

# Abstract Data Type Components

*Marko van Dooren*      *Eric Steegmans*

*Report CW439, March 2006*



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

# Abstract Data Type Components

*Marko van Dooren*      *Eric Steegmans*

*Report CW439, March 2006*

Department of Computer Science, K.U.Leuven

## Abstract

Although abstract data types are the foundation of object-oriented programming, the use of application independent ADTs as components for other ADTs is surprisingly difficult. No existing language allows the ADTs of high-level elements like associations, bounded values, graphs, ... to be reused conveniently as components. As a result, their instance variables and methods are implemented over and over again.

To allow such elements to be reused, we raise the abstraction level of the language by introducing a new multiple inheritance mechanism with two relations: *is-a* and *has-a*. The *is-a* relation is for subtyping and code inheritance, and is tailored for *classification*. The *has-a* relation is for code inheritance only, and is tailored for composition of ADTs. Using this inheritance mechanism, the implementation of the foundation of a program becomes a trivial task.

A case study shows a 22% decrease in the size of the code, while the second best inheritance mechanism gets no further than 5%. In addition, the mechanism reduces the dependency between the implementation of a class and its hierarchy.

We present a formal model of the inheritance mechanism, along with a proof of type soundness.

**Keywords :** Multiple inheritance, abstract data type, component.

**CR Subject Classification :** D.3.3, D.3.1, D.2.13

# Abstract Data Type Components

Marko van Dooren and Eric Steegmans  
K.U.Leuven  
Department of Computer Science

---

## 1. INTRODUCTION

In current object-oriented programming languages, implementing even a simple application can be a difficult task. Take for example the application in Figure 1. Designing its high-level model using well-known high-level characteristics like associations, bounded values, graphs, ... is a trivial task. Implementing the application, however, is quite expensive.

The ADTs and implementations of the characteristics must be incorporated in the application classes<sup>1</sup>. Their features<sup>2</sup> are given names that are meaningful in the context of the application class, and part of their functionality may be specialized. But aside from the specialized part and renaming, both the ADT and the implementation remain essentially unchanged. Consequently, both should be reused, and customized where necessary. But it turns out that even with the best object-oriented inheritance mechanism, reusing such characteristics is far from trivial.

From Figure 1, it is clear that inheriting the characteristics is impossible in most languages. First of all, a separate code inheritance relation is required because, for example, an account is not an association. Second, because an application class can have multiple characteristics, possibly of the same kind, we need repeated inheritance. This eliminates all languages forbidding repeated inheritance, like Java [37], C# [30], Scala [61], Cecil [18], traits [74], ... Third, we need renaming to solve name conflicts, and give the features meaningful names, ruling out C++ [80] and Timor [44]. Only languages similar to SmartEiffel 2.2 [20] and the upcoming ECMA-367 version of Eiffel [62] remain.

But even these languages are not suitable for reusing the characteristics via inheritance. First, massive renaming is required because every feature must be renamed individually. Second, care must be taken to completely separate the characteristics since the default policy of these languages is to share features. Third, for every method that a characteristic must invoke on another characteristic, an abstract method must be added because the final name of the invoked method is not known in the definition of the former characteristic. The inheriting class must then dispatch the invocation to the correct method. Third, methods involving multiple instance of a characteristic cannot be reused for the characteristics of the application classes because there is no subtyping relation between them. Fourth, providing a lot of functionality in the characteristics forces the developer of the application class to resolve many conflicts, and either provide a bloated interface, or hide the functionality from external clients, preventing reuse. Fifth, their mechanism for allowing non-conformance in absence of subtyping is not modular. This leaves us with empty hands ...

Our case study shows that this overhead cancels out most of the saved work. As a result,

---

<sup>1</sup>We focus on classes, but the paper also applies to prototypes

<sup>2</sup>A feature of a class is an instance variable or a method.

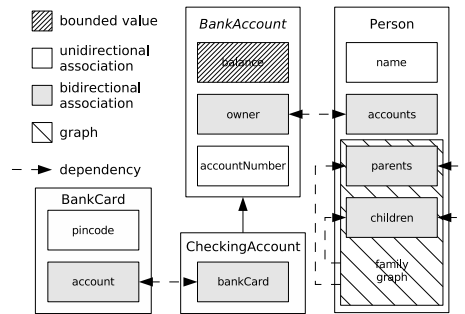


Figure 1: High-level design of an application.

the application is typically built by implementing the ADTs of the characteristics from scratch for each class, as done in Figure 23 for the bank account. This not only costs a lot of time, but is also error-prone.

In [61], Odersky and Zenger identify three abstractions for removing hard references from components to increase their reusability: *abstract type members*, *selftype annotations*, and *modular mixin composition*. Abstract type members and selftypes specify the required services of a component, and mixins perform the composition. But while these abstractions are scalable with respect to the size of the components, they are not scalable in the way components are used. The problem is that both selftypes, and mixins as used in Scala, prohibit any composition involving multiple components of the same kind, or components containing features with the same name. So although the authors claim that these abstractions can lift an arbitrary assembly of static program parts to a component system, they already fail for the application in Figure 1, which is little more than an assembly of four kinds of static program parts.

In this paper, we introduce an inheritance mechanism that allows general purpose classes to be reused conveniently as building blocks for other classes. To achieve this, we introduce renaming parameters and first-class code inheritance relations. Using this inheritance mechanism, the implementation of the application in Figure 1 becomes as simple as its high-level design.

## Overview

In Section 2, we present the *component relation*, which is used for code inheritance. In Section 3, we present the impact on subtyping relation. We give an implementation of Figure 1 in Section 4. To evaluate the inheritance mechanism, we present a case study in Section 5. We present a part of the formal model, and the proof of type soundness in Section 6. Related work and future work are discussed in Sections 7 and 8, and we conclude in Section 9.

## 2. THE COMPONENT RELATION

In this section, we present the *component relation* for building classes using general purpose classes as building blocks. Whereas other code inheritance relations are nearly identical to subtyping relations, the component relation has some important differences. The new features of the component relation are a result of tailoring it for composition of ADTs. Traditional code inheritance can of course also be achieved, but since that is already well

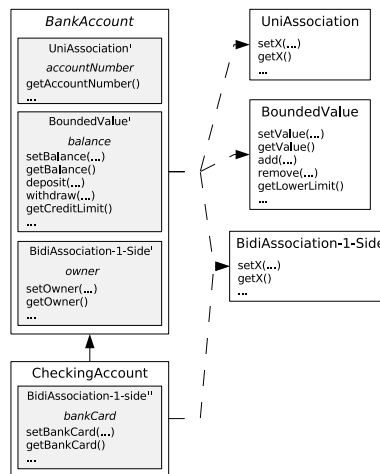


Figure 2: Creating classes from components.

documented, we focus on composition of ADTs. The component relation is denoted by the symbol  $\llcorner$ , while the subtyping relation is denoted by  $\leq$ .

Not all proposed language constructs are required to build the application of Figure 1 using components. We distinguish two kinds of constructs: constructs that are essential for composition of ADTs, and constructs that can further increase code reuse. As mentioned before, a separate code inheritance relation, repeated inheritance, and renaming are existing essential language constructs.

## 2.1 Composition of ADTs

Figure 1 illustrates the example that will be used throughout the rest of this section. The application contains classes for persons, bank accounts, and bank cards. The rectangles inside a class are the characteristics of that class. They are high-level elements used to create the high-level design of the application. An account has a balance, which is a number that lies between the credit limit and an upper bound. In addition, it has a unidirectional association with its account number, and a bidirectional association with its owner. A person has a unidirectional association with his or her name, and bidirectional associations with his or her parents, children, and bank accounts. The associations for the parents and children form a graph offering different iteration strategies. As shown by the dashed arrows in the figure, there are dependencies between the characteristics. For example, the owner characteristic of an account must know the name of the setter method of the accounts characteristic of a person in order to keep the association consistent. Since these characteristics are general and frequently used, we want to turn them into classes, and use them as components to create other classes.

Figure 2 shows the class diagram for `BankAccount`. The solid lines represent subtyping relations, the dashed lines represent component relations. The inherited classes are used as components for implementing the inheriting class. We have created class `BidiAssociation-1-Side` for the 1-side of a bidirectional association, `UniAssociation` for unidirectional associations, and `BoundedValue` for bounded values, to make them reusable. Class `BankAccount` inherits them through component

relations. The actual components in `BankAccount` have types `BoundedValue'` and `BidiAssociation-1-Side'`, and `UniAssociation'`, which are subtypes of the inherited classes. This is because features of the inherited classes can be overridden in `BankAccount`. These subtyping relations, however, are not part of the code, they are *implicit*.

Class `CheckingAccount` additionally has an association with a bank card, and thus has another component relation with `BidiAssociation1` to implement this characteristic. The resulting component is *completely separated* from the component for the association with the owner of the account.

What is important is that we no longer use instance variables and methods as primitives to implement the characteristics of a class, or create a new language construct for a characteristic like e.g. full-blown relationships in [8], or events in C# [30]. Instead, we reuse the characteristics by encapsulating them in a class. As such, we have raised the abstraction level of the programming language.

## 2.2 Syntax

Figure 3 shows the syntax of the component relation. It consists of the keyword `component` followed by the name of the inherited class, including any generic parameters. After the inherited class, there can optionally be a name for the component relation, component parameters, and a configuration block. Both the relation and the name can have an access modifier.

The name and access modifiers of the component relation and component parameters will be discussed in Section 2.5. The configuration block is similar to that of Eiffel: it consists of a comma-separated list of configuration clauses. The `assignment` is used for renaming, the `undefine` clause for selecting an implementation in case of conflicts, the `override` clause if a feature is overridden, and the `export` clause for changing the visibility of a feature. The `direct` and `indirect` clauses are used to determine how a feature is inherited, and are discussed in Section 2.5.2.

Figure 4 shows the component relations for the class of bank accounts. They state that the class of bank accounts has a component `owner` that behaves like a `BidiAssociation-1-Side`, a component `balance` that behaves like a `BoundedValue`, and a component `accountNumber` that behaves like a `UniAssociation`. The inheritance names, component parameters, and assignments in the configuration blocks are discussed in the remainder of this section.

## 2.3 General Semantics

The component relation is essentially a code inheritance relation. This means that if A has a component relation with B, A inherits the features of B, but not its type. For example, the class of bank accounts inherits all features of a bounded value, but a bank account is no bounded value.

As a result, it is allowed for a class to have a component relation with a final class, while it is not allowed to have a subtyping relation with that class. The features of the final class, however, cannot be overridden in the inheriting class. Forbidding a component relation with a final class makes no sense because it is equivalent to manually forwarding all feature calls to an object of the final class. You can reuse, but not override, the code, so the essence of the `final` modifier is respected.

Instance variables are properties that, unless declared `final`, can be overridden and

```

ComponentClause:
  AccessModifier? component Type Config?
Config:
  Name? CompParams? ConfigurationBlock?
Name:
  AccessModifier? Identifier
CompParams:
  "(" Identifier ( , Identifier)* ")"
ConfigurationBlock:
  "[" ConfigurationClause ( , ConfigurationClause)* "]"
ConfigurationClause:
  Identifier = Identifier
undefine "{" IdentifierList "}"
override "{" IdentifierList "}"
export AccessModifier "{" IdentifierList "}"
direct "{" IdentifierList "}"
indirect "{" IdentifierList "}"

```

Figure 3: Grammar for the component relation.

```

component BidiAssociation-1-Side<Account,Person>
  owner (accounts) [X=Owner]
component BoundedValue<long> balance
  [X=Balance,LOW=LowerLimit,HI=UpperLimit];
component UniAssociation<int> accountNumber
  [X=AccountNumber, export private {setAccountNumber}]

```

Figure 4: Examples of the component relation.

merged. Even though this requires a *vtable* lookup for an instance variable due to multiple inheritance and renaming, it is not less efficient than Java, where getter and setter methods used to encapsulate instance variables also require a lookup.

*Duplication.* If a feature is inherited via different inheritance paths, a choice must be made to decide if the feature is inherited once, or multiple times. The default policy for features inherited via a component relation is duplication because, generally, the components will not overlap. This means that if a feature is inherited via a component relation and again via another inheritance relation, there is a conflict, even if the definitions are the same. This conflict must be resolved explicitly via merging or renaming. As a result, features inherited via a component relation do not participate in the rule-of-dominance used for subtyping in Section 3.2. In Section 3.2, we discuss why duplication is forbidden for subtyping. To avoid an explosion of the number of renaming clauses, we introduce *renaming parameters* in Section 2.4 and *indirect inheritance* in Section 2.5.2.

As in SmartEiffel, binding of features in inherited methods is done within the inheritance relation through which they are inherited. This is required to allow separation of the components. In Figure 2, for example, `CheckingAccount` inherits the getter method of `BidiAssociation-1-Side` twice: once for the association with the owner, and once for the association with the bank card. Both getters must of course use the instance variable

```

Set<TO> getXs;
void addX(TO x);
void removeX(TO x);
void replaceX(TO x, TO y);
boolean isValidX(TO x);
boolean containsX(TO x);

```

**Figure 5: Name pattern of associations.**

of their own component.

*Conformance.* For the external clients of a class, it is not a problem if features inherited via a component relation are overridden in a non-conforming manner, e.g. by narrowing the argument types of a method. Such clients cannot access these feature using the interface of the inherited class since they are not part of a subtyping relation, so their expectations cannot be broken. The features of the inheriting class, however, are also invoked by method of the class that is inherited via the component relations, and they expect them to behave according to their original definition. As a result, we currently require that the signatures of an overriding method conform to the signatures of all overridden methods. Providing a different, non-conforming signature to external clients remains future work. In Section 2.6, we discuss how such non-conformance could be obtained while maintaining modularity.

## 2.4 Renaming Parameters

Without intervention, using duplication as the default for the component relation would force a developer to explicitly rename almost every method of the component. The case study in Section 5 shows that renaming is a significant problem, and thus renaming parameters are also an essential language construct for composition of ADTs. We introduce a lightweight macro system to minimize the effort of renaming features.

The names in the features of characteristics often exhibit patterns. For example, the names of the methods of the N side of an association are `getX`, `addX`, `removeX`, `replaceX`, `containsX`, `...`. To avoid these patterns from getting lost in the implementation, we introduce *renaming parameters*. They can be written in the names of non-private features, and allow an inheriting class to rename a number of features with a single renaming declaration.

A renaming parameter is a parameter of a class and is written between square brackets. It can be given a default value; otherwise its name will serve as the default value. The parameter can be used in feature names by writing its name between two % characters. An inheriting class can then assign a value to the parameter in the configuration block of the inheritance relation. The value of the parameter can be any string that is valid for all feature names containing the parameter – which are all visible to the inheriting class.

Figure 6 illustrates the use of renaming parameters. Class `BidiAssociation-N-Side` contains a renaming parameter to customize the names of the methods. Parameter `X` is used as the name of the other end of the association. Parameter `XS` represents the plural of `X` and by default equals the value of `X` appended with an 's'. For the `children` component of `Person` both parameters are assigned because the default value of `XS` is not appropriate.

The name of the renaming parameter itself must be unique for that class but may be

```

class Association<FROM,TO> ... [X] {
    boolean isValid%X%(TO x);
    boolean contains%X%(TO x);
    ...
}
class BidiAssociation-N-Side<FROM,TO> ... [X,XS=%X%s] {
    subtype Association <FROM,TO> [X=%X%] {
        Set<TO> get%X%S% {...}
        void add%X%(TO x) {...}
        void remove%X%(TO x) {...}
        void replace%X%(TO x, TO y) {...}
        ...
    }
}

class Person
    component BidiAssociation-N-Side<Person, BankAccount>
        accounts ... [X=Account]
    component BidiAssociation-N-Side<Person, Person>
        parents ... [X=Parent]
    component BidiAssociation-N-Side<Person, Person>
        children ... [X=Child,XS=Children]
    {...}

```

**Figure 6: Using renaming parameters.**

equal to a renaming parameter of a parent class. The name must also be different from the names of the methods in that class to avoid ambiguities in the configuration blocks of subclasses. In addition, no conflicts may occur when each renaming parameter is assigned its default value.

In order to propagate the value of a renaming parameter to superclasses, the parameter can be used in the right-hand side of the assignments of a configuration block. To avoid conflicts when a class uses a renaming parameter with the same name as a renaming parameter in the superclass, parameters in the left-hand side of the assignment are resolved in the superclass, while parameters in the right-hand side are resolved in the current class.

In Figure 6, class `BidiAssociation-N-Side` uses a renaming parameter `X` with the same name as a renaming parameter of its parent class. In the configuration block, the left-hand side of the assignment references the renaming parameter of the parent class, while the right-hand side references the renaming parameter of the current class. The effect of this assignment is that the renaming parameter of `Association` has the same value as the renaming parameter of `BidiAssociation`.

Methods whose names contain a renaming parameter can be renamed both by assigning a value to a renaming parameter and by renaming the method directly. In order to avoid ambiguities, explicitly renaming an individual method has priority over the renaming that would be done by an assignment to a renaming parameter. The method must be renamed as if the renaming parameter had its default value. The order in which both declarations are written in the configuration block is irrelevant.

In the body of the class declaring a renaming parameter, invocations features containing that renaming parameter must also contain the renaming parameter if the target of the

```

class BankAccount
  component BidiAssociation-N-Side<BankAccount,BankCard>...
    [X=BankCard, addX=attachBankCard]

```

**Figure 7: Priority of renaming.**

invocation is `this`. If the target is another object of that class, the default value for the parameter must be used since the value of the parameter for that object may be different. More general, a feature containing a renaming parameter must be accessed with the most specific value known at the site where it is accessed.

Renaming parameters cannot be filled in during object construction. If the class of which an object is being constructed has renaming parameters, they will assume the default value. Because the object can only be used using the type of its class, or one of its super classes, the renaming could never be visible anyway.

Figure 7 illustrates the priority during renaming. Class `BankAccount` now uses the renaming parameter `X` to rename all methods inherited from `BidiAssociation-N-Side` except for the `addX` method, which is individually renamed to `attachBankCard`.

When overriding a method whose name contains one or more renaming parameters, there are two options. For the first option, the name of the overriding method is equal to the name of the overridden method, including renaming parameters which have the same values as the renaming parameters in the overridden method. For the second option, the method must be renamed to its new name. Otherwise, the connection between the methods would be lost.

## 2.5 First-Class Component Relations

In this section, we solve four remaining problems. First, we must write a lot of low-level code to resolve dependencies. Second, not all feature names contain renaming parameters. Third, we must make a trade-off between the size of a class and the provided functionality. Fourth, the lack of a subtyping relation prevents a component to be used as an object of the inherited type. These problems can be solved by turning component relations – and thus components – into first-class citizens.

A component relation can have a name, which typically represents the role of the component in the inheriting class. Figure 8 illustrates this for classes `BankAccount` and `Person`. The `BidiAssociation-1-Side` component of `BankAccount` is named `owner`, and the `BidiAssociation-N-Side` component of `Person` is named `accounts`. The renaming clauses are not displayed, they are discussed in Section 2.4.

Note that aside from the component relation having a name, this is completely different from named inheritance relations in Timor [44]. This is discussed with other related work in Section 7.

Component relations can also be abstract if the inheriting class is abstract. In this case, concrete subtypes must either override the component relation, implement all abstract methods inherited via the component relation, or a combination of both. If a component relation is declared as `final`, it cannot be overridden. Overriding of component relations is presented in Section 3.3.

In the remainder of this section we explain how the problems mentioned above are solved using the names of component relations. We will use them to connect components,

```

class BankAccount
  component BidiAssociation-1-Side<BankAccount,Person> owner ...
  ...

class Person
  component BidiAssociation-N-Side<Person,BankAccount> accounts ...
  ...

```

**Figure 8: First-class component relations.**

access additional functionality, and use components as if they were separate objects.

2.5.1 *Component Parameters.* Some characteristics depend on other characteristics. Examples of such dependencies are the methods to set up and break down bidirectional associations, as shown in Figure 9. The `setOwner` method of `BankAccount` must know which `register` method to invoke on the `Person` to keep the association consistent. The `registerOwner` method needs an `unregister` method to correctly remove the old back-pointer, and `getAccounts` is needed in the specification of `setOwner`. Because `Person` has multiple associations, these dependencies cannot be resolved automatically. The developer of `BankAccount` must connect these methods to the appropriate methods in `Person`.

If the association characteristic is not reused, as in Figure 23 of Section 4, the appropriate methods are invoked directly in `setOwner`. Figure 10 shows how the association is reused with traditional code inheritance techniques. It uses the `insert` relation of SmartEiffel for code inheritance. The `getThis` method is discussed in Section 2.6.1. Abstract methods are introduced for each dependency. These methods are then used by the implementation of `BidiAssociation-1-Side` to invoke methods on the other component. The actual resolution of the dependencies is done by inheriting classes. They must rename and implement the abstract methods, and forward the call to the appropriate method.

As is shown by Figure 10, and confirmed by our case study, this is a very cumbersome practice. In addition, if more internal dependencies are added to `BidiAssociation-1-Side`, all inheriting classes must be adapted, even though they will be connected to methods of the same association as the other dependencies.

Using Eiffel agents or C# delegates, it is possible to remove the abstract methods at the cost of a bigger memory footprint. The association class can use a delegate for every dependency, but this requires an extra object and a reference for every dependency of every characteristic of every object. For the `Person` class of our application that would result in eleven additional references and eleven delegate objects for every `Person` object, even with just three dependencies per bidirectional association, and two dependencies for a graph. Sharing the delegate objects reduces the memory usage, but barely reduces the code size.

To resolve these dependencies in a more elegant manner, we use the names of the component relations. Figure 11 shows what we want to achieve. We want to connect the `owner` characteristic of `BankAccount` to the `accounts` characteristic of `Person` and vice versa with a single high-level connection on each side.

A class can have a number of formal *component parameters*, which are used *only at*

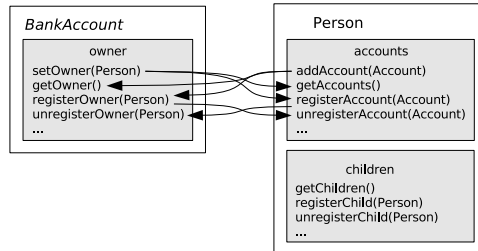


Figure 9: Traditional dependency resolution.

```

class BidiAssociation-1-Side<FROM,TO>
  subtype BidiAssociation<FROM,TO> {
    protected abstract void otherRegister(TO to, FROM from);
    protected abstract void otherUnregister(TO to, FROM from);
    protected abstract FROM otherGetX(TO to);

    public void setX(TO other) {
      ...
      otherRegister(other,getThis());
      ...
    }
    public void registerX(TO other) {
      ...
      otherUnregister(other,getThis());
      ...
    }
    ...
  }
class BankAccount ... {
  insert BidiAssociation-1-Side<BankAccount, Person>
    rename otherRegister as registerAccount
      otherUnregister as unregisterAccount
      otherGetX as getOwnerAccount
    ...

  protected void registerAccount(Person person, Account account)
    {person.registerAccount(account);}
  protected void unregisterAccount(Person person, Account account)
    {person.unregisterAccount(account);}
  protected Account getOwnerAccount(Person person)
    {return person.getAccount();}
}

```

Figure 10: Implementing traditional dependency resolution.

*compile-time*. They are declared after the generic parameters of a class between parentheses, and have the form  $T \rightarrow C$  name. In this declaration,  $T \rightarrow C$  is a constraint on the component relation passed through the parameter.  $T$  is the type containing the relation, and  $C$  is the target type of the relation. Finally, name is the name of the parameter.

Figure 12 illustrates the declaration of a component parameter in `BidiAssociation-1-Side`. The formal parameter expects the name of a relation that *a*) is a relation of the class at the other side of the association (TO), and *b*) is a `BidiAssociation` representing an association in the opposite direction (from TO to FROM). Take for example the component relations of Figure 8. If we substitute the generic types, we see that component relation `owner` requires the name of a component relation with type `BidiAssociation<Person, BankAccount>` that is contained in `Person`. Since the `accounts` component of `Person` satisfies these constraints, we can connect the `owner` component to the `accounts` component. Similarly, the `owner` component satisfies the constraints of the `accounts` component. The high-level connection of both components is shown in Figure 13. The `owner` component of `BankAccount` is connected to the `accounts` component of `Person`, and vice versa. Note that `BidiAssociation1` does not require the components to be mutually connected. Such constraints are not in the scope of this paper.

A component parameter can be used to invoke methods on, or access fields of, the actual component passed through the parameter. Method invocations and field accesses are performed using the following expressions: `expr@param.m(args)` and `expr@param.f`. In such expressions, `expr` must be of type  $T$ , and `m` or `f` must be applicable to type  $C$ . We use a symbol different from the dot to emphasize the difference with a regular invocation. In addition, this avoids confusion about the meaning of `expr.param` if a feature with name `param` is added to  $T$ .

The `setX` method in Figure 12 shows how the component parameter is used to invoke methods. Instead of using an abstract `registerOther` method to let the other end of the association set its reference to the current object, we now use the component parameter to invoke the `register` method directly. The invocation of `register` is applied to the `otherEnd` component of `other`. The method that will be invoked, is the `register` method of the actual component relation passed through the parameter, which may be overridden or renamed in the actual class TO. In the example of Figure 13, the `setOwner` method inherited by `BankAccount` will invoke the `registerAccount` method inherited by `Person`.

Currently, component parameters are not part of the type of a class, unlike generic parameters. Consequently, `A(someComponent)` and `A(otherComponent)` have the same type: `A`. Incorporating component parameters in the type of a class together with wildcards remains future work.

An important additional advantage of component parameters is that dependencies can be added without modifying the inheriting classes. For example, we can add an `isSibling` method to `BidiAssociation` to check if some object is its sibling. This method would invoke the `contains` method on the other end of the association, introducing another dependency. Inheriting classes (clients), however, do not need to be modified.

By comparing the code in Figure 10 with that of Figure 13, it becomes clear that first-class component relations and component parameters are essential for composition of ADTs. Like Odersky and Zenger in [61], we identify *abstract type members* or *generic*

*parameters* as existing essential language constructs. Without one of these constructs, component parameters lose a lot of flexibility.

Technically, component parameters can be replaced by component references and delegates or agents, but that would make component references an essential language construct. A drawback of that choice is that the connection is not made where it should be made: at the class level. Instead, the connection would be made at the object level. In addition, using delegates and component references, it is possible to change the connection at run-time, which seems to make little sense for connections of characteristics.

*2.5.2 Direct and Indirect inheritance.* In this section, we use the name of a component relation to solve two remaining problems. First, renaming parameters alone are not yet sufficient to get rid of all renaming conflicts. Second, the developer of a class typically balances the size of its interface to limit its complexity but still provide enough functionality for the anticipated use of the class. But by doing this, he may deprive clients from important functionality.

*Reducing Name Conflicts.* Even with renaming parameters, the inheritance mechanism will become practically useless if we do not intervene. Not all method and field names will contain renaming parameters, and even if that is the case, such methods and fields are not necessarily interesting for the inheriting class. For example, even a small number of features in top-level classes, like `equals` and `hashCode`, would cause an explosion of the number of rename clauses. These methods would have to be renamed for every component relation. Figure 14 illustrates this for class `BankAccount`. It inherits four different definitions of `equals` and `hashCode`. But usually, the definitions inherited via the components are not interesting for the inheriting class, yet they still cause conflicts that must be resolved.

*Interface Bloat vs. Reuse.* Another problem is the size of the class interface. To improve the understandability of an application class, it is desirable to keep its interface simple. But to increase code reuse, it is desirable to provide as much functionality as possible.

This is illustrated in Figure 15. In order to maximize code reuse, it is desirable to put many features in the association classes. Examples are applying some action to all referenced elements, a universal and an existential quantifier, accumulation, validation, . . . . But if all these features must be inherited by `Person`, either its interface get bloated, or the developer of `Person` must do a lot of work to hide the functionality. Of course the argument of the uninteresting name conflicts applies here as well.

*Indirect Inheritance.* To solve both problems, we can take advantage of the fact that code inheritance and subtyping are not tied together. Because a component relation does not imply a subtyping relation, it is not mandatory *for the interface of the inheriting class* to contain all features of the inherited class. As a result, we can make a distinction between *directly* and *indirectly* inherited features. A directly inherited feature is present in the interface of the inheriting class, while an indirectly inherited feature is not, and thus cannot cause a conflict.

Technically, the extra renaming can be avoided by using renaming parameters in every feature of a class representing a characteristic, even if it is not appropriate. As such, indirect inheritance is not an essential construct for composition of ADTs, but it can significantly increase code reuse while maintaining simple interfaces for the application classes.

An indirectly inherited feature, however, *can still be accessed* if the component relation has been given a name. The feature can then be invoked as `myObject.inheritanceName.feature`. As such, the client resolves the conflict by using the name of the component relation. Nevertheless, it is still the responsibility of the programmer to provide a balanced interface for the class with meaningful names. Using inheritance names to access features should not be the standard way of using a class.

Figure 16 illustrates this for class `Person`. For the `children` component, only the `add`, `remove`, and `get` methods are inherited directly. The `parents` component additionally inherits the `replace` and `isValid` methods. The other methods must be invoked indirectly via the name of the inheritance relation.

Directly inherited features are accessible via either their simple name, or via the inheritance name and their original name. Both expressions of course invoke the same method. Super calls in the inheriting class must be performed using `inheritanceName.super.feature`. More about super calls is presented in Section 3.4.

The inheriting class must specify which features are inherited directly. This is done in the configuration block by either including them with a `direct` declaration, or renaming, selecting, or merging them. All other features are inherited indirectly.

To facilitate selecting directly inherited features, the features of a class can be put in groups as in Eiffel, Smalltalk, and C# (using the `#region` directive). This way, inheriting classes can directly inherit an entire group of methods with little effort. For example, the basic functionality of a class can be put into a single group while more advanced functionality can be put in other groups. To select which features or groups are inherited directly, the programmer can use `direct` and `indirect` declarations in the configuration block of the component relation. They can be used with both feature groups and individual features. For an individual feature, the final name of that feature – after performing the renaming – must be used. Features are only inherited directly if they are listed in a `direct` declaration, and not listed in an `indirect` declaration. Every component relation automatically has a `direct` declaration for the group named `default`. To avoid ambiguities, group names are put in the same namespace as method and field names. The mechanism can still be made more flexible, but that is outside the scope of this paper.

Figure 18 illustrates simple use of indirect inheritance. Class `BankAccount` is a subtype of `Object`, and uses a component relation with `BidiAssociation` for its bidirectional association with class `Person`. Because `BidiAssociation` is also a subtype of `Object`, it also has an `equals` method, which can for example be used to check whether two ends of associations reference the same set of objects. Since the `equals` method in `BidiAssociation` is not in the `default` group and there are no `direct` or `indirect` declarations, it is inherited indirectly by `BankAccount`. It is not present in the interface of the class, but can still be invoked as `someAccount.owner.equals`.

Figure 17 illustrates the use of the selection mechanism. Both the `children` and `parents` components inherit the `get`, `add`, and `remove` methods, which are in the `default` group. The `children` component of `Person` excludes `replaceChild` from the direct interface since it is barely useful in that context, and would otherwise be inherited since it is in the `default` group. The `parents` component additionally includes the `isValidParent` method, which may contain conditions for the adoption of a child, in its interface.

*2.5.3 Alternative Approaches.* At first sight, using the name of an inheritance relation to access a feature may seem identical to using manual delegation. For this technique, there is a reference to an object of the type of the component that can be used to access the functionality. There are some important disadvantages to this technique, however. First, you cannot easily provide direct access to the functionality of the component. You need to do this because e.g. a client wants to query the owner of an account by invoking the `getOwner` method, not by invoking `owner.getOther`, which is more difficult to read and write. Second, it is impossible to override methods of the component if it is not explicitly designed for such use. Last, this technique results in inefficient programs. For every component of every object, a separate object is needed, increasing the number of memory lookups.

Automatic delegation as offered by delegation based inheritance mechanisms offer no alternative either. They fall in the same category as other class based inheritance mechanisms since they put every inherited method in the interface of the inheriting class.

First class subtyping relations and reuse variables of Timor cannot be used to achieve direct and indirect inheritance. Reuse variables only provide methods needed to implement the types of a class, leaving other methods unavailable to clients. First class subtyping relations can only resolve naming conflicts by prepending the name to the method. Direct access using a different name is impossible. In addition, adding a subtype relation, or even adding a method to an inherited type can break existing clients without warning since the direct name automatically is removed from the class interface.

*2.5.4 Visibility.* By default, component relations are public because they are typically used for the characteristics of a class. A public client can see their name, type, and configuration. The reason for this default choice is understandability. If a programmer knows the behavior of class `C`, he also knows the behavior of a component of type `C`. But if the relation is not visible, must study the contracts of the inherited features again in order to understand their behavior. If the component relation is used for traditional code inheritance, e.g. to implement a `Stack` using an `Array`, it should be hidden from the client. Restricting visibility can be done at different levels: the name, the type, and the individual features can be given different visibilities. If the type of the relation is invisible to a client, the name is also invisible to that client. If the name of the relation is visible to a client, all features that are visible to that client can also be invoked if they are inherited indirectly. This is discussed in Section 2.5.2.

## 2.6 Object Identity

Since features are bound within the component relation through which they are inherited, the `this` reference in the implementation of such features can be seen as a reference to a subobject. For example, when a method of `BidiAssociation-N-Side` is executed in the context of `Person`, the `this` reference can be interpreted as a reference to the subobject representing the `BidiAssociation-N-Side` component, e.g. `this.children`. As long as `this` in `BidiAssociation-N-Side` is used only as the target of a method invocation or field access, no problems can arise because of the conformance requirements for overriding, which ensure substitutability. But an important question is what happens when `this` is used as a separate expression, since there is no subtyping relation between the inheriting class and the inherited class.

If `this` can be used as a separate expression in a class inherited via a component rela-

tion, it not only can be thought of as a reference to a subobject, it really *is* a reference to a subobject. As such, it can be passed as an argument to other objects, and stored in variables of the type of the inherited class. We will call such references *component references* for the rest of this discussion.

Component references are not an essential language construct for composition of ADTs, they offer additional possibilities for reusing code. Class `BoundedValue`, for example, has an `equals` method to check if its value, upper limit, and lower limit are equal to that of another `BoundedValue`. With component references, it is trivial to check if the balance component of one account is equal to that of another account using `myAccount.balance.equals(yourAccount.balance)`. Moreover, all operations involving multiple `BoundedValue` objects can be reused without any limits. You can use them with `BoundedValue` components of different classes or separate objects of class `BoundedValue`. Without component references, such code cannot be reused because the inheriting class is no subtype of `BoundedValue`. A drawback of component references is their impact on object aliasing and non-conformance.

Component references can cause additional problems with object aliasing [41]. This is illustrated in Figure 19. In the scenario on the left, variable `a` references a subobject of the object referenced by `b`. In the scenario on the right, the variables reference different subobjects that have a shared state because they were merged in the component relations. In both scenarios, side-effects of operations on reference `a` may be visible through reference `b` and vice versa, which can be unexpected. This can also happen without component references, but in that case it is less likely that concepts like bounded values get aliased because they would typically be implemented over and over again.

The presence of component references also affects the ability to completely remove features inherited via a component relation, or to override them in a non-conforming manner as discussed in Section 2.3. For example, if `BankAccount` wants to remove the method for modifying the lower limit of the balance, it can inherit the method indirectly, and make the component relation `balance` inaccessible to its clients. But if class `BoundedValue` has methods that leak the `this` reference, the hidden setter method can still be accessed.

*Component Classes.* If non-conformance or removal of features is allowed in presence of component references, or if component references are forbidden, it is important to prevent `this` from being used as a separate expression in the inherited class. To achieve this, a class that is explicitly designed as a reusable component can use the `component` modifier. Within the body of such a class, it is not allowed to use `this` as a separate expression. As a result, the component cannot leak a reference to itself. The `this` expression, however, can still be used as a target of method invocations or field accesses. Subtypes of a component class must also be component classes since it is a promise to the clients. A `possible_component` modifier indicates that the class itself satisfies the properties of a component class, but does not promise anything about its subtypes. This modifier is needed for the top class `Object` since both component and non-component classes are subclasses of `Object`, and the component classes need the non-leaking guarantee.

In addition, the name of a component relation with a component class cannot be used as a separate expression if features are removed. Similar to `this` in a component class, it can still be used as the target of method invocations and field accesses if they are still accessible after taking the `export` clauses of the relation into account.

In SmartEiffel, safe non-conformance is achieved by type-checking the implementation

of a class inherited via an *insert* relation. If a class A inserts class B, non-overridden features of B are type-checked as if they were written in A. The problem with this approach, however, is that it is not modular. A change in the implementation of class B can break class A.

Note that having component classes is not the same as having a completely separate concept for components, as with traits. Component classes can still be classified and instantiated.

2.6.1 *This*. Some characteristics need a reference to the object that uses them as a component. For example, the bidirectional association classes must pass an object of type FROM to the other end of the association. But since the class is typically inherited via a component relation, they cannot use `this` to obtain that reference. The `this` expression has the type of the association class, not FROM.

A first solution is to store a reference to the FROM object. While simple and effective, this solution requires an additional reference for every association of every object that simply references the outer object.

A more efficient solution is to use an Eiffel type anchor to introduce a *final* method `getThis` with type like `Current` in class ANY, the default but not mandatory super class. This is shown in Figure 20. The method simply returns a reference to the current object. A component class that needs a reference to the outer object is not a subtype of Any, but of `General`, the mandatory top-level class. It introduces an abstract `getThis` method with the appropriate type which it uses to get a reference to the outer object. In the application class, both features are merged. Because duplication is the default behavior for component relations, an optional modifier `shared` can make a feature shared by default. Otherwise, `getThis` would have to be merged for every component relation that requires `getThis`. In the `BidiAssociation1` class of Figure 24, this construction is used to obtain a reference to the object at this side of the association.

A third solution is to use a construction similar to self types in Scala. A class could put a constraint on the classes that inherit from it using e.g. the constraint `outer T`. An inheriting class must then either have the same constraint, or be a subtype of T. The reference to the object of the class satisfying the constraint can be accessed using `T.this`.

*Choices For A Language Developer.* Below, we present a number of possible choices for a language developer regarding component references and component classes. We prefer choice 3 because of its flexibility.

- (1) The language has no support for component references. This option prevents the additional aliasing issues, and non-conformance as discussed in Section 2.3 is allowed.
- (2) The language has component references and no component classes. This option forbids non-conformance.
- (3) The language has component references and component classes. A class can have component relations with any class. For a component relation with a non-component class, non-conformance is not allowed. For a component relation with a component class, it is allowed.

### 3. THE SUBTYPING RELATION

#### 3.1 Syntax

Figure 21 shows the syntax of the subtyping relation. It consists of the keyword `subtype` followed by the name of the super type, including any generic parameters. There can optionally be a name for the relation, component parameters, and a configuration block. The name of the relation is discussed in Section 3.4.

#### 3.2 General Semantics

The subtyping relation is tailored for *classification*. Because we use the component relation for code reuse, we can simplify the subtyping relation.

The subtyping relation has the same meaning as in SmartEiffel. The inheriting class inherits both the implementation and the type of the inherited class.

As with the component relation, inherited features can be renamed. A first use of renaming for classification is providing better names for features inherited from general classes. For example, the `getParent` method in the top-level class of our metamodel can be renamed to `getMethod` for the class `Implementation`. In addition, renaming is required to solve conflicts and merge similar features when inheriting from independently developed classes. Without renaming, reusability of such classes is severely decreased. In addition to method and fields, component relation can also be renamed. Indirectly inherited features are then accessed using the new inheritance name.

To ensure consistency, component parameters passed to the same class via different subtyping relations must be identical. This is similar to the rule for generic parameters in Eiffel and SmartEiffel.

Just as in SmartEiffel and Cecil, duplication of features that are inherited via a subtyping relation is forbidden because it is confusing for classification, hereby avoiding the diamond problem for subtyping. For example, it makes perfect sense that a `Zebra` is a special type of `Horse`, but being 1.7 times a `Horse` does not. An object is either a horse, or it is not a horse. In more technical terms, if duplication is allowed, only one of the duplicated features can be used on an object of type `Horse`. The other feature is actually a new feature that is introduced in `Zebra`. This should be done via code inheritance, not subtyping. As a consequence, features with the same origin that are inherited via subtyping relations must be given the same name.

For the subtyping relation, we use a rule-of-dominance like in C++ [80]. If one definition of a feature inherited via subtyping overrides all others, that definition is inherited and there is no conflict. Since the subtyping relation is nominal, and conformance is enforced, behavioral subtyping is preserved. For overriding methods, the standard conformance rules apply. For overriding instance variables, which are properties, the type must be preserved. Overriding of component relations is discussed in Section 3.3.

Features can be merged by giving them the same name. This is needed to inherit from independently developed classes that share a concept. To choose a definition, either a new one can be provided, or an existing definition can be selected by undefining the others. The final feature must of course conform to all inherited features. Merging of component relations is discussed in Section 3.3. Contrary to SmartEiffel, we do not forbid merging instance variables. Method groups [79] or data groups [51] can be used to prevent separating dependent instance variables [70] by e.g. duplicating only one of them. This topic, however, is not in the scope of this paper.

For constructors, either the approach of C++ or Eiffel/Smalltalk can be chosen. In C++, the most specific class must invoke both the constructors of the direct super classes, and the constructors for every virtually inherited class. In Eiffel and Smalltalk, a constructor is a regular method without special requirements.

### 3.3 Overriding and Merging Components

A component relation can be overridden in a subtype if they are not declared `final` as illustrated in Figure 22. The `BidiAssociation'` component in `B` overrides the `Only` the subtyping relations represented by the solid arrows and the component relations represented by the dotted arrows are in the code, the dashed subtyping relations are implicit. There are two rules that must be satisfied by the overriding component.

First, standard subtyping conformance is required. The overriding component (`BidiAssociation'`) must not only be a subtype of the target class of the component relation (`BidiAssociation`), but also of all overridden components (`Association'`). Because of this subtyping hierarchy of the overriding component, we again use the rule-of-dominance to minimize the number of methods that must be overridden in the overriding component (`BidiAssociation'`).

Second, conformance of the component interface is required. This means that every feature that is inherited directly in an overridden component relation must be inherited directly in the overriding component relation. The features of an overridden component are automatically renamed to the corresponding new names defined by the overriding component. Renaming of features within the type hierarchy of the component is of course taken into account. This is illustrated in Figure 22. If `getX` is renamed in `Association'` to `x`, in `BidiAssociation` to `y`, and in `BidiAssociation'` to `z`, then `x` will automatically be renamed to `z` in the subtyping relation between `A` and `B`.

Note that overriding a component relation does not cause any problems for super constructor calls to that relation. In Figure 22, the constructor call to `Association` remains valid since memory allocation has already been done, and the overriding component conforms to the overridden component.

Corresponding component parameters of the overriding component do not have to be equal. For the association classes, such a constraint can seem natural, but it is not always appropriate. Suppose that we add a `Graph` component to the elements of our metamodel to represent the lexical structure of the elements. In subclasses, this component will typically be overridden with a `AssociationGraph` component that is linked to the associations of that class that are part of the lexical structure. In further subclasses, even more associations can be added to the structure. Nevertheless, classes like the association classes may wish to impose such constraints in addition to consistency constraints. Such constraints remain future work.

Component relations can also be merged. As with merging of methods and fields, either an existing component must be selected, or an overriding component must be defined. The final component must satisfy both rules mentioned above. As with overriding, the names are automatically mapped to the corresponding names of the final component.

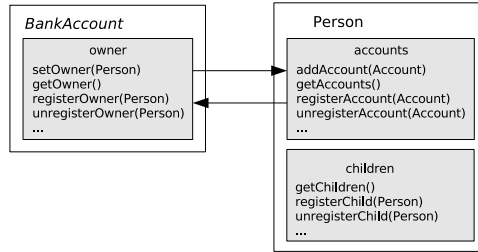


Figure 11: High-level dependency resolution.

```

class BidiAssociation-1-Side<FROM,TO>
  (TO → BidiAssociation<TO,FROM> otherEnd)
  subtype BidiAssociation<FROM,TO> {

  private TO other;

  public void setX(TO other) {
    ...
    other@otherEnd.register(getThis());
    ...
  }
  protected void register(TO other) {...}
  ...
}
  
```

Figure 12: Component parameters.

```

class BankAccount
  component BidiAssociation-1-Side<BankAccount,Person> owner
  (accounts)
  ...

class Person
  component BidiAssociation-N-Side<Person,BankAccount> accounts
  (owner) ...
  ...
  
```

Figure 13: Implementing high-level dependency resolution.

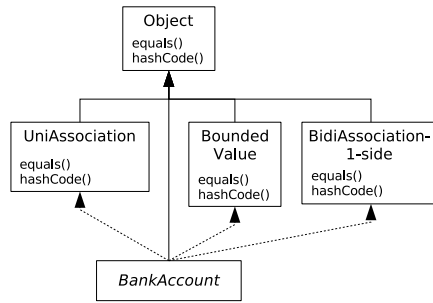


Figure 14: Uninteresting methods cause conflicts.

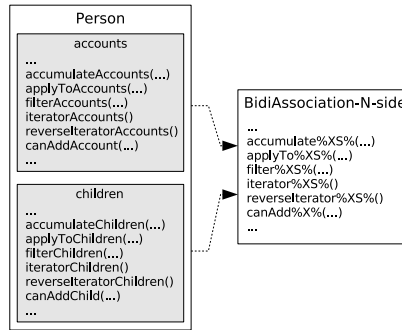


Figure 15: Interface bloat.

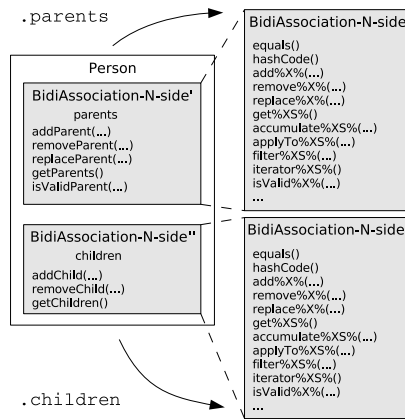


Figure 16: Indirect Inheritance.

```

class BidiAssociation-N-Side<FROM,TO> ... [X,XS=%X%s]
  boolean equals(Object other) {...}
  int hashCode() {...}
  group default {
    Set<TO> get%XS% {...}
    void add%X%(TO x) {...}
    void remove%X%(TO x) {...}
    void replace%X%(TO x, TO y) {...}
  }
  group advanced {
    isValid%X%(TO x) {...}
    applyTo%XS%(Command<TO>) {...}
    ...
  }
}

class Person
  component BidiAssociation-N-Side<Person,Person> children (parents)
    [X=Child,XS=Children, indirect{replaceChild}]
  component BidiAssociation-N-Side<Person,Person> parents (children)
    [X=Parent, direct{isValidParent}]
  ...

```

Figure 17: Selecting directly inherited features.

```

class Object {
  boolean equals(Object other);
}

class BankAccount
  subtype Object
  component BidiAssociation<BankAccount,Person>
    owner [X=Owner]{
  }
}

```

Figure 18: Indirect inheritance.

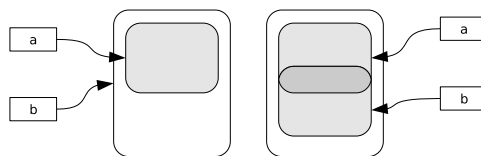


Figure 19: Object aliasing.

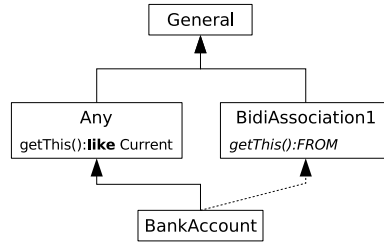


Figure 20: Reaching the outer object via `getThis()`.

*SubtypeClause:*

**subtype** *Type Identifier? CompParams? ConfigurationBlock?*

Figure 21: Grammar for the subtyping relation.

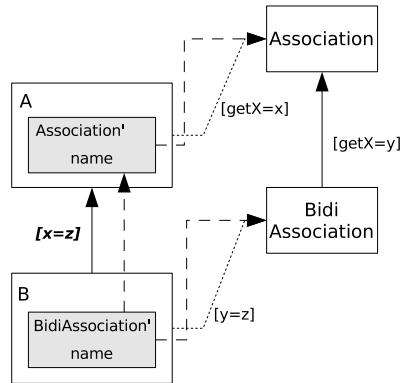


Figure 22: Overriding components.

### 3.4 Reducing Hierarchy Dependencies

In [74], it is argued that `super` calls in class-based languages with multiple inheritance increase the dependency of code on the class hierarchy. In such a language, multiple methods with the same name can be inherited by a class, so in order to disambiguate super calls to such methods, they must be qualified with the name of the direct super class containing the method that must be invoked. Examples of languages using this approach are C++, Cecil, Trellis/OWL, Eiffel, and SmartEiffel. This problem does not occur with inheritance mechanisms that linearize the class hierarchy, or in the prototype-based language Self [19], where super calls can be directed to a named parent slot.

These additional dependencies can be removed by also giving a name to a subtyping relation. The name of a subtyping relation is private since only the inheriting class can invoke super calls. If a subtype relation has a name, it is possible to qualify a super call using the name of that inheritance relation instead of the name of the super class. Consequently, the call remains valid if the actual super class for that relation is changed, as long as an appropriate method is available in the new super class. This is similar to directed resends in Self. For our inheritance mechanism, a super call is either unqualified, or qualified by the name of an inheritance relation.

Technically, reuse variables in the class-based language Timor also reduce this dependency, but in their paper [44], the authors do not present this insight.

## 4. EXAMPLE

In this section, we show how the application of Figure 1 is implemented using our inheritance mechanism. Although the example is extreme in the sense that the program does not contain much application specific behavior, it illustrates what can be achieved with the component relation. Section 5 presents a realistic case study.

Figure 23 shows the Java implementation of `BankAccount`. It has characteristics for the balance, the owner, and the number of an account. The `unregister` is needed in order to be able to connect to an N association end. Note that inlining the auxiliary `register` method does not make the code shorter, and additionally requires a `contains` method to prevent an infinite loop of setter calls.

Figures 24 and 25 show basic versions of two association classes, which are in a library. Advanced functionality like validation, sending of events, ... is not shown for reasons of space. Such additional methods do not impact the size of the application classes since they are not in the default group. Class `BidiAssociation-1-Side` uses a `UniAssociation` for the reference to the other end, and the corresponding getter and setter methods. The setter method is overridden in order to keep the association in a consistent state. The invocations of the `registerX` and `unregisterX` methods on the component parameters do not contain `%X%` because they may have been renamed using a different value for the renaming parameter, hence the default value of the parameter is used. Note that the `unregisterX` method takes the object to be unregistered as an argument. This signature is inherited from the super class `BidiAssociation`, and must be general enough to allow different multiplicities.

Figure 26 shows the *entire* implementation of Figure 1. The implementation can be done almost completely by configuring existing components. Only the constructors are actually implemented. This is a very important result, because it means that the implementation can be made by simply drawing a class diagram, and filling in some parameters.

```

class BankAccount {
    public BankAccount(int number) {
        this.creditLimit = -1000;
        this.upperLimit = 1000000;
        this.accountNumber = number;
    }
    private Person owner;
    public Person getOwner() {
        return owner;
    }
    public void setOwner(Person owner) {
        if(this.owner != owner) {
            registerOwner(owner);
            if(owner != null)
                owner.registerAccount(this);
        }
    }
    protected void registerOwner(Person owner) {
        if (this.owner != null)
            this.owner.unregisterAccount();
        this.owner = owner;
    }
    protected void unregisterOwner() {
        owner = null;
    }

    private long balance;
    private long upperLimit;
    private long creditLimit;

    public long getBalance() {
        return balance;
    }
    public void deposit(long amount) {
        if ((amount > 0) &&
            (balance <= Long.MAX_VALUE - amount) &&
            (balance + amount <= upperLimit))
            balance += amount;
    }
    public void withdraw(long amount) {
        if ((amount > 0) &&
            (balance >= Long.MIN_VALUE + amount) &&
            (balance - amount >= creditLimit))
            balance -= amount;
    }
    public long getUpperLimit() {
        return upperLimit;
    }
    public long getCreditLimit() {
        return creditLimit;
    }
    private final int accountNumber;
    public int getAccountNumber() {
        return accountNumber;
    }
}

```

Figure 23: The Java version of BankAccount.

```

component class BidiAssociation-1-Side<FROM,TO> [X,XS=%X% s]
    (TO → BidiAssociation<TO,FROM> otherEnd)
subtype BidiAssociation<FROM,TO> [X=%X%]
component UniAssociation<TO> [X=%X%, override set%X%]
{
  group default {
    public abstract shared FROM getThis();
    public void set%X%(TO other) {
      if(this.other%X% != other) {
        register%X%(other);
        if(other != null)
          other@otherEnd.registerX(FROM.this);
      }
    }
  }
protected void register%X%(TO other) {
  if (this.other%X% != null)
    this.other%X%@otherEnd.unregisterX(FROM.this);
  this.other%X% = other;
}
protected void unregister%X%(TO other) {
  this.other%X% = null;
}
  ...
}

```

Figure 24: Library class `BidiAssociation-1-Side`.

On top of that, the high-level concepts of the diagram cannot get lost because they are directly present in the code. In current CASE tools, such concepts are lost because they are translated into low-level code, leading to synchronization problems. The library classes for the characteristics are component classes, so we can completely remove methods. For class `BankAccount`, the setter methods for the balance, the credit limit, the upper limit, and the account number are removed. The constructors of component relations are invoked using a `super` call qualified by the inheritance name for reasons of clarity.

The entire implementation of Figure 1 using high-level elements is not only (a bit) shorter than the traditional implementation of the single class `BankAccount`, but it is also significantly less complex. The component version only does a bit of trivial work like renaming and excluding methods. The Java version, however, must itself ensure consistency of the bidirectional associations, and protect against overflow and underflow in the `withdraw` and `deposit` methods. While this code may seem simple, we must note that even Meyer's implementation of bi-linkable elements on page 597 of [58] is wrong. The old back-pointer is not set to `Void` when a bi-linkable element is attached to a new element. This cannot be fixed in the `put_left` method that he leaves as an exercise to the reader, since the reference to the inconsistent object is lost. This bug can be hard to find in a real program.

In languages without repeated inheritance, such results cannot be achieved because a class often has more than one characteristic of a certain kind. This is the case for

```

component class UniAssociation<TO> [X]
  subtype Association<TO> [X=%X%]
  {
    protected TO other%X%;
    group default {
      public TO get%X%() {
        return other%X%;
      }
      public void set%X%(TO other) {
        this.other%X% = other;
      }
    }
  }
  ...
}

```

Figure 25: Library class `UniAssociation`.

`BankAccount`. Both `UniAssociation` and `BidiAssociation1` are subtypes of `Association`. In `SmartEiffel`, the components can be reused, but there are so many methods to be renamed for clarity and conflict resolution that most of the reduction in code size is lost. Nevertheless, the result would still be far less complex than the Java code since renaming is much simpler. More about the effects of renaming is discussed in Section 5.

## 5. CASE STUDY

We compared our inheritance mechanism with manual delegation, and the inheritance mechanisms of Java, `SmartEiffel`, and a version of traits with repeated inheritance [66], by comparing their impact on the size of an application. We used *Inome* [87], our metamodel for Java, and *Chameleon* [87], our framework for metamodels of programming languages. Together they contain 9763 lines of Java code. The details can be found in the technical report [90].

We modified the Java programs using our inheritance mechanism<sup>3</sup>, and then calculated the size for the other techniques based on the overhead of renaming, dependency resolution, encapsulation of state, and manual delegation for each technique.

To study the impact of the size and the nature of extensions of the components, we repeated the experiment for two simulated code bases. In the first one, all associations send events when they are modified. This extension is application independent because managing the listeners and invoking `notify` is always the same. In the second one, the associations also check the validity of the elements. For this extension, the validity condition is application specific and must be overridden, while other supporting code can be reused.

We do not take specifications into account in the study. We assume that the average specification of the removed methods and variables equal the averages for the entire program. Consequently, the relative gain will not differ when taking the specifications into account.

<sup>3</sup>We must note that the resulting code does not currently compile because our compiler is not yet complete.

```

class BankAccount
  component BoundedValue<long> balance
    [X=Balance,LOW=CreditLimit,
     HI=UpperLimit,increaseBalance=deposit,
     decreaseBalance=withdraw,
     export private {setUpperLimit,setLowerLimit,setBalance}]
  component BidiAssociation1<BankAccount,Person> owner
    (accounts) [X=Owner]
  component UniAssociation<int> accountNumber
    [X=AccountNumber,
     export private {setAccountNumber}]
{
  public BankAccount(int accountID) {
    balance.super(0,-1000,1000000);
    accountNumber.super(accountID);
  }
}

class CheckingAccount
  subtype BankAccount
  component BidiAssociation-1-Side<CheckingAccount,BankCard>
    bankCard (account) [X=BankCard]
{
  public BankAccount(int number) {
    super(number);
  }
}

class Person
  component BidiAssociation-N-Side<Person,BankAccount>
    accounts (owner) [X=Accounts]
  component BidiAssociation-N-Side<Person,Person>
    parents (children) [X=Parents]
  component BidiAssociation-N-Side<Person,Person>
    children (parents) [X=Children]
  component UniAssociation<String> [X=Name]
  component Graph<Person> family (parents,children)
{
  public Person(String name, Person mother, Person father) {
    setName(name);
    addParent(mother);
    addParent(father);
  }
}

class BankCard
  component BidiAssociation1<BankCard,CheckingAccount>
    account (bankCard) [X=Account]
  component UniAssociation<int> [X=PinCode]
{}

```

Figure 26: Implementation of Figure 1.

Standard Code Base	Java	Components	SmartEiffel	Traits	Delegation
1. Size (LOC)	9763	7672	9326	10521	8712
2. Reduction (LOC)	N/A	2091	437	-758	1051
3. Reduction (%)	N/A	21.42%	4.48%	-7.76%	10.77%
4. Added Cost Java (%)	N/A	27.25%	4.69%	-7.20%	12.06%
5. Share of Characteristics	N/A	87.57%	40.50%	100.00%	100.00%
6. Share of Multiple Inheritance	N/A	12.43%	59.50%	0.00%	0.00%

**Figure 27: Results for the standard code base.**

Events Code Base	Java	Components	SmartEiffel	Traits	Delegation
1. Size (LOC)	11683	7672	9966	12121	9672
2. Reduction (LOC)	N/A	4011	1717	-438	2011
3. Reduction (%)	N/A	34.33%	14.70%	-3.75%	17.21%
4. Added Cost Java (%)	N/A	52.28%	17.23%	-3.61%	20.79%
5. Share of Characteristics	N/A	93.52%	84.86%	100.00%	100.00%
6. Share of Multiple Inheritance	N/A	6.48%	15.14%	0.00%	0.00%

**Figure 28: Results for the events code base.**

Validation Code Base	Java	Components	SmartEiffel	Traits	Delegation
1. Size (LOC)	12406	7992	10877	13032	10520
2. Reduction (LOC)	N/A	4414	1529	-626	1886
3. Reduction (%)	N/A	35.58%	12.32%	-5.05%	15.20%
4. Added Cost Java (%)	N/A	55.23%	14.06%	-4.80%	17.93%
5. Share of Characteristics	N/A	94.11%	83.00%	100.00%	100.00%
6. Share of Multiple Inheritance	N/A	5.89%	17.00%	0.00%	0.00%

**Figure 29: Results for the events and validation code base.**

### Counting LOC

We count the LOC using the *sloccount* utility of David Wheeler. Because import statements and lines containing only braces are automatically generated by modern programming environments like Eclipse [38], we ignore them in the line count. They do not represent work done by a programmer. We subtracted such lines from the results of the *sloccount* utility because it includes such lines.

Note that this is in our own disadvantage for the first result because the methods that will be removed are very short. As a result, such lines make up a significant part of the LOC that are removed using the *sloccount* numbers.

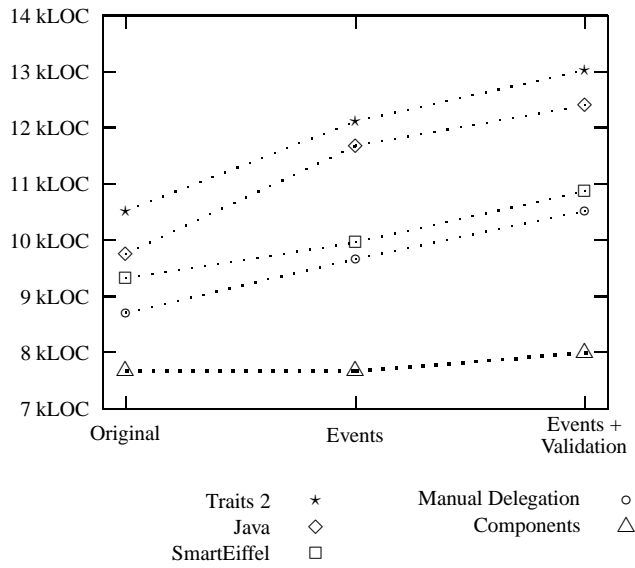


Figure 30: Code size.

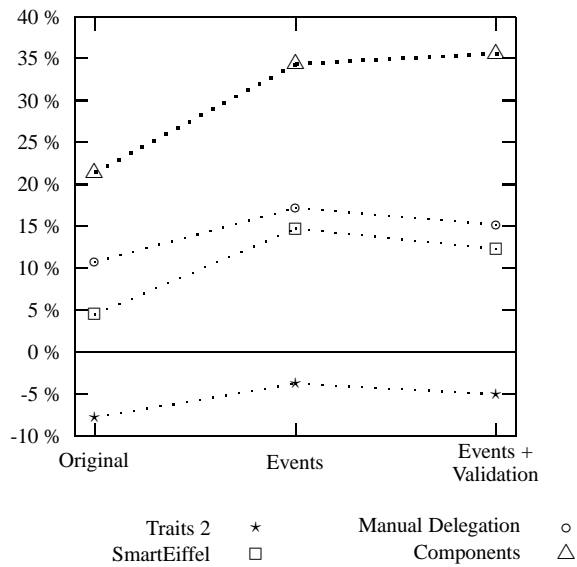


Figure 31: Reduction.

We also chose to ignore import statements because they are all automatically generated by the IDE when performing autocompletion. Selecting the appropriate type from a drop-down box – if required at all – will not take more effort than writing the type name. There are about 1900 import statements in the code base, which would make up almost 20% of

the original code base.

### Lies, Terrible Lies, and Statistics

For all our result we give two numbers. The percentage of decrease of the original code base, and the percentage of increase that is required to make the new code base as big as the original one. While both numbers are valid conclusions, the first percentage will always be smaller than the second one. For percentages up to 10%, the difference is small, but for 20% and above, the difference is significant.

### The Original Code Base

In the original code base, we used manual forwarding to implement bidirectional associations. For a 1-side of an association, that means forwarding call to `get` and `set`, 2 LOC each, and 1 LOC for the instance variable for the object managing the association, giving a total of 5 LOC for each 1-side. For an n-side, we need to forward `get`, `add`, `remove`, and `replace`. Note that the `replace` method cannot be substituted by calls to `add` and `remove` because this would not respect the order of elements in source files, which is not acceptable for statements and still annoying for methods, variables, and inner classes. Actually, the code would need additional checks for a `null` reference, but we will ignore them since they are not present for every association and since `replace` is also missing most of the time. Taking that into account, we conclude that on average an n-side uses 9 LOC. The code base uses 84 1-sides, and 25 n-sides, which means that the associations make up 645 LOC.

### The Standard Code Base

If we had used standard techniques for implementing the bidirectional associations, our code base would have been larger. For the unidirectional associations we already used the standard technique.

The general implementation for a 1-side of a bidirectional association contains 14 LOC as shown in Figure 32. For an n-side, that is 21 LOC as shown in Figure 33. The `register` methods are required for modifying the other end of an association.

If we would use this technique for associations, the associations would take up 1991 LOC, which is 1046 LOC more than in the small code base. The results are displayed in Figure 27.

### The Events Code Base

We are currently building an Eclipse editor for Chameleon that operates independent of the actual object-oriented programming language of the source file. For this, it is essential that the user interface elements such as the text file and the outline remain synchronized with the underlying metamodel instance. A common technique for achieving this is using the Observer pattern [34]. Every time an object of the model is modified, it notifies the listeners to trigger updates of the user interface. If we use component to model all association, unidirectional and bidirectional, we can get this functionality for free by using the appropriate components. If we add this functionality to the standard model, that would require 1 LOC for the collection of listeners, 3 LOC for `addListener`, 3 LOC for `removeListener` (both including a call to the method for lazy creation of the collection), 3 LOC for lazy creation of the collection of listeners, and at least 2 LOC for notifying the listeners using an iteration, giving a total of 12 extra LOC per association.

```

private T other;

public T getX() {
    return other;
}

public void setX(T other) {
    if(this.other != other) {
        register(other);
        if(other != null) {
            other.register(this);
        }
    }
}

void register(T other) {
    if (this.other != null) {
        this.other.unregister();
    }
    this.other = other;
}

void unregister() {
    other = null;
}

```

Figure 32: Standard implementation of a 1-side.

For a total of 160 associations, this results in an additional 1920 LOC.

### SmartEiffel

For SmartEiffel, the methods of the associations must be renamed to resolve the conflicts, and abstract methods must be renamed and implemented for resolving dependencies. For a 1-side, this results in an additional 14 LOC (5 renaming + 3 dependencies). For an n-side this is 16 LOC (7 renaming + 3 dependencies).

If events are added, this costs an additional 4 renaming per characteristic.

### Discussion

Figure 30 shows the code size for the different techniques and code bases. Figure 31 shows the reduction in size compared to Java. Almost all of the reduction is obtained in the domain model, which takes up 70% of the software. The other 30% consists of input and output algorithms.

First of all, we must note that the reduction in code size is *not* the same as the reduction in complexity. Renaming clauses and manual delegation are much simpler than the reused methods.

Both figures clearly show that our inheritance mechanism results in a much bigger reduction than other approaches. The difference is caused by the additional overhead mentioned above. Manual delegation and code inheritance in SmartEiffel reduce the size much less

```

private SomeOrderedSet<T> others;

public SomeOrderedSet<T> getX() {
    return new SomeCollection<T>(other);
}

public void addX(T other) {
    if(other != null) {
        other.register(this);
    }
    register(other);
}

public void removeX(T other) {
    if(other != null) {
        other.unregister(this);
    }
    unregister(other);
}

public void replace(T original, T replacement) {
    int index = others.indexOf(original);
    if((index != -1) && (original != replacement)) {
        others.set(index, replacement);
        replacement.register(this);
        original.unregister(this);
    }
}

void register(T other) {
    others.add(other);
}

void unregister(T other) {
    others.remove(other);
}

```

**Figure 33: Standard implementation of a 1-side.**

than our mechanism, but are still a big improvement over the Java version. Using traits, however, the code size even increases. The additional getter and setter methods – traits cannot contain state – cause so much additional overhead that the application becomes bigger than the original Java application.

An important result is the impact of adding functionality that is not overridden in the application. Adding support for sending events requires *no modification* of the version using our inheritance mechanism. The renaming parameters, component parameters, and indirect inheritance avoid the need for additional code if all methods and variables added to the `default` group contain existing renaming parameters. With all other techniques, code

$$\begin{aligned}
P &::= \overline{L} e \\
L &::= \text{class } C(\overline{\alpha}) \overline{ST} \overline{SC} \{ \overline{F} \overline{K} \overline{M} \} \\
\alpha &::= T \rightarrow C \\
\delta &::= \alpha \mid i \\
ST &::= \text{subtype } C(\overline{\delta}) [\overline{n} = \overline{\sigma}] \\
SC &::= \text{component } C(\overline{\delta}) i [\overline{n} = \overline{\sigma}] \\
M &::= C m(\overline{C} \overline{x}) \{ \text{return } e i \} \\
e &::= x \mid i \mid e.f \mid e.i \mid e@ \alpha \mid e.m(\overline{e}) \mid \text{new } C(\overline{e}) \mid (C)e
\end{aligned}$$

Figure 34: Syntax.

must be added to the applications for renaming clauses, manual delegations, dependency methods, or state encapsulation. The more functionality offered by the component, the more modifications are required by other techniques. This is a very important practical result. A first consequence is that the developer of a component can now add functionality without breaking any client code. A second consequence is that he can now provide lots of functionality without putting a huge burden on his clients.

Another important result shows up if validation is added to the associations, and specific validation rules are implemented in the applications. The version using our inheritance mechanism is the only one in which less code must be added than in the Java version, as shown by the gradients in the right part of Figure 30. This means that it is still beneficial to reuse small components, or small parts of bigger components, using our inheritance mechanism. Using the other techniques, the additional overhead makes reuse unattractive in these scenarios.

## 6. FORMAL MODEL

In this section, we present a part of our formal model. More details can be found in the technical report [90]. Our model is based both on ClassicJava [33] and Featherweight Java [42]. Because our inheritance mechanism supports renaming, the static type of the target is required to determine the invoked method or accessed field. We use the type elaboration of ClassicJava to incorporate that information in the program. The rest of the model is based on Featherweight Java because of its simplicity.

To model the essence of our inheritance mechanism, we added multiple inheritance, separation of subtyping and code inheritance, named inheritance relations, component parameters, component references, indirect inheritance, and simple renaming to the Featherweight Java model. Other elements have been omitted to keep the model simple. In case of a conflict, the conflicting elements must be overridden by a new definition. Because we do not model component classes, non-conformance and feature hiding are not allowed. We assume that all component relations have been given a name, and that classes with component parameters are abstract.

### 6.1 Syntax

The syntax of the language is shown in Figure 34. The differences with Featherweight Java are the component parameters  $\alpha$ , the two inheritance relations, and the expressions  $e.i$  for component references and  $e@ \alpha$  for invocations on component parameters. The subtyping relation cannot have a name. Variable  $\delta$  ranges over both component parameters and inheritance names.

$$\begin{array}{c}
C <: C \\
\frac{C <: D \quad D <: E}{C <: E} \\
\frac{\text{class } C \dots \text{subtype } D \dots \{ \dots \}}{C <: D}
\end{array}$$

**Figure 35: Subtyping**

$$\begin{array}{c}
C < C \\
\frac{C <: D}{C < D} \\
\frac{C < D \quad D < E}{C < E} \\
\frac{\text{class } C \dots \text{component } D \dots \{ \dots \}}{C < D}
\end{array}$$

**Figure 36: Subclassing**

## 6.2 Type Elaboration

Because methods can be renamed, we must perform type elaboration, as done in [33] for static methods. They combine the elaboration rules and the well-formedness rules. The elaboration rules for our model are almost the same as those for ClassicJava. The only difference is that we also elaborate the static type of the target of a method invocation while in ClassicJava this is done only for instance variables. We do not repeat them here. The only effect is the insertion of the static type of the target of a method invocation or instance variable access.

$$\begin{array}{l}
e.m(\text{args}) \Rightarrow \underline{e:\Gamma(e)}.m(\text{args}) \\
e.f \Rightarrow \underline{e:\Gamma(e)}.f
\end{array}$$

The typing of the non-elaborated program is almost identical to that of the elaborated program except that the actual type are used instead of the static types in rules  $T - \text{field}$ ,  $T - \text{Comp}$ ,  $T - \text{Comp} - \text{Param}$ , and  $T - \text{invk}$ .

The well-formedness rules are written separate from the elaboration.

## 6.3 Subtyping and Subclassing

The subtyping rules and subclassing are shown in Figures 35 and 36. The subtyping rules come straight from Featherweight Java[42]. The subclassing rules are similar. Note that the second judgment declares that the subtyping relation implies the subclassing relation. The subtyping relation is represented by the  $<:$  relation, the subclassing relation by the  $<$  relation.

## 6.4 Class Well-formedness

Figure 37 shows the class well-formedness rules. Rules  $V\text{-No-ST-Loops}$  and  $V\text{-No-CO-Loops}$  ensure that no cycles are present in the inheritance relations. Note that  $V\text{-No-CO-Loops}$

$$\begin{array}{c}
\frac{E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC} \{ \overline{F} \overline{K} \overline{M} \}}{\frac{\forall \text{subtype } T \langle \bar{\delta} \rangle \quad [ \bar{n} = \bar{p} ] \in \overline{ST} : \neg T \langle : C}{E \text{ NO-ST-LOOPS OK}} \text{ (V-No-ST-Loops)}} \\
\frac{E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC} \{ \overline{F} \overline{K} \overline{M} \}}{\frac{\forall \text{component } T \langle \bar{\delta} \rangle \quad i \quad [ \bar{n} = \bar{p} ] \in \overline{ST} : \neg T \langle : C}{\frac{\forall \text{component } T \langle \bar{\delta} \rangle \quad i \quad [ \bar{n} = \bar{p} ] \in \overline{ST} : \neg T \langle : C}{E \text{ NO-CO-LOOPS OK}} \text{ (V-No-CO-Loops)}} \\
\frac{\forall T, \forall ST_a, ST_b \in \overline{ST} : \quad ST_a \langle : T \wedge ST_b \langle : T \Rightarrow \text{params}(ST_a, T) = \text{params}(ST_b, T)}{\frac{E \text{ COMPONENT-PARAM OK}}{\text{ (V-Component-Params)}} \\
\frac{E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC} \{ \overline{F} \overline{K} \overline{M} \}}{\frac{\forall X, Y \in \text{componentNames}(C) \cup \text{fields}(C) \cup \text{methods}(C) : \quad X \neq Y \Rightarrow \text{name}(X) \neq \text{name}(Y)}{E \text{ NAMESPACE OK}} \text{ (T-Namespace)}} \\
\frac{\text{componentNames}(C) = \{ i \mid \text{component}(i, C) = X \text{ CO} \}}{\frac{E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC} \{ \overline{F} \overline{K} \overline{M} \}}{\frac{\forall \text{class } T(\bar{\gamma}) \dots : \quad ST_i = \text{subtype } T(\bar{\delta}) \dots \Rightarrow \bar{\delta} \leq \bar{\gamma} \quad \forall ST_i = \text{component } T(\bar{\delta}) \dots : \bar{\delta} \leq \bar{\gamma}}{E \text{ COMP PARAMS OK}} \text{ (T-Inherited-Classes)}} \\
\frac{\alpha = T \rightarrow C \quad \delta = i \quad \text{component}(i, T) = S \rightarrow D \quad D \langle : C \quad T \langle : S}{\delta \leq \alpha} \\
\frac{\alpha = T \rightarrow C \quad \delta = S \rightarrow D \quad D \langle : C \quad T \langle : S}{\delta \leq \alpha}
\end{array}$$

Figure 37: Class Well-formedness.

*Loops* has an extra judgements to forbid cycles mixing both kinds of relations. Rule (*V-Component-Params*) ensures that if component parameterse are passed to the same type via different inheritance paths, the values are the same in both cases. This is similar to the rule for generic parameters in Eiffel and SmartEiffel. Rule *T-Namespace* put all named elements of a class in the same namespace and demands that all names are unique within a class. Rule (*T-Inherited-Classes*) demands that the types and component parameters of all inheritance relations are valid.

These rules must be conjugated together with the rules for fields and methods, which are discussed further on.

## 6.5 Components

Figure 38 shows the lookup rules for components.

Figure 40 shows the rules for overriding components. Figure 41 shows the relations between components.

$$\begin{array}{c}
\frac{E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC}\{\overline{FKM}\} \quad \text{component } T \langle \bar{\beta} \rangle [\bar{n}=\bar{p}] \quad i \in \overline{SC}}{\text{component}(i, C) = C \text{ component } T \langle \bar{\beta} \rangle [\bar{n}=\bar{p}] \quad i} \\
E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC}\{\overline{FKM}\} \quad \text{component } T \langle \bar{\beta} \rangle [\bar{n}=\bar{p}] \quad i \notin \overline{SC} \\
\text{subtype } T [\bar{n}=\bar{p}] \quad s \in \overline{ST} \quad \text{component}([\bar{n}/\bar{p}]i, T) = X \text{ component } Y [\bar{u}=\bar{v}] \quad x \\
\hline
\text{component}(i, C) = X \text{ component } Y [\bar{u}=\bar{v}] \quad x \\
\hline
\text{component}(i, \text{subtype } T \langle \bar{n} = \bar{p} \rangle [\dots]j) = \text{component}([\bar{p}/\bar{n}]i, T)
\end{array}$$

Figure 38: Lookup of components.

$$\begin{array}{c}
\text{params}(\text{class } T \langle \bar{\alpha} \rangle \overline{ST}, T, \bar{\delta}) = \bar{\alpha} \\
\text{params}(\text{class } T \langle \bar{\alpha} \rangle \overline{ST}, S, \bar{\delta}) = \text{params}([\bar{\delta}/\bar{\alpha}]\overline{ST}, S) \\
\frac{T_j <: S \quad E = \text{class } T_j \langle \bar{\alpha} \rangle \overline{ST} \overline{SC}\{\overline{FKM}\}}{\text{params}(\text{subtype } T \langle \bar{\delta} \rangle [\bar{n}=\bar{p}], S) = \text{params}(E, S, \sigma(\bar{\delta}, \bar{\alpha}))} \\
\frac{\alpha = T \rightarrow C \quad \delta = i}{\sigma(\delta, \alpha) = T.i} \\
\frac{\alpha = T \rightarrow C \quad \delta = S \rightarrow D}{\sigma(\delta, \alpha) = \delta}
\end{array}$$

Figure 39: Component parameters.

$$\begin{array}{c}
E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC}\{\overline{FKM}\} \quad ST_i = \text{subtype } T \langle \bar{\delta} \rangle [\bar{o}=\bar{p}] \\
SC_j = \text{component } c S \langle \bar{\gamma} \rangle [m=n] \quad \text{component}([\bar{p}/\bar{o}]c, T) = U \text{ component } V \langle \bar{\varepsilon} \rangle x [\bar{o}=\bar{p}] \\
\hline
C \text{ } SC_j \text{ overrides } U \text{ component } V \langle \bar{\varepsilon} \rangle x [\bar{q}=\bar{r}] \\
\frac{C \text{ } F \text{ overrides } D \text{ } G \quad D \text{ } G \text{ overrides } E \text{ } H}{C \text{ } F \text{ overrides } E \text{ } H} \\
E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC}\{\overline{FKM}\} \quad SC_i = \text{component } S \langle \bar{\gamma} \rangle [\bar{o}=\bar{q}] \quad s \\
\forall X, SC_T = \text{component } T \langle \bar{\delta} \rangle [\bar{n}=\bar{p}] \quad t : \\
C \text{ } SC_i \text{ overrides } X \text{ } SC_T \Rightarrow S <: T \\
\hline
C \text{ } S_i \text{ OVERRIDE OK}
\end{array}$$

Figure 40: Component overriding.

$$\begin{array}{c}
\frac{\exists E \text{ } O : (C \text{ } M \text{ overrides } E \text{ } O) \wedge (D \text{ } N \text{ overrides } E \text{ } O)}{C \text{ } M \text{ related to } D \text{ } N} \\
\frac{C \text{ } M \text{ overrides } D \text{ } N}{C \text{ } M \text{ related to } D \text{ } N}
\end{array}$$

Figure 41: Component relations.

$$\begin{array}{c}
E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC}\{\overline{F} K \overline{M}\} \\
\frac{C \overline{SC} \text{ OVERRIDE OK}}{E \text{ COMPONENT OVERRIDE OK}} \text{ (T-Comp-Override)} \\
E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC}\{\overline{F} K \overline{M}\} \\
\frac{\forall n : |\{i | \text{component}(n, ST_i) = \dots\}| > 1 \Rightarrow \\
\exists j : SC_j = \text{componentT}\langle \dots \rangle [\dots] \quad n}{E \text{ COMPONENT SELECT OK}} \text{ (T-Comp-Select)} \\
E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC}\{\overline{F} K \overline{M}\} \\
\frac{\forall i, j, n_1, n_2 : \\
\text{component}(n_1, ST_i) \text{ related to } \text{component}(n_2, ST_j) \Rightarrow n_1 = n_2}{E \text{ NO COMPONENT DUPLICATION OK}} \text{ (T-No-Comp-Duplication)}
\end{array}$$

**Figure 42: Component well-formedness.**

6.5.1 *Component Well-formedness.* Figure 42 shows the well-formedness rules for component relations.

## 6.6 Fields

6.6.1 *Field Lookup.* A field is represented as  $P \ F$ , where  $F$  is the definition of the field, and  $P$  is its enclosing class. This is needed to determine the origin of a field. Indirect inheritance is modeled by giving indirectly inherited fields the name `inheritanceName.f`.

Figure 43 shows the definition of the *fields* function. Rules 1 and 2 are trivial. Rules 3 and 4 determine which fields *are* inherited by the subtyping and component relations respectively. The difference between both functions is that *inh<sub>st</sub>* incorporates the rule-of-dominance. It ignores definitions that are overridden by a definition inherited via another subtyping relation. In addition, if the same definition is inherited more than once via subtyping with different names, the type rules demand that all versions are given the same name. This makes them syntactically equal, after which the set definition merges them. This is not the case for the component relation, where duplication is the default policy. Rules 5 and 6 determine which fields *can be* inherited via a specific subtyping or component relation. They take the fields of the inherited class, and apply the renaming  $\tau_{st}$  and  $\tau_{co}$ . These functions are defined in rules 7-10 and 11-12 respectively. The  $\tau_{st}$  function is divided in four parts. In rules 8 and 9, no renaming is done. Rule 7 deals with the renaming of an individual field, while rule 10 deals with renaming as a consequence of renaming a component relation. Note that the latter rules change the parent class of the method to the inheriting class. Rules 11 and 12 do the same for the component relation, but they also change the `this` reference by `this.inheritanceName`.

Figure 44 shows the *field* function, which is used to find a field, given its name and the static (T) and actual (C) types of the target. Rule 13 covers the case where the name of the requested field is in *fields(T)*. Rule 14 covers method that are inherited directly, but are accessed indirectly. They are not present directly in *fields(T)*, but there is a trail of *overrides* and *same as* relations between the requested method and a method in *fields(T)*. Renaming of component relations is taken into account by looking up the actual relation, which may have a different name than `head`.

The *overrides* relation is shown in Figure 46. Rule 15 describes the standard overriding relation for subtyping. Rule 16 for the component relation is similar, but it inserts the name of the component relation before the name of the overridden field to distinguish it from other inheritance instances of the same field. In addition, it introduces indirectly inherited fields in the inheriting class. If the field is inherited directly, the indirect version can still be used, and is dynamically bound because of the *overrides* relation. Rule 17 makes the *overrides* relation transitive, and rule 18 takes the *same as* relation into account. Conformance of fields is enforced by rule 19, and rule 20 determines if a field is overridden in a class.

The *same as* relation is shown in Figure 47. Rules 21 and 24 state that the *same as* relation is reflexive and transitive. Rules 22 and 23 state that a renamed feild is the same as the field with the previous name. As for overriding, the name of the component relation is added for fields renamed in a component relation.

Figure 45 shows the field type lookup.

6.6.2 *Field Well-formedness.* Figure 48 shows the field well-formedness rules.



$$\frac{field(f, T, C) = U \ D \ g}{ftype(f, T, C) = D}$$

**Figure 45: Field type lookup.**

$$\frac{\begin{array}{l} ST_k = \text{subtype } T < \bar{\delta} > \ [ \bar{\alpha} = \bar{\beta} ] \\ E = \text{class } C < \bar{\alpha} > \overline{ST} \overline{SC} \{ \bar{F} \bar{K} \bar{M} \} \quad \text{class } T < \bar{\beta} > \ \dots \\ \tau_{st}(\bar{\delta}, \bar{\alpha} = \bar{\beta}, \bar{\beta}, C, D \ A \ g) = C \ A \ f_j \quad D \ A \ g \in \text{fields}(T) \end{array}}{C \ F_j \ \text{overrides } D \ A \ g} \quad (15)$$

$$\frac{\begin{array}{l} ST_k = \text{component } T < \bar{\delta} > \ i \ [ \bar{\alpha} = \bar{\beta} ] \\ E = \text{class } C < \bar{\alpha} > \overline{ST} \overline{SC} \{ \bar{F} \bar{K} \bar{M} \} \quad \text{class } T < \bar{\beta} > \ \dots \\ \tau_{co}(T, \bar{\delta}, i, \bar{\alpha} = \bar{\beta}, \bar{\beta}, C, D \ A \ g) = C \ A \ f_j \quad D \ A \ g \in \text{fields}(T) \end{array}}{C \ F_j \ \text{overrides } D \ A \ i.g} \quad (16)$$

$$\frac{\begin{array}{l} C \ F \ \text{overrides } D \ G \vee C \ F \ \text{same as } D \ G \\ D \ G \ \text{overrides } E \ H \end{array}}{C \ F \ \text{overrides } E \ H} \quad (17)$$

$$\frac{\begin{array}{l} C \ F \ \text{overrides } D \ G \quad D \ G \ \text{same as } E \ H \\ C \ F \ \text{overrides } E \ H \end{array}}{C \ F \ \text{overrides } E \ H} \quad (18)$$

$$\frac{\begin{array}{l} B = T \ f \\ \forall C \ D = C \ U \ g : A \ B \ \text{overrides } C \ D \Rightarrow T = U \\ A \ B \ \text{OVERRIDE OK} \end{array}}{A \ B \ \text{OVERRIDE OK}} \quad (19)$$

$$\frac{\begin{array}{l} E = \text{class } C < \bar{i} > \overline{ST} \overline{SC} \{ \bar{F} \bar{K} \bar{M} \} \quad g \in \bar{F} \\ D \ g \ \text{overridden in } E \end{array}}{D \ g \ \text{overridden in } E} \quad (20)$$

**Figure 46: Field overriding.**

$$\frac{}{C \ F \ \text{same as } C \ F} \quad (21)$$

$$\frac{\begin{array}{l} E = \text{class } C < \bar{i} > \overline{ST} \overline{SC} \{ \bar{F} \bar{K} \bar{M} \} \\ ST_i = \text{subtype } T \ [ \bar{\alpha} = \bar{\beta} ] \ i \quad D \ A \ g \in \text{fields}(T) \\ \tau_{st}([\bar{\alpha} = \bar{\beta}], C, i, D \ G) = C \ A \ h \quad \text{-h overridden in } E \end{array}}{C \ A \ h \ \text{same as } D \ A \ g} \quad (22)$$

$$\frac{\begin{array}{l} E = \text{class } C < \bar{i} > \overline{ST} \overline{SC} \{ \bar{F} \bar{K} \bar{M} \} \\ ST_i = \text{component } T \ [ \bar{\alpha} = \bar{\beta} ] \ i \quad D \ A \ g \in \text{fields}(T) \\ \tau_{co}([\bar{\alpha} = \bar{\beta}], i, C, D \ G) = C \ A \ h \quad \text{-h overridden in } E \end{array}}{C \ A \ h \ \text{same as } D \ A \ i.g} \quad (23)$$

$$\frac{\begin{array}{l} C \ F \ \text{same as } D \ G \quad D \ G \ \text{same as } E \ H \\ C \ F \ \text{same as } E \ H \end{array}}{C \ F \ \text{same as } E \ H} \quad (24)$$

**Figure 47: Field equivalence.**

$$\begin{array}{c}
E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC} \{\overline{F} \ K \ \overline{M}\} \\
\frac{\forall D \ N, E \ O \in \text{fields}(\overline{ST}, C) : D \ N \text{ related to } E \ O \Rightarrow n = o}{E \ \text{NO FIELD SUBTYPING DUPLICATION} \quad (\text{F-No-Subtyping-Duplication})} \\
\\
E = \text{class } C < \bar{i} > \overline{ST} \overline{SC} \{\overline{F} \ K \ \overline{M}\} \\
\frac{\forall o : \left( \begin{array}{l} |\{N=P \ D \ g|g=o \wedge U \ N \in (\text{inh}_{f, st}(E))\}| > 1 \\ \Rightarrow \exists B \ o \in \overline{F} \end{array} \right)}{\text{class } C < \bar{i} > \overline{ST} \overline{SC} \{\overline{F} \ K \ \overline{M}\} \text{ F-SELECT OK} \quad (\text{F-Select})} \\
\\
E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC} \{\overline{F} \ K \ \overline{M}\} \\
SC_i = \text{component } S < \bar{\gamma} > j \ [\bar{\gamma} = \bar{z}] \\
\forall X, CO = \text{component } T < \bar{\beta} > \ [\bar{q} = \bar{r}] \ i : C \ SC_i \ \text{overrides } X \ CO \Rightarrow \\
\forall U \ N \in \text{fields}(\overline{ST}, C), V \ O \in \text{fields}(T) : \\
U \ N \ \text{related to } X \ \dots i.o(\dots) \ \{\dots\} \Rightarrow \\
\forall C \ P \in \text{fields}(SC_i, C) : \\
C \ P \ \text{related to } B=W \ \dots j.s(\dots) \ \{\dots\} \wedge \text{field}(o, T, S) = B \Rightarrow \\
n=p \\
\hline
(\text{F-Component-Override-Renaming}) \\
E \ \text{COMPONENT FIELD OVERRIDE RENAME OK} \\
\\
E = \text{class } C < \bar{\alpha} > \overline{ST} \overline{SC} \{\overline{F} \ K \ \overline{M}\} \\
\frac{C \ \overline{F} \ \text{OVERRIDE OK}}{\text{class } C < \bar{i} > \overline{ST} \overline{SC} \{\overline{F} \ K \ \overline{M}\} \text{ FIELD OVERRIDE OK} \quad (\text{F-Override})}
\end{array}$$

Figure 48: Field well-formedness.

## 6.7 Methods

6.7.1 *Method Lookup.* A method is represented as  $P \ M$ , where  $M$  is the definition of the method, and  $P$  is its enclosing class. This is needed to determine the origin of a method. Indirect inheritance is modeled by giving indirectly inherited methods the name `inheritanceName.m`.

Figure 49 shows the definition of the *methods* function. Rules 25 and 26 are trivial. Rules 27 and 28 determine which methods *are* inherited by the subtyping and component relations respectively. The difference between both functions is that *inh<sub>st</sub>* incorporates the rule-of-dominance. It ignores definitions that are overridden by a definition inherited via another subtyping relation. In addition, if the same definition is inherited more than once via subtyping with different names, the type rules demand that all versions are given the same name. This makes them syntactically equal, after which the set definition merges them. This is not the case for the component relation, where duplication is the default policy. Rules 29 and 30 determine which methods *can be* inherited via a specific subtyping or component relation. They take the methods of the inherited class, and apply the renaming, and substitution of component parameters by using  $\tau_{st}$  and  $\tau_{co}$ . These functions are defined in rules 31-34 and 35-36 respectively. The  $\tau_{st}$  function is divided in four parts. In rules 32 and 33, no renaming is done. Rule 31 deals with the renaming of an individual method, while rule 34 deals with renaming as a consequence of renaming a component relation. Note that the latter rules change the parent class of the method to the inheriting class. Rules 35 and 36 do the same for the component relation, but they also change the `this` reference by `this.inheritanceName`. Finally, the  $\tau$  function in rules 37 and 38, substitutes the component parameters. If the replacement is a component parameter, the `@` symbol must be kept. If the value is the name of an actual component relation, it is replaced by a dot.

Figure 50 shows the *method* function, which is used to find a method, given its name and the static (T) and actual (C) types of the target. Rule 39 covers the case where the name of the requested method is in *methods(T)*. Rule 40 covers method that are inherited directly, but are accessed indirectly. They are not present directly in *methods(T)*, but there is a trail of *overrides* and *same as* relations between the requested method and a method in *methods(T)*. Renaming of component relations is taken into account by looking up the actual relation, which may have a different name than `head`.

The *overrides* relation is shown in Figure 53. Rule 41 describes the standard overriding relation for subtyping. Rule 43 for the component relation is similar, but it inserts the name of the component relation before the name of the overridden method to distinguish it from other inheritance instances of the same method. In addition, it introduces indirectly inherited methods in the inheriting class. If the method is inherited directly, the indirect version can still be used, and is dynamically bound because of the *overrides* relation. Rule 42 makes the *overrides* relation transitive, and rule 44 takes the *same as* relation into account. Conformance of methods is enforced by rule 45, and rule 46 determines if a method is overridden in a class.

The *same as* relation is shown in Figure 54. Rules 47 and 48 state that the *same as* relation is reflexive and transitive. Rules 49 and 50 state that a renamed method is the same as the method with the previous name. As for overriding, the name of the component relation is added for methods renamed in a component relation.

Figure 51 shows the method type lookup, and Figure 52 shows the method body lookup.

Both rules are trivial.

### 6.7.2 Method Well-formedness.

## 6.8 Auxiliary functions

6.8.1 *Abstract.* Since we the formal model demands that inheritance parameters are filled in by a subclass, we make classes with inheritance parameters abstract.

## 6.9 Expression Typing

Because fields, methods, and inheritance relations can be renamed, we need to perform type elaboration as done in [33].

Because there is no type for  $a . b$  if  $b$  is an inheritance relation, there can be no ambiguity for  $\text{expr} . f$  in case of e.g.  $a . b . c . d . e$ . If  $b$  and  $c$  are variables, the only valid match is  $\text{expr}=a . b . c$  and  $f=d . e$ .

Note that there are no reduction rules for  $T - \text{Comp} - \text{Field}$  and  $T - \text{Comp} - \text{Invk}$  because they cannot occur in a running program. Types with inheritance parameters are abstract, and all parameters must be filled in by subclasses.

Because the formal model allows references to subcomponents, we must add a condition to  $T - \text{Field}$  and  $T - \text{Ink}$  to ensure that only a single rule can be applicable at a time. Otherwise, there would be two possibilities for  $e.i.f$  and  $e.i.m()$ , being either  $(e.i).f$  and  $(e.i).m()$  or  $(e).i.f$  and  $(e).i.m()$ . The same goes for  $e@i.f$  and  $e@i.m()$ .

## 6.10 Reduction Rules

Note that the  $\Delta$  environment is not needed because it is only required for the compile-time type-check. Also note that there is no rule for  $e@i$  because the parameter will be substituted during method selection by the actual component name. This is proven in Lemma 6.9.

Also note that there is not even a rule for  $e.i$  if  $i$  is an inheritance relation. The expression will remain unchanged until it is the target of a method invocation or field access. In both case only one rule applies:  $(e).i.f$  and  $(e).i.m()$ .

$$\frac{E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC} \{\overline{F} \overline{K} \overline{M}\}}{\text{methods}(C) = \text{methods}(E)} \quad (25) \quad \frac{E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC} \{\overline{F} \overline{K} \overline{M}\}}{\text{methods}(E) = \overline{C} \overline{F} \cup \text{inh}_{st}(E) \cup \text{inh}_{co}(E)} \quad (26)$$

$$\text{inh}_{st}(E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC} \{\overline{F} \overline{K} \overline{M}\}) = \{ \cup N | \neg N \text{ overridden in } E \wedge \cup N \in \text{methods}(\overline{ST}, C) \wedge \exists V O \in \text{methods}(\overline{ST}, C) : V O \neq \cup N \wedge V O \text{ overrides } \cup N \} \quad (27)$$

$$\text{inh}_{co}(E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC} \{\overline{F} \overline{K} \overline{M}\}) = \{ \cup N | \neg N \text{ overridden in } E \wedge \cup N \in \text{methods}(\overline{SC}, C) \} \quad (28)$$

$$\frac{\text{class } T(\bar{\alpha}) \dots \quad \text{methods}(T) = \overline{P} \overline{M}}{\text{methods}(\text{subtype } T(\bar{\delta}) \text{ } [\bar{n}=\bar{o}], C) = \tau_{st}(\bar{\delta}, \bar{n} = \bar{o}, \bar{\alpha}, C, \overline{P} \overline{M})} \quad (29)$$

$$\frac{\text{class } T(\bar{\alpha}) \dots \quad \text{methods}(T) = \overline{P} \overline{M}}{\text{methods}(\text{component } T(\bar{\delta}) \text{ } i \text{ } [\bar{n}=\bar{o}], C) = \tau_{co}(T, \bar{\delta}, i, \bar{n} = \bar{o}, \bar{\alpha}, C, \overline{P} \overline{M})} \quad (30)$$

$$\frac{m \in \bar{n}}{\tau_{st}(\bar{\delta}, \bar{n} = \bar{o}, \bar{\alpha}, C, P \ B \ m(\overline{B} \ \overline{x}) \ \{\text{return } e;\}) = C \ B \ [\bar{o}/\bar{n}]m(\overline{B} \ \overline{x}) \ \{\text{return } \tau(\bar{\delta}, \bar{\alpha}, e);\}} \quad (31)$$

$$\frac{m \notin \bar{n} \quad . \notin m}{\tau_{st}(\bar{\delta}, \bar{n} = \bar{o}, \bar{\alpha}, C, P \ B \ m(\overline{B} \ \overline{x}) \ \{\text{return } e;\}) = P \ B \ m(\overline{B} \ \overline{x}) \ \{\text{return } \tau(\bar{\delta}, \bar{\alpha}, e);\}} \quad (32)$$

$$\frac{m \notin \bar{n} \quad m = \text{head.tail} \quad . \notin \text{head} \quad \text{head} \notin \bar{n}}{\tau_{st}(\bar{\delta}, \bar{n} = \bar{o}, \bar{\alpha}, C, P \ B \ m(\overline{B} \ \overline{x}) \ \{\text{return } e;\}) = P \ B \ m(\overline{B} \ \overline{x}) \ \{\text{return } \tau(\bar{\delta}, \bar{\alpha}, e);\}} \quad (33)$$

$$\frac{m \notin \bar{n} \quad m = \text{head.tail} \quad . \notin \text{head} \quad \text{head} \in \bar{n}}{\tau_{st}(\bar{\delta}, \bar{n} = \bar{o}, \bar{\alpha}, C, P \ B \ m(\overline{B} \ \overline{x}) \ \{\text{return } e;\}) = C \ B \ ([\bar{o}/\bar{n}]\text{head}).\text{tail}(\overline{B} \ \overline{x}) \ \{\text{return } \tau(\bar{\delta}, \bar{\alpha}, e);\}} \quad (34)$$

$$\frac{m \in \bar{n}}{\tau_{co}(T, \bar{\delta}, i, \bar{n} = \bar{o}, \bar{\alpha}, C, P \ B \ m(\overline{B} \ \overline{x}) \ \{\text{return } e;\}) = C \ B \ [\bar{o}/\bar{n}]m(\overline{B} \ \overline{x}) \ \{\text{return } [\underline{\text{this:C.i}}/\underline{\text{this:T}}]\tau(\bar{\delta}, \bar{\alpha}, e);\}} \quad (35)$$

$$\frac{m \notin \bar{n}}{\tau_{co}(T, \bar{\delta}, i, \bar{n} = \bar{o}, \bar{\alpha}, C, P \ B \ m(\overline{B} \ \overline{x}) \ \{\text{return } e;\}) = C \ B \ i.m(\overline{B} \ \overline{x}) \ \{\text{return } [\underline{\text{this:C.i}}/\underline{\text{this:T}}]\tau(\bar{\delta}, \bar{\alpha}, e);\}} \quad (36)$$

$$\frac{\delta = T \rightarrow C}{\tau(\delta, \alpha, e) = [\delta/\alpha]e} \quad (37) \quad \frac{\delta = i}{\tau(\delta, \alpha, e) = [\delta/\alpha]e} \quad (38)$$

Figure 49: Methods of a class.

$$\frac{\begin{array}{l} \text{methods}(\mathbb{T}) = \overline{\mathbb{V}}\overline{\mathbb{N}} \quad \text{name} = n_i \quad \cup \text{M} \in \text{methods}(\mathbb{C}) \\ \cup \text{M overrides } \mathbb{V}_i \mathbb{N}_i \vee \cup \text{M same as } \mathbb{V}_i \mathbb{N}_i \end{array}}{\text{method}(\text{name}, \mathbb{T}, \mathbb{C}) = \text{M}} \quad (39)$$

$$\begin{array}{l} \text{name} = \text{head.tail} \\ \quad . \notin \text{head} \\ \text{component}(\text{head}, \mathbb{T}) = \text{X component } \mathbb{Y}(\overline{\delta}) \text{ i } [\overline{\mathbb{n}}=\overline{\sigma}] \\ \quad \mathbb{N} = \mathbb{B} \text{ i.tail}(\overline{\mathbb{B}} \overline{\mathbb{x}}) \{ \dots \} \\ \quad \cup \mathbb{B} \text{ tail}(\overline{\mathbb{B}} \overline{\mathbb{x}}) \{ \dots \} \in \text{methods}(\mathbb{Y}) \\ \quad (\cup \text{M overrides } \text{X } \mathbb{N} \vee \cup \text{M same as } \text{X } \mathbb{N}) \\ \quad \cup \text{M} \in \text{methods}(\mathbb{C}) \\ \hline \text{method}(\text{name}, \mathbb{T}, \mathbb{C}) = \text{M} \end{array} \quad (40)$$

**Figure 50: Method lookup.**

$$\frac{\text{method}(\text{m}, \mathbb{T}, \mathbb{C}) = \cup \mathbb{B} \text{ n}(\overline{\mathbb{B}} \overline{\mathbb{x}}) \{ \text{return } e; \}}{\text{mtype}(\text{m}, \mathbb{T}, \mathbb{C}) = \overline{\mathbb{B}} \rightarrow \mathbb{B}}$$

**Figure 51: Method type lookup.**

$$\frac{\text{method}(\text{m}, \mathbb{T}, \mathbb{C}) = \cup \mathbb{B} \text{ n}(\overline{\mathbb{B}} \overline{\mathbb{x}}) \{ \text{return } e; \}}{\text{mbody}(\text{m}, \mathbb{T}, \mathbb{C}) = \overline{\mathbb{x}}.e}$$

**Figure 52: Method body lookup.**

$$\begin{array}{c}
ST_j = \text{subtype } T(\bar{\delta}) \text{ } [\bar{\sigma}=\bar{\rho}] \quad \text{class } T\langle\bar{\beta}\rangle \dots \\
E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC}\{\bar{F} \bar{K} \bar{M}\} \quad N = A \text{ n}(\bar{A} \bar{x})\{\text{return } e_i\} \\
\tau_{st}(\bar{\delta}, \bar{\sigma}=\bar{\rho}, \bar{\beta}, C, D \text{ N}) = C \text{ A } m_i \dots \quad D \text{ N} \in \text{methods}(T) \\
\hline
C \text{ M}_i \text{ overrides } D \text{ N} \quad (41)
\end{array}$$

$$\begin{array}{c}
D \text{ N overrides } E \text{ O} \\
\hline
C \text{ M overrides } D \text{ N} \vee C \text{ M same as } D \text{ N} \quad (42) \\
C \text{ M overrides } E \text{ O}
\end{array}$$

$$\begin{array}{c}
SC_j = \text{component } T(\bar{\delta}) \text{ } i \text{ } [\bar{\sigma}=\bar{\rho}] \quad \text{class } T\langle\bar{\beta}\rangle \dots \\
E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC}\{\bar{F} \bar{K} \bar{M}\} \quad N = A \text{ n}(\bar{A} \bar{x})\{\text{return } e_i\} \\
\tau_{co}(T, \bar{\delta}, i, \bar{\sigma}=\bar{\rho}, \bar{\beta}, C, D \text{ N}) = C \text{ A } m_i \dots \quad D \text{ N} \in \text{methods}(T) \\
\hline
C \text{ M}_i \text{ overrides } D \text{ A } i.\text{n}(\bar{A} \bar{x})\{\text{return } e_i\} \quad (43)
\end{array}$$

$$\begin{array}{c}
C \text{ M overrides } D \text{ N} \quad D \text{ N same as } E \text{ O} \\
\hline
C \text{ M overrides } E \text{ O} \quad (44)
\end{array}$$

$$\begin{array}{c}
M = B \text{ m}(\bar{B} \bar{x})\{\text{return } e_i\} \\
\forall D \text{ N}=A \text{ n}(\bar{A} \bar{y})\{\dots\} : C \text{ M overrides } D \text{ N} \Rightarrow (B <: A \wedge \bar{B} = \bar{A}) \\
\hline
C \text{ M OVERRIDE OK} \quad (45)
\end{array}$$

$$\begin{array}{c}
E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC}\{\bar{F} \bar{K} \bar{M}\} \quad m \in \bar{M} \\
\hline
B \text{ m}(\bar{B} \bar{x}) \{\text{return } e_i\} \text{ overridden in } E \quad (46)
\end{array}$$

**Figure 53: Method overriding.**

$$\frac{}{C \ M \ same \ as \ C \ M} \quad (47) \quad \frac{C \ F \ same \ as \ D \ G \quad D \ G \ same \ as \ E \ H}{C \ F \ same \ as \ E \ H} \quad (48)$$

$$\begin{aligned} & E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC} \{ \overline{FK} \overline{M} \} \\ & ST_j = \text{subtype } T(\bar{\delta}) \ [ \overline{\sigma} = \overline{\rho} ] \\ & \tau_{st}(\bar{\delta}, [ \overline{\sigma} = \overline{\rho} ], \bar{\beta}, C, D \ N) = C \ A \ h(\bar{A} \ \bar{x}) \{ \text{return } e; \} \\ & \quad \text{class } T(\bar{\beta}) \ \dots \\ & \quad N = A \ n(\bar{A} \ \bar{x}) \{ \text{return } e; \} \\ & \quad D \ N \in \text{methods}(T) \\ & \quad \text{-h overridden in } E \\ & \hline & C \ A \ h(\bar{A} \ \bar{x}) \{ \text{return } e; \} \ same \ as \ D \ N \quad (49) \end{aligned}$$

$$\begin{aligned} & E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC} \{ \overline{FK} \overline{M} \} \\ & SC_j = \text{component } T(\bar{\delta}) \ i \ [ \overline{\sigma} = \overline{\rho} ] \\ & \tau_{co}(T, \bar{\delta}, i, [ \overline{\sigma} = \overline{\rho} ], \bar{\beta}, C, D \ N) = C \ A \ h(\bar{A} \ \bar{x}) \{ \text{return } g; \} \\ & \quad \text{class } T(\bar{\beta}) \ \dots \\ & \quad N = A \ n(\bar{A} \ \bar{x}) \{ \text{return } e; \} \\ & \quad D \ N \in \text{methods}(T) \\ & \quad \text{-h overridden in } E \\ & \hline & C \ A \ h(\bar{A} \ \bar{x}) \{ \text{return } g; \} \ same \ as \ D \ A \ i.n(\bar{A} \ \bar{x}) \{ \text{return } e; \} \quad (50) \end{aligned}$$

Figure 54: Method equivalence.

$$\frac{\exists E \ O : (C \ M \ overrides \ E \ O) \wedge (D \ N \ overrides \ E \ O)}{C \ M \ related \ to \ D \ N} \quad (51) \quad \frac{C \ M \ overrides \ D \ N}{C \ M \ related \ to \ D \ N} \quad (52)$$

$$\frac{C \ M \ same \ as \ D \ N}{C \ M \ related \ to \ D \ N} \quad (53)$$

Figure 55: Method relations.

$$\begin{array}{c}
E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC} \{ \overline{F} \ K \ \overline{M} \} \\
\frac{\forall D \ N, E \ O \in \text{methods}(\overline{ST}, C) : D \ N \text{ related to } E \ O \Rightarrow n = o}{E \ \text{NO METHOD SUBTYPING DUPLICATION}} \text{ (M-No-Subtyping-Duplication)} \\
\\
E = \text{class } C < \bar{i} > \overline{ST} \overline{SC} \{ \overline{F} \ K \ \overline{M} \} \\
\forall o : \left( \begin{array}{l} |\{N=T \ A \ n(\overline{A} \ \overline{Y}) \{ \text{return } f; \} |_{n=o} \wedge \cup N \in (\text{inheritedMethods}(E))\}| > 1 \\ \Rightarrow \exists B \ o(\overline{B} \ \overline{X}) \{ \text{return } e; \} \in \overline{M} \end{array} \right) \\
\hline
\text{class } C < \bar{i} > \overline{ST} \overline{SC} \{ \overline{F} \ K \ \overline{M} \} \text{ METHOD SELECT OK} \text{ (M-Select)} \\
\\
E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC} \{ \overline{F} \ K \ \overline{M} \} \\
SC_i = \text{component } S < \overline{\gamma} > [ \overline{Y} = \overline{Z} ] \ j \\
\forall X, CO = \text{component } T < \overline{\beta} > [ \overline{Q} = \overline{R} ] \ i : C \ SC_i \text{ overrides } X \ CO \Rightarrow \\
\forall \cup N \in \text{methods}(\overline{ST}, C), \forall O \in \text{methods}(T) : \\
\cup N \text{ related to } X \ \dots i.o(\dots) \ \{ \dots \} \Rightarrow \\
\forall C \ P \in \text{methods}(SC_i, C) : \\
C \ P \text{ related to } B=W \ \dots j.s(\dots) \ \{ \dots \} \wedge \text{method}(o, T, S) = B \Rightarrow \\
n=p \\
\hline
\text{(M-Component-Override-Renaming)} \\
E \ \text{COMPONENT METHOD OVERRIDE RENAME OK} \\
\\
E = \text{class } C < \bar{\alpha} > \overline{ST} \overline{SC} \{ \overline{F} \ K \ \overline{M} \} \\
\frac{C \ \overline{M} \ \text{OVERRIDE OK}}{\text{class } C < \bar{i} > \overline{ST} \overline{SC} \{ \overline{F} \ K \ \overline{M} \} \text{ METHOD OVERRIDE OK}} \text{ (T-Override)} \\
\\
E = \text{class } C < \bar{\alpha} > \overline{ST} \overline{SC} \{ \overline{F} \ K \ \overline{M} \} \\
\frac{C \ \overline{M} \ \text{IMPLEMENTATION OK}}{\text{class } C < \bar{i} > \overline{ST} \overline{SC} \{ \overline{F} \ K \ \overline{M} \} \text{ IMPLEMENTATION OK}} \text{ (T-Implementation)} \\
\\
\frac{\overline{x} : \overline{C}, \text{this} : C \vdash e_0 : E_0 \quad E_0 <: C_0}{C \ C_0 \ m(\overline{C} \ \overline{x}) \{ \text{return } e_0 \} \text{ IMPLEMENTATION OK}}
\end{array}$$

Figure 56: Method well-formedness

$$\frac{E = \text{class } C < \bar{\alpha} > \overline{ST} \overline{SC} \{ \overline{F} \ K \ \overline{M} \} \quad \bar{\alpha} \neq \emptyset}{C \ \text{abstract}}$$

Figure 57: The abstract judgement.

$$\begin{array}{c}
\frac{}{\Delta; \Gamma \vdash x : \Gamma(x)} \text{ (T-var)} \\
\frac{\Delta; \Gamma \vdash e_0 : C_0 \quad \text{field}(f, T, C_0) = D \ g \quad . \notin f}{\Delta; \Gamma \vdash \underline{e_0} : T.f : D} \text{ (T-Field)} \\
\frac{\text{component}(i, T) = \text{component } D \langle \bar{\beta} \rangle \ [ \bar{n} = \bar{p} ] \ j}{\Delta; \Gamma \vdash \underline{e_0} : T.i : D} \text{ (T-Comp)} \\
\frac{\Delta(\alpha) = T \rightarrow C}{\Delta; \Gamma \vdash \underline{e_0} : T@\alpha : C} \text{ (T-Comp-Param)} \\
\frac{\Delta; \Gamma \vdash e_0 : C_0 \quad \text{mtype}(m, T, C_0) = \bar{D} \rightarrow C \quad \Delta; \Gamma \vdash \bar{e} : \bar{C} \quad \bar{C} <: \bar{D} \quad . \notin m}{\Delta; \Gamma \vdash \underline{e_0} : T.m(\bar{e}) : C} \text{ (T-Invk)} \\
\frac{\neg C \text{ abstract} \wedge \text{fields}(C) = \bar{T} \ \bar{D} \ \bar{F} \quad \Delta; \Gamma \vdash \bar{e} : \bar{C} \quad \bar{C} <: \bar{D}}{\Delta; \Gamma \vdash \text{new } C(\bar{e}) : C} \text{ (T-New)} \\
\frac{\Delta; \Gamma \vdash e_0 : D \quad D <: C}{\Delta; \Gamma \vdash (C)e_0 : C} \text{ (T-UCast)} \\
\frac{\Delta; \Gamma \vdash e_0 : D \quad C <: D \quad C \neq D}{\Delta; \Gamma \vdash (C)e_0 : C} \text{ (T-DCast)} \\
\frac{\Delta; \Gamma \vdash e_0 : D \quad C \not<: D \quad D \not<: C \quad \text{stupid warning}}{\Delta; \Gamma \vdash (C)e_0 : C} \text{ (T-SCast)}
\end{array}$$

Figure 58: Expression typing.

$$\begin{array}{c}
\frac{\text{field}(f, T, C) = D \ g \quad \text{fields}(C) = \bar{U} \ \bar{D} \ \bar{G} \quad g = g_i}{\text{new } C(\bar{e}) : T.f \rightarrow e_i} \text{ (R-Field)} \\
\frac{\text{mbody}(m, T, C) = \bar{x}.e_0}{\text{new } C(\bar{e}) : T.m(\bar{d}) \rightarrow [\bar{d}/\bar{x}, \text{new } C(\bar{e})/\text{this}]e_0} \text{ (R-Invk)} \\
\frac{C <: D}{(D)(\text{new } C(\bar{e})) \rightarrow \text{new } C(\bar{e})}
\end{array}$$

Figure 59: Computation Rules.

$$\begin{array}{c}
\frac{e_0 \rightarrow e'_0}{\underline{e_0} : \underline{T.f} \rightarrow \underline{e'_0} : \underline{T.f}} \quad (RC - Field) \\
\\
\frac{e_0 \rightarrow e'_0}{\underline{e_0} : \underline{T.m(\bar{e})} \rightarrow \underline{e'_0} : \underline{T.m(\bar{e})}} \quad (RC - Invk - Rcv) \\
\\
\frac{e_i \rightarrow e'_i}{\underline{e_0} : \underline{T.m(\dots, e_i, \dots)} \rightarrow \underline{e_0} : \underline{T.m(\dots, e'_i, \dots)}} \quad (RC - Invk - Arg) \\
\\
\frac{e_i \rightarrow e'_i}{\text{new } C(\dots, e_i, \dots) \rightarrow \text{new } C(\dots, e'_i, \dots)} \quad (RC - New - Arg) \\
\\
\frac{e_0 \rightarrow e'_0}{(C)e_0 \rightarrow (C)e'_0} \quad (RC - Cast)
\end{array}$$

**Figure 60: Congruence Rules.**

## 6.11 Proof of Type Soundness

We prove the type soundness of our model by isolating the assumptions made in the proof of Featherweight Java, and proving that our inheritance mechanism satisfies those assumptions.

6.11.1 *Subject Reduction.* In Lemma 6.6, we must make a change because our *mtype* function has an extra argument, being the static type of the target. In the proof, the lemma is used to prove that the type of a method invocation is a subtype after substitution. In other words, it suffices to prove that for a given static type, a more specific actual type will result in a more specific return type.

LEMMA 6.1. *The component judgement defines a function.*

PROOF. This follows directly from rule  $T - \text{Comp} - \text{Select}$ .  $\square$

LEMMA 6.2. *The method judgement defines a function.*

PROOF. From the definition of *method*, it follows that it is a function if *methods* is a function. Rule  $T - \text{Namespace}$  ensures that for methods directly defined in a class  $C$ , there can only be one match for a given name. The other methods in  $\text{methods}(C)$  come from either  $\text{inh}_{st}$  or  $\text{inh}_{co}$ . But because these functions remove methods with the same name as a method in  $C$ , there can never be more than one result.  $\square$

LEMMA 6.3. *Every method  $m$  in a supertype or component of class  $C$  is either in the set of methods of  $C$ , or an overriding or equal method is present in that set.*

$$E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC} \{ \overline{F} \overline{K} \overline{M} \} \wedge U \ N \in \text{methods}(ST_i) \cup \text{methods}(SC_i)$$

$\Downarrow$

$$\exists V \ O \in \text{methods}(C) : V \ O \text{ overrides } U \ N \vee V \ O \text{ same as } U \ N$$

PROOF. The only reason a method  $m$  of a supertype or component of class  $C$  can be absent from the set of methods of  $C$  is that it has been removed by either the  $\text{inh}_{st}$  or  $\text{inh}_{co}$  function.

For  $\text{inh}_{st}$ , this means that the method is either overridden in the class itself, or an overriding method has also been inherited. In both cases there is an overriding relation between the removed method and a method in  $\text{methods}(C)$  according to the definition of *overrides*. If the method is not removed, the *same as* relation holds.

The proof for  $\text{inh}_{co}$  is similar.  $\square$

LEMMA 6.4. *The field judgement defines a function.*

PROOF. Similar to that of Lemma 6.2  $\square$

LEMMA 6.5. *Every field  $f$  in a supertype or component of class  $C$  is either in the set of fields of  $C$ , or a field overriding it is present in that set.*

$$E = \text{class } C(\bar{\alpha}) \overline{ST} \overline{SC} \{ \overline{F} \overline{K} \overline{M} \} \wedge U \ D \ G \in \text{fields}(ST_i) \cup \text{fields}(SC_i)$$

$\Downarrow$

$$\exists V \ E \ h \in \text{fields}(C) : V \ E \ h \text{ overrides } U \ D \ G \vee V \ E \ h \text{ same as } U \ D \ G$$

PROOF. Similar to that of Lemma 6.3.  $\square$

LEMMA 6.6. *If  $\text{mtype}(m, T, C) = \bar{B} \rightarrow B_0$  then  $\text{mtype}(m, T, D) = \bar{B} \rightarrow E_0$  with  $E_0 <: B_0$  for all  $S <: T, D <: S$*

PROOF. From Lemma 6.3, it follows that  $method(m, T, D)$  selects a method that either overrides or is the same as  $method(m, T, C)$ . As a result, this lemma follows from the `OVERRIDE OK` judgement.  $\square$

Because we also allow overriding of fields, we must have a similar lemma for the type of a field. The only difference is that the type of a field cannot change.

LEMMA 6.7. *If  $f_{type}(\mathcal{F}, T, C) = \bar{B}$  then  $f_{type}(\mathcal{F}, T, D) = \bar{B}$  for all  $S \prec: T, D \prec: S$*

PROOF. Similar to the proof of Lemma 6.6  $\square$

Lemma 6.8 needs to be modified from the Featherweight Java version because we have two new type expressions.

LEMMA 6.8. *If  $\Delta; \Gamma, \bar{x} : \bar{B} \vdash e : D$ , and  $\Gamma \vdash \bar{d} : \bar{A}$  where  $\bar{A} \prec: \bar{B}$ , then  $\Gamma \vdash [\bar{d}/\bar{x}]e : C$  for some  $C \prec: D$ .*

PROOF. Cases  $T - Var$ ,  $T - New$ ,  $T - UCast$ ,  $T - DCast$ , and  $T - SCast$  remain unchanged. Case  $T - Invk$  only requires a syntactic modification to pass the static type from the type elaboration to Lemma 6.6. Case  $T - Field$  become nearly identical to  $T - Invk$  because of the possible overriding.

Case T-Comp.  $e = \underline{e_0:T}.i$

The actual type of  $e_0$  is not used in the type of  $\underline{e_0:T}.i$ , hence the type of  $e$  will not change under substitution. Safety is guaranteed by induction. The component relation will exist.

Case T-Comp.  $e = \underline{e_0:T}@_\alpha$

Again, the actual type of  $e_0$  is not used in the type of  $\underline{e_0:T}@_\alpha$ , hence the type of  $e$  will not change under substitution. Safety is guaranteed by induction. The component relation will exist.  $\square$

LEMMA 6.9. *The expression  $e : T@_\alpha$  will not be encountered during evaluation of a program.*

PROOF. Classes with component parameters are abstract and thus cannot be instantiated. From the fact that all parameters must be filled in by subtype and component clauses, and the substitution of component parameters in the *methods* function, it follows that such expressions cannot occur in methods of concrete classes. The proof follows from the fact that a program is of the form `new C( $\bar{\epsilon}$ )` and must be well-typed.  $\square$

LEMMA 6.10. *If  $\Delta; \Gamma \vdash e : C$ , then  $\Delta; \Gamma, x : D \vdash e : C$*

PROOF. Straightforward.  $\square$

LEMMA 6.11. *The transformation of the method body of a method inherited via a subtyping relation is type safe.*

$$\begin{array}{c}
\text{class } T(\bar{\alpha}) \dots \wedge \\
\bar{\alpha} : \bar{B} \rightarrow \bar{D}, \bar{x} : \bar{X}, \text{this} : T \vdash e : E \wedge \bar{\delta} \leq \bar{\alpha} \wedge \\
\text{class } S(\bar{\beta}) \dots \text{subtype } T(\bar{\delta}) \dots \\
\Downarrow \\
\bar{x} : \bar{X}, \text{this} : S \vdash \tau(\bar{\delta}, \bar{\alpha}, e) : F \wedge F <: E.
\end{array}$$

PROOF. The lemma follows directly from the definition of the  $\tau$  function and Lemma 6.8.  $\square$

LEMMA 6.12. *The transformation of the method body of a method inherited via a component relation is type safe.*

$$\begin{array}{c}
\text{class } T(\bar{\alpha}) \dots \wedge \\
\bar{\gamma} : \bar{G}, \bar{x} : \bar{X}, \text{this} : T \vdash e : E \wedge \bar{\delta} \leq \bar{\alpha} \wedge \\
\text{class } S(\bar{\beta}) \dots \text{component } T(\bar{\delta}) \text{ i } \dots \\
\Downarrow \\
\bar{x} : \bar{X}, \text{this} : S \vdash [\underline{\text{this}} : S. i / \text{this} : T] \tau(\bar{\delta}, \bar{\alpha}, e) : F \wedge \\
F <: E.
\end{array}$$

PROOF. Because of rule  $T - \text{Comp}$ , the type of  $\underline{\text{this}} : C. i$  is  $T$ , so the lemma follows from Lemmas 6.8 and 6.11.  $\square$

For Lemma 1.4 of the Featherweight Java proof, we provide a different lemma because the method body is altered when inherited instead of during the evaluation.

LEMMA 6.13. *If  $\text{type}(m, T, C) = \bar{D} \rightarrow D$ , and  $\text{mbody}(m, T, C)$ , then  $\bar{x} : \bar{D}, \text{this} : C_0 \vdash e : C$  with  $C <: D$*

PROOF. In case  $\text{method}(m, T, C)$  is defined in  $C$ , rule  $T - \text{Implementation}$  proves the Lemma. In case  $\text{method}()$  is inherited either through a subtype relation or a component relation, we must prove that the altered method body maintains a valid type. This follows directly from Lemmas 6.11 and 6.12.  $\square$

THEOREM 6.14 (SUBJECT REDUCTION). *For a well-typed expression  $e$  of an elaborated program:*

*If  $\Gamma \vdash e : C$  and  $e \rightarrow e'$ , then  $\Gamma \vdash e' : C'$  for some  $C' <: C$*

PROOF. The proof is nearly identical to the proof of Theorem 2.4.1 of Featherweight Java.  $\square$

### 6.11.2 Progress

THEOREM 6.15 (PROGRESS). *Suppose  $e$  is a well-typed expression in the evaluation of an elaborated program.*

- (1) *If  $e$  includes  $\text{new } C_0(\bar{e}).f$  as a subexpression, then  $\text{fields}(C_0) = \bar{C} \bar{T} \bar{F}$  and  $f \in \bar{F}$  for some  $\bar{C}, \bar{T}$ , and  $\bar{F}$ .*
- (2) *If  $e$  includes  $\text{new } C_0(\bar{e}).m(\bar{d})$  as a subexpression, then  $\text{mbody}(m, C_0) = \bar{x}.e_0$  and  $\#(\bar{x}) = \#(\bar{d})$  for some  $\bar{x}$  and  $e_0$ .*

PROOF. Because component parameters cannot occur in the evaluation of an elaborated program (Lemma 6.9), and operations performed on component references are treated as

method invocations and field accesses, the proof is the same as that of Theorem 2.4.2 of Featherweight Java except that it now uses Lemmas 6.2, 6.3, 6.4, and 6.5 to prove that  $m$  and  $f$  are present.  $\square$

6.11.3 *Type Soundness.* The domain of values is the same as that of Featherweight Java.

$$v ::= \text{new } C(\bar{v})$$

**THEOREM 6.16 (TYPE SOUNDNESS).** *Suppose  $e$  is an expression of an elaborated program. If  $\emptyset \vdash e : C$  and  $e \rightarrow^* e'$  with  $e'$  in normal form, then  $e'$  is either a value  $v$  with  $\emptyset \vdash v : D$  and  $D < C$ , or an expression containing  $(D)\text{new } C(\bar{v})$  where  $C < D$ .*

**PROOF.** Immediate from Theorems 6.14 and 6.15.  $\square$

## 7. RELATED WORK

In [61], Odersky and Zenger identify three scalable component abstractions for removing hard references from components to increase their reusability: *abstract type members*, *self-type annotations*, and *modular mixin composition*. Abstract type members and selftypes specify the required services of a component, and mixins perform the composition. But while these abstractions are scalable with respect to the size of the components, they are not scalable in the way components are used. The problem is that both selftypes, and mixins as used in Scala, prohibit any composition involving multiple components of the same kind, or components containing features with the same name. Despite the claim that these abstractions can lift an arbitrary assembly of static program parts to a component system, they already fail for our simple example application, which is little more than an assembly of four kinds of static program parts. The authors argue that nesting of classes is essential because otherwise, the amount of wiring would become substantial. This contradicts our findings. In this paper, we built an application using components without using nested classes. So while nested classes provide certain benefits, they are not a requirement for component composition. In both approaches, components are classes, and the result of the composition of components is again a class.

In [88], we introduced anchored exception declarations to remove hard references from the exceptional specification of a component. They allow the exceptional specification of a method to be declared relative to other methods. This increases both the adaptability and reusability of code using checked exceptions. More specifically, they simplify the reuse of higher-order functions by taking the exceptional behavior of the actual function parameter into account instead of providing an inflexible upper bound that forces the programmer to write many inconvenient and dangerous error handlers.

ArchJava [2] uses ports to connect components. A component declares the methods provided and required by a port. Composition of components is done by connecting ports to each other. The difference with our approach is that in ArchJava, ports are used to enforce communication constraints, while component relations are used to compose an ADT from other ADTs. In our approach, a port corresponds to a regular class which is then used in a component using a component relation that inherits all features indirectly. The class representing the port can declare its requirements using instance variables, abstract getter methods, or component parameters. So as a composition mechanism, our inheritance mechanism is more flexible, but it does not enforce communication constraints. Our

```

component CToDoModular is compose {
    provides IQueue q;
    provides ISupervisor su;
    intro CToDoExtension toDo;
    intro CList list;
    plug list.list into toDo.list;
    plug todo.q into q;
    plug toDo.su into su;
}

```

**Figure 61: The ComponentJ version of CToDoModular.**

```

class CToDoModular
  component CToDoExtension<CTodoModular> (list) [direct{q,su}]
  component CList list
  {}

```

**Figure 62: Our version of CToDoModular.**

mechanism for preventing component references – using component classes – is simpler and more effective than that of ArchJava. In ArchJava, this is accomplished by prohibiting the use of the type of a component in the ports and public interfaces of a component type, and types of instance variables. In addition, an exception is thrown if a cast to a component type is thrown. We prevent the use of `this`, and names of component relations as a separate expressions for component classes.

ComponentJ [76] only uses ports for composition of components, and thus is less flexible than our approach. In addition, it is more verbose, as shown by Figures 61 and 62. In Figure 62, the `CToDoExtension` component is connected to the `CList` component using a component parameter. Another alternative is to use an instance variable or an abstract getter method.

In [8], the authors present a language construct for first-class full-blown relationships. Such a language construct is also advocated by Rumbaugh in [68]. With our inheritance mechanism, this language construct can be replaced by component classes. In this paper, we used relationships without attributes, but a component for full-blown relationships can be built on top of them. An example implementation is given in Figure 63. A `PassiveBidiAssociation` is a `BidiAssociation` that cannot be modified from that end of the association. This is necessary to ensure consistency of the relationship.

In the 1997 version of Eiffel [58], the inheritance relation is used both for subtyping and code inheritance. It is possible to duplicate features when inheriting more than once from the same class, which is confusing for classification purposes as argued in Section 3.2. The resulting diamond problem for repeated inheritance is often considered to make the language more difficult [11; 70]. In addition, a subclass can use covariant argument types for a method, or even remove features, which makes a whole program analysis required to ensure type safety. In SmartEiffel 2.2 [20] and the new Eiffel specification [62], the inheritance mechanism has been extended with non-conforming inheritance. In SmartEiffel 2.2, duplication of features and narrowing their visibility is no longer permitted. Using covariant argument types, however, remains possible. SmartEiffel ensures type-safety by

```

public abstract class Relationship<FROM,TO,
    KIND extends Relationship<FROM,TO,KIND>>
    [FromName,ToName]
    (FROM → PassiveBidiAssociation<FROM,KIND> from)
    (TO → PassiveBidiAssociation<TO,KIND> to)
component BidiAssociation1<KIND,FROM>
    (from) [X=%FromName%,
        override unregister%FromName%]
        export private {set%FromName%,register%FromName%}] {
component BidiAssociation1<KIND,TO>
    (to) [X=%ToName%,
        override unregister%ToName%,
        export private {set%ToName%,register%ToName%}] {
public Relationship(from,to) {
    set%FromName%(from)
    set%ToName%(to)
}
protected void unregister%FromName%(FROM from) {
    super(from);
    set%ToName%(null);
}
protected void unregister%ToName%(TO to) {
    super(to);
    set%FromName%(null);
}
...
}

public class Attends
    subtype Relationship<Student,Course,Attends> (courses,students)
        [FromName=Student,ToName=Course]
    component UniAssociation<int> [X=Mark] {...}

public class Student
    component PassiveBidiAssociationSet<Student,Attends> courses
    ...
public class Course
    component PassiveBidiAssociationSet<Course,Attends> students
    ...

```

Figure 63: Full-blown relationships.

type-checking the code of an inserted class in the context of the inheriting class, but this violates the modularity principle, as argued in Section 2.6. Because sharing is the default policy for the *insert* relation, accidental merging of components is possible. Our work can mainly be thought of as an extension of the SmartEiffel inheritance mechanism to allow convenient composition of abstract data types.

Sather [84] and Timor [44] separate types and classes, and the relations between them. Types can inherit from multiple other types, and classes can *include* other classes for code

inheritance. In addition, classes can implement types. We do not favor the mandatory separation of types and classes since it always requires a heavyweight solution.

Timor has support for named subtyping relations [44] to support repeated inheritance. We think this is a bad idea because it does not fit in the classification metaphor. We think it is very confusing for an object to be 1.9 times a `CassettePlayer`, as in their example. They also use the inheritance names to disambiguate conflicting names, but for reusing characteristics this approach is not practical. A severe problem with their mechanism is that name conflicts are automatically resolved by removing direct access to the involved methods. As a result, adding a subtyping relation, or even adding a method to an inherited type can break existing clients without even a warning because conflicts can be introduced. The names of subtyping relations can be used as component references. Timor also has support for *reuse variables*. Features of the classes referenced by such variables are inherited if they are needed for the types implemented by the class. If they are not inherited, however, they are not available to clients since they are not part of the types via which the class can be used. The mechanism can be seen as delegation-by-value. Reuse variables also reduce the dependency of the implementation of a class on its hierarchy, but the authors do not present this insight.

Traits [74] not only use a separate relation for code inheritance, but also a separate concept – a *trait* – for a set of methods that can be reused via code inheritance. Unlike traits, we do not have a separate concept to represent a component, it is just a class. If the component relation could only be used with special building blocks, unanticipated reuse would be impossible. On top of that, programmers must deal with an extra concept which is just a degenerate abstract class. Another motivation for our choice is the possibility to instantiate characteristics. We see no reason to forbid a programmer to create an object that represents a bounded value. In addition, classification of characteristics is necessary. To reuse almost any kind of association, it is necessary to create a hierarchy of association classes. The relation between classes capturing choices like mutability, arity, . . . and class `Association` is a subtyping relation, not just a code inheritance relation. Methods inherited via traits automatically override methods inherited from classes although there is no relation between them. This form of structural subtyping can lead to bugs that are hard to find. In addition, dependencies of traits must be resolved individually, and repeated trait-inheritance is not possible. As such, traits allow far less code reuse than our inheritance mechanism. In [10], traits are used to refactor the Smalltalk collection classes. The authors report a 12% reduction in code size.

In [66], Reppy and Turon present trait-based metaprogramming. They add renaming and hiding to traits to allow using a trait more than once in a class. Similar to SmartEiffel, name conflicts and dependencies must be resolved one at a time. Because traits cannot contain state, the overhead is larger than in SmartEiffel.

Languages like CLOS [25], most mixin-based [11] languages like Scala [61], and many others use linearized multiple inheritance. The linearization of the class hierarchy, however, complicates its use [78; 19; 74]. It is not possible to determine the meaning of a single inheritance relation of a class without looking at the others because some of its methods may be overridden by methods of other classes that happen to have the same name. This makes it easy for methods to be overridden by accident [78]. Repeated inheritance, which is essential language for composition of classes is impossible in these languages. The abstract super class of a mixin, however, allows for reusable refinements, which cannot

easily be created using our approach. More research is required to decide how abstract super classes should be integrated in our inheritance mechanism.

Cecil [18] supports multiple inheritance. Repeated inheritance, however, is forbidden, and name conflicts result in compilation errors. The language uses properties for instance variables, making it possible to override them. Subtyping and code inheritance relations can be used both separately or combined.

In Self [19], inheritance relations are given a priority. For relations with identical priorities, name conflicts result in an error. For relations with different priorities, conflicts are resolved automatically by inheriting the feature of the relation with the highest priority. The *Sender Path Tiebreaker Rule* resolves additional conflicts by giving priority to methods within the same inheritance path in case of ambiguities. Renaming is not supported. Directed resends do not increase the dependency between the implementation and the inheritance hierarchy because they are sent to named slots, which is very similar to using named inheritance relations.

C++ [80] has limited support for repeated inheritance. A class cannot inherit from the same base class more than once, making it unsuitable for building classes from components. In addition, it has no support for renaming, forcing clients to resolve name conflicts. The language supports separation of subtyping and code inheritance through public and private inheritance.

The Sina/ST language [1] offers an *interface predicate* to determine how calls to an object are dispatched. The type can dispatch calls to the current object or to an object declared in its interface. The predicates (`target.method(args)`) are matched from left to right. If a call matches the name and argument types of a predicate, it is dispatched to its target. The predicate, while providing a lot of flexibility, also brings with it a lot of complexity. Dynamic binding, however, must explicitly be designed in the super class using a `server` call.

There are several mechanisms for building hierarchies of inheritance hierarchies [63; 64; 32; 60; 61]. In these approaches, a hierarchy of classes can be extended by extending the existing classes and introducing new classes. With hierarchy inheritance, extensions to a class of a hierarchy are visible to all other classes of the hierarchy. This approach is complementary to ours: multiple inheritance cannot be used to achieve the benefits of hierarchy inheritance and vice versa. Adding such a mechanism to ours will result in an even higher reusability of code.

In Jigsaw [12], inheritance is presented as an operation on modules. The authors define a number of basic operators to model multiple inheritance, mixins, instantiation, and other techniques. Contrary to their approach, we define two highly specialized operators to match the classification and building block metaphors. To model our inheritance mechanism in Jigsaw, operators must be added to model e.g. indirect inheritance and component parameters. Another difference is that their approach is mainly technical, while ours is more focussed on methodology by focussing on easy to understand metaphors.

In [70], Sakkinen argues that with the possibility of sharing or duplicating state, it is possible that dependent state variables are split because they are not all shared or duplicated. Data groups [51] or method groups [79] can be used to prevent this problem. All state within a group should either be shared or duplicated. Integration of this functionality in our inheritance mechanism remains future work.

In [86], Tobin-Hochstadt and Allen present a calculus of metaclasses which allows them

to build arbitrary hierarchies with *instance of* relations. This allows them to capture aspects of the world that cannot otherwise be expressed [93]. In addition to the *extends* relation for subtyping and code inheritance, they use a *kind* relation to declare that a class is an instance of another class.

The Java Syntactic Extender [5] is a complete macro system for Java [37]. Our renaming parameters provide only an extremely basic macro system, which can only be used to conveniently rename features of a class.

## 8. FUTURE WORK

An important task is to finish our compiler and create a library of reusable components. These include, but are not limited to, a hierarchy of association classes allowing choices like multiplicity, value or reference semantics, mutability, constraints. With these associations, graphs can be built to reuse any iteration over an object structure by incorporating Joost Visser's work on visitor combination and traversal control [91].

The error handling strategy of a class is fixed at this moment. For example, class `BoundedValue` must choose how to deal with invalid input: use preconditions, throw exceptions, or provide a default behavior. That means that to provide all choices to an application developer, we need three versions of the same characteristic. It would be more interesting to have a single version that provides a number of strategies for dealing with errors, and allowing the application developer to choose one.

The abstract super classes of mixins allow a developer to create reusable refinements – classes that wrap their super class to customize its behavior. This is not possible with our inheritance mechanism. More research is needed to decide how abstract super classes should be integrated in our inheritance mechanism.

## 9. CONCLUSION

We have shown that current object-oriented programming languages do not offer the abstraction level required to easily compose an ADT using other ADTs as components. This prevents a developer from reusing high-level concepts like associations, bounded values, graphs, . . . .

Our inheritance mechanism is the first to make this kind of reuse practical. By using renaming parameters and making component relations first-class citizens, we eliminate the problems encountered in current languages. They allow a programmer to easily exploit name patterns, connect components, provide both a simple class interface and lots of functionality, and use components as if they were separate objects. Together, these improvements raise the abstraction level of the programming language, since it is no longer required to create a new language construct or write lots of low-level wiring code to reuse a characteristic.

The case study confirms that our inheritance mechanism yields much better results (21% to 36% reduction) than other mechanisms (4% to 15% reduction). Moreover, it shows that only using our mechanism, components can be extended without the need to modify inheriting classes. In addition, it is still beneficial to reuse small components, or small parts of big components with our inheritance mechanism, contrary to the other techniques.

To allow even easier and more code reuse, we identified *component references* as an existing construct, and introduced two new constructs. *Component classes* modularly restrict access to features of components in presence of component references.

We have made a formal model for our inheritance mechanism, and proved that it is type sound.

## 10. ACKNOWLEDGMENTS

We especially thank Adriaan Moors, Jan Smans, and Tom Schrijvers for their advice. We thank Dominique Colnet, Guillem Marpons, and Frederic Merizen of the SmartEiffel team for their valuable feedback. We also want to thank Bart Jacobs, Jeroen Boydens, Sven De Labey, and Koen Vanderkimpen, who provided many insightful comments. We thank Jan Dockx for the discussions on software reuse, which were very helpful.

## REFERENCES

- M. Aksit and A. Tripathi. Data abstraction mechanisms in SINA/ST. In *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 267–275, New York, NY, USA, 1988. ACM Press.
- J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In *ECOOP*, pages 334–367, 2002.
- P. America. Inheritance and subtyping in a parallel object-oriented language. In *European conference on object-oriented programming on ECOOP '87*, pages 234–242, London, UK, 1987. Springer-Verlag.
- D. Ancona, G. Lagorio, and E. Zucca. Jam - a smooth extension of Java with mixins. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 154–178, London, UK, 2000. Springer-Verlag.
- J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 31–42, New York, NY, USA, 2001. ACM Press.
- M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004 proceedings*, 2004.
- K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford, and P. T. Withington. A monotonic superclass linearization for dylan. *SIGPLAN Not.*, 31(10):69–82, 1996.
- G. M. Bierman and A. Wren. First-class relationships in an object-oriented language. In *ECOOP*, pages 262–286, 2005.
- A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the emerald system. In *OOPSLA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 78–86, New York, NY, USA, 1986. ACM Press.
- A. P. Black, N. Schärli, and S. Ducasse. Applying traits to the Smalltalk collection classes. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–64, New York, NY, USA, 2003. ACM Press.
- G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, New York, NY, USA, 1990. ACM Press.
- G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 282–290, Washington, DC, 1992. IEEE Computer Society.
- L. Cardelli. A semantics of multiple inheritance. *Inf. Comput.*, 76(2-3):138–164, 1988.
- L. Cardelli, editor. *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*, volume 2743 of *Lecture Notes in Computer Science*. Springer, 2003.
- B. Carré; and J.-M. Geib. The point of view notion for multiple inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 312–321, New York, NY, USA, 1990. ACM Press.
- C. Chambers. Object-oriented multi-methods in cecil. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, pages 33–56, London, UK, 1992. Springer-Verlag.
- C. Chambers. Predicate classes. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 268–296, London, UK, 1993. Springer-Verlag.

- C. Chambers. The Cecil language specification and rationale: Version 3.2. 2004.
- C. Chambers, D. Ungar, B.-W. Chang, and U. Hölzle. Parents are shared parts of objects: inheritance and encapsulation in SELF. *Lisp Symb. Comput.*, 4(3):207–222, 1991.
- D. Colnet, G. Marpons, and F. Merizen. Reconciling subtyping and code reuse in object-oriented languages: Using inherit and insert in SmartEiffel, the GNU Eiffel compiler. In *ICSR*, 2006.
- D. Colnet, P. Ribet, C. Adrian, F. Merizen, and G. Marpons. Smarteiffel 2.2, 2005. <http://smarteiffel.loria.fr>.
- W. R. Cook. A proposal for making eiffel type-safe. *Comput. J.*, 32(4):305–311, 1989.
- W. R. Cook, W. Hill, and P. S. Canning. Inheritance is not subtyping. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135, New York, NY, USA, 1990. ACM Press.
- G. Curry, L. Baer, D. Lipkie, and B. Lee. Traits: An approach to multiple-inheritance subclassing. *ACM SIGOA Newsletter*, 3(1-2):1–9, 1982.
- L. G. DeMichiel and R. P. Gabriel. The common lisp object system: An overview. In *ECOOP*, pages 151–170, 1987.
- R. Dixon, T. McKee, M. Vaughan, and P. Schweizer. A fast method dispatcher for compiled languages with multiple inheritance. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 211–214, New York, NY, USA, 1989. ACM Press.
- R. Ducournau and M. Habib. On some algorithms for multiple inheritance in object-oriented programming. In *European conference on object-oriented programming on ECOOP '87*, pages 243–252, London, UK, 1987. Springer-Verlag.
- R. Ducournau, M. Habib, M. Huchard, and M. L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 16–24, New York, NY, USA, 1992. ACM Press.
- D. Duggan and C.-C. Techaubol. Modular mixin-based inheritance for application frameworks. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 223–240, New York, NY, USA, 2001. ACM Press.
- ECMA Technical Committee 39 (TC39) Task Group 2 (TG2). *C# Language Specification*. ECMA, 2 edition, December 2002.
- E. Ernst. Propagating class and method combination. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 67–91, London, UK, 1999. Springer-Verlag.
- E. Ernst. Higher-order hierarchies. In L. Cardelli, editor, *Proceedings ECOOP 2003*, LNCS 2743, pages 303–329, Heidelberg, Germany, July 2003. Springer-Verlag.
- M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183. ACM Press, 1998.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.
- J. Y. Gil and I. Maman. Micro patterns in java code. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 97–116, New York, NY, USA, 2005. ACM Press.
- D. S. Goldberg, R. B. Findler, and M. Flatt. Super and inner: together at last! In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 116–129, New York, NY, USA, 2004. ACM Press.
- J. Gosling et al. *The Java Language Specification, Second Edition*. Addison-Wesley Longman Publishing Co., Inc., 2000.
- O. Gruber, B. J. Hargrave, J. McAffer, P. Rapicault, and T. Watson. The Eclipse 3.0 platform: adopting OSGi technology. *IBM Syst. J.*, 44(2):289–299, 2005.
- I. Harry H. Porter. Separating the subtype hierarchy from the inheritance of implementation. *J. Object Oriented Program.*, 4(6):20–29, 1992.
- T. Hartmann, R. Jungclaus, and G. Saake. Aggregation in a behaviour oriented object model. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, pages 57–77, London, UK, 1992. Springer-Verlag.
- J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, 1992.

- A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
- R. E. Johnson. Type-checking smalltalk. In *OOPSLA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 315–321, New York, NY, USA, 1986. ACM Press.
- J. L. Keedy, C. Heinlein, and G. Menger. Inheriting multiple and repeated parts in Timor. *Journal of Object Technology*, 3(10):99–120, 2004.
- J. L. Keedy, G. Menger, and C. Heinlein. Support for subtyping and code re-use in timor. In *CRPITS '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 35–43, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.
- G. Kniesel. Delegation for java: Api or language extension?
- G. Kniesel. Type-safe delegation for run-time component adaptation. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 351–366, London, UK, 1999. Springer-Verlag.
- J. L. Knudsen. Name collision in multiple classification hierarchies. In *on ECOOP '88 (European Conference on Object-Oriented Programming)*, pages 93–109, London, UK, 1988. Springer-Verlag.
- T. Korson and J. D. McGregor. Understanding object-oriented: a unifying paradigm. *Commun. ACM*, 33(9):40–60, 1990.
- G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, 2000.
- K. R. M. Leino. Data groups: specifying the modification of extended state. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 144–153, New York, NY, USA, 1998. ACM Press.
- K. R. M. Leino, A. Poetsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 246–257, New York, NY, USA, 2002. ACM Press.
- K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: an objective sense of style. In *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 323–334, New York, NY, USA, 1988. ACM Press.
- M. V. Limberghen and T. Mens. Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems. *Object Oriented Systems*, 3:1–30, 1996.
- B. Liskov and J. M. Wing. A new definition of the subtype relation. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 118–141, London, UK, 1993. Springer-Verlag.
- O. L. Madsen, B. Magnusson, and B. Møller Pedersen. Strong typing of object-oriented languages revisited. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 140–150, New York, NY, USA, 1990. ACM Press.
- O. L. Madsen and B. Møller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 397–406, New York, NY, USA, 1989. ACM Press.
- B. Meyer. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., 1997.
- M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings of OOPSLA '02, Sigplan Notices*, 37(11), pages 52–67, 2002.
- N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 99–115, New York, NY, USA, 2004. ACM Press.
- M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 41–57, New York, NY, USA, 2005. ACM Press.
- T. G. . of Technical Committee 39. *ECMA-367 Standard: Eiffel Analysis, Design and Programming Language*. ECMA International, 2005.
- H. Ossher and W. Harrison. Combination of inheritance hierarchies. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 25–40, New York, NY, USA, 1992. ACM Press.

- K. Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 89–110, London, UK, 2002. Springer-Verlag.
- J. Palsberg and M. I. Schwartzbach. Static typing for object-oriented programming. *Sci. Comput. Program.*, 23(1):19–53, 1994.
- J. Reppy and A. Turon. A foundation for trait-based metaprogramming, 2006. International Workshops on Foundations of Object-Oriented Languages.
- J. G. Rossie Jr., D. P. Friedman, and M. Wand. Modeling subobject-based inheritance. *Lecture Notes in Computer Science*, 1098:248–??, 1996.
- J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 466–481, New York, NY, USA, 1987. ACM Press.
- M. Sakkinen. On the darker side of C++. In *on ECOOP '88 (European Conference on Object-Oriented Programming)*, pages 162–176, London, UK, 1988. Springer-Verlag.
- M. Sakkinen. Disciplined inheritance. In *ECOOP*, pages 39–56, 1989.
- C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/Owl. *SIGPLAN Not.*, 21(11):9–16, 1986.
- N. Schärli, A. P. Black, and S. Ducasse. Object-oriented encapsulation for dynamically typed languages. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 130–149, New York, NY, USA, 2004. ACM Press.
- N. Schärli, S. Ducasse, and O. Nierstrasz. Classes = traits + states + glue (beyond mixins and multiple inheritance). In *Proceedings of the International Workshop on Inheritance*, 2002.
- N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.
- N. Schärli, O. Nierstrasz, S. Ducasse, R. Wuyts, and A. Black. Traits: The formal model. Technical Report IAM-02-006, Institut für Informatik, Universität Bern, Switzerland, Nov. 2002. Also available as Technical Report CSE-02-013, OGI School of Science & Engineering, Beaverton, Oregon, USA.
- J. C. Seco and L. Caires. A basic model of typed components. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 108–128, London, UK, 2000. Springer-Verlag.
- Y. Smaragdakis and D. S. Batory. Implementing layered designs with mixin layers. In *ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 550–570, London, UK, 1998. Springer-Verlag.
- A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45, New York, NY, USA, 1986. ACM Press.
- R. Stata and J. V. Guttag. Modular reasoning in the presence of subclassing. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 200–214, New York, NY, USA, 1995. ACM Press.
- B. Stroustrup. *The C++ programming language (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1991.
- P. F. Sweeney and J. Y. Gil. Space and time-efficient memory layout for multiple inheritance. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 256–275, New York, NY, USA, 1999. ACM Press.
- C. Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australia, 1996.
- C. A. Szyperski. Import is not inheritance - why we need both: Modules and classes. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, pages 19–32, London, UK, 1992. Springer-Verlag.
- C. Szyperski, S. Omohundro, and S. Murer. Engineering a programming language: The type and class system of Sather. Technical Report TR-93-064, Berkeley, CA, 1993.
- A. Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479, 1996.

- S. Tobin-Hochstadt and E. Allen. A core calculus of metaclasses, 2005. International Workshops on Foundations of Object-Oriented Languages.
- M. van Dooren and N. Smeets. Jnome, 2006. <http://www.jnome.org>.
- M. van Dooren and E. Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 455–471, New York, NY, USA, 2005. ACM Press.
- M. van Dooren and E. Steegmans. Language constructs for improving reusability in object-oriented software. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 118–119, New York, NY, USA, 2005. ACM Press.
- M. van Dooren and E. Steegmans. Abstract data type components. Technical Report CW 439, K.U.Leuven, March 2006.
- J. Visser. Visitor combination and traversal control. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 270–282, New York, NY, USA, 2001. ACM Press.
- P. Wegner and S. B. Zdonik. Inheritance as an incremental modification mechanism or what Like is and isn't Like. In *on ECOOP '88 (European Conference on Object-Oriented Programming)*, pages 55–77, London, UK, 1988. Springer-Verlag.
- C. A. Welty and D. A. Ferrucci. What's in an instance? Technical report, Rochester Polytechnic Institute Computer Science Dept., 1994.