The Fragile Base Class Problem and Its Solution

Leonid Mikhajlov
Emil Sekerinski

Turku Centre for Computer Science
TUCS Technical Report No 117
June 1997
ISBN 952-12-0020-0
ISSN 1239-1891

Abstract

In this paper we study the fragile base class problem. This problem occurs in open object-oriented systems employing code inheritance as an implementation reuse mechanism. System developers unaware of extensions to the system developed by its users may produce a seemingly acceptable revision of a base class which may damage its extensions. The fragile base class problem becomes apparent during maintenance of open object-oriented systems, but requires consideration during design. We formulate the fragile base class problem in terms of a flexibility property. By means of five orthogonal examples demostrating that the flexibility property does not hold, we introduce requirements for disciplining inheritance. We extend the refinement calculus to accommodate for classes, objects, class-based inheritance, and class refinement. We formulate and formally prove a flexibility theorem showing that the restrictions we impose on inheritance are sufficient to permit substituting a base class with its revision in presence of extension classes. We also present three check lists for system designers, developers, and extenders to allow application of our results in practice informally.

Keywords: fragile base class problem, code inheritance, self-recursion, substitutability, class refinement.

TUCS Research Group
Programming Methodology Research Group

1 Introduction

The name *fragile base class problem* gives a good intuitive understanding of the problem. Classes in the foundation of an object-oriented system can be fragile. The slightest attempt to modify these foundation classes may damage the whole system.

The problem does not depend on the system implementation language. However, systems employing code inheritance as an implementation reuse mechanism along with self-recursion [25] are vulnerable to it. Such systems are delivered to users as a collection of classes. The users can reuse functionality provided by the system by inheriting from system classes. Moreover, when the open system is an object-oriented framework, the users can extend system functionality by substituting its classes with derived ones. In general, system developers are unaware of extensions developed by the users. Attempting to improve functionality of the system, the developers may produce a seemingly acceptable revision of system classes which may damage the extensions. In a closed system all extensions are under control of system developers, who, in principle, can analyze the effect of certain base class revisions on the entire system. Although possible in principle, in practice this becomes infeasible [26, 21]. In an open system the fragile base class problem requires consideration during design [29].

We have encountered several different interpretations of the problem in the technical literature on this topic. Often, during a discussion of component standards, the name is used to describe the necessity to recompile extension and client classes when base classes are changed [14]. While being apparently important, that problem is only a technical issue. Even if recompilation is not necessary, system developers can make inconsistent modifications. Another interpretation is the necessity to guarantee that objects generated by an extension class can be safely substituted for objects of the corresponding base class [29]. Only in this case, the extension class can be safely substituted for the base class in all clients. However, objects of the extension class can be perfectly substitutable for objects of the base class even if the latter is fragile.

At first glance the problem might appear to be caused by inadequate system specification or user assumptions of undocumented features, but our study reveals that it is more involved. We consider an example which illustrates how concealed the problem can be.

We take a more formal look on the nature of the problem. We abstract the essence of the problem into a flexibility property and explain why unrestricted code inheritance violates this property. We present orthogonal examples which suggest five restrictions for disciplining inheritance. We extend refinement calculus [5, 17, 18] with notions of classes, objects, class-based inheritance, and refinement on classes. For formalization of inheritance we adopt a model suggested by Cook and Palsberg in [8]. We formulate and formally prove a flexibility theorem showing that the restrictions we impose on inheritance are sufficient to permit substituting a base class with its revision in presence of extension classes. Next, we present three check lists for system designers, developers, and extenders to allow application of our results in practice informally. Finally, we discuss related work and draw some conclusions.

2 Fragile Base Class Problem

We assume a fairly standard model of object-oriented programming employing objects, classes, and single inheritance. A class is a template that defines a set of instance variables and methods. Instantiating a class creates a new object with instance variables initialized with initial values and method bodies defined by the class. A subclass inherits instance variables and method definitions from a superclass. All methods are dynamically bound. Every method has an implicit parameter *self*, which must be referred explicitly when a method of the class calls another method of the same class. Methods of a subclass can refer to methods of a superclass through another implicit parameter *super*. As the problem does not depend on the implementation language, we use a simple language independent notation in our examples.

2.1 Example of the Fragile Base Class Problem

We begin with an example which gives an intuitive understanding of the problem and shows how disguised the fragile base class problem can be.[1] Suppose that a class *Bag* is provided by some object-oriented system, for example, an extensible container framework. In an extensible framework user extensions can be called by both the user application and the framework. The class *Bag* has an instance variable b : *bag of char* which is initialized with an empty bag. It also has methods *add* inserting a new element into b, *addAll* invoking the *add* method to add a group of elements to the bag simultaneously, and *cardinality* returning the number of elements in b.

Suppose now that a user of the framework decides to extend it. To do so, the user derives a class *CountingBag* which introduces an instance variable n and overrides *add* to increment n every time a new element is added to the bag. The user also overrides the *cardinality* method to return the value of n which should be equal to the number of elements in the bag. Note that the user is obliged to verify that *CountingBag* is substitutable for *Bag* to be

[1] This example is adopted from [23]

safely used by the framework.

$$
\begin{array}{ll}
\begin{array}{l}
Bag \ = \ \textbf{class} \\
\quad b : bag\ of\ char \\
\\
\quad init \mathrel{\widehat{=}} b := \llbracket\ \rrbracket \\
\quad add(\textbf{val}\ x : char) \mathrel{\widehat{=}} \\
\qquad b := b \cup \llbracket x \rrbracket \\
\quad addAll(\textbf{val}\ bs : bag\ of\ char) \mathrel{\widehat{=}} \\
\qquad \textbf{begin var}\ y \mid y \in bs \cdot \\
\qquad\quad \textbf{while}\ bs \neq \llbracket\ \rrbracket\ \textbf{do} \\
\qquad\qquad self.add(y); \\
\qquad\qquad bs := bs - \llbracket y \rrbracket \\
\qquad\quad \textbf{od} \\
\qquad \textbf{end} \\
\quad cardinality(\textbf{res}\ r : int) \mathrel{\widehat{=}} \\
\qquad r := |b| \\
\textbf{end}
\end{array}
&
\begin{array}{l}
CountingBag \ = \ \textbf{class} \\
\textbf{inherits}\ Bag \\
\quad n : int \\
\\
\quad init \mathrel{\widehat{=}} n := 0; super.init \\
\quad add(\textbf{val}\ x : char) \mathrel{\widehat{=}} \\
\qquad n := n + 1; super.add(x) \\
\\
\\
\\
\\
\\
\\
\quad cardinality(\textbf{res}\ r : int) \mathrel{\widehat{=}} \\
\qquad r := n \\
\textbf{end}
\end{array}
\end{array}
$$

After some time a system developer decides to improve the efficiency of the class *Bag* and releases a new version of the system. The "improved" Bag' implements *addAll* without invoking *add*:

$$
\begin{array}{l}
Bag' \ = \ \textbf{class} \\
\quad b : bag\ of\ char \\
\quad init \mathrel{\widehat{=}} b := \llbracket\ \rrbracket \\
\quad add(\textbf{val}\ x : char) \mathrel{\widehat{=}} b := b \cup \llbracket x \rrbracket \\
\quad addAll(\textbf{val}\ bs : bag\ of\ char) \mathrel{\widehat{=}} b := b \cup bs \\
\quad cardinality(\textbf{res}\ r : int) \mathrel{\widehat{=}} r := |b| \\
\textbf{end}
\end{array}
$$

Naturally, the system developer claims that the new version of the system is fully compatible with the previous one. It definitely appears to be so if considered in separation of the extensions. However, when trying to use Bag' instead of *Bag* as the base class for *CountingBag*, the framework extender suddenly discovers that the resulting class returns the incorrect number of elements in the bag.

Here we face the fragile base class problem. Any open system applying code inheritance and self-recursion in an ad-hoc manner is vulnerable to this problem.

2.2 Failure of the Ad-Hoc Inheritance Architecture

Let us analyze the reasons for the failure in modifying a system relying on ad-hoc code inheritance. Assume that we have a base class C and an extension class E inheriting from it. We say that E is equivalent to $(M$ **mod** $C)$[2], where M corresponds to the extending part of the definition of E, and the operator **mod** combines M with the inherited part C. We refer to M as a *modifier* [28]. The model of single inheritance employing the notion of modifiers was proved by Cook and Palsberg in [8] to correspond to the form of inheritance used in object-oriented systems.[3] For instance, in our previous example M has the form:

$$
\begin{array}{l}
M \;=\; \textbf{modifier} \\
\qquad n : int \\
\qquad init \mathrel{\widehat{=}} n := 0; super.init \\
\qquad add(x : \; char) \mathrel{\widehat{=}} n := n + 1; super.add(x) \\
\textbf{end}
\end{array}
$$

Therefore, we have that C belongs to the system, while $(M$ **mod** $C)$ represents a user extension of this system. When system developers state that the new version of their system is fully compatible with the previous one, they essentially say that a revision class C' is a *refinement* of C. We say that some class C *is refined by* class C', if the externally observable behavior of objects generated by C' is the externally observable behavior of objects generated by C or an improvement of it. In other words, objects generated by C' must be substitutable for objects generated by C in any possible context.[4] Ensuring substitutability of the user extension $(M$ **mod** $C)$ for the framework class C amounts to verifying that C *is refined by* $(M$ **mod** $C)$.

Under these two conditions all participating parties, i.e. the system developers and its extenders, rely on the following property:

if C *is refined by* C' and C *is refined by* $(M$ **mod** $C)$
then C *is refined by* $(M$ **mod** $C')$

Further on we refer to it as the *flexibility property.* Unfortunately, this flexibility property does not hold in general, as demonstrated by our example. This consideration brings us to the question, what are the shortcommings of inheritance and what are the restrictions we need to make to discipline it.

3 Disciplined Inheritance

In this paper we confine the fragile base class problem in a number of ways. First, we consider a class to be a closed entity. This means that method

[2]We read **mod** as modifies.

[3]In their paper modifiers are referred to as wrappers. We prefer the term modifier, because the term wrapper is usually used in the context of object aggregation.

[4]Readers familiar with the notion of behavioral subtyping [16, 1] can think of class refinement as behavioral subtyping. Further on in the paper we give a formal definition of class refinement.

parameters and instance variables that are objects of some other classes are textually substituted with definitions of the corresponding classes. Therefore, without loss of generality we consider method parameters and instance variables to be of simple types. Secondly, for the time being we consider the case when a base class revision and an extension class have as many methods as the corresponding base class. We also assume that a base class does not have mutually recursive methods. As we have stated before, our language provides only for single inheritance.

When an extension class invokes its base class method, we say that an *up-call* has occurred; when a base class invokes a method from a class derived from it, we refer to such an invocation as a *down-call*. A call of a method from another method in the same class is referred to as a *self-call*. We refer to an invocation of a base class method by an extension class as a *super-call*.

Now let us consider five examples invalidating the flexibility property and illuminating the shortcomings of code inheritance.[5] Note that if we regard the definition of a base class as the specification of its functionality, we cannot blame modifier developers for relying on undocumented features and, therefore, inducing the problem. The examples are orthogonal to each other, meaning that all of them illustrate different aspects of the problem. Therefore, they suggest requirements for disciplining inheritance.

Suppose that C describes a class with a state x initially equal to 0 and two methods m and n both incrementing x by 1. A modifier M overrides n so that it calls m.

$$\begin{array}{ll}
C \;=\; \textbf{class} & \qquad M \;=\; \textbf{modifier} \\
\quad x := 0 & \\
\quad m \mathrel{\widehat{=}} x := x + 1 & \\
\quad n \mathrel{\widehat{=}} x := x + 1 & \qquad\quad n \mathrel{\widehat{=}} \mathit{self}.m \\
\textbf{end} & \qquad \textbf{end}
\end{array}$$

Now, if a revision C' reimplements m by calling the method n, which has an implementation exactly as it was before, we run into a problem.

$$\begin{array}{l}
C' \;=\; \textbf{class} \\
\quad x := 0 \\
\quad m \mathrel{\widehat{=}} \mathit{self}.n \\
\quad n \mathrel{\widehat{=}} x := x + 1 \\
\textbf{end}
\end{array}$$

When the modifier M is applied to C', the methods m and n of the resulting class (M **mod** C') become mutually recursive. Apparently, a call to either one would leads to a never terminating loop. This example suggests

[5] As we tried to keep these examples as concise as possible, they might appear slightly artificial.

a formulation of the first requirement.

"No cycles" requirement: *A base class revision and a modifier should not jointly introduce new cyclic method dependencies.*

To illustrate a next shortcoming of code inheritance, it is sufficient to provide only a specification of a base class. The base class C calculates the square and fourth roots of a given real number. Its specification is given in terms of **pre** and **post** conditions which state that, given a non-negative real number x, the method will find such r that its power of two and four respectively equals x.

A modifier M overrides the $Sqrt$ method so that it would return a negative value.[6] Such an implementation of $Sqrt$ is a refinement of the original one, because it reduces non-determinism.

$$
\begin{array}{ll}
C \;=\; \textbf{class} & M \;=\; \textbf{modifier} \\
\quad m(\textbf{val}\ x : \mathit{Real}, \textbf{res}\ r : \mathit{Real}) \mathrel{\widehat{=}} & \quad m(\textbf{val}\ x : \mathit{Real}, \textbf{res}\ r : \mathit{Real}) \mathrel{\widehat{=}} \\
\qquad \textbf{pre}\ x \geq 0 & \qquad r := -\sqrt{x} \\
\qquad \textbf{post}\ r'^2 = x & \\
\quad n(\textbf{val}\ x : \mathit{Real}, \textbf{res}\ r : \mathit{Real}) \mathrel{\widehat{=}} & \\
\qquad \textbf{pre}\ x \geq 0 & \\
\qquad \textbf{post}\ r'^4 = x & \\
\textbf{end} & \textbf{end}
\end{array}
$$

A revision C' of the base class implements the specification of the square root by returning a positive square root of x. The implementation of the fourth root relies on this fact and merely calls m from itself twice, without checking that the result of the first application is positive. Note that C' is a refinement of C.

$$
\begin{array}{l}
C' \;=\; \textbf{class} \\
\quad m(\textbf{val}\ x : \mathit{Real}, \textbf{res}\ r : \mathit{Real}) \mathrel{\widehat{=}} \\
\qquad r := \sqrt{x} \\
\\
\quad n(\textbf{val}\ x : \mathit{Real}, \textbf{res}\ r : \mathit{Real}) \mathrel{\widehat{=}} \\
\qquad \mathit{self}.m(x, r); \mathit{self}.m(r, r) \\
\textbf{end}
\end{array}
$$

Imagine now that we have an instance of a class (M **mod** C'). The call to n will lead to a failure, because the second application of the square root will get a negative value as a parameter. This counter example leads us to the second requirement.

[6] By convention $\sqrt{x}$ returns a positive square root of x.

"No revision self-calling assumptions" requirement: *Revision class methods should not make any additional assumptions about the behavior of the other methods of itself. Only the behavior described in the base class may be taken into consideration.*

In fact, a somewhat analogous requirement must be imposed on an extension class. This can be illustrated by the following example.

$$
\begin{array}{ll}
C \;=\; \textbf{class} & M \;=\; \textbf{modifier} \\
\quad x := 0 & \\
\quad l(\textbf{val}\ v : \mathit{int}) \;\widehat{=}\; \{v \geq 0\}; x := v & \quad l(\textbf{val}\ v : \mathit{int}) \;\widehat{=}\; [v \geq 0]; \mathit{super}.l(v) \\
\quad m(\textbf{val}\ v : \mathit{int}) \;\widehat{=}\; \mathit{self}.l(v) & \\
\quad n(\textbf{val}\ v : \mathit{int}) \;\widehat{=}\; x := v; \mathit{self}.m(v) & \quad n(\textbf{val}\ v : \mathit{int}) \;\widehat{=}\; \mathit{self}.m(v) \\
\textbf{end} & \textbf{end}
\end{array}
$$

$$
\begin{array}{l}
C' \;=\; \textbf{class} \\
\quad x := 0 \\
\quad l(\textbf{val}\ v : \mathit{int}) \;\widehat{=}\; \{v \geq 0\}; x := v \\
\quad m \;\widehat{=}\; \{v \geq 0\}; \mathit{self}.l(v) \\
\quad n \;\widehat{=}\; x := v; \mathit{self}.m(v) \\
\textbf{end}
\end{array}
$$

Here $[v \geq 0]$ and $\{v \geq 0\}$ are the so-called *guard* and *assertion* statements. If the predicate $v \geq 0$ in a state is *true*, both of these statements skip, in the opposite case guard is equivalent to a *magic* statement, which refines any other statement, while assertion is equivalent to *abort*.[7] This example suggests our third requirement.

"No base class down-calling assumptions" requirement: *Methods of a modifier should disregard the fact that base class self-calls can get redirected to the modifier itself. In this case bodies of the corresponding methods in the base class should be considered instead, as if there were no dynamic binding.*

Developers of a revision C' may want to improve the efficiency of C by modifying its data representation. The following example demonstrates that, in general, C' cannot change the data representation of C in presence of inheritance.

A base class C represents its state by an instance variable x. It declares two methods m and n increasing x by 1 and 2 respectively. A modifier M provides a harmless (as it appears by looking at C) override of the method n, which does exactly what the corresponding method in C do, i.e. increases

[7] The precise semantics of these statements is given in Sec. 4.1.

n by two.

$$
\begin{array}{ll}
C \;=\; \textbf{class} & \qquad M \;=\; \textbf{modifier} \\
\quad x := 0 & \\
\quad m \mathrel{\widehat{=}} x := x + 1 & \\
\quad n \mathrel{\widehat{=}} x := x + 2 & \qquad\quad n \mathrel{\widehat{=}} x := x + 2 \\
\textbf{end} & \qquad \textbf{end}
\end{array}
$$

A revision C' introduces an extra instance variable y, initialized to 0. Methods m and n increase x and y by 1 and by 2, but indirectly via y. Therefore, the methods of C' implicitly maintain an invariant, $x = y$.

$$
\begin{array}{l}
C' \;=\; \textbf{class} \\
\quad x := 0; y := 0 \\
\quad m \mathrel{\widehat{=}} y := y + 1; x := y \\
\quad n \mathrel{\widehat{=}} y := y + 2; x := y \\
\textbf{end}
\end{array}
$$

Now if we consider an object *obj* to be an instance of class (M **mod** C'), obtained by substituting C' for C, and the sequence of methods calls *obj.n;obj.m*, we face the problem. By looking at C we could assume that the sequence of method calls makes x equal to 3, whereas, in fact, x is assigned only 1.

An analogous problem was described by Alan Snyder in [21]. He notices that "Because the instance variables are accessible to clients of the class, they are (implicitly) part of the contract between the designer of the class and the designers of descendant classes. Thus, the freedom of the designer to change the implementation of a class is reduced". In our example, since M is allowed to modify the instance variables inherited from C directly, it becomes impossible to change the data representation in C'. Therefore, we can formulate our fourth requirement.

"No direct access to the base class state" requirement: *An extension class may not access the state of its base class directly, but only through calling base class methods.*

When creating a modifier its developer usually intends it for a particular base class. A common practice is an introduction of new variables in the modifier and binding their values with the values of the intended base class instance variables. Such a binding can be achieved even without explicitly referring to the base class variables. We can say that such a modifier carries an *invariant function* which is applied during the modifier application and returns an invariant of the resulting class. The following example demonstrates that, in general, such an invariant function can lead to problems.

A class C has an instance variable x. A modifier M introduces a new instance variable y and binds its value to the value of instance variable x of the base class the modifier is supposed to be applied to. An override of the method n verifies this fact by first making a super-call to the method l and asserting that the returned value is equal to y.

$$
\begin{array}{ll}
C \;=\; \textbf{class} & M \;=\; \textbf{modifier} \\
\quad x := 0 & \quad y := 0 \\
\quad l(\textbf{res}\ r : int) \mathrel{\widehat{=}} r := x & \\
\quad m \mathrel{\widehat{=}} x := x + 1; \mathit{self}.n & \quad m \mathrel{\widehat{=}} y := y + 1; \mathit{super}.m \\
\quad n \mathrel{\widehat{=}} \begin{array}{l} \textbf{begin var}\ r \cdot \\ \quad \mathit{self}.l(r) \\ \textbf{end} \end{array} & \quad n \mathrel{\widehat{=}} \begin{array}{l} \textbf{begin var}\ r \cdot \\ \quad \mathit{super}.l(r); \{r = y\} \\ \textbf{end} \end{array} \\
\textbf{end} & \textbf{end}
\end{array}
$$

It is easy to see that before and after execution of any method of (M **mod** C) predicate $x = y$ holds. We can say that the class maintains the invariant $x = y$. The full definition of the method m in an instance of the class (M **mod** C) effectively has the form $y := y + 1; x := x + 1; \{x = y\}$, where the assertion statement skips since the preceding statements establish the invariant.

Now if a revision C' reimplements m by first self-calling n and then incrementing x as illustrated below, we run into a problem.

$$
\begin{array}{l}
C' \;=\; \textbf{class} \\
\quad x := 0 \\
\quad l(\textbf{res}\ r : int) \mathrel{\widehat{=}} r := x \\
\quad m \mathrel{\widehat{=}} \mathit{self}.n; x := x + 1 \\
\quad n \mathrel{\widehat{=}} \begin{array}{l} \textbf{begin var}\ r \cdot \\ \quad \mathit{self}.l(r) \\ \textbf{end} \end{array} \\
\textbf{end}
\end{array}
$$

The body of the method m in the instance class (M **mod** C') is effectively of the form $y := y + 1; \{x = y\}; x := x + 1$ and naturally it aborts. This leads us to formulation of our fifth and the last requirement.

"No modifier invariant function" requirement: *A modifier should not bind values of its instance variables with values of the intended base class instance variables to generate an invariant.*

Let us now summarize all restrictions disciplining the code inheritance mechanism:

- **no cycles**

- **no revision self-calling assumptions**
- **no base class down-calling assumptions**
- **no direct access to base class state**
- **no modifier invariant function.**

The presented examples demostrate that the flexibility property does not hold in general. We claim that if disciplining code inheritance according to the five requirements presented above, we can formulate and prove a flexibility theorem which permits substituting a base class with its revision in presence of extension classes. In the next section we consider a formal basis necessary for formalizing this theorem.

4 Formal Basis

4.1 Refinement Calculus Basics

This section is based on the work by Back and von Wright as presented in [5]. The behavior of a program statement can be characterized by Dijkstra's weakest precondition predicate transformer [9]. For a statement S and a predicate p, the weakest precondition $wp(S,p)$ is such that the statement S terminates in a state satisfying the postcondition p. Since the relation between pre- and postconditions is all we are interested in for a statement, we can identify the statement with a function mapping postconditions to preconditions.

The predicates over a state space (type) Σ are the functions from Σ to *Bool*, denoted $\mathcal{P}\Sigma$. The relations from Σ to Γ are the functions from Σ to a predicate (set of values) over Γ, denoted by $\Sigma \leftrightarrow \Gamma$. The predicate transformers from Σ to Γ are the functions mapping predicates over Γ to predicates over Σ, denoted $\Sigma \mapsto \Gamma$ (note the reversion of the direction):

$$\begin{aligned}
\mathcal{P}\Sigma &\mathrel{\widehat{=}} \Sigma \to Bool \\
\Sigma \leftrightarrow \Gamma &\mathrel{\widehat{=}} \Sigma \to \mathcal{P}\Gamma \\
\Sigma \mapsto \Gamma &\mathrel{\widehat{=}} \mathcal{P}\Gamma \to \mathcal{P}\Sigma
\end{aligned}$$

The entailment ordering $p \le q$ on predicates $p, q : \mathcal{P}\Sigma$ is defined by universal implication:

$$p \le q \mathrel{\widehat{=}} (\forall \sigma : \Sigma.\ p\ \sigma \ \Rightarrow\ q\ \sigma)$$

The predicates *true* and *false* over Σ map every $\sigma : \Sigma$ to the boolean values *true* and *false*, respectively. Conjunction $p \wedge q$, disjunction $p \vee q$, and negation $\neg p$ are all defined by pointwise extension of the corresponding operations on *Bool*.

The *composition* of relations is defined for relations $P : \Sigma \leftrightarrow \Gamma$ and $Q : \Gamma \leftrightarrow \Delta$ by

$$(P;Q)\ \sigma\ \delta \mathrel{\widehat{=}} \exists \gamma \cdot (P\ \sigma\ \gamma)\ \wedge\ (Q\ \gamma\ \delta),$$

where $\sigma : \Sigma$, $\delta : \Delta$, and $\gamma : \Gamma$.

The cartesian product of state spaces Σ_1 and Σ_2 is written $\Sigma_1 \times \Sigma_2$. For relations $P_1 : \Sigma_1 \leftrightarrow \Gamma_1$ and $P_2 : \Sigma_2 \leftrightarrow \Gamma_2$ their *product*, $P_1 \times P_2$, is a relation of type $(\Sigma_1 \times \Sigma_2) \leftrightarrow (\Gamma_1 \times \Gamma_2)$, where for $\sigma_1 : \Sigma_1$, $\sigma_2 : \Sigma_2$, $\gamma_1 : \Gamma_1$ and $\gamma_2 : \Gamma_2$, we have:

$$(P_1 \times P_2)\ (\sigma_1, \sigma_2)\ (\gamma_1, \gamma_2) \mathrel{\widehat{=}} (P_1\ \sigma_1\ \gamma_1)\ \wedge\ (P_2\ \sigma_2\ \gamma_2)$$

The relational product operator is right associative and binds stronger than the relational composition operator.

The *identity relation*, *Id* $: \Sigma \leftrightarrow \Sigma$, is defined for $\sigma_1, \sigma_2 : \Sigma$ as follows:

$$\mathit{Id}\ \sigma_1\ \sigma_2 \mathrel{\widehat{=}} \sigma_1 = \sigma_2$$

A predicate transformer $S : \Sigma \mapsto \Gamma$ is said to be *monotonic* if for all predicates p and q, $p \leq q$ implies $S\ p \leq S\ q$. Statements from Σ to Γ are identified with monotonic predicate transformers from Σ to Γ. Statements of this kind may be concrete, i.e. executable, or abstract, i.e. specifications, and may have different initial and final state spaces.

The sequential composition of statements $S : \Sigma \mapsto \Gamma$ and $T : \Gamma \mapsto \Delta$ is modeled by their functional composition. Let q be a predicate over Δ, then

$$(S;T)\ q \mathrel{\widehat{=}} S\ (T\ q).$$

Meet of (similarly-typed) predicate transformers is defined pointwise:

$$(S \sqcap T)\ q \mathrel{\widehat{=}} (S\ q\ \wedge\ T\ q)$$

Meet of statements models *nondeterministic choice* between executing statements S and T. It is required that both alternatives establish the postcondition.

The **abort** statement does not guarantee any outcome or termination, therefore, it maps every postcondition to *false*. The **magic** statement is *miraculous*, since it is always guaranteed to establish any postcondition. The **skip** statement leaves the state unchanged. We define for any state space Σ and $q : \mathcal{P}\Sigma$

$$\begin{array}{l} \textbf{abort}\ q \mathrel{\widehat{=}} \mathit{false} \\ \textbf{magic}\ q \mathrel{\widehat{=}} \mathit{true} \\ \textbf{skip}\ q \mathrel{\widehat{=}} q. \end{array}$$

For a predicate $p : \mathcal{P}\Gamma$, the assertion $\{p\}$ is **abort** if p does not hold, and **skip** otherwise. The *guard* statement $[p]$ behaves as **skip** if p holds, otherwise it behaves as **magic**. Let q be a predicate over Γ, then:

$$\begin{array}{l} \{p\}\ q \mathrel{\widehat{=}} p\ \wedge\ q \\ [p]\ q \mathrel{\widehat{=}} p\ \Rightarrow\ q \end{array}$$

Given a relation $P : \Sigma \leftrightarrow \Gamma$, the *angelic update* statement $\{P\} : \Sigma \mapsto \Gamma$ and the *demonic update* statement $[P] : \Sigma \mapsto \Gamma$ are defined by:

$$\{P\}\ q\ \sigma \mathrel{\widehat=} (\exists \gamma : \Gamma \cdot (P\ \sigma\ \gamma)\ \wedge\ (q\ \gamma))$$
$$[P]\ q\ \sigma \mathrel{\widehat=} (\forall \gamma : \Gamma \cdot (P\ \sigma\ \gamma)\ \Rightarrow\ (q\ \gamma))$$

When started in a state σ, both $\{P\}$ and $[P]$ choose a new state γ such that $P\ \sigma\ \gamma$ holds. If no such state exists, then $\{P\}$ aborts, whereas $[P]$ can establish any postcondition.

We say that a statement S is conjunctive if

$$S\ (\wedge i \in I \cdot p_i)\ =\ (\wedge i \in I \cdot S\ p_i),$$

where $I \neq \emptyset$. An arbitrary conjunctive statement can be represented by a sequential composition $\{q\};[Q]$ for some predicate q and relation Q.

Assignment can be modeled as an update statement. If the state space Σ is partitioned by the variables $x : T$ and $y : U$, i.e. $\Sigma = T \times U$, then

$$x := e \mathrel{\widehat=} [R], \text{ where } R\ (x, y)\ (x', y') = (x' = e\ \wedge\ y' = y).$$

A specification statement with a precondition over initial state space and a postcondition relating initial and final state spaces is defined by a composition of assertion and demonic update:

$$\textbf{pre}\ p\ (x, y)\ \textbf{post}\ R\ (x, y)\ (x', y') \mathrel{\widehat=} \{p\};[R]$$

For statements S and T of the same type and a postcondition q, the conditional is defined by:

$$(\textbf{if}\ p(x, y)\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{fi}) \mathrel{\widehat=} ([p]; S) \sqcap ([\neg p]; S)$$

The refinement ordering $S \sqsubseteq T$, read S *is refined by* T, on statements $S, T : \Sigma \mapsto \Gamma$ is defined as a universal entailment:

$$S \sqsubseteq T \mathrel{\widehat=} (\forall q : \mathcal{P}\Gamma \cdot S\ q \leq T\ q)$$

For example, a specification statement is refined if its precondition is weakened and the postcondition is strengthened:

$$(\textbf{pre}\ p(x, y)\ \textbf{post}\ R\ (x, y)\ (x', y')\ ) \sqsubseteq (\textbf{pre}\ p'(x, y)\ \textbf{post}\ R'\ (x, y)\ (x', y'))$$
$$\text{if } p(x, y)\ \Rightarrow\ p'(x, y) \text{ and } R'\ (x, y)\ (x', y')\ \Rightarrow\ R\ (x, y)\ (x', y')$$

Other rules for refinement of specifications into programs and transforming programs are given in [3, 17, 10].

An iteration statement is defined by the least fixed point of a function F with respect to the refinement ordering:

$$\textbf{while}\ p(x)\ \textbf{do}\ S\ \textbf{od} \mathrel{\widehat=} \mu\ F,$$
$$\text{where } F\ X = \textbf{if}\ p(x)\ \textbf{then}\ S; X\ \textbf{else skip fi}$$

According to the theorem of Knaster-Tarski, a monotonic function has a unique least fixed point in a complete lattice. Statements form a complete lattice with the refinement ordering $\sqsubseteq$, and the statement **if** $p(x)$ **then** $S; X$ **else skip fi** is indeed monotonic with respect to refinement ordering $\sqsubseteq$ in X, therefore, $\mu\ F$ exists and is unique [27]. Intuitively, defining iteration in this way implies that a non-terminating loop behaves as **abort**.

A *block* construct allows temporary adding a new *local* state component to the present *global* state. An entry statement **enter** adds a new state component to the present state and initializes it in accordance with a given predicate p. An exit statement **exit** removes the added state component.

$$
\begin{array}{l}
\mathbf{enter}\, z \,|\, p(x,y,z) \;\widehat{=} \\
\qquad [R^{-1}], \text{where } R^{-1}\ (x,y)\ (z,x',y') = (x = x' \ \wedge\ y = y' \ \wedge\ p\ (x,y,z')) \\
\mathbf{exit}\, z \;\widehat{=} \\
\qquad \{R\}, \text{where } R\ (z,x,y)\ (x',y') = (x = x' \ \wedge\ y = y' \ \wedge\ p\ (x,y,z'))
\end{array}
$$

Accordingly, the block construct is defined as follows:

$$\mathbf{begin\ var}\, z \,|\, p(x,y,z) \cdot S\, \mathbf{end} \quad \widehat{=} \quad \mathbf{enter}\, z \,|\, p(x,y,z); S; \mathbf{exit}\, z$$

In general, different state spaces can be coerced using *wrapping* and *unwrapping* operators. Statements S and S' operating on state spaces Σ and Σ' respectively can be combined using a relation $R : \Sigma' \leftrightarrow \Sigma$ which when lifted to predicate transformers gives the update statements $\{R\} : \Sigma' \mapsto \Sigma$ and $[R^{-1}] : \Sigma \mapsto \Sigma'$. We use these statements to define *wrapping* and *unwrapping* operators as follows:

$$
\begin{array}{ll}
\mathit{Wrapping}: & S\!\uparrow\! R \;\widehat{=}\; [R^{-1}]; S; \{R\} \\
\mathit{Unwrapping}: & S'\!\downarrow\! R \;\widehat{=}\; \{R\}; S'; [R^{-1}]
\end{array}
$$

Sometimes it is necessary to wrap a statement operating on an extended state into a relation R. In this case we consider the relation R to be automatically extended into $\mathit{Id} \times R$ or $R \times \mathit{Id}$. For example, to wrap a statement $S : \Sigma \times \Gamma \mapsto \Sigma \times \Gamma$, the relation R is extended to $R \times \mathit{Id}$.

The wrapping and unwrapping operators are left associative and have the lowest precedence of all relational operators. We make use of the following rule:

$$\mathit{Wrapping\ rule}: \quad S\!\uparrow\! R\!\downarrow\! R \sqsubseteq S$$

For tuples of statements we have:

$$
\begin{array}{lcl}
(S_1, ..., S_n)\!\uparrow\! R & \widehat{=} & (S_1\!\uparrow\! R, ..., S_n\!\uparrow\! R) \\
(S_1, ..., S_n)\!\downarrow\! R & \widehat{=} & (S_1\!\downarrow\! R, ..., S_n\!\downarrow\! R)
\end{array}
$$

A definition of (non-recursive) procedure with value and result parameters is modeled by an appropriate extension and reduction of state space:

proc m(**val** $x : U$, **res** $y : V$) $\widehat{=}$ M stands for $m \,\widehat{=}\,$ **enter** $y \,|\, \mathit{true}; M;$ **exit** x, where y is initialized to an arbitrary value. A call to m first extends the state by value parameters and then removes result parameters:

$$m(e,\ z) \mathrel{\widehat{=}} \textbf{enter}\, x \mid x = e; m; z := y; \textbf{exit}\, y$$

4.2 Formalization of Classes

A class C is defined as a pair with initial values c_0 of instance variables and a tuple with method bodies $c_1, ..., c_n$. Within the method bodies, other methods are referred via *self*.m_i. Since these calls may be redirected due to inheritance, *self*.$m_1, ...,$ *self*.m_n are treated as parameters of $c_1, ..., c_n$. Abbreviating *self*.$m_1, ...,$ *self*.m_n by *self*, we have

$$C \;=\; (c_0, cm), \text{ where } cm = \lambda \mathit{self} \cdot (c_1, ..., c_n)$$

and assume that cm is monotonic in the *self* parameter. This model of classes as self-referential structures follows closely that of Cook and Palsberg [8]. However, there are only functional methods in their model, and for our purposes this is insufficient. We declare the class C, defined in this way, as follows:

$$C \;=\; \textbf{class}\ c := c_0,\ m_1 \mathrel{\widehat{=}} c_1, ..., m_n \mathrel{\widehat{=}} c_n\ \textbf{end}$$

A class can be depicted as in Fig. 1. An incoming arrow represents an interface of the class C, an outcoming arrow stands for self-calls of C.

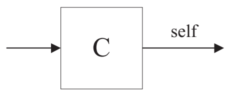

Figure 1: Illustration of a class

The type of instance variables of C is taken to be Σ. However, due to dynamic binding, methods of C operate not only on Σ, but also on the state space of some modifier whose type is unknown until the modifier is applied. Accordingly, we model the methods of C by predicate transformers of type $\alpha \times \Sigma \mapsto \alpha \times \Sigma$, where α is a type variable. It is instantiated with the type of modifier instance variables at modifier application. Hence, cm has the following type:

$$((\alpha{\times}\Sigma \mapsto \alpha{\times}\Sigma), ..., (\alpha{\times}\Sigma \mapsto \alpha{\times}\Sigma)) \to ((\alpha{\times}\Sigma \mapsto \alpha{\times}\Sigma), ..., (\alpha{\times}\Sigma \mapsto \alpha{\times}\Sigma))$$

In fact, the methods of C operate on the state space extended with the type of method parameters. We omit types of method parameters for simplicity.

A definition of method with value and result parameters, e.g., $m_i(\textbf{val}\ x : U,\ \textbf{res}\ y : V) \mathrel{\widehat{=}} c_i$ and a call to such a method, e.g., *self*.$m_i(e, z)$ are treated identically to the definition and the call of procedure.

Calling a method of a class means that any further calls to *self* are redirected to the class methods themselves. Modeling this formally amounts to taking the least fixed point of the methods. We define the operation of taking the least fixed point of a class by taking initial values of its instance variables and taking the least fixed point of the methods:

$$\mu\ C \mathrel{\widehat{=}} (c_0, \mu\ cm)$$

For tuples of methods $(c_1, .., c_n)$ and $(c'_1, ..., c'_n)$, where c_i and c'_i are of the same type, the refinement ordering is defined elementwise:

$$(c_1, ..., c_n) \sqsubseteq (c'_1, ..., c'_n) \mathrel{\widehat{=}} c_1 \sqsubseteq c'_1 \ \wedge\ ...\ \wedge\ c_n \sqsubseteq c'_n$$

Statement tuples form a lattice with the above refinement ordering and cm is monotonic, therefore, $\mu\ C$ exists and is unique. Fig. 2 illustrates taking a fixed point of the class C. If we model objects as tuples of instance variable values and method bodies, then taking the least fixed point of the class C corresponds to creating an object of this class. Thus, we can write:

$$\mathit{create}\ C \mathrel{\widehat{=}} \mu\ C$$

It suffices to consider only those modifiers which redefine all methods of the base class. In case some method should remain unchanged, the corresponding method of the modifier calls the former via super. A modifier

$$L = \textbf{modifier}\ l := l_0,\ m_1 \mathrel{\widehat{=}} l_1, ..., m_n \mathrel{\widehat{=}} l_n\ \textbf{end}$$

is modeled by a pair with initial values of new instance variables and a tuple of method bodies which may refer to *self*.$m_1, ...,$ *self*.m_n and *super*.$m_1, ...,$ *super*.m_n. Abbreviating *super*.$m_1, ...,$ *super*.m_n by *super*, we have

$$L = (l_0,\ lm), \text{where } lm = \lambda \mathit{self} \cdot \lambda \mathit{super} \cdot (l_1, ..., l_n)$$

and assume that lm is monotonic in the *self* and *super* arguments. See Fig. 3 for an illustration of modifiers. As with the class diagram, an incoming arrow

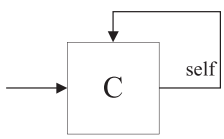

Figure 2: Illustration of taking a fixed point of C

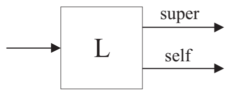

Figure 3: Illustration of modifiers

represents methods of the modifier, whereas outcoming arrows stand for self and super-calls of the modifier.

Similarly to methods of class C, methods of the modifier L operate on the state space $\alpha' \times (\Gamma \times \beta)$, where α' and β are type variables and Γ is a type of modifier instance variables. Hence, the type of lm is as follows:

$$\begin{array}{l} ((\alpha' \times \Gamma \times \beta \mapsto \alpha' \times \Gamma \times \beta), ..., (\alpha' \times \Gamma \times \beta \mapsto \alpha' \times \Gamma \times \beta)) \rightarrow \\ \quad ((\alpha' \times \Gamma \times \beta \mapsto \alpha' \times \Gamma \times \beta), ..., (\alpha' \times \Gamma \times \beta \mapsto \alpha' \times \Gamma \times \beta)) \rightarrow \\ \quad\quad ((\alpha' \times \Gamma \times \beta \mapsto \alpha' \times \Gamma \times \beta), ..., (\alpha' \times \Gamma \times \beta \mapsto \alpha' \times \Gamma \times \beta)) \end{array}$$

The role of the type variables will become clear when we consider creating classes using modifiers.

In a class resulting from combining the modifier L with the base class C methods of L are making upcalls to methods of C. However, C and L operate on different state spaces, and we need to wrap super-calls in a relation that would coerce the states. Consider a relation of the following form:

$$\widehat{\pi_\Sigma} \mathrel{\widehat{=}} \lambda(\gamma, \sigma) : (\Gamma \times \Sigma) \cdot \lambda\sigma' \cdot \sigma = \sigma'$$

In fact, this relation is deterministic and could instead be represented as a function taking a state $(\gamma, \sigma) : (\Gamma \times \Sigma)$ to a state $\sigma : \Sigma$.

Two operators **mod** and **upcalls** can be used for creating a class from the base class C and the modifier L:

$$\begin{array}{lcl} L \ \mathbf{mod}\ C & \widehat{=} & ((l_0, c_0),\ \lambda \mathit{self} \cdot\ lm\ \mathit{self}\ (cm\ \mathit{self}{\uparrow}(Id \times \widehat{\pi_\Sigma})){\downarrow}(Id \times \widehat{\pi_\Sigma})) \\ L \ \mathbf{upcalls}\ C & \widehat{=} & ((l_0, c_0),\ \lambda \mathit{self} \cdot\ lm\ \mathit{self}\ (\mu\ cm){\downarrow}(Id \times \widehat{\pi_\Sigma})) \end{array}$$

In both cases the modifier L is said to be applied to the base class C. See Fig. 4 for an illustration of modifier application. Note that in case of the **mod** operator self-calls of C become down-calls; whereas, with the **upcalls** operator self-calls of C remain within C itself.

Application of the modifier L instantiates its type variable β with the type Σ of the base class C. Hence, the classes created by the application of L to C have methods operating on $\alpha' \times (\Gamma \times \Sigma)$, where α' is a type variable to be instantiated in the next modifier application.

4.3 Refinement on Classes

For defining refinement of classes, we first consider data refinement of statements. Let $S : \Sigma \mapsto \Sigma$ and $S' : \Sigma' \mapsto \Sigma'$ be statements and $R : \Sigma' \leftrightarrow \Sigma$ a

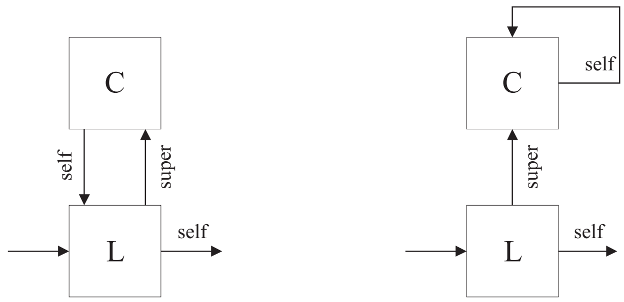

Figure 4: Illustration of operators **mod** and **upcalls**

relation between the state spaces of these statements. We define data refinement between S and S' as in [5]:

$$S \sqsubseteq_R S' \mathrel{\widehat{=}} S \sqsubseteq S' \uparrow R \text{ or, equivalently,}$$
$$S \sqsubseteq_R S' \mathrel{\widehat{=}} S \downarrow R \sqsubseteq S'$$

We can express class refinement in terms of a slight generalization of the refinement on abstract data types [13, 11, 4, 10]. An abstract data type T can be represented in the form

$$T = (t_0, tp),$$

where t_0 is an initial value of an internal state of type Σ, and tp is a tuple of procedures modifying this internal state. The procedures are of type $\alpha \times \Sigma \mapsto \alpha \times \Sigma$, where α is a type variable on which the procedures skip. We say that the abstract data type $T = (t_0, tp)$ is data refined by an abstract data type $T' = (t'_0, tp')$ via a relation R if initialization establishes R and all procedures preserve R:

$$T \ \sqsubseteq_R \ T' \mathrel{\widehat{=}} (R\ t'_0\ t_0) \ \wedge \ tp \ \sqsubseteq_{Id \times R} \ tp',$$

where $Id : \alpha \leftrightarrow \alpha$ is the identity relation, and $R : \Sigma' \leftrightarrow \Sigma$ relates the corresponding internal states.

Now class refinement can be defined as follows. Let C be as above and $C' = (c'_0, cm')$, where $cm'\ \mathit{self} = (c'_1, ..., c'_n)$, and $R : \Sigma' \leftrightarrow \Sigma$ then

$$C \sqsubseteq_R C' \mathrel{\widehat{=}} \mu\ C \ \sqsubseteq_R \ \mu\ C'.$$

Class refinement ensures that all objects of the refining class will behave as objects of the refined class. This notion of class refinement allows the instance variables of C' extend those of C or to be completely different. The refinement relation can be also applied to a pair of abstract or concrete classes.

5 Flexibility Theorem

Recall that the flexibility property had the form

if C *is refined by* (L **mod** C) and C *is refined by* D
then C *is refined by* (L **mod** D),

where C is a base class, L is a modifier and D is a revision of the class C. We have to strengthen this property by the requirements of Sec. 3.

Let us review the state spaces of the participating classes. The class C operates on the state space $\alpha \times \Sigma$, the class D operates on $\alpha \times \Sigma'$, the class (L **mod** C) operates on $\alpha' \times (\Gamma \times \Sigma)$, and, finally, ($L$ **mod** D) operates on $\alpha' \times (\Gamma \times \Sigma')$.

Consider the abstraction relations coercing the corresponding state spaces of participating classes. Let $\widehat{R} : \Sigma' \leftrightarrow \Sigma$ be a relation coercing the state space component Σ' of the revision D to the component Σ of the base class C. Let $\widehat{\pi_\Sigma} : \Gamma \times \Sigma \leftrightarrow \Sigma$ and $\widehat{\pi_{\Sigma'}} : \Gamma \times \Sigma' \leftrightarrow \Sigma'$ be the projection relations coercing the corresponding state spaces. Let us also adopt the following abbreviations:

$$
\begin{aligned}
&R \mathrel{\widehat{=}} Id \times \widehat{R}\\
&\pi_\Sigma \mathrel{\widehat{=}} Id \times \widehat{\pi_\Sigma}\\
&\pi_{\Sigma'} \mathrel{\widehat{=}} Id \times \widehat{\pi_{\Sigma'}}
\end{aligned}
$$

Now we can formulate the *flexibility theorem* based on the definitions given in the previous sections:

Flexibility Theorem *Let C and D be classes and L a modifier, such that $C = (c_0, cm)$ with $cm = \lambda self \cdot (c_1, ..., c_n)$, $D = (d_0, dm)$ with $dm = \lambda self \cdot (d_1, ..., d_n)$, and $L = (l_0, lm)$ with $lm = \lambda self \cdot \lambda super \cdot (l_1, ..., l_n)$. Let R, π_Σ and $\pi_{\Sigma'}$ be the relations as described above and Id be of type $\alpha \leftrightarrow \alpha$.*
If

1. *"no cycles" requirement holds*
2. *"no revision self-calling assumptions" requirement holds, i.e.*
   $\mu\ C\ \sqsubseteq_R\ (d_0,\ dm\ ((\mu\ cm){\downarrow}R))$
3. *"no base class down-calling assumptions" and*
4. *"no direct access to base class state" and*
5. *"no modifier invariant function" requirements hold, i.e*
   $C\ \sqsubseteq_{\pi_\Sigma}\ (L$ **upcalls** $C)$

then

$C\ \sqsubseteq_{(Id \times R);\pi_\Sigma}\ (L$ **mod** $D)$

Proof In accordance with the definition of class refinement, we first need to show that under the assumption $(\widehat{R}\ d_0\ c_0)$ it is true that $R;\widehat{\pi_\Sigma}\ (l_0, d_0)\ c_0$.

We prove this goal as follows:

$$
\begin{array}{l}
\quad R;\widehat{\pi_\Sigma}\ (l_0, d_0)\ c_0 \\
= \{\textit{definitions of } R \textit{ and relational composition}\} \\
\quad \exists l' : \Gamma, c' : \Sigma \cdot (Id \times \widehat{R})\ (l_0, d_0)\ (l', c')\ \wedge\ \widehat{\pi_\Sigma}\ (l', c')\ c_0 \\
= \{\textit{definition of } \times \textit{ on relations then definition of Id and } \widehat{\pi_\Sigma} \textit{relations}\} \\
\quad \exists l' : \Gamma, c' : \Sigma \cdot l_0 = l'\ \wedge\ \widehat{R}\ d_0\ c'\ \wedge\ c' = c_0 \\
= \{\textit{one point rule}\} \\
\quad \widehat{R}\ d_0\ c_0 \\
\Leftarrow \{\textit{assumption}\} \\
\quad \textit{true}
\end{array}
$$

Now we need to show that all methods of μ (L **mod** D) refine the corresponding methods of μC with respect to the relation $(Id \times R); \pi_\Sigma$. To express the first assumption, we represent methods by functions of the called methods. We rearrange the methods in linear order so that each method calls only previous ones. We assign an index to every method with respect to this order. Without loss of generality, we can consider the case when for every distinct index there is only one method.

$$
\begin{array}{lll}
C_1 = \lambda() \cdot c_1, & \ldots & C_n = \lambda(\textit{self.m}_1, \ldots, \textit{self.m}_{n-1}) \cdot c_n \\
D_1 = \lambda() \cdot d_1, & \ldots & D_n = \lambda(\textit{self.m}_1, \ldots, \textit{self.m}_{n-1}) \cdot d_n \\
L_1 = \lambda() \cdot & \ldots & L_n = \lambda(\textit{self.m}_1, \ldots, \textit{self.m}_{n-1}) \cdot \\
\qquad \lambda(\textit{super.m}_1) \cdot l_1, & & \qquad \lambda(\textit{super.m}_1, \ldots, \textit{super.m}_n) \cdot l_n
\end{array}
$$

There are no free occurrences of *self* and *super* in C_i, D_i and L_i. For example, for the class C we have that:

$$
cm = \lambda \textit{self} \cdot (C_1\ (), C_2\ (\textit{self.m}_1), \ldots, C_n\ (\textit{self.m}_1, \ldots, self.m_{n-1}))
$$

The theorem can then be rewritten as

$$
\begin{array}{l}
(\mathcal{C}_1, \ldots, \mathcal{C}_n) \sqsubseteq_R (\mathcal{D}_1, \ldots, \mathcal{D}_n)\ \wedge\ (\mathcal{C}_1, \ldots, \mathcal{C}_n) \sqsubseteq_{\pi_\Sigma} (\mathcal{L}_1, \ldots, \mathcal{L}_n)\ \Rightarrow \\
\qquad (\mathcal{C}_1, \ldots, \mathcal{C}_n) \sqsubseteq_{(Id \times R);\pi_\Sigma} (\mathcal{M}_1, \ldots, \mathcal{M}_n),
\end{array}
$$

where $\mathcal{C}_1, \ldots, \mathcal{C}_n$ are the methods $C_1, \ldots, C_n$ with all calls to *self* substituted with the appropriate method bodies, and similarly for $\mathcal{D}, \mathcal{L}, \mathcal{M}$ and $\mathcal{T}$ i.e.:

$$
\begin{array}{lll}
\mathcal{C}_1 = C_1\ (), & \ldots & \mathcal{C}_n = C_n\ (\mathcal{C}_1, \ldots, \mathcal{C}_{n-1}) \\
\mathcal{D}_1 = D_1\ (), & \ldots & \mathcal{D}_n = D_n\ (\mathcal{C}_1, \ldots, \mathcal{C}_{n-1}) \downarrow R \\
\mathcal{L}_1 = L_1\ ()\ \mathcal{C}_1 \downarrow \pi_\Sigma, & \ldots & \mathcal{L}_n = L_n\ (\mathcal{L}_1, \ldots, \mathcal{L}_{n-1})\ (\mathcal{C}_1, \ldots, \mathcal{C}_n) \downarrow \pi_\Sigma \\
\mathcal{M}_1 = L_1\ ()\ \mathcal{T}_1 \downarrow \pi_{\Sigma'}, & \ldots & \mathcal{M}_n = L_n\ (\mathcal{M}_1, \ldots, \mathcal{M}_{n-1})\ (\mathcal{T}_1, \ldots, \mathcal{T}_n) \downarrow \pi_{\Sigma'} \\
\mathcal{T}_1 = D_1\ (), & \ldots & \mathcal{T}_n = D_n\ (\mathcal{M}_1, \ldots, \mathcal{M}_{n-1}) \uparrow \pi_{\Sigma'}
\end{array}
$$

For example, $\mathcal{M}$ and $\mathcal{T}$ jointly represent the least fixed point of the methods of (L **mod** D). Note that $\mathcal{M}$ stands for the methods of the modifier L with calls to *self* substituted with $\mathcal{M}$ itself and calls to *super* substituted with $\mathcal{T}$. Analogously, $\mathcal{T}$ represents methods of the revision D with calls to *self* substituted with $\mathcal{M}$.

Rather than proving the goal above, we prove a stronger goal

$$\begin{array}{l}(\mathcal{C}_1,\ldots,\mathcal{C}_n) \sqsubseteq_R (\mathcal{D}_1,\ldots,\mathcal{D}_n) \ \wedge \ (\mathcal{C}_1,\ldots,\mathcal{C}_n) \sqsubseteq_{\pi_\Sigma} (\mathcal{L}_1,\ldots,\mathcal{L}_n) \ \Rightarrow \\ (\mathcal{L}_1,\ldots,\mathcal{L}_n) \sqsubseteq_{Id\times R} (\mathcal{M}_1,\ldots,\mathcal{M}_n) \ \wedge \ (\mathcal{D}_1,\ldots,\mathcal{D}_n) \sqsubseteq (\mathcal{T}_1,\ldots,\mathcal{T}_n)\end{array}$$

from which the target goal follows by transitivity, since:

$$A \sqsubseteq_P B \ \wedge \ B \sqsubseteq_Q C \ \Rightarrow \ A \sqsubseteq_{Q;P} C$$

For proving this stronger goal we need the following

$$\begin{array}{l}\textit{wrapping theorem :} \\ \qquad L_i\ (\mathcal{M}_1,\ldots,\mathcal{M}_{i-1})\!\uparrow\!(Id\times R)\ (\mathcal{D}_1,\ldots,\mathcal{D}_i)\!\uparrow\!(Id\times R) \sqsubseteq \\ \qquad\qquad (L_i\ (\mathcal{M}_1,\ldots,\mathcal{M}_{i-1})\ (\mathcal{D}_1,\ldots,\mathcal{D}_i))\!\uparrow\!(Id\times R)\end{array}$$

proved in Appendix A. Also we need two lemmas proved in Appendix B:

$$\begin{array}{ll}\textit{lemma 1 :} & S\!\downarrow\!\pi_\Sigma\!\downarrow\!(Id\times R) = S\!\downarrow\!R\!\downarrow\!\pi_{\Sigma'} \\ \textit{lemma 2 :} & S\!\uparrow\!(Id\times R)\!\uparrow\!\pi_\Sigma = S\!\uparrow\!\pi_{\Sigma'}\!\uparrow\!R\end{array}$$

We prove the theorem by induction on the index of methods.
Base case: Under the assumptions $\mathcal{C}_1 \sqsubseteq_R \mathcal{D}_1 \ \wedge \ \mathcal{C}_1 \sqsubseteq_{\pi_\Sigma} \mathcal{L}_1$ the goal $\mathcal{L}_1 \sqsubseteq_{Id\times R} \mathcal{M}_1 \ \wedge \ \mathcal{D}_1 \sqsubseteq \mathcal{T}_1$ must hold. We prove this goal as follows:

$$\begin{array}{l}
\quad \mathcal{L}_1 \sqsubseteq_{Id\times R} \mathcal{M}_1 \ \wedge \ \mathcal{D}_1 \sqsubseteq \mathcal{T}_1 \\
= \{\textit{definitions, definition of data refinement}\ \} \\
\quad L_1\ ()\ \mathcal{C}_1\!\downarrow\!\pi_\Sigma \sqsubseteq (L_1\ ()\ \mathcal{T}_1\!\downarrow\!\pi_{\Sigma'})\!\uparrow\!(Id\times R) \ \wedge \ D_1() \sqsubseteq D_1() \\
\Leftarrow \{\textit{monotonicity of } L_1\textit{, assumption, logic}\ \} \\
\quad L_1\ ()\ \mathcal{D}_1\!\uparrow\!R\!\downarrow\!\pi_\Sigma \sqsubseteq (L_1\ ()\ \mathcal{T}_1\!\downarrow\!\pi_{\Sigma'})\!\uparrow\!(Id\times R) \\
\Leftarrow \{\textit{wrapping theorem}\} \\
\quad L_1\ ()\ \mathcal{D}_1\!\uparrow\!R\!\downarrow\!\pi_\Sigma \sqsubseteq L_1\ ()\ \mathcal{T}_1\!\downarrow\!\pi_{\Sigma'}\!\uparrow\!(Id\times R) \\
\Leftarrow \{\textit{monotonicity of } L_1\ \} \\
\quad \mathcal{D}_1\!\uparrow\!R\!\downarrow\!\pi_\Sigma \sqsubseteq \mathcal{T}_1\!\downarrow\!\pi_{\Sigma'}\!\uparrow\!(Id\times R) \\
= \{\textit{definition of data refinement}\ \} \\
\quad \mathcal{D}_1\!\uparrow\!R\!\downarrow\!\pi_\Sigma\!\downarrow\!(Id\times R) \sqsubseteq \mathcal{T}_1\!\downarrow\!\pi_{\Sigma'} \\
= \{\textit{lemma 1}\ \} \\
\quad \mathcal{D}_1\!\uparrow\!R\!\downarrow\!R\!\downarrow\!\pi_{\Sigma'} \sqsubseteq \mathcal{T}_1\!\downarrow\!\pi_{\Sigma'} \\
\Leftarrow \{\textit{wrapping rule}\ \} \\
\quad \mathcal{D}_1\!\downarrow\!\pi_{\Sigma'} \sqsubseteq \mathcal{T}_1\!\downarrow\!\pi_{\Sigma'} \\
\Leftarrow \{\textit{monotonicity of wrappers}\ \} \\
\quad \mathcal{D}_1 \sqsubseteq \mathcal{T}_1 \\
= \{\textit{definitions}\ \} \\
\quad \textit{true}
\end{array}$$

Inductive case: Our inductive assumption states that when the participating entities have n methods, the goal holds. After transformations, our proof obligation for the case of $n+1$ methods is:

$$
\begin{array}{l}
(\mathcal{C}_1, \ldots, \mathcal{C}_{n+1}) \sqsubseteq_R (\mathcal{D}_1, \ldots, \mathcal{D}_{n+1}) \;\wedge\; (\mathcal{C}_1, \ldots, \mathcal{C}_{n+1}) \sqsubseteq_{\pi_\Sigma} (\mathcal{L}_1, \ldots, \mathcal{L}_{n+1}) \;\wedge \\
(\mathcal{L}_1, \ldots, \mathcal{L}_n) \sqsubseteq_{Id \times R} (\mathcal{M}_1, \ldots, \mathcal{M}_n) \;\wedge\; (\mathcal{D}_1, \ldots, \mathcal{D}_n) \sqsubseteq (\mathcal{T}_1, \ldots, \mathcal{T}_n) \;\Rightarrow \\
\quad \mathcal{L}_{n+1} \sqsubseteq_{Id \times R} \mathcal{M}_{n+1} \;\wedge\; \mathcal{D}_{n+1} \sqsubseteq \mathcal{T}_{n+1}
\end{array}
$$

We prove this goal as follows:

$$
\begin{array}{l}
\quad \mathcal{L}_{n+1} \sqsubseteq_{Id \times R} \mathcal{M}_{n+1} \;\wedge\; \mathcal{D}_{n+1} \sqsubseteq \mathcal{T}_{n+1} \\
= \{\textit{definitions, definition of data refinement}\} \\
\quad L_{n+1}\ (\mathcal{L}_1, \ldots, \mathcal{L}_n)\ (\mathcal{C}_1, \ldots, \mathcal{C}_{n+1}){\downarrow}\pi_\Sigma \sqsubseteq \\
\qquad (L_{n+1}\ (\mathcal{M}_1, \ldots, \mathcal{M}_n)\ (\mathcal{T}_1, \ldots, \mathcal{T}_{n+1}){\downarrow}\pi_{\Sigma'}){\uparrow}(Id \times R) \;\wedge\; \mathcal{D}_{n+1} \sqsubseteq \mathcal{T}_{n+1} \\
\Leftarrow \{\textit{assumption } (\mathcal{L}_1, \ldots, \mathcal{L}_n) \sqsubseteq (\mathcal{M}_1, \ldots, \mathcal{M}_n){\uparrow}(Id \times R), \\
\quad \textit{assumption } (\mathcal{C}_1, \ldots, \mathcal{C}_{n+1}) \sqsubseteq (\mathcal{D}_1, \ldots, \mathcal{D}_{n+1}){\uparrow}R, \textit{monotonicity of } L_{n+1}\} \\
\quad L_{n+1}\ (\mathcal{M}_1, \ldots, \mathcal{M}_n){\uparrow}(Id \times R)\ (\mathcal{D}_1, \ldots, \mathcal{D}_{n+1}){\uparrow}R{\downarrow}\pi_\Sigma \sqsubseteq \\
\qquad (L_{n+1}\ (\mathcal{M}_1, \ldots, \mathcal{M}_n)\ (\mathcal{T}_1, \ldots, \mathcal{T}_{n+1}){\downarrow}\pi_{\Sigma'}){\uparrow}(Id \times R) \;\wedge\; \mathcal{D}_{n+1} \sqsubseteq \mathcal{T}_{n+1} \\
\Leftarrow \{\textit{wrapping theorem }\} \\
\quad L_{n+1}\ (\mathcal{M}_1, \ldots, \mathcal{M}_n){\uparrow}(Id \times R)\ (\mathcal{D}_1, \ldots, \mathcal{D}_{n+1}){\uparrow}R{\downarrow}\pi_\Sigma \sqsubseteq \\
\qquad L_{n+1}\ (\mathcal{M}_1, \ldots, \mathcal{M}_n){\uparrow}(Id \times R)\ (\mathcal{T}_1, \ldots, \mathcal{T}_{n+1}){\downarrow}\pi_{\Sigma'}{\uparrow}(Id \times R) \;\wedge \\
\qquad\quad \mathcal{D}_{n+1} \sqsubseteq \mathcal{T}_{n+1} \\
\Leftarrow \{\textit{monotonicity of } L_{n+1}\} \\
\quad (\mathcal{D}_1, \ldots, \mathcal{D}_{n+1}){\uparrow}R{\downarrow}\pi_\Sigma \sqsubseteq (\mathcal{T}_1, \ldots, \mathcal{T}_{n+1}){\downarrow}\pi_{\Sigma'}{\uparrow}(Id \times R) \;\wedge\; \mathcal{D}_{n+1} \sqsubseteq \mathcal{T}_{n+1} \\
\Leftarrow \{\textit{definitions of data refinement}\} \\
\quad (\mathcal{D}_1, \ldots, \mathcal{D}_{n+1}){\uparrow}R{\downarrow}\pi_\Sigma{\downarrow}(Id \times R) \sqsubseteq (\mathcal{T}_1, \ldots, \mathcal{T}_{n+1}){\downarrow}\pi_{\Sigma'} \;\wedge\; \mathcal{D}_{n+1} \sqsubseteq \mathcal{T}_{n+1} \\
= \{\textit{lemma 1 }\} \\
\quad (\mathcal{D}_1, \ldots, \mathcal{D}_{n+1}){\uparrow}R{\downarrow}R{\downarrow}\pi_{\Sigma'} \sqsubseteq (\mathcal{T}_1, \ldots, \mathcal{T}_{n+1}){\downarrow}\pi_{\Sigma'} \;\wedge\; \mathcal{D}_{n+1} \sqsubseteq \mathcal{T}_{n+1} \\
\Leftarrow \{\textit{wrapping rule }\} \\
\quad (\mathcal{D}_1, \ldots, \mathcal{D}_{n+1}){\downarrow}\pi_{\Sigma'} \sqsubseteq (\mathcal{T}_1, \ldots, \mathcal{T}_{n+1}){\downarrow}\pi_{\Sigma'} \;\wedge\; \mathcal{D}_{n+1} \sqsubseteq \mathcal{T}_{n+1} \\
\Leftarrow \{\textit{monotonicity of wrappers, assumption } (\mathcal{D}_1, \ldots, \mathcal{D}_n) \sqsubseteq (\mathcal{T}_1, \ldots, \mathcal{T}_n)\} \\
\quad \mathcal{D}_{n+1} \sqsubseteq \mathcal{T}_{n+1} \\
= \{\textit{definitions}\} \\
\quad D_{n+1}\ (\mathcal{C}_1, \ldots, \mathcal{C}_n){\downarrow}R \sqsubseteq D_{n+1}\ (\mathcal{M}_1, \ldots, \mathcal{M}_n){\uparrow}\pi_{\Sigma'} \\
\Leftarrow \{\textit{assumption } (\mathcal{C}_1, \ldots, \mathcal{C}_{n+1}) \sqsubseteq (\mathcal{L}_1, \ldots, \mathcal{L}_{n+1}){\uparrow}\pi_\Sigma, \textit{monotonicity of } L_{n+1}\ \} \\
\quad D_{n+1}\ (\mathcal{L}_1, \ldots, \mathcal{L}_n){\uparrow}\pi_\Sigma{\downarrow}R \sqsubseteq D_{n+1}\ (\mathcal{M}_1, \ldots, \mathcal{M}_n){\uparrow}\pi_{\Sigma'} \\
\Leftarrow \{\textit{assumption } (\mathcal{L}_1, \ldots, \mathcal{L}_{n+1}) \sqsubseteq (\mathcal{M}_1, \ldots, \mathcal{M}_{n+1}){\uparrow}(Id \times R), \\
\quad \textit{monotonicity of } L_{n+1}\ \} \\
\quad D_{n+1}\ (\mathcal{M}_1, \ldots, \mathcal{M}_n){\uparrow}(Id \times R){\uparrow}\pi_\Sigma{\downarrow}R \sqsubseteq D_{n+1}\ (\mathcal{M}_1, \ldots, \mathcal{M}_n){\uparrow}\pi_{\Sigma'} \\
= \{\textit{lemma 2}\} \\
\quad D_{n+1}\ (\mathcal{M}_1, \ldots, \mathcal{M}_n){\uparrow}\pi_{\Sigma'}{\uparrow}R{\downarrow}R \sqsubseteq D_{n+1}\ (\mathcal{M}_1, \ldots, \mathcal{M}_n){\uparrow}\pi_{\Sigma'} \\
\Leftarrow \{\textit{wrapping rule }\} \\
\quad D_{n+1}\ (\mathcal{M}_1, \ldots, \mathcal{M}_n){\uparrow}\pi_{\Sigma'} \sqsubseteq D_{n+1}\ (\mathcal{M}_1, \ldots, \mathcal{M}_n){\uparrow}\pi_{\Sigma'} \\
= \;\; \textit{true}
\end{array}
$$

□

6 Application

The refinement calculus allows for reasoning about correctness of refinement between statements or programming constructs in general. In this respect it does not differentiate between abstract specification statements and executable ones. Therefore, our results are applicable to any group consisting of a base class, its extension, and a modification of the base class. In particular, two scenarios are of interest:

- A base class C is a specification class, i.e. its methods contain specification statements, and C' is its implementation;
- A base class C is an executable class and C' is its revision.

The fragile base class problem was first noticed while framework maintenance, and corresponds to the second scenario. However, this problem occurs in the first one as well.

We formulate three check lists for system designers, developers (i.e. implementors or maintainers) and its extenders. By verifying that every item in a list holds, all parties can make sure that they avoid the fragile base class problem. First, we present the *check list for system designers*:

1. An extension class should not have a direct access to the inherited state. In some languages [24] this can be enforced by declaring attributes storing the class state as *private*. If future extensions are expected to require more freedom in modifying the class state than is allowed by the class client interface, the class can provide a number of low-level state modifying methods. Since having these methods as part of the class client interface may be inappropriate, we suggest declaring them as *protected* [21, 24]. Protected methods are accessible only to derived classes.
2. Method bodies must explicitly indicate all *self* method calls.

Now let us consider the *check list for system implementors and system maintainers*:

1. A method of a revision class may self-call only those methods that are self-called by its counterpart in the base class.
2. When verifying that some method of a revision is a refinement of a matching method in the base class, instead of considering the bodies of self-called methods, one should consider the bodies of the corresponding methods in the base class.

And finally, *check list for system extenders* is as follows:

1. A method of an extension class may only self-call those methods that are self-called by its counterpart in the base class, or it may make up-calls to these methods. Plus, an extension method may always make up-calls to its counterpart in the base class.

2. When verifying that some method of an extension is a refinement of a matching method in the base class, one should disregard the fact that due to dynamic binding the base class can make down-calls to extension class methods. One should consider that the base class calls its own methods instead.

3. A method of an extension class may modify the state of its base class only by making up-calls to methods of the latter.

4. An extension class may not establish an invariant binding values of inherited instance variables with values of its own instance variables.

7 Related Work and Conclusions

The fragile base class problem is of particular importance for object-oriented software development. As soon as one defines a base class, an extension, and wants to change the base class, one faces this problem. Even in case of a large closed system this problem becomes non-trivial and painful to deal with. In case of an open system it becomes crucial.

The name fragile base class problem was introduced while discussing component standards [29, 14], since it has critical significance for component systems. Modification of the components by their developers should not affect component extensions of their users in any respect. Firstly, recompilation of derived classes should be avoided if possible [14]. This issue constitutes a syntactic aspect of the problem. While being apparently important, that problem is only a technical issue. Even if recompilation is not necessary, component developers can make inconsistent modifications. Such inconsistent base class modifications constitute a semantic aspect of the problem, which is the focus of our study. This aspect of the problem was recognized by COM developers [29]. They see the root of the problem in inheritance violating data encapsulation and choose to abandon inheritance by employing the forwarding architecture. Although solving the problem, this approach comes at the cost of reduced flexibility. We choose to carry out a detailed study of the problem which demonstrates that, even when encapsulation is violated, consistent modifications of base classes are still possible.

Other papers consider the question of ensuring substitutability of classes created by inheritance from some base class for this base class [1, 16, 22]. This problem constitutes only a part of the fragile base class problem, and as such, is one of the flexibility theorem premises.

The fragile base class problem in our formulation (although they do not refer to it by this name) is considered by Steyaert et al. in [23] and by Kiczales and Lamping in [15]. The first paper is most closely related to our work. The authors introduce *reuse contracts* "that record the protocol between managers and users of a reusable asset". Acknowledging that "reuse contracts provide only syntactic information", they claim that "this is enough to firmly increase the likelihood of behaviorally correct exchange of parent classes". Such syntactic reuse contracts are, in general, insufficient to guard against the fragile base class problem.

As we have confined the problem, three of the four conflicts Steyaert et al. describe (those dealing with new method introduction in a base class) do not apply. The fourth conflict falls into the domain of our analysis. Our first example is, in fact, adopted from their paper. They blame the revision Bag' of the base class for modifying the structure of self-calls of Bag and, therefore, causing the problem.

They state that, in general, such method inconsistency can arise only when a revision chooses to eliminate a self-call to the method which is overridden in a modifier. From this one could conclude that preserving the structure of self-calls in a revision would safeguard against inconsistencies. Our examples demonstrate that this is not the case.

Our analysis reveals different causes of the *Bag/CountingBag* problem. The extension class *CountingBag* apparently violates our "no modifier invariant function" requirement. The class maintains an invariant $n = |b|$ binding values of an instance variable n with the number of elements in the inherited instance variable b. This invariant is no longer preserved when Bag is substituted with Bag' and, therefore, the *cardinality* method of the resulting class returns an incorrect value.

The objective of the paper by Kiczales and Lamping is to develop a methodology for informally specifying a framework, so that the specification would accurately describe its functionality and provide the framework user with appropriate leeway for extending and reusing it. However, their recommendations are based only on empirical expertise. We believe that such methodology should be grounded on a mathematical basis and developing such methodology constitutes the ultimate goal of our research.

The application of formal methods to analyzing the fragile base class problem gives us an opportunity to gain a deeper insight into the problem, which was impossible without it. In fact, only the "no direct access to base class state" requirement was known from the literature. The other requirements emerged while attempting to formally prove the flexibility property. The correspondence between the requirements we formulate and good programming style guidelines is not incidental. We believe that our theory provides a mathematical foundation for the empirical expertise.

In this respect, the simplicity of formal model has crucial importance for its applicability to studies of the kind described in this paper. Hense in

[12] gives a full denotational semantics of an object-oriented programming language. His model provides for classes with state, self-referencing, and multiple inheritance. The latter is described using wrappers of Cook and Palsberg [8]. However, a number of notions crucial for our purposes, such as, e.g. refinement ordering, are not defined. Our model includes only the constructs necessary for considering the fragile base class problem. Application of the refinement calculus to formalizing classes makes our class semantics more succinct. The refinement calculus also gives us a mathematical framework for formal reasoning about substitutability of objects. Note also that we express class refinement through abstract data type refinement which is a well established theory. This allows applying a large collection of refinement laws for verification.

By formulating the problem in terms of the flexibility property and formally proving it, we demonstrate that the restrictions imposed on inheritance are sufficient to circumvent the problem.

The analysis of the fragile base class problem has revealed that the inheritance operator **mod** is not monotonic in its base class argument. The monotonicity property

$$\text{if } C \textit{ is refined by } C' \text{ then } (M \textbf{ mod } C) \textit{ is refined by } (M \textbf{ mod } C')$$

is stronger than the flexibility property. If inheritance were monotonic, not only the fragile base class problem would be avoided, but also the extension $(M \textbf{ mod } C)$ would be fully substitutable with the new extension $(M \textbf{ mod } C')$, whenever C is substitutable with C'. Extra restrictions should be imposed on inheritance to make it monotonic. Examining these restrictions is the subject of our future research. Effects of disciplining inheritance, the way we propose, on component and object-oriented languages and systems require separate consideration and also constitute the subject of future research. The other research direction is in generalizing the results by relaxing the confinements on the problem that we have made and weakening the restrictions we have imposed on the inheritance mechanism.

Acknowledgments

We would like to thank Ralph Back, Jockum von Wright, Anna Mikhajlova, Jim Grundy, Wolfgang Weck and Linas Laibinis for many useful comments.

References

[1] Pierre America. Designing an object-oriented programming language with behavioural subtyping. *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June*

1990, volume 489 of Lecture Notes in Computer Science, pp. 60–90, Springer–Verlag, 1991.

[2] Apple Computer, Inc. OpenDoc programmer's guide. *Addison-Wesley Publishing Company,* Draft. Apple Computer, Inc., 9/4/95

[3] R.J.R. Back. Correctness preserving program refinements: proof theory and applications. *vol. 131 of Mathematical Center Tracts.* Amsterdam: Mathematical Center, 1980.

[4] R.J.R. Back. Changing data representation in the refinement calculus. *21st Hawaii International Conference on System Sciences*, 1989.

[5] R.J.R. Back, J. von Wright. Refinement Calculus, A Systematic Introduction. *to be published by Springer–Verlag*, 1997.

[6] R.J.R. Back and M.J. Butler. Exploring summation and product operators in the refinement calculus. *Mathematics of Program Construction, 1995*, volume 947 of Lecture Notes in Computer Science, Springer–Verlag, 1995.

[7] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys,* 17(4):471–522, 1985.

[8] William Cook, Jens Palsberg. A denotational semantics of inheritance and its correctness. *OOPSLA '89 Proceedings*, volume 24 of SIGPLAN Notices, pp. 433–443, October 1989.

[9] E.W.Dijkstra, A discipline of programming. *Prentice–Hall International*, 1976.

[10] P.H.B. Gardiner and C.C. Morgan. Data refinement of predicate transformers. *Theoretical Computer science*, 87:143-162, 1991.

[11] Jifeng He, C.A.R.Hoare and J.W.Sanders. Data Refinement Refined. In *European Symposium on Programming*, volume 213 of Lecture Notes in Computer Science, Springer–Verlag, 1986.

[12] Andreas V. Hense. Denotational semantics of an object-oriented programming language with explicit wrappers. Volume 5 of *Formal Aspects of Computing*, pp. 181.207, Springer–Verlag, 1993.

[13] C.A.R. Hoare. Proofs of correctness of data representation. *Acta informatica,* 1(4), 1972.

[14] IBM Corporation. IBM's System Object Model (SOM): Making reuse a reality. *IBM Corporation, Object Technology Products Group,* Austin, Texas.

[15] Gregor Kiczales, John Lamping. Issues in the design and specification of class libraries. *OOPSLA '92 Proceedings*, volume 27 of SIGPLAN Notices, pp. 435–451, October 1992.

[16] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems,*16(6):1811–1841, November 1994.

[17] Carroll C.Morgan. Programming from specifications. *Prentice-Hall,* 1990.

[18] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming,* 9, 287–306, 1987.

[19] Cuno Pfister, Clemens Szyperski. Oberon/F framework. Tutorial and reference. *Oberon microsystems, Inc.*, 1994.

[20] Dick Pountain, Clemens Szyperski. Extensible software systems. *Byte Magazine,* 19(5): 57–62, May 1994.
http://www.byte.com/art/9405/sec6/art1.html.

[21] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. *OOPSLA '86 Proceedings*, volume 21 of SIGPLAN Notices, pp. 38–45, 1986.

[22] Raymie Stata, John V. Guttag. Modular reasoning in the presence of subtyping. *OOPSLA '95 Proceedings*, volume 30 of SIGPLAN Notices, pp. 200–214, 1995.

[23] Patric Steyaert, Carine Lucas, Kim Mens, Theo D'Hondt. Reuse contracts: managing the evolution of reusable assets. *OOPSLA '96 Proceedings*, volume 31 of SIGPLAN Notices, pp. 268–285, 1996.

[24] Bjarne Stroustrup. The C++ programming language. *Addison-Wesley,* 1986.

[25] Clemencs Szyperski. Independently extensible systems. Software engineering potential and challenges. *Proceedings of the 19th Australasian Computer Science Conference* Melbourne, Australia, February 1996.

[26] David Taenzer, Murthy Gandi, Sunil Podar. Problems in object-oriented software reuse. *Proceedings ECOOP'89, S. Cook (Ed.), Cambridge University Press* Nottingham, July 10-14, 1989, pp. 25–38.

[27] A. Tarski. A Lattice Theoretical Fixed Point Theorem and its Applications. volume 5 of *Pacific J. Mathematics*, pp. 285–309, 1955.

[28] Peter Wegner, Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. *Proceedings ECOOP'88,* volume 322 of Lecture Notes in Computer Science, pp. 55–77, Springer-Verlag, Oslo, August 1988.

[29] S. Williams and C. Kinde. The Component Object Model: Technical Overview. *Dr. Dobbs Journal,* December 1994.

Appendix A

Before proving the wrapping theorem, we first need to consider a number of theorems and laws of the refinement theory.

Without loss of generality, an arbitrary statement calling procedures $m_1, ..., m_n$ can be rewritten as

$$\textbf{begin var } l \mid p \cdot \textbf{do } g_1 \to m_1; S_1 \, [] \, ... \, [] \, g_n \to m_n; S_n \textbf{ od end}$$

which, using the **while** statement, is equivalently expressed as

$$\textbf{begin var } l \mid p \cdot \textbf{while } (g_1 \vee ... \vee g_n) \textbf{ do } [g_1]; m_1; S_1 \sqcap ... \sqcap [g_n]; m_n; S_n \textbf{ od end}$$

for some initialization predicate p, guards g_i, and conjunctive statements S_i. Further on we write S_i **for** $i = 1..n$ as an abbreviation for $S_1 \sqcap ... \sqcap S_n$.

In particular, we can represent the body of an arbitrary method m_i of a modifier in the following form:

$$\begin{array}{rl} m_i = & \textbf{begin var } l \mid p \cdot \\ & \quad \textbf{while } g_1 \vee ... \vee g_{i-1} \vee g'_1 \vee ... \vee g'_i \textbf{ do} \\ & \quad\quad [g_j]; \mathit{self}.m_j; S_j \textbf{ for } j = 1..i-1 \sqcap \\ & \quad\quad [g'_j]; \mathit{super}.m_j; S'_j \textbf{ for } j = 1..i \\ & \quad \textbf{od} \\ & \textbf{end} \end{array}$$

for some local variable l, initialization predicate p, guards $[g_i]$, $[g'_i]$, and conjunctive statements S_i, S'_i.

We make use of the following rules for data refinement of sequential composition of statements, nondeterministic choice of statements, and **while** statement presented in [5]:

$$\begin{array}{rl} ;\ \mathit{rule}: & (S_1; S_2){\downarrow}R \sqsubseteq (S_1{\downarrow}R); (S_2{\downarrow}R) \\ \sqcap\ \mathit{rule}: & (S_1 \sqcap S_2){\downarrow}R \sqsubseteq (S_1{\downarrow}R \sqcap S_1{\downarrow}R) \\ \mathit{While\ rule}: & \textbf{while } g \textbf{ do } S \textbf{ od}{\downarrow}R \sqsubseteq \textbf{while } \{R\}\ g \textbf{ do } [[R]\ g]; (S{\downarrow}R) \textbf{ od} \end{array}$$

We define an *indifferent statement* [5] as a statement that does not refer to components of the state coerced by a data refinement relation. We say that the statement $S : \Gamma \times \Delta \mapsto \Gamma \times \Delta$, which modifies only the Δ component of the state, is indifferent to $R : \Gamma \times \Delta \leftrightarrow \Sigma \times \Delta$ which modifies only the Γ component of the state. We employ the following result for calculating a data refinement of an indifferent statement:

$$\mathit{Indifference\ rule}: \quad S{\downarrow}R \sqsubseteq S, \mathit{where\ S\ is\ conjunctive\ and\ indifferent\ to\ R}$$

In case when a guard g of the **while** statement is indifferent towards a relation R, it is possible to show that the while-loop rule can be slightly modified:

$$\mathit{Indifferent\ while\ rule}: \ \textbf{while } g \textbf{ do } S \textbf{ od}{\downarrow}R \sqsubseteq \textbf{while } g \textbf{ do } S{\downarrow}R \textbf{ od}$$

Wrapping Theorem *Let $L, \mathcal{M}, \mathcal{T}$ and R be as were defined in Sec. 5, then*

$$L_i\ (\mathcal{M}_1, \ldots, \mathcal{M}_{i-1}){\uparrow}(Id \times R)\ (\mathcal{T}_1, \ldots, \mathcal{T}_i){\downarrow}\pi_{\Sigma'}{\uparrow}(Id \times R)\ \sqsubseteq$$
$$(L_i\ (\mathcal{M}_1, \ldots, \mathcal{M}_{i-1})\ (\mathcal{T}_1, \ldots, \mathcal{T}_i){\downarrow}\pi_{\Sigma'}){\uparrow}(Id \times R)$$

Proof Let us denote $\mathcal{T}_j{\downarrow}\pi_{\Sigma'}$ by $\mathcal{B}_j$ for $j = 1..i$ and $(Id \times R)$ by P. Then we can rewrite our goal as was described above:

$$\begin{array}{lcl}
\textbf{begin var } l \mid p\cdot & & \textbf{begin var } l \mid p\cdot \\
\ \textbf{while } g_1 \vee ... \vee g_{i-1} & & \ \textbf{while } g_1 \vee ... \vee g_{i-1} \\
\qquad\quad \vee g'_1 \vee ... \vee g'_i \textbf{ do} & & \qquad\quad \vee g'_1 \vee ... \vee g'_i \textbf{ do} \\
\ [g_j]; \mathcal{M}_j{\uparrow}P; S_j \textbf{ for } j = 1..i-1\ \sqcap & & \ [g_j]; \mathcal{M}_j; S_j \textbf{ for } j = 1..i-1\ \sqcap \\
\ [g'_j]; \mathcal{B}_j{\uparrow}P; S'_j \textbf{ for } j = 1..i & \sqsubseteq & \ [g'_j]; \mathcal{B}_j; S'_j \textbf{ for } j = 1..i \\
\ \textbf{od} & & \ \textbf{od} \\
\textbf{end}{\downarrow}P & & \textbf{end}
\end{array}$$

Consider the typing of the participating constructs. If the type of the local variable l is Λ, then state predicates p, g_j, g'_j and statements S_j are working on the state space $\Lambda \times \alpha \times \Gamma \times \beta$. Therefore, application of the appropriately extended relation P effectively coercing Σ' into Σ cannot influence these constructs, in other words, they are indifferent to P.

We prove the theorem by refining the left hand side to match the right hand side:

$$\begin{array}{ll}
& \textbf{begin var } l \mid p\cdot \\
& \quad \textbf{while } g_1 \vee ... \vee g_{i-1} \vee g'_1 \vee ... \vee g'_i \textbf{ do} \\
& \qquad [g_j]; \mathcal{M}_j{\uparrow}P; S_j \textbf{ for } j = 1..i-1\ \sqcap \\
& \qquad [g'_j]; \mathcal{B}_j{\uparrow}P; S'_j \textbf{ for } j = 1..i \\
& \quad \textbf{od} \\
& \textbf{end}{\downarrow}P \\
= \{ & \textit{definition of block } \} \\
& (\textbf{enter}\, l \mid p; \\
& \textbf{while } g_1 \vee ... \vee g_{i-1} \vee g'_1 \vee ... \vee g'_i \textbf{ do} \\
& \quad [g_j]; \mathcal{M}_j{\uparrow}P; S_j \textbf{ for } j = 1..i-1\ \sqcap \\
& \quad [g'_j]; \mathcal{B}_j{\uparrow}P; S'_j \textbf{ for } j = 1..i \\
& \textbf{od}; \\
& \textbf{exit}\, l){\downarrow}P \\
\sqsubseteq \{ & ;\ \textit{rule } \} \\
& (\textbf{enter}\, l \mid p){\downarrow}P; \\
& \textbf{while } g_1 \vee ... \vee g_{i-1} \vee g'_1 \vee ... \vee g'_i \textbf{ do} \\
& \quad [g_j]; \mathcal{M}_j{\uparrow}P; S_j \textbf{ for } j = 1..i-1\ \sqcap \\
& \quad [g'_j]; \mathcal{B}_j{\uparrow}P; S'_j \textbf{ for } j = 1..i \\
& \textbf{od}{\downarrow}P; \\
& (\textbf{exit}\, l){\downarrow}P
\end{array}$$

$\sqsubseteq$ { **enter** *and* **exit** *are conjunctive and indifferent to* P*, thus indifference rule, indifferent while rule* }

$$
\begin{array}{l}
\textbf{enter}\, l \mid p; \\
\textbf{while}\ g_1 \vee ... \vee g_{i-1} \vee g'_1 \vee ... \vee g'_i\ \textbf{do} \\
\quad ([g_j]; \mathcal{M}_j \uparrow P; S_j\ \textbf{for}\ j = 1..i-1 \sqcap \\
\quad [g'_j]; \mathcal{B}_j \uparrow P; S'_j\ \textbf{for}\ j = 1..i) \downarrow P \\
\textbf{od}; \\
\textbf{exit}\, l
\end{array}
$$

$\sqsubseteq$ { $\sqcap$ *rule*, ; *rule* }

$$
\begin{array}{l}
\textbf{enter}\, l \mid p; \\
\textbf{while}\ g_1 \vee ... \vee g_{i-1} \vee g'_1 \vee ... \vee g'_i\ \textbf{do} \\
\quad [g_j] \downarrow P; \mathcal{M}_j \uparrow P \downarrow P; S_j \downarrow P\ \textbf{for}\ j = 1..i-1 \sqcap \\
\quad [g'_j] \downarrow P; \mathcal{B}_j \uparrow P \downarrow P; S'_j \downarrow P\ \textbf{for}\ j = 1..i \\
\textbf{od}; \\
\textbf{exit}\, l
\end{array}
$$

$\sqsubseteq$ {$[g_j]$ *are indifferent to* P*, and* S_j *are conjunctive, thus indifference rule; wrapping rule* }

$$
\begin{array}{l}
\textbf{enter}\, l \mid p; \\
\textbf{while}\ g_1 \vee ... \vee g_{i-1} \vee g'_1 \vee ... \vee g'_i\ \textbf{do} \\
\quad [g_j]; \mathcal{M}_j; S_j\ \textbf{for}\ j = 1..i-1 \sqcap \\
\quad [g'_j]; \mathcal{B}_j; S'_j\ \textbf{for}\ j = 1..i \\
\textbf{od}; \\
\textbf{exit}\, l
\end{array}
$$

$=$ { *definition of block* }

$$
\begin{array}{l}
\textbf{begin}\,\textbf{var}\, l \mid p \cdot \\
\quad \textbf{while}\ g_1 \vee ... \vee g_{i-1} \vee g'_1 \vee ... \vee g'_i\ \textbf{do} \\
\quad\quad [g_j]; \mathcal{M}_j; S_j\ \textbf{for}\ j = 1..i-1 \sqcap \\
\quad\quad [g'_j]; \mathcal{B}_j; S'_j\ \textbf{for}\ j = 1..i \\
\quad \textbf{od} \\
\textbf{end}
\end{array}
$$

$\square$

Appendix B

Proofs of lemma 3, lemma 4, lemma 1 and lemma 2 rely on the following theorems of refinement calculus.

Inverse of relational cross product :	$(R \times P)^{-1} = R^{-1} \times P^{-1}$
inverse of relational composition :	$(R;P)^{-1} = P^{-1};R^{-1}$
homomorphism of angelic update :	$\{R;P\} = \{R\};\{P\}$
homomorphism of demonic update :	$[R;P] = [R];[P]$
composition of products :	$(R \times R');(P \times P') = R;P \times R';P'$

Lemma 3 *For relations* $R : \Sigma' \leftrightarrow \Sigma$, $\pi_\Sigma : \Gamma \times \Sigma \leftrightarrow \Sigma$ *and* $\pi_{\Sigma'} : \Gamma \times \Sigma' \leftrightarrow \Sigma'$, *the following holds:* $(Id \times R);\pi_\Sigma = \pi_{\Sigma'};R$.

Proof

$$
\begin{aligned}
& (Id \times R);\pi_\Sigma = \pi_{\Sigma'};R \\
= & \{\textit{extensionality}\ \} \\
& \forall (\gamma, \sigma'), \sigma \cdot (Id \times R);\pi_\Sigma\ (\gamma, \sigma')\ \sigma = \pi_{\Sigma'};R\ (\gamma, \sigma')\ \sigma \\
= & \{\textit{definition of relational composition}\ \} \\
& \forall (\gamma, \sigma'), \sigma \cdot \exists \gamma', \sigma'' \cdot (Id \times R)\ (\gamma, \sigma')\ (\gamma', \sigma'')\ \wedge\ \pi_\Sigma\ (\gamma', \sigma'')\ \sigma = \\
& \qquad \exists \sigma'' \cdot \pi_{\Sigma'}\ (\gamma, \sigma')\ \sigma''\ \wedge\ R\ \sigma''\ \sigma \\
= & \{\textit{definition of relational product and Id relation, and projection relations}\ \} \\
& \forall (\gamma, \sigma'), \sigma \cdot \exists \gamma', \sigma'' \cdot \gamma = \gamma'\ \wedge\ (R\ \sigma'\ \sigma'')\ \wedge\ \sigma'' = \sigma = \\
& \qquad \exists \sigma'' \cdot \sigma' = \sigma''\ \wedge\ R\ \sigma''\ \sigma \\
= & \{\textit{one point rule, logic}\ \} \\
& \forall \sigma', \sigma \cdot R\ \sigma'\ \sigma = R\ \sigma'\ \sigma \\
= & \ \textit{true}
\end{aligned}
$$

□

Lemma 4 *For relations* $R : \Sigma' \leftrightarrow \Sigma$, $\pi_\Sigma : \Gamma \times \Sigma \leftrightarrow \Sigma$ *and* $\pi_{\Sigma'} : \Gamma \times \Sigma' \leftrightarrow \Sigma'$, *the following holds:* $\pi_\Sigma^{-1};(Id \times R^{-1}) = R^{-1};\pi_{\Sigma'}^{-1}$.

Proof The proof is similar to the one of lemma 3. □

Lemma 1 *For* $S : \alpha' \times \Gamma \times \Sigma \mapsto \alpha' \times \Gamma \times \Sigma$ *and relations* $R : \Sigma' \leftrightarrow \Sigma$, $\pi_\Sigma : \Gamma \times \Sigma \leftrightarrow \Sigma$ *and* $\pi_{\Sigma'} : \Gamma \times \Sigma' \leftrightarrow \Sigma'$, *the following holds:*

$$S \downarrow (Id \times \pi_\Sigma) \downarrow (Id \times Id \times R) = S \downarrow (Id \times R) \downarrow (Id \times \pi_{\Sigma'})$$

Proof

$$
\begin{aligned}
& S \downarrow (Id \times \pi_\Sigma) \downarrow (Id \times Id \times R) \\
= & \{\textit{definitions of wrappings}\ \} \\
& \{Id \times Id \times R\}; \{Id \times \pi_\Sigma\}; S; [Id \times \pi_\Sigma^{-1}]; [Id \times Id \times R^{-1}] \\
= & \{\textit{homomorphisms}\ \} \\
& \{(Id \times Id \times R);(Id \times \pi_\Sigma)\}; S; [(Id \times \pi_\Sigma^{-1});(Id \times Id \times R^{-1})] \\
= & \{\textit{composition of products}\ \} \\
& \{Id;Id \times (Id \times R);\pi_\Sigma)\}; S; [Id;Id \times \pi_\Sigma^{-1};(Id \times R^{-1})] \\
= & \{\textit{lemma 3, lemma 4}\ \} \\
& \{Id \times \pi_{\Sigma'};R\}; S; [Id \times R^{-1};\pi_{\Sigma'}^{-1}] \\
= & \{\textit{composition of products}\ \} \\
& \{(Id \times \pi_{\Sigma'});(Id \times R)\}; S; [(Id \times R^{-1});(Id \times \pi_{\Sigma'}^{-1})] \\
= & \{\textit{homomorphisms, definitions of wrapping, definitions of relations}\ \} \\
& S \downarrow (Id \times R) \downarrow (Id \times \pi_{\Sigma'})
\end{aligned}
$$

□

Lemma 2 *For* $S : \alpha' \times \Gamma \times \Sigma' \mapsto \alpha' \times \Gamma \times \Sigma'$ *and relations* $R : \Sigma' \leftrightarrow \Sigma$, $\pi_\Sigma : \Gamma \times \Sigma \leftrightarrow \Sigma$ *and* $\pi_{\Sigma'} : \Gamma \times \Sigma' \leftrightarrow \Sigma'$, *the following holds:*

$$S \uparrow (Id \times Id \times R) \uparrow (Id \times \pi_\Sigma) = S \uparrow (Id \times \pi_{\Sigma'}) \uparrow (Id \times R)$$

Proof The proof is similar to the one of lemma 1. □

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi

University of Turku
- **Department of Mathematical Sciences**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**