The constraint $d(Y, DY_1)$ becomes active and rule ac is tried. Now, let all constraints in which Y appears, be arc consistent, then rule pc is tried with $d(Y, DY)$ as active constraint, whereas there still might be a constraint in which X appears and that is not arc consistent yet. Rule priorities ensure that these constraints are made arc consistent first. \square

3.2 Constraint Store Invariants

It is often desirable to impose certain representational invariants on the constraints in the CHR constraint store. An example of such an invariant is set semantics: no two syntactically equal constraints can exist in the constraint store. These invariants may be violated when asserting new constraints (both built-in and CHR) and we

¹ Pragmas are compiler directives. They instruct the compiler on *how* to compile a program.

can use special purpose CHR rules for restoring them. Such rules should fire before any rule that expects (some of) the invariants.

Example 2. Assume we want to check whether two graphs, G_1 and G_2 , are equal. We do this by removing those edges that are common to both graphs. If there are still edges left after reaching a fixed point, then the graphs are different. We represent the edges of graph G_1 and G_2 by the edge constraints $e_1/2$ and $e_2/2$ respectively. This gives us the following program:

```
s1 @ e1(X,Y) \ e1(X,Y) <=> true pragma priority(1).
s2 @ e2(X,Y) \ e2(X,Y) <=> true pragma priority(1).

rc @ e1(X,Y), e2(X,Y) <=> true pragma priority(2).
```

Edges obey set semantics, as implemented by the rules s_1 and s_2 . The rule rc (remove common) removes those edges that appear both in graph G_1 and in graph G_2 . Now consider the query:

```
?- e1(X,X), e2(X,Y), e2(Y,X), X = Y.
```

When executing this query using the refined operational semantics, ignoring the rule priorities, the $X = Y$ built-in constraint causes the sequential activation of all three CHR constraints. If the $e_1/2$ constraint is activated before any of the $e_2/2$ constraints, then rule rc fires before the s_2 rule is tried, which results in a final constraint store containing the $e_2(X,X)$ constraint which wrongfully indicates that the graphs are different.

We already mentioned in the introduction that rule priorities require that the highest priority rule for which a fireable rule instance exists, fires. This is a global notion in that it does not matter which constraints participate in the firing rule instance, and in particular, there is no concept of an active constraint. So using rule priorities, the set semantics rules will always be tried before the lower priority rule rc and when the highest priority fireable rule instance is one of priority 2 (or less), then there will be no two syntactically equal $e_1/2$ or $e_2/2$ constraints.

The above example *can* be implemented correctly using the refined semantics, but this leads to inefficient and unlogical code:

```
s1 @ e1(X,Y) \ e1(X,Y) <=> true.
s2 @ e2(X,Y) \ e2(X,Y) <=> true.
s3 @ e1(_,_), e1(X,Y) \ e1(X,Y) <=> true.
s4 @ e1(_,_), e2(X,Y) \ e2(X,Y) <=> true.
s5 @ e2(_,_), e1(X,Y) \ e1(X,Y) <=> true.
s6 @ e2(_,_), e2(X,Y) \ e2(X,Y) <=> true.

rc @ e1(X,Y), e2(X,Y) <=> true. □
```

3.3 Dynamic Rule Priorities

Rule priorities are called *dynamic* if they depend on (the arguments of) the constraints that form a rule instance.

Example 3 (Dijkstra's Shortest Path). The program below implements Dijkstra's shortest path algorithm using dynamic rule priorities.

```
d1 @ source(V) ==> dist(V,0) pragma priority(1).
d2 @ dist(V,D1) \ dist(V,D2) <=> D1 < D2 | true pragma priority(1).
d3 @ dist(V,D), e(V,C,U) ==> dist(U,D+C) pragma priority(D+2).
```

Here, an $e(V, C, U)$ constraint represents an edge from node V to node U with cost C . A `source/1` constraint fixes the source for the (single-source) shortest path algorithm. The algorithm computes `dist/2` constraints representing the distance of the shortest path from the source node to a given node. Rule d_1 states that the distance from the source node to itself is zero. Rule d_2 removes redundant `dist/2` constraints. Finally, rule d_3 “labels” the nodes connected to the node that is closest to the source node and for which this has not been done yet. The (dynamic) rule priority ensures that there exists no node closer to the source node for which this labeling has not been done yet. \square

Example 4 (Sudoku). The Sudoku solver from the CHR website keeps track of the number of possible values for each cell, and chooses a value from the most constrained cell first. To do so, the following code is used:

```
fillone(N), f(A,B,C,D,N,L) <=> member(V,L), f(A,B,C,D,V), fillone(1).
fillone(N) <=> N < 9 | fillone(N+1).
fillone(_) <=> true.
```

Initially, the store contains the constraint `fillone(1)`. The argument of this constraint is increased until a match is found and a rule fires. After the rule has fired, it is reset to 1. In CHR^{P} we can get the same effect using only one rule:

```
f(A,B,C,D,N,L) <=> member(V,L), f(A,B,C,D,V) pragma priority(N).  $\square$ 
```

4 CHR^P: CHR with Rule Priorities

CHR^{P} extends CHR with user-defined rule priorities. Compared to CHR under the refined operational semantics, it offers a high-level, flexible and declarative alternative for specifying the execution strategy of a CHR program. Uptil now, CHR programmers were forced to either write completely confluent programs that are correct under any execution strategy supported by the theoretical operational semantics of CHR, or, to take into account the refined operational semantics. The latter is useful for developers of CHR compiler, but is not suitable for a declarative form of execution control from the CHR programmer perspective. In this section, we introduce the syntax and semantics of CHR^{P} and look at some of its theoretical properties.

4.1 Syntax

The syntax of CHR^{P} is compatible with the syntax of (regular) CHR. A CHR^{P} simpagation rule looks as follows:

$$r @ H^k \setminus H^r \iff g | B \text{ pragma priority}(p)$$

where r , H^k , H^r , g and B are as defined in Section 2.1. Simplification and propagation rules have a similar structure. The rule *priority* p is an arithmetic expression for which holds that $\text{vars}(p) \subseteq (\text{vars}(H^k) \cup \text{vars}(H^r))$ (i.e. all variables in p also appear in the heads. A rule in which $\text{vars}(p) = \emptyset$ is called a *static* priority rule: its priority is known at compile time and equal for all rule instances. A rule in which $\text{vars}(p) \neq \emptyset$ is called a *dynamic* priority rule: its priority is only known at runtime and different rule instances of the same rule may fire at different priorities.

4.2 The Priority Semantics ω_p

We propose a formal operational semantics for CHR^{P} . It is called the priority semantics and denoted by ω_p . It consists of a refinement of the ω_t semantics with a minimal amount of determinism to support rule priorities. The ω_p semantics uses the same state representation as the ω_t semantics. Its transitions are shown below.

1. **Solve** $\langle \{c\} \uplus G, S, B, T \rangle_n \xrightarrow{\omega_p} \langle G, S, c \wedge B, T \rangle_n$ where c is a built-in constraint.
2. **Introduce** $\langle \{c\} \uplus G, S, B, T \rangle_n \xrightarrow{\omega_p} \langle G, \{c\#n\} \cup S, B, T \rangle_{n+1}$ where c is a CHR constraint.
3. **Apply** $\langle \emptyset, H_1 \cup H_2 \cup S, B, T \rangle_n \xrightarrow{\omega_p} \langle C, H_1 \cup S, \theta \wedge B, T \cup \{t\} \rangle_n$ where P contains a rule of priority p of the form

$$r @ H_1' \setminus H_2' \iff g \mid C \text{ pragma priority}(p)$$

and a matching substitution θ such that $\text{chr}(H_1) = \theta(H_1')$, $\text{chr}(H_2) = \theta(H_2')$, $\mathcal{D} \models B \rightarrow \exists_B(\theta \wedge g)$ and $t = \text{id}(H_1) ++ \text{id}(H_2) ++ [r] \notin T$. Furthermore, no rule of priority p' and substitution θ' exists with $\theta'(p') < \theta(p)$ for which the above conditions hold.

The ω_p semantics restricts the applicability of the **Apply** transition with respect to the ω_t semantics. The **Solve** and **Introduce** transitions are unchanged. The **Apply** transition is only applicable to states with an empty goal. If it fires a rule instance of priority p in state σ , there exists no apply transition $\sigma \xrightarrow{\omega_t} \sigma'$ under ω_t that fires a rule instance of a higher priority.

4.3 Correspondence

We now look more formally to the correspondence between the ω_t and ω_p semantics. In particular, we show that every ω_p derivation is also a derivation under ω_t (Theorem 1). We then show how the ω_p semantics respects the rule priorities (Theorem 2). Finally, for CHR^{P} programs in which all rule priorities are equal, we show that every ω_t derivation corresponds to an ω_p derivation (Theorem 3).

Theorem 1. *Every derivation D under ω_p , is also a derivation under ω_t . If a state σ is a final state under ω_p , then it is also a final state under ω_t .*

Proof. The first part of the theorem holds because the ω_p only adds restrictions to the applicability of ω_t transitions. For the second part, suppose that state σ is a final state under ω_p , but not under ω_t . The only transition applicable under ω_t must be the **Apply** transition, since the **Solve** and **Introduce** transitions are equal in both semantics. This means that the goal must be empty.

From all **Apply** transitions that are applicable in state σ under ω_t , we can choose the one that fires the highest priority rule instance. It is clear that this transition is also valid under ω_p , which contradicts our assumption. This proves the second part of the theorem. \square

Theorem 2. *If an **Apply** transition applies on a state σ under ω_p , firing a rule of priority p , then there exists no derivation under ω_t and starting in σ that fires a rule of a higher priority first.*

Proof. When applying the **Apply** transition on state σ under ω_p , it fires the highest priority rule that can fire given the current built-in store, CHR store and propagation history. If there exists a derivation starting in σ under ω_t that fires a higher priority rule first, then there must be a **Solve** or **Introduce** transition that updates the built-in store or CHR store, that needs to be applied first. Since the **Apply** transition under ω_p requires that the goal is empty, this is not possible, so no such derivation can exist. \square

For every state $\sigma = \langle G, S, B, T \rangle_n$, there exists a derivation $\sigma \xrightarrow{\omega_p^*} \sigma^*$ where $\sigma^* = \langle \emptyset, S \cup S', B \wedge B', T \rangle_{n+|S'|}$ with $G = B' \uplus \text{chr}(S')$, B' a multi-set of built-in constraints, S' a set of identified CHR constraints and $|S'|$ the number of elements in S' . The derivation is formed by solving all built-in constraints in the goal G and introducing all CHR constraints in it. We call state σ^* a *normalization* of σ . There are $|S'|!$ such normalizations, one for each order in which the CHR constraints of the goal are introduced.

Theorem 3. *For a given CHR^{FP} program P in which all priorities are equal, it holds that for every non-failing derivation D under ω_t , if $\sigma_1 \xrightarrow{\omega_t} \sigma_2 \in D$ then for every normalization σ_2^* of σ_2 , there exists a normalization σ_1^* of σ_1 such that $\sigma_1^* \xrightarrow{\omega_p^*} \sigma_2^* \in D'$. If a state σ is a final state under ω_t , then it is also a final state under ω_p .*

Proof. Given $\sigma_1 \xrightarrow{\omega_t} \sigma_2 \in D$. We look at each of the three possible transitions:

1. **Solve** $\sigma_1 = \langle \{c\} \uplus G, S, B, T \rangle_n$ and $\sigma_2 = \langle G, S, B \cup \{c\}, T \rangle_n$. Clearly all normalizations of σ_1 and σ_2 are equal.
2. **Introduce** $\sigma_1 = \langle \{c\} \uplus G, S, B, T \rangle_n$ and $\sigma_2 = \langle G, S \cup \{c\#n\}, B, T \rangle_{n+1}$. All normalizations of σ_2 are also normalizations of σ_1 .
3. **Apply** $\sigma_1 = \langle G, H^r \cup S, B, T \rangle_n$ and $\sigma_2 = \langle C \uplus G, S, \theta \wedge B, T' \rangle_n$. In any normalization of σ_1 , the same rule instance can fire because introducing CHR constraints to the store nor solving built-in constraints from the goal can prevent a rule instance from being fireable.² So we have for every normalization σ_1^* of σ_1 that $\sigma_1^* = \langle \emptyset, H^r \cup S \cup S', B \wedge B', T \rangle_{n'} \xrightarrow{\omega_p^*} \langle C, S \cup S', B \wedge B', T' \rangle_{n'} = \sigma_2'$. It is easy to see that every normalization of σ_2 corresponds to a normalization of σ_2' .

We conclude that for every transition $\sigma_1 \xrightarrow{\omega_t} \sigma_2$, there exists a corresponding derivation $\sigma_1^* \xrightarrow{\omega_p^*} \sigma_2^*$ for every normalization σ_2^* of σ_2 .

Theorem 3 implies that for CHR^{FP} programs in which all rule priorities are equal, every execution strategy under ω_t is consistent with ω_p , and so such programs can be executed using the refined operational semantics as implemented by existing CHR implementations. While such CHR^{FP} programs are obviously degenerate, execution under the ω_r semantics or slightly altered versions of it, is possible a larger class of programs.

Let B be the body of a rule instance of priority p . We can execute B using the refined operational semantics if

1. The constraints in B do not “observe” eachother. This means that the derivation of a constraint $c \in B$ does not fire rule instances that contain a CHR constraint from $B \setminus \{c\}$ or depends on a built-in constraint from $B \setminus \{c\}$.
2. All rule instances in the derivation of a constraint $c \in B$ have a priority higher or equal to p and lower or equal to the lowest priority rule instance fired during the derivation of all previous constraints in B .
3. The same conditions hold for the rule bodies of the fired rule instances.

Example 5. Consider the following CHR^{FP} program and initial goal $\{e, a\}$

```
r1 @ e \ b <=> true pragma priority(1).
r2 @ c \ e <=> true pragma priority(2).
r3 @ b      <=> fail pragma priority(2).
r4 @ c      <=> true pragma priority(3).
r5 @ a      <=> b, c pragma priority(3).
```

² We do not consider non-monotone guards like Prolog’s `var/1`. They are not allowed in pure CHR.

For the body of rule **r5** we can use the refined operational semantics as it will fire the following sequence of rules: [**r1**, **r2**, **r4**] which is consistent with ω_p . For the goal $\{\mathbf{a}\}$ we get [**r3**] which is also consistent with ω_p . \square

Alternative conditions can be found under which (parts of) a CHR^{IP} program can be executed using the ω_r semantics.

4.4 Confluence

In [1], a criterion is given for deciding whether a (terminating) CHR program is confluent under the ω_t semantics.

Definition 1 (Joinability). *Two states σ_1 and σ_2 are joinable given program P and operational semantics ω if $\sigma_1 \xrightarrow{\omega}_P^* \sigma_1^*$ and $\sigma_2 \xrightarrow{\omega}_P^* \sigma_2^*$ with σ_1^* and σ_2^* variants with respect to the variables of σ .*

Two execution states are variants with respect to a set of variables \mathcal{V} if they are identical module renaming of constraint identifiers and renaming of the variables in \mathcal{V} , after removing redundant propagation history tuples.

Definition 2 (Confluence). *A CHR program P is confluent under operational semantics ω if for all states σ , σ_1 and σ_2 : if $\sigma \xrightarrow{\omega}_P^* \sigma_1$ and $\sigma \xrightarrow{\omega}_P^* \sigma_2$ then σ_1 and σ_2 are joinable.*

Confluence follows from *local* confluence if P terminates. Under ω_t , local confluence can be proven by checking all *critical pairs* for joinability. A critical pair consists of two *minimal* states σ_1 and σ_2 which follow from firing respectively rule instance $\theta_1(r_1)$ and $\theta_2(r_2)$ in common ancestor state σ such that $\theta_2(r_2)$ cannot fire in σ_1 and $\theta_1(r_1)$ cannot fire in σ_2 . Clearly, under ω_p , a critical pair only follows from firing rule instances with equal priority.

Under ω_t , if a rule instance $\theta(r)$ is fireable in a state $\langle G, S, B, T \rangle_n$, then it is also fireable in any (non-failed) “larger” state $\langle G \uplus G', S \cup S', B \cup B', T \setminus T' \rangle_{n'}$. This no longer holds under ω_p because adding CHR or built-in constraints to the store or removing propagation history tuples, can cause a *higher priority* rule instance to become fireable. A similar problem was found in [8] for the refined operational semantics, although its exact cause was not clearly identified.

Example 6. Consider the following example, borrowed from [8] and extended with rule priorities:

```

r1 @ p, q(_) <=> r pragma priority(1).
r2 @ q(_), q(_) \ r <=> true pragma priority(2).
r3 @ r \ q(_) <=> true pragma priority(3).
r4 @ r <=> true pragma priority(4).

```

A critical pair for rule **r1** is $(\langle \{r\}, \{q(1)\#1\}, true, T_1 \rangle_{n_1} \langle \{r\}, \{q(2)\#2\}, true, T_2 \rangle_{n_2})$ with common ancestor state $\langle \emptyset, \{q(1)\#1, q(2)\#2, p\#3\}, true, T \rangle_n$. Both states further derive into $\langle \emptyset, \emptyset, true, T' \rangle_{n'}$ and so it seems that there is confluence. However, given the initial goal $\{p, q(1), q(2), q(3)\}$ we get the final stores $\{q(1), q(2)\}$, $\{q(2), q(3)\}$ or $\{q(1), q(3)\}$. The problem is that in a minimal state, rule **r2** is not applicable and rules **r3** and **r4** can fire. In a larger state, **r2** may become applicable and cause rules **r3** and **r4** to be not applicable anymore. \square

Clearly, it is not sufficient to look at minimal states only. We can limit the number of potential states though by noting that in the ancestor state of a critical pair, no higher priority rule instance could fire. A practical confluence test is outside the scope of this paper.

4.5 Examples

We illustrate the different behavior of the ω_p semantics and the ω_r semantics on some small examples. The first example shows that rule priorities are ‘stronger’ than rule order under the ω_r semantics.

Example 7. Consider the following program:

```
r1 @ a ==> write('rule 1\n'), b pragma priority(1).
r2 @ a, b ==> write('rule 2\n') pragma priority(2).
r3 @ a <=> write('rule 3\n') pragma priority(3).
r4 @ a, b ==> write('rule 4\n') pragma priority(4).
```

Using the refined operational semantics, the rule priority declarations are ignored. For the initial goal **a**, we get the following output:

ω_r Semantics	ω_p Semantics
rule 1	rule 1
rule 2	rule 2
rule 4	rule 3
rule 3	

In the ω_p semantics, constraint **a** is removed by rule **r₃** before rule **r₄** is tried. This causes rule **r₄** to be not applicable anymore. There is no rule ordering that can cause rule **r₃** to be fired after rule **r₂** but before rule **r₄** in the ω_r semantics. \square

The following two examples illustrate the non-determinism in the ω_p semantics.

Example 8. Consider the following program:

```
r1 @ a(X) ==> write(r1:X), n1 pragma priority(1).
r2 @ a(X) ==> write(r2:X), n1 pragma priority(2).
```

For the initial goal **a(1)**, **a(2)**, we get the following output:

ω_r Semantics	ω_p Semantics			
r1:1	r1:1	r1:1	r1:2	r1:2
r2:1	r1:2	r1:2	r1:1	r1:1
r1:2	r2:1	r2:2	r2:1	r2:2
r2:2	r2:2	r2:1	r2:2	r2:1

Here the non-determinism is caused by the existence of different rule instances for the same rule. \square

Example 9. Consider the following program:

```
r1 @ a ==> write('rule 1\n') pragma priority(1).
r2 @ a ==> write('rule 2\n') pragma priority(1).
```

In this example, rules **r₁** and **r₂** have an equal priority. For the initial goal **a**, we get the following output:

ω_r Semantics	ω_p Semantics	
rule 1	rule 1	rule 2
rule 2	rule 2	rule 1

Here, the non-determinism is caused by the existence of different rules with equal priority. \square

If there are $\mathcal{O}(n)$ priority classes and $\mathcal{O}(m)$ rule instances in each priority class, then there are $\mathcal{O}((m!)^n)$ orderings of the rule firings.

5 Discussion

In this section, we discuss some issues of CHR^{P} and look at alternatives and extensions for rule priorities.

5.1 Issues

In this subsection, we show that certain programming patterns that are often used in CHR programs based on the refined operational semantics, lead to problems when using the priority semantics. We show how alternative formulations lead to the desired result using rule priorities.

Set Semantics and the Propagation History Consider the following example:

```
r1 @ a \ a <=> true pragma priority(1).
r2 @ a ==> a pragma priority(2).
```

and the initial goal $G = \{a\}$. Under the refined operational semantics, ignoring rule priorities, this program terminates after two rule firings: first rule **r2** fires, adding a new constraint **a**, which is then removed by rule **r1**. Under the priority semantics, it is possible that not the newly added occurrence of **a** is removed, but instead the one that was in the initial goal. The new occurrence has no propagation history tuple for rule **r2** and so the rule can fire again. If rule **r1** always removes the oldest occurrence of **a**, the program does not terminate.

We introduce a new constraint **new_a** that is removed if a constraint **a** already exists, and replaced by a new **a** constraint otherwise. The following program terminated under the priority semantics for the initial goal $G = \{a\}$.

```
r4 @ a \ new_a <=> true pragma priority(1).
r5 @ new_a <=> a pragma priority(2).
r6 @ a ==> new_a pragma priority(3).
```

Non-ground constraints can be made syntactically equal by unification. In this case, the approach presented does not work, but on the other hand, the refined operational semantics can then also not guarantee termination.

```
r7 @ a(X) \ a(X) <=> true pragma priority(1).
r8 @ a(X) ==> a(Y), X=Y pragma priority(2).
```

CHR Constraints in Guards Although it is strictly speaking not allowed to use CHR constraints in guards of rules, many useful programs do this anyway. In particular, it is often used to detect the absence of a constraint, as in the following example:

```
r1 @ a(X) \ b(Y) <=> no_c(X) | d(X,Y).
r2 @ c(X) \ no_c(X) <=> fail.
r3 @ no_c(_) <=> true.
```

The **no_b** constraint acts as an ask constraint. In particular, if asserting it succeeds, neither the built-in constraint store, nor the CHR constraint store are changed. In practical implementations based on the the refined operational semantics, such a constraint is solved for immediately, and so it does not really pose a problem. It is not clear how a CHR^{P} implementation can support CHR constraints in the guard. If they are processed as any other constraint in the body, then it would be scheduled and the guard might incorrectly be assumed to be implied by the built-in store.

The following code essentially does the same as the example, but now without CHR constraints in the guard.

```

r4 @ b(X) <=> b_id(X,_) pragma priority(1).
r5 @ a(X), b_id(Y,Id) ==> r1_instance(X,Y,Id) pragma priority(3).
r6 @ c(X) \ r1_instance(X,_,_) <=> true pragma priority(1).
r7 @ r1_instance(X,Y,Id), b(_,Id) <=> d(X,Y) pragma priority(2).

```

We introduce a unique identifier argument (a fresh variable) to the removed constraints of the original rule **r1**. This is because only if the rule fires, they are removed and the decision to fire or not depends on whether rule **r6** or **r7** fires. Consider that we do not use identifiers, but instead remove a syntactically equal constraint as shown below.

```

r8 @ a(X), b(Y) ==> r1_instance(X,Y) pragma priority(3).
r9 @ c(X) \ r1_instance(X,_) <=> true pragma priority(1).
r10 @ r1_instance(X,Y), b(Y) <=> d(X,Y) pragma priority(2).

```

Now rule **r10** can remove a constraint **b/1** for which rule **r8** does not have a propagation history tuple, while leaving a syntactically equal **b/1** constraint in the store for which such a tuple does exist. With an initial goal $\{a(1), b(2), b(2)\}$ it depends on the matching order whether one or both **b/1** constraints are removed.

We note that neither rule **r1** nor rule **r5** triggers when a **c/1** constraint is removed and so only a passive form of negation as absence is supported. For more on negation as absence in CHR, see [23].

Sequential Host Language Statements Rule bodies often contain host language statements that are not constraints. Amongst others, this is the case for IO calls. Another example is the Prolog **is/2** predicate which throws an exception if the second argument (right hand side) is not ground. Under the refined operational semantics, rule bodies are processed from left to right and all constraints are solved for as soon as they are encountered. In contrast, under the priority semantics, the order in which different constraints in a rule body are processed is not determined and CHR constraints are only introduced into the CHR constraint store (i.e. they are not solved immediately).

We can ensure correct behavior of these host language statements by encapsulating them into CHR constraints. Consider the following rule:

```

r1 @ a(X) <=> write('Give a number'), read(Y), Z is X - Y, b(Z).

```

The **write/1** statement should be executed before the **read/1** statement and the call to **is/2** should be delayed until $X - Y$ is ground.

```

r2 @ a(X) <=> io(yes,write('Give a number'),Next),
             io(Next,read(Y),_), safe_is(Z,X - Y),
             b(Z) pragma priority(3).
r3 @ io(yes,Call,Done) ==> call(Call) pragma priority(1).
r4 @ io(yes,Call,Done) <=> Done = yes pragma priority(2).
r5 @ safe_is(X,Y) <=> ground(Y) | X is Y pragma priority(1).

```

IO calls are encapsulated into $io(Ready, Call, Done)$ constraints where *Ready* equals **yes** if the IO call is ready to be executed, *Call* is the encapsulated IO call and *Done* equals **yes** if the IO call is done executing. Rule **r3** fires if an encapsulated IO call is ready to be executed. ($Ready = yes$). Its body only contains the encapsulated call. Rule **r4** fires after rule **r3** has fired, and indicates that the IO call has been executed by instantiating *Done* to **yes**. This allows the next IO call to be executed. The **is/2** call is encapsulated in a **safe_is/2** constraint. It is called by rule **r5** as soon as its second argument is ground.

5.2 Starvation and Priority Aging

Under the priority semantics, a given rule instance does not fire as long as there exist a higher priority fireable rule instance. To prevent that a rule instance has to “wait” very long, or even never gets to fire (“starvation”), we can increase its priority over time. In operating systems this is called priority aging.

We can extend the rule priority declaration with a function that updates the actual priority of a given rule instance, and a declaration of how often this update should happen (expressed in number of rule firings). For example:

```
| r1 @ a(X) <=> b(X) pragma priority(X,update(Old,Old-1),10).
```

which means that a rule instance of rule **r1** initially has priority X and this priority increases by one every 10 rule firings. In the term $\text{update}(Old,New)$, the new priority New of a rule instance is a function of its old priority Old . When a rule instance has priority 1, its priority cannot be increased anymore.

The main difficulty with this approach is determining when a rule instance becomes fireable. This seems to be only possible if we use an eager matching technique like in the RETE algorithm [10], where all fireable rule instances are eagerly generated and kept in memory. This approach is costly memory-wise. If firing a rule instance invalidates other rule instances, then a lot of useless computation is performed. Also, traversing all rule instances to update their priority, including the cost of updating the internal priority queue, can be expensive.³

We can emulate the proposed extension of rule priorities by using a source transformation. Consider the following, somewhat more complex, rule:

```
| r2 @ a(X) \ b <=> X mod 2 := 1 |
   | c(X) pragma priority(X,update(Old,Old-1),10).
```

We add a unique identifier as extra argument to all CHR constraints and transform the rule as follows:

```
| r3 @ a(X,Id1), b(Id2) ==> X mod 2 := 1 |
   | r2_instance(X,1,Id1,Id2) pragma priority(1).
r4 @ update \ r2_instance(Old,10,Id1,Id2) <=> Old > 1 |
   | r2_instance_passive(Old-1,1,Id1,Id2) pragma priority(1).
r5 @ update \ r2_instance(Old,Step,Id1,Id2) <=> Step < 10 |
   | r2_instance_passive(Old,Step+1,Id1,Id2) pragma priority(1).
r6 @ update <=> true pragma priority(2).
r7 @ r2_instance_passive(P,Step,Id1,Id2) <=>
   | r2_instance(P,Step,Id1,Id2) pragma priority(3).
r8 @ a(X,Id1) \ b(Id2), r2_instance(P,Step,Id1,Id2) <=>
   | c(X,_), update pragma priority(P+3).
r9 @ r2_instance(P,_,_,_) <=> true pragma priority(P+4).
```

Rule **r3** fires when a fireable rule instance is found. It generates a constraint representing this instance with its initial priority and a step count that denotes how many rule firings have taken place since the last update of priority. Rule instance priorities are updated after each rule firing by the `update` constraint. This constraint is added to the body of each rule (see rule **r8**). Rule **r4** increases the priority of a rule instance of rule **r2** whenever the step count equals 10 and the instance does not have the highest priority yet ($Old > 1$). Rule **r5** increases the step count by one if it is less than 10. Both rules **r4** and **r5** generate a passive version of the rule instance to ensure that it is only updated once after every rule firing. Rule **r6** removes the

³ When using Fibonacci heaps [11] as priority queue, the (amortized) cost of increasing the priority of an item is constant.

`update` constraint after all rule instances have been updated and rule `r7` converts the passive versions of the rule instances into active versions which are updated as soon as a new `update` constraint is asserted, i.e. after the next rule firing. Rule `r8` fires a rule instance if it has the highest priority and all head constraints are still in the store and rule `r9` removes a rule instance if some of the head constraints are not in the store anymore.

The above approach assumes that guards are monotone: once a guard is implied by the built-in constraint store, this remains true in all later states. It is clear that aging should only be used if rule priorities are used for efficiency and not if they are used for correctness of the program.

5.3 Alternatives for Rule Priorities

In this subsection, we discuss two alternatives for rule priorities that also allow for execution control. These are constraint priorities and execution in phases.

Constraint Priorities CHR constraints can often be divided into (active) operation constraints and (passive) data constraints (see for example the CHR implementation of the union-find algorithm in [18] and Fibonacci heaps in [20]). This is in particular the case when using CHR as a general purpose programming language instead of a language to describe constraint solvers in. Often such programs do not have a declarative reading in classical logic, but do have one in intuitionistic linear logic [3].

By using constraint priorities, we can prioritize certain operation constraints and hence certain operations. The priority of a constraint depends on its type and potentially also on its arguments. An obvious semantics for constraint priorities is the following: if in a given execution state, c is the highest priority constraint for which a fireable rule instance exists, then this constraint (or a constraint with equal priority) must participate in the next rule firing.

In the following rule, let p_1, \dots, p_n be the constraint priorities of constraints $c_1(\bar{X}), \dots, c_n(\bar{X})$ respectively and let lower numbers denote higher priorities.

$$c_1(\bar{X}_1), \dots, c_n(\bar{X}_n) \iff g \mid B$$

We can use rule priorities to get the same execution strategy as proposed for constraint priorities:

$$c_1(\bar{X}_1), \dots, c_n(\bar{X}_n) \iff g \mid B \text{ pragma priority}(\min(p_1, \dots, p_n))$$

Clearly, constraint priorities are subsumed by rule priorities. The opposite does not hold, as the following simple example illustrates.

```
r1 @ a(X) <=> b(X) pragma priority(1).
r2 @ a(X) <=> c(X) pragma priority(X).
```

We can of course add an extra head to each rule, whose constraint priority equals the rule priority, as shown below:

```
r3 @ r3_priority \ a(X) <=> b(X).
r4 @ r4_priority(X) \ a(X) <=> c(X).
```

where the constraint `r3_priority` has priority 1 and the constraint `r4_priority(X)` has priority X . However, these constraints have to be asserted first, which is feasible for static priority rule `r3`, but not for dynamic priority rule `r4` as there are infinitely many `r4_priority/1` constraints.

The Extended Constraint Handling Rules library (`ech`) of the ECLⁱPS^e Constraint Logic Programming system [22] supports a form of static constraint priorities. Similar to under the refined semantics, it is based on the concept of an *active* constraint. If a new CHR constraint is asserted in the body of a rule, it becomes active if it has a higher priority than the current active constraint, and otherwise it is scheduled for activation at its own (lower) priority. If a built-in constraint wakes up a set of CHR constraints, then these are activated from highest to lowest priority as long as their priority is higher than that of the current active constraint, and scheduled for activation otherwise. In `ech`, a CHR constraint has a priority between 1 and 11. These priorities can be specified absolutely, or relative to the priority of the CHR solver (which is 9 by default). The priority system is shared with other constraint solver libraries.

Example 10. Consider the following CHR program:

```
:- constraints
    a/1:at_absolute_priority(1),
    b/0:at_absolute_priority(2),
    c/1:at_absolute_priority(3),
    d/0:at_absolute_priority(4),
    e/1:at_absolute_priority(5).

r1 @ d ==> c(X), a(X), e(X), X = 1, b.

r2 @ a(1) <=> write(a).
r3 @ b      <=> write(b).
r4 @ c(1) <=> write(c).
r5 @ d      <=> write(d).
r6 @ e(1) <=> write(e).
```

and query “?- d.”. Constraint `d` becomes active and fires rule `r1`. Constraints `c(X)`, `a(X)`, `e(X)` are all (sequentially) inserted into the constraint store. Of these constraints, `c(X)` and `a(X)` are activated as soon as they are asserted, whereas `e(X)` is scheduled for activation at priority 5. The built-in constraint `X = 1` wakes up `a(1)`, `c(1)` and `e(1)` in priority order: `a(1)` fires rule `r2` and is then removed; `c(1)` fires rule `r4` and is removed; `e(1)` is (again) scheduled for activation at priority 5. Then `b` is asserted and becomes active immediately. It fires rule `r3` and is removed. Now the body of rule `r1` has been processed completely and constraint `d` searches for the next fireable rule instance. This next rule instance is an instance of `r5`. It fires and `d` is removed. Finally, `e(1)` is activated, fires rule `r6` and is removed. The (screen) output of this query hence is “`acbde`”.

The constraint priorities behave somewhat unlogical in that a constraint becomes active as soon as it has a higher priority than the current active constraint, while there might be higher priority constraints in the remaining part of the rule body. If rule bodies were processed completely before activating a higher priority constraint, the example program would output “`abcde`”. □

Execution in Phases Several rule based algorithms partition their rules into phases. Only the rules of the current phase are allowed to fire and the current phase changes either after firing a rule, or when no rule is applicable in it. A similar idea is used in term rewriting systems. In TAMPR [4], a set of rewrite rules is partitioned into a sequence of subsets of these rules. All rules in a given subset are applied until reaching a fixpoint, after which the same is done for the next subset. Stratego [21] uses a similar system where rewrite rules are divided into *stages*. A single rule can belong to multiple stages.

Example 11 (Miss Manners). The Manners program is a well known benchmark for production rule systems [5]. It has 5 phases: `start`, `assign_seats`, `make_path`, `check_done` and `print_results`. The following table shows the phase transitions.

Current Phase	Rule Name	Next Phase
<code>start</code>	<code>assign_first_seat</code>	<code>assign_seats</code>
<code>assign_seats</code>	<code>find_seating</code>	<code>make_path</code>
<code>make_path</code> <code>make_path</code>	<code>make_path</code> <code>path_done</code>	<code>make_path</code> <code>check_done</code>
<code>check_done</code> <code>check_done</code>	<code>are_we_done</code> <i>(default)</i>	<code>print_results</code> <code>assign_seats</code>
<code>print_results</code>	<code>print_results</code>	<code>print_results</code>

The code below implements the Manners program. We use the *pragma phase/2* with first argument the rule phase and second argument the next phase (which becomes the current phase after firing the rule). A default phase transition (when no rule applies in the current phase) is represented by a *phase/2* fact with similar semantics.

```

assign_first_seat @ guest(Name,_,_) \ count(Cnt) <=>
    seating(1,Name,Name,1,Cnt,0,yes), path(Cnt,Name,1),
    count(Cnt + 1) pragma phase(start, assign_seats).

find_seating @ seating(_,_,Name,Seat,Id,Pid,yes),
    guest(Name,Sex1,Hobby), guest(Guest,Sex2,Hobby) \ count(Id) <=>
    Sex1 \== Sex2, not_path(Id,Guest), not_chosen(Id,Guest,Hobby) |
    seating(Seat,Name,Guest,Seat + 1,Cnt,Id,no), count(Cnt + 1),
    path(Cnt,Guest,Seat + 1), chosen(Id,Guest,Hobby)
    pragma phase(assign_seats, make_path).

make_path @ seating(_,_,_,_,Id,Pid,no), path(Pid,Name,Seat) ==>
    not_path(Id,Name) | path(Id,Name,Seat)
    pragma phase(make_path, make_path).

path_done @ seating(Seat1,Name1,Name2,Seat2,Id,Pid,no) <=>
    seating(Seat1,Name1,Name2,Seat2,Id,Pid,yes)
    pragma phase(make_path, check_done).

are_we_done @ last_seat>LastSeat), seating(_,_,_,LastSeat,_,_,_) ==>
    true pragma phase(check_done, print_results).

print_results @ seating(_,_,_,Seat2,Id,_,_), last_seat(Seat2) \
    path(Id,Name,Seat) <=> write(Name:Seat), nl
    pragma phase(print_results, print_results).

phase(check_done, assign_seats).

```

We need two auxiliary constraints to implement negation as absence:

```

path(X,Y,_) \ not_path(X,Y) <=> fail.
not_path(_,_) <=> true.

chosen(X,Y,Z) \ not_chosen(X,Y,Z) <=> fail.
not_chosen(_,_,_) <=> true.

```

□

It is possible to implement rule priorities using phases. For static priorities, this is fairly straightforward. Every rule belongs to a phase corresponding to its priority. After a rule fires, the phase changes to the highest priority phase. When no rule is applicable in the current phase, it is changed to the next (lower priority) phase. Things become more complicated when dynamic rule priorities are involved. In this case, a rule belongs to a dynamic phase based on the arguments of the constraints involved in its instances. It remains unclear how the phase changes should be modeled as in theory there are infinitely many phases. In any case, while it seems possible to model rule priorities using phases, it will require a highly specialized implementation to be efficient.

Phases offer a more flexible form of execution control compared to rule priorities, and so it proves to be difficult to implement phases using priorities. The above discussion shows that the opposite is possible, but has a considerable performance penalty if rule priorities are not treated in a specialized way.

6 Conclusions

In this paper, we have proposed to extend the Constraint Handling Rules language with user-defined rule priorities. The extended language, called CHR^{P} , supports a high-level, flexible and declarative form of execution control. It bridges the gap between using the high-level but also highly non-deterministic theoretical operational semantics of CHR, and programming with the refined operational semantics in mind. Clearly, the textual order of the rules in a program, combined with flag constraints and similar constructs, is not a good way to specify the execution strategy of a program. In contrast, rule priorities allow a clear separation between the logic and the control of a CHR program and is therefore a much better alternative.

We have formalized the syntax and semantics of CHR^{P} and investigated its theoretical properties. We looked at some issues that come up when using certain programming patterns from (regular) CHR and we compared rule priorities with constraint priorities and execution in phases. In future work, we plan to work on an efficient implementation of CHR^{P} , create a practical confluence test, and more formally define which class of CHR^{P} programs can be (partially) executed using the refined operational semantics of CHR.

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *3rd Intl. Conf. Principles and Practice of Constraint Programming*, volume 1330 of *LNCS*, pages 252–266, 1997.
2. J. Baeten, J. Bergstra, and J. W. Klop. Term rewriting systems with priorities. In *2nd Intl. Conf. Rewriting Techniques and Applications*, volume 256 of *LNCS*, pages 83–94, 1987.
3. H. Betz and T. W. Frühwirth. A linear-logic semantics for constraint handling rules. In *11th Intl. Conf. Principles and Practice of Constraint Programming*, volume 3709 of *LNCS*, pages 137–151, 2005.
4. J. Boyle, T. Harmer, and V. Winter. The TAMPR program transformation system: Design and applications. In *Modern Software Tools for Scientific Computing*. Birkhauser, 1997.
5. D.A. Brant, T. Grose, B. Lofaso, and D.P. Miranker. Effects of database size on rule system performance: Five case studies. In *17th Intl. Conf. Very Large Data Bases*, 1991.
6. R. Caferra, R. Echached, and N. Peltier. Rewriting term-graphs with priority. In *8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 109–120, 2006.

7. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *9th Intl. Symposium on Programming Languages: Implementation, Logics and Programs*, volume 1292 of *LNCS*, pages 191–206, 1997.
8. G. J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, University of Melbourne, Victoria, Australia, Dec 2005.
9. G. J. Duck, P. J. Stuckey, M. García de la Banda, and C. Holzbaur. The refined operational semantics of constraint handling rules. In *20th Intl. Conf. Logic Programming*, volume 3132 of *LNCS*, pages 90–104, 2004.
10. C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
11. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.
12. T. W. Frühwirth. Theory and practice of constraint handling rules. *J. Log. Program.*, 37(1-3):95–138, 1998.
13. H. Ganzinger and D. A. McAllester. Logical algorithms. In *18th Intl. Conf. Logic Programming*, volume 2401 of *LNCS*, pages 209–223, 2002.
14. L. De Koninck, T. Schrijvers, and B. Demoen. The correspondence between the logical algorithms language and CHR. Technical Report CW 480, K.U.Leuven, Belgium, March 2007.
15. F. Morawietz. Chart parsing and constraint programming. In *18th Intl. Conf. Computational Linguistics*, pages 551–557. Morgan Kaufmann, 2000.
16. G. Ringwelski and M. Hoche. Impact- and cost-oriented propagator scheduling for faster constraint propagation. In *19th Workshop on (Constraint) Logic Programming*, Ulmer Informatik-Berichte, pages 88–98. Universität Ulm, Germany, 2005.
17. T. Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, Jun 2005.
18. T. Schrijvers and T. Frühwirth. Optimal union-find in constraint handling rules. *Theory and Practice of Logic Programming*, 6(1&2), 2006.
19. C. Schulte and P. J. Stuckey. Speeding up constraint propagation. In *10th Intl. Conf. Principles and Practice of Constraint Programming*, volume 3258 of *LNCS*, pages 619–633, 2004.
20. J. Sneyers, T. Schrijvers, and B. Demoen. Dijkstra’s algorithm with Fibonacci heaps: An executable description in CHR. In *20th Workshop on Logic Programming*, volume 1843-06-02 of *INFSYS Research Report*, pages 182–191. Technische Universität Wien, Austria, 2006.
21. E. Visser. Stratego: A language for program transformation based on rewriting strategies. In *12th Intl. Conf. Rewriting Techniques and Applications*, volume 2051 of *LNCS*, pages 357–362, 2001.
22. M. Wallace, S. Novello, and J. Schimpf. ECLⁱPS^e: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, 1997.
23. P. Van Weert, J. Sneyers, T. Schrijvers, and B. Demoen. Extending chr with negation as absence. In *3rd Workshop on CHR*, Technical Report CW 452. K.U.Leuven, Belgium, 2006.