

A Study of The Fragile Base Class Problem

Leonid Mikhajlov¹ and Emil Sekerinski²

¹ Turku Centre for Computer Science,
Lemminkäisenkatu 14A, Turku 20520, Finland;
lmikhajl@aton.abo.fi

² McMaster University,
1280 Main Street West, Hamilton, Ontario, Canada, L8S4K1;
emil@ece.eng.mcmaster.ca

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 express the fragile base class problem in terms of a flexibility property. By means of five orthogonal examples, violating the flexibility property, we demonstrate different aspects of the problem. We formulate requirements for disciplining inheritance, and extend the refinement calculus to accommodate for classes, objects, class-based inheritance, and class refinement. We formulate and formally prove a flexibility theorem demonstrating that the restrictions we impose on inheritance are sufficient to permit safe substitution of a base class with its revision in presence of extension classes.

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 [28] are vulnerable to it. Such systems are delivered to users as a collection of classes. The users can reuse the functionality provided by the system by inheriting from system classes. Moreover, when the system is an object-oriented framework, the users can extend the 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 [29, 23]. In an open system the fragile base class problem requires consideration during design [32].

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 [15]. 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 [32]. 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. By means of five orthogonal examples violating the flexibility property, we demonstrate different aspects of the problem. Then we formulate requirements disciplining inheritance. We extend the refinement calculus [8, 19, 20] with notions of classes, objects, class-based inheritance, and refinement on classes. For the formalization of inheritance we adopt a model suggested by Cook and Palsberg in [10]. We formulate, prove, and explain 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. 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.

```

Bag = class
  b : bag of char

  init  $\hat{=}$  b := []
  add(val x : char)  $\hat{=}$ 
    b := b  $\cup$  [x]
  addAll(val bs : bag of char)  $\hat{=}$ 
    while bs  $\neq$  [] do
      begin var y | y  $\in$  bs.
        self.add(y);
        bs := bs - [y]
      end
    od
  cardinality(res r : int)  $\hat{=}$ 
    r := |b|
end

CountingBag = class
  inherits Bag
  n : int
  init  $\hat{=}$  n := 0; super.init
  add(val x : char)  $\hat{=}$ 
    n := n + 1; super.add(x)
  cardinality(res r : int)  $\hat{=}$ 
    r := n
end

Bag' = class
  b : bag of char
  init  $\hat{=}$  b := []
  add(val x : char)  $\hat{=}$  b := b  $\cup$  [x]
  addAll(val bs : bag of char)  $\hat{=}$  b := b  $\cup$  bs
  cardinality(res r : int)  $\hat{=}$  r := |b|
end

```

Fig. 1. Example of the fragile base class problem

2.1 Example of the Fragile Base Class Problem

We begin with an example, presented in Fig. 1, 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 safely used by the framework.

After some time a system developer decides to improve the efficiency of the class *Bag* and releases a new version of the system. An “improved” *Bag'* im-

¹ This example is adopted from [26]

plements *addAll* without invoking *add*. 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 $(M \text{ mod } C)^2$, 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* [31]. Therefore, we have that *C* belongs to the system, while $(M \text{ mod } C)$ represents a user extension of this system. The model of single inheritance employing the notion of modifiers was proved by Cook and Palsberg in [10] to correspond to the form of inheritance used in object-oriented systems.³ For instance, in our previous example *M* has the form:

```

M = modifier
  n : int := 0
  add(x : char)  $\hat{=}$  n := n + 1; super.add(x)
end

```

We accept the view that a base class and a modifier initialize their own instance variables.

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 a class *C* is *refined by* a 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. Ensuring substitutability of the user extension $(M \text{ mod } C)$ for the framework class *C* amounts to verifying that *C* is *refined by* $(M \text{ mod } C)$.

Thus, all participating parties, i.e. the system developers and its extenders, rely on a *flexibility property* :

```

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

```

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.

Unfortunately, the flexibility property does not hold in general, as demonstrated by our example. In our opinion, this fact constitutes the essence of the fragile base class problem. This consideration brings us to the question, what are the shortcomings of inheritance and what are the restrictions we need to make in order to evade the problem.

3 Aspects of the Problem

Now let us consider five examples invalidating the flexibility property and illuminating the shortcomings of inheritance.⁴ 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.

Let us first briefly introduce the used terminology. A call of a method through the implicit parameter *self* is referred to as a *self-call*. Analogously, we refer to an invocation of a method through the implicit parameter *super* as a *super-call*. When an extension class invokes a base class method, we say that an *up-call* has occurred; when a base class invokes a method of a class derived from it, we refer to such an invocation as a *down-call*.

3.1 Unanticipated Mutual Recursion

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 . 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.

$C = \mathbf{class}$	$M = \mathbf{modifier}$	$C' = \mathbf{class}$
$x : int := 0$		$x : int := 0$
$m \hat{=} x := x + 1$		$m \hat{=} self.n$
$n \hat{=} x := x + 1$	$n \hat{=} self.m$	$n \hat{=} x := x + 1$
\mathbf{end}	\mathbf{end}	\mathbf{end}

When the modifier M is applied to C' , the methods m and n of the resulting class ($M \mathbf{mod} C'$) become mutually recursive. Apparently, a call to either one leads to a never terminating loop.

This example demonstrates that the problem might occur due to unexpected appearance of mutual recursion of methods in the resulting class.

⁴ As we tried to keep these examples as concise as possible, they might appear slightly artificial.

3.2 Unjustified Assumptions in Revision Class

To illustrate the next shortcoming of inheritance, it is sufficient to provide only the specification of a base class. The base class C calculates the square and the 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 method m so that it would return a negative value.⁵ Such an implementation of m is a refinement of the original specification, because it reduces nondeterminism.

```
 $C = \text{class}$ 
   $m(\text{val } x : \text{real}, \text{res } r : \text{real}) \hat{=}$ 
    pre  $x \geq 0$ 
    post  $r^2 = x$ 
   $n(\text{val } x : \text{real}, \text{res } r : \text{real}) \hat{=}$ 
    pre  $x \geq 0$ 
    post  $r^4 = x$ 
end

 $M = \text{modifier}$ 
   $m(\text{val } x : \text{real}, \text{res } r : \text{real}) \hat{=}$ 
     $r := -\sqrt{x}$ 
end
```

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 .

```
 $C' = \text{class}$ 
   $m(\text{val } x : \text{real}, \text{res } r : \text{real}) \hat{=}$ 
     $r := \sqrt{x}$ 
   $n(\text{val } x : \text{real}, \text{res } r : \text{real}) \hat{=}$ 
     $\text{self}.m(x, r); \text{self}.m(r, r)$ 
end
```

Suppose now that we have an instance of a class ($M \text{ mod } C'$). 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 example demonstrates that the problem might occur due to an assumption in the revision class that, while a self-call, the body of a method, as defined in the revision class itself, is guaranteed to be executed. However, due to inheritance and dynamic binding this assumption is not justified.

3.3 Unjustified Assumptions in Modifier

Participants of this example are rather meaningless; however, from the more formal point of view, they are composed from legal constructs, and therefore,

⁵ By convention \sqrt{x} returns a positive square root of x .

should satisfy our flexibility property. We use an assertion statement $\{p\}$, where p is a state predicate. If p is true in a current state, the assertion skips, otherwise it aborts. Thus, the assertion statement can be seen as an abbreviation for the corresponding conditional.

```

C = class                                M = modifier
   $l(\mathbf{val} \ v : \mathit{int}) \hat{=} \{v \geq 5\}$      $l(\mathbf{val} \ v : \mathit{int}) \hat{=} \mathbf{skip}$ 
   $m(\mathbf{val} \ v : \mathit{int}) \hat{=} \mathit{self}.l(v)$ 
   $n(\mathbf{val} \ v : \mathit{int}) \hat{=} \mathbf{skip}$             $n(\mathbf{val} \ v : \mathit{int}) \hat{=} \mathit{self}.m(v)$ 
end                                       end

C' = class
   $l(\mathbf{val} \ v : \mathit{int}) \hat{=} \{v \geq 5\}$ 
   $m(\mathbf{val} \ v : \mathit{int}) \hat{=} \{v \geq 5\}$ 
   $n(\mathbf{val} \ v : \mathit{int}) \hat{=} \mathbf{skip}$ 
end

```

Let us compute full definitions of the classes $(M \ \mathbf{mod} \ C)$ and $(M \ \mathbf{mod} \ C')$.

```

(M mod C) = class                        (M mod C') = class
   $l(\mathbf{val} \ v : \mathit{int}) \hat{=} \mathbf{skip}$             $l(\mathbf{val} \ v : \mathit{int}) \hat{=} \mathbf{skip}$ 
   $m(\mathbf{val} \ v : \mathit{int}) \hat{=} \mathit{self}.l(v)$      $m(\mathbf{val} \ v : \mathit{int}) \hat{=} \{v \geq 5\}$ 
   $n(\mathbf{val} \ v : \mathit{int}) \hat{=} \mathit{self}.l(v)$      $n(\mathbf{val} \ v : \mathit{int}) \hat{=} \{v \geq 5\}$ 
end                                       end

```

It is easy to see that while C is refined by $(M \ \mathbf{mod} \ C)$, the class C is not refined by $(M \ \mathbf{mod} \ C')$. Due to the presence of assertion $\{v \geq 5\}$ in the method n of $(M \ \mathbf{mod} \ C')$, its precondition is stronger than the one of the method n of C , while to preserve refinement its precondition could have only been weakened.

Therefore, the problem might occur due to an assumption made in a modifier that in a particular layout base class self-calls are guaranteed to get redirected to the modifier itself. However such an assumption is unjustified, because the revision class can modify the self-calling structure.

3.4 Direct Access to the Base Class State

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 of C does, i.e. increases n by 2.

```

C = class                                M = modifier
   $x : \mathit{int} := 0$ 
   $m \hat{=} x := x + 1$ 
   $n \hat{=} x := x + 2$ 
end                                       end

```

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, $x = y$.

```

C' = class
   $x : int := 0; y : int := 0$ 
   $m \hat{=} y := y + 1; x := y$ 
   $n \hat{=} y := y + 2; x := y$ 
end

```

Now, if we consider an object obj to be an instance of class $(M \text{ mod } C')$, obtained by substituting C' for C , and the sequence of method 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 [23]. 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' .

3.5 Unjustified Assumption of Binding Invariant in Modifier

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 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 then asserting that the returned value is equal to y .

<pre> C = class $x : int := 0$ $l(\text{res } r : int) \hat{=} r := x$ $m \hat{=} x := x + 1; self.n$ $n \hat{=} \text{skip}$ end </pre>	<pre> M = modifier $y : int := 0$ $m \hat{=} y := y + 1; super.m$ begin var r· $n \hat{=} super.l(r); \{r = y\}$ end end </pre>
---	--

It is easy to see that before and after execution of any method of $(M \text{ mod } C)$ the predicate $x = y$ holds. We can say that $(M \text{ mod } C)$ maintains the invariant $x = y$. The full definition of the method m in an instance of the class $(M \text{ mod } C)$ 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.

```

C' = class
  x : int := 0
  l(res r : int) ≐ r := x
  m ≐ self.n; x := x + 1
  n ≐ skip
end

```

The body of the method m in an instance of the class ($M \text{ mod } C'$) is effectively of the form $y := y + 1; \{x = y\}; x := x + 1$, and, naturally, it aborts.

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. Thus the resulting extension class maintains an invariant binding values of inherited instance variables with the new instance variables. Such an invariant can be violated when the base class is substituted with its revision. If methods of the modifier rely on such an invariant, a crash might occur.

3.6 Discussion

The presented examples demonstrate different aspects of the fragile base class problem. However, this list of aspects is by no means complete. We have chosen these aspects, because in our opinion they constitute the core of the problem. Also among these basic aspects of the problem there are some which were apparently overlooked by other researchers, as we discuss in our conclusions.

Further on 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 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. Second, we consider the case when a base class revision and extension have as many methods as the corresponding base class. Third, for the time being we consider only functional modifiers, i.e. modifiers that do not have instance variables of their own. Modeling modifiers with state adds considerable complexity and constitutes a topic of current separate research. We also assume that a base class does not have recursive and mutually recursive methods. As we have stated above, our language provides only for single inheritance.

The first four shortcomings of inheritance illustrated above lead to the formulation of the four requirements disciplining it:

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

2. **“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.*
3. **“No base class down-calling assumptions” requirement:** *Modifier methods 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.*
4. **“No direct access to the base class state” requirement:** *An extension class should not access the state of its base class directly, but only through calling base class methods.*

We claim that if disciplining inheritance according to these four requirements, 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 formulating 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 [3, 6–8]. The behavior of a program statement can be characterized by Dijkstra’s weakest precondition predicate transformer [11]. 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 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):

$$\begin{aligned} \mathcal{P}\Sigma &\hat{=} \Sigma \rightarrow Bool \\ \Sigma \leftrightarrow \Gamma &\hat{=} \Sigma \rightarrow \mathcal{P}\Gamma \\ \Sigma \mapsto \Gamma &\hat{=} \mathcal{P}\Gamma \rightarrow \mathcal{P}\Sigma \end{aligned}$$

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

$$p \subseteq q \hat{=} (\forall \sigma : \Sigma . p \sigma \Rightarrow q \sigma)$$

The predicates *true* and *false* over Σ map every $\sigma : \Sigma$ to the boolean values T and F , 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 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:

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

The relational product operator is right associative.

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

$$Id \sigma_1 \sigma_2 \hat{=} \sigma_1 = \sigma_2$$

A predicate transformer $S : \Sigma \mapsto \Gamma$ is said to be *monotonic* if for all predicates p and q , $p \subseteq q$ implies $S p \subseteq S q$. 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

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

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

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

Meet of statements models *nondeterministic choice* between executing statements S and T . It is required that both alternatives establish the postcondition. For predicate transformers $S_1 : \Sigma_1 \mapsto \Gamma_1$ and $S_2 : \Sigma_2 \mapsto \Gamma_2$, their product $S_1 \times S_2$ is a predicate transformer of type $\Sigma_1 \times \Sigma_2 \mapsto \Gamma_1 \times \Gamma_2$ whose execution has the same effect as simultaneous execution of S_1 and S_2 :

$$(S_1 \times S_2) q \hat{=} (\cup q_1, q_2 \mid q_1 \times q_2 \subseteq q \cdot S_1 q_1 \times S_2 q_2)$$

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. For any predicate $q : \mathcal{P}\Sigma$

$$\begin{aligned} \mathbf{abort} q &\hat{=} \mathit{false} \\ \mathbf{magic} q &\hat{=} \mathit{true} \\ \mathbf{skip} q &\hat{=} q. \end{aligned}$$

For a predicate $p : \mathcal{P}\Gamma$, the assertion $\{p\}$ it behaves as **abort** if p does not hold, and as **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:

$$\begin{aligned} \{p\} q &\hat{=} p \wedge q \\ [p] q &\hat{=} p \Rightarrow q \end{aligned}$$

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:

$$\begin{aligned} \{P\} q \sigma &\hat{=} (\exists \gamma : \Gamma \cdot (P \sigma \gamma) \wedge (q \gamma)) \\ [P] q \sigma &\hat{=} (\forall \gamma : \Gamma \cdot (P \sigma \gamma) \Rightarrow (q \gamma)) \end{aligned}$$

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.

A statement S is said to be 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 variables $x : T$ and $y : U$, i.e. $\Sigma = T \times U$, then

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

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

$$\mathbf{pre } p \mathbf{ post } q \hat{=} \{p\};[R], \text{ where } q(x, y) = \forall(x', y') \cdot R(x, y) (x', y')$$

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

$$\mathbf{if } p \mathbf{ then } S \mathbf{ else } T \mathbf{ fi} \hat{=} ([p]; S) \sqcap ([\neg p]; S)$$

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:

$$S \sqsubseteq T \hat{=} (\forall q : \mathcal{P}\Gamma \cdot S q \subseteq T q)$$

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

$$\begin{aligned} \mathbf{pre } p \mathbf{ post } q &\sqsubseteq \mathbf{pre } p' \mathbf{ post } q' \\ \text{if } p &\subseteq p' \text{ and } q' \subseteq q \end{aligned}$$

Other rules for refinement of specifications into programs and transforming programs are given in [3, 19, 12].

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

$$\begin{aligned} \mathbf{while } p \mathbf{ do } S \mathbf{ od} &\hat{=} \mu F, \\ \text{where } F X &= \mathbf{if } p \mathbf{ then } S; X \mathbf{ else skip fi} \end{aligned}$$

According to the theorem of Knaster-Tarski [30], 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 $\mathbf{if } p(x) \mathbf{ then } S; X \mathbf{ else skip fi}$

is monotonic with respect to \sqsubseteq in X , therefore, μF exists and is unique. 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.

$$\begin{aligned} \mathbf{enter} z | p &\hat{=} [P], \\ &\text{where } P(x, y) (z', x', y') = (x' = x \wedge y' = y \wedge p(z', x', y')) \\ \mathbf{exit} z &\hat{=} [Q], \\ &\text{where } Q(z, x, y) (x', y') = (x' = x \wedge y' = y) \end{aligned}$$

Accordingly, the block construct is defined as follows:

$$\mathbf{begin} \text{ var } z | p \cdot S \mathbf{end} \hat{=} \mathbf{enter} z | p; S; \mathbf{exit} z$$

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:

$$\begin{aligned} \mathit{unwrapping} : S \downarrow R &\hat{=} \{R\}; S; [R^{-1}] \\ \mathit{wrapping} : S' \uparrow R &\hat{=} [R^{-1}]; S'; \{R\} \end{aligned}$$

Thus, after wrapping we have that $S \downarrow R$ and $S' \uparrow R$ operate on the state spaces Σ' and Σ respectively. 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 $Id \times R$ or $R \times Id$. For example, to wrap a statement $S : \Sigma \times \Gamma \mapsto \Sigma \times \Gamma$, the relation $R : \Sigma' \leftrightarrow \Sigma$ is extended to $R \times Id$.

The wrapping and unwrapping operators are left associative and have the lower precedence than the relational product. Further on, we make use of the following

$$\mathit{wrapping} \text{ rule} : S \uparrow R \downarrow R \sqsubseteq S$$

For tuples of statements we have:

$$\begin{aligned} (S_1, \dots, S_n) \uparrow R &\hat{=} (S_1 \uparrow R, \dots, S_n \uparrow R) \\ (S_1, \dots, S_n) \downarrow R &\hat{=} (S_1 \downarrow R, \dots, S_n \downarrow R) \end{aligned}$$

4.2 Formalization of Classes and Modifiers

We model classes as self referential structures as proposed by Cook and Palsberg in [10]. This model applies to most mainstream object-oriented languages like C++, Java and Smalltalk. However, in our formalization classes have internal state, unlike in their model.

For simplicity, we model method parameters by global variables that both methods of a class and its clients can access. We precede a formal method parameter with a keyword **val** to indicate that the method only reads the value of this parameter without changing it. Similarly, we precede the formal parameter with **res** to indicate that the method returns a value in this parameter.

In practice a call of a method m_j by a method m_i of the same class has the form $self.m_j$. Due to inheritance and dynamic binding such a call can get redirected to a definition of m_j in an extension class. We model the method m_i as a function of an argument $self.m_j$. In general, a method of a class may invoke all the other methods of the same class. Thus if there are n methods in the class we have

$$m_i = \lambda(x_1, \dots, x_n).c_i,$$

where c_i is a statement representing the body of the method m_i and x_j stands for $self.m_j$. Accordingly, we define a class C by

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

and where $c_0 : \Sigma$ is an initial value of the internal class state and $self$ is an abbreviation for the tuple x_1, \dots, x_n . We assume that cm is monotonic in the $self$ parameter. For our purposes it suffices to model the state space of all parameters of all methods as an extra component Δ of the class state space. Thus cm has the type

$$\begin{aligned} & ((\Sigma \times \Delta \mapsto \Sigma \times \Delta) \times \dots \times (\Sigma \times \Delta \mapsto \Sigma \times \Delta)) \rightarrow \\ & ((\Sigma \times \Delta \mapsto \Sigma \times \Delta) \times \dots \times (\Sigma \times \Delta \mapsto \Sigma \times \Delta)). \end{aligned}$$

We declare the class C , defined in this way, as follows:

$$C = \mathbf{class} \ c := c_0, \ m_1 \hat{=} c_1, \dots, m_n \hat{=} c_n \ \mathbf{end}$$

A class can be depicted as in Fig. 2. The incoming arrow represents calls to the class C , the outgoing arrow stands for self-calls of C .

In our model classes are used as a templates for creating objects. Objects have all of the self-calls resolved with the methods of the same object. Modeling this formally amounts to taking the least fixed point of the function cm . For tuples of statements (c_1, \dots, c_n) and (c'_1, \dots, c'_n) , where c_i and c'_i are of the same type, the refinement ordering is defined elementwise:

$$(c_1, \dots, c_n) \sqsubseteq (c'_1, \dots, c'_n) \hat{=} c_1 \sqsubseteq c'_1 \wedge \dots \wedge c_n \sqsubseteq c'_n$$

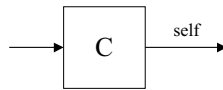


Fig. 2. Illustration of a class

Statement tuples form a complete lattice with the above refinement ordering. Also cm is monotonic in its argument. These two conditions are sufficient to guarantee that the least fixed point of the function cm exists and is unique. 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 its methods cm :

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

Fig. 3 illustrates taking the fixed point of the class C .

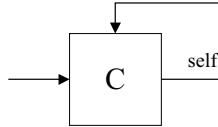


Fig. 3. Illustration of taking a fixed point of C

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:

$$create C \hat{=} \mu C$$

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 $L = \mathbf{modifier} m_1 \hat{=} l_1, \dots, m_n \hat{=} l_n \mathbf{end}$ is modeled by a function

$$L = \lambda self \cdot \lambda super \cdot (l_1, \dots, l_n),$$

where *self* and *super* are abbreviations of the tuples x_1, \dots, x_n and y_1, \dots, y_n respectively. We assume that L is monotonic in both arguments. See Fig. 4 for an illustration of modifiers. As with the class diagram, the incoming arrow represents calls to methods of the modifier, whereas the outgoing arrows stand for self and super-calls of the modifier.

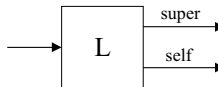


Fig. 4. Illustration of modifiers

Under the condition that the signatures of modifier methods match those of the base class, the modifier can be applied to an arbitrary base class. The modifier does not access the state of the base class directly, but only by making super-calls. As the state space of the base class is unknown until modifier application, we say that the methods of the modifier L operate on the state space $\alpha \times \Delta$, where α is a type variable to be instantiated with the type of the internal state of the base class while modifier application, and Δ is a type of the state component representing all parameters of all methods of the modifier. Hence, the type of L is as follows:

$$\begin{aligned} & ((\alpha \times \Delta \mapsto \alpha \times \Delta) \times \dots \times (\alpha \times \Delta \mapsto \alpha \times \Delta)) \rightarrow \\ & ((\alpha \times \Delta \mapsto \alpha \times \Delta) \times \dots \times (\alpha \times \Delta \mapsto \alpha \times \Delta)) \rightarrow \\ & ((\alpha \times \Delta \mapsto \alpha \times \Delta) \times \dots \times (\alpha \times \Delta \mapsto \alpha \times \Delta)) \end{aligned}$$

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

$$\begin{aligned} L \mathbf{mod} C & \hat{=} (c_0, \lambda self \cdot lm \ self \ (cm \ self)) \\ L \mathbf{upcalls} C & \hat{=} (c_0, \lambda self \cdot lm \ self \ (\mu \ cm)) \end{aligned}$$

In both cases the modifier L is said to be applied to the base class C . See Fig. 5 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.

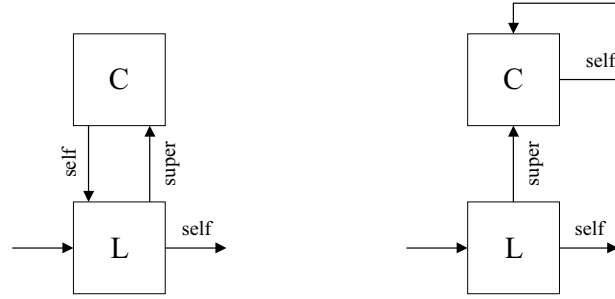


Fig. 5. Illustration of operators **mod** and **upcalls**

Application of the modifier L instantiates its type variable α with the type Σ of the base class C . Hence, the classes created by the application of L to C have methods operating on the state space $\Sigma \times \Delta$.

4.3 Refinement on Classes

Before 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 rela-

tion between the state spaces of these statements. We define data refinement between S and S' as in [8]:

$$\begin{aligned} S \sqsubseteq_R S' &\hat{=} S \sqsubseteq S' \uparrow R \text{ or, equivalently,} \\ S \sqsubseteq_R S' &\hat{=} S \downarrow R \sqsubseteq S' \end{aligned}$$

We can express class refinement in terms of refinement on abstract data types [14, 4, 12]. 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 $(\Sigma \times \Delta) \mapsto (\Sigma \times \Delta)$, where Δ is the state space component representing all parameters of all procedures. 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 $R : \Sigma' \leftrightarrow \Sigma$ if initialization establishes R and all procedures preserve R :

$$T \sqsubseteq_R T' \hat{=} R t'_0 t_0 \wedge tp \sqsubseteq_{R \times Id} tp'$$

By convention, when R is equal to Id , the subscript on the refinement relation is omitted.

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

$$C \sqsubseteq_R C' \hat{=} \mu C \sqsubseteq_R \mu C'.$$

Class refinement ensures that all objects of the refining class are safely substitutable for 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 has the form

$$\begin{aligned} &\text{if } C \text{ is refined by } (L \text{ mod } C) \text{ and } C \text{ is refined by } D \\ &\text{then } C \text{ is refined by } (L \text{ mod } D), \end{aligned}$$

where C is a base class, L a modifier and D a revision of the class C . As was illustrated in the previous sections, this property does not hold in general. In the following sections we formulate, prove, and explain a flexibility theorem, which provides for safe substitutability of C with $(L \text{ mod } D)$ by strengthening the premises of the flexibility property, following the requirements formulated in Sec. 3.6.

5.1 Formulating and Proving Flexibility Theorem

Consider classes C , D and a modifier L , such that $C = (c_0, cm)$ with $cm = \lambda self \cdot (c_1, \dots, c_n)$, $D = (d_0, dm)$ with $dm = \lambda self \cdot (d_1, \dots, d_n)$, and $L = \lambda self \cdot \lambda super \cdot (l_1, \dots, l_n)$. As we have assumed that the classes and the modifier do not have recursive and mutually recursive methods, it is always possible to rearrange their methods in linear order so that each method calls only previous ones. We assign an index to every method with respect to this order. We assume that the corresponding methods in C , D and L receive the same index. Without loss of generality, we can consider the case when for every distinct index there is only one method. We represent methods by functions of the called methods:

$$\begin{aligned} C_1 &= \lambda() \cdot c_1, & \dots & C_n = \lambda(x_1, \dots, x_{n-1}) \cdot c_n \\ D_1 &= \lambda() \cdot d_1, & \dots & D_n = \lambda(x_1, \dots, x_{n-1}) \cdot d_n \\ L_1 &= \lambda() \cdot \lambda(y_1) \cdot l_1, & \dots & L_n = \lambda(x_1, \dots, x_{n-1}) \cdot \lambda(y_1, \dots, y_n) \cdot l_n \end{aligned}$$

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

$$cm = \lambda self \cdot (C_1 (), C_2 (x_1), \dots, C_n (x_1, \dots, x_{n-1}))$$

Let $R : \Sigma' \leftrightarrow \Sigma$ be a relation coercing the state space component Σ' of the revision D to the component Σ of the base class C . Now we can formulate the *flexibility theorem*:

Flexibility Theorem *Let C , D be the classes, L the modifier, and R the state coercing relation as defined above. Then the following holds:*

$$\begin{aligned} \mu C \sqsubseteq_R (d_0, dm ((\mu cm) \downarrow (R \times Id))) \wedge \mu C \sqsubseteq \mu (L \mathbf{upcalls} C) \Rightarrow \\ \mu C \sqsubseteq_R (L \mathbf{mod} D) \end{aligned}$$

Proof In accordance with the definition of abstract data type refinement, we first need to show that $R d_0 c_0 \Rightarrow R d_0 c_0$, which is trivially true.

Next we need to show the following goal:

$$\begin{aligned} snd(\mu C) \sqsubseteq_{R \times Id} dm ((\mu cm) \downarrow (R \times Id)) \wedge snd(\mu C) \sqsubseteq snd(\mu (L \mathbf{upcalls} C)) \Rightarrow \\ snd(\mu C) \sqsubseteq_{R \times Id} snd(\mu (L \mathbf{mod} D)) \end{aligned}$$

Note that in this goal data refinement relations connect tuples of predicate transformers, which correspond to the method bodies with all self and super-calls resolved with the methods of the same class. Thus, we can rewrite this goal as

$$\begin{aligned} (\mathcal{C}_1, \dots, \mathcal{C}_n) \sqsubseteq_{R \times Id} (\mathcal{D}_1, \dots, \mathcal{D}_n) \wedge (\mathcal{C}_1, \dots, \mathcal{C}_n) \sqsubseteq (\mathcal{L}_1, \dots, \mathcal{L}_n) \Rightarrow \\ (\mathcal{C}_1, \dots, \mathcal{C}_n) \sqsubseteq_{R \times Id} (\mathcal{M}_1, \dots, \mathcal{M}_n), \end{aligned}$$

where \mathcal{C} , \mathcal{D} , \mathcal{L} , and \mathcal{M} are defined as follows:

$$\begin{aligned} \mathcal{C}_1 &= C_1 (), & \dots & \mathcal{C}_n = C_n (\mathcal{C}_1, \dots, \mathcal{C}_{n-1}) \\ \mathcal{D}_1 &= D_1 () \downarrow (R \times Id), & \dots & \mathcal{D}_n = D_n (\mathcal{C}_1, \dots, \mathcal{C}_{n-1}) \downarrow (R \times Id) \\ \mathcal{L}_1 &= L_1 () \mathcal{C}_1, & \dots & \mathcal{L}_n = L_n (\mathcal{L}_1, \dots, \mathcal{L}_{n-1}) (\mathcal{C}_1, \dots, \mathcal{C}_n) \\ \mathcal{M}_1 &= L_1 () \mathcal{T}_1, & \dots & \mathcal{M}_n = L_n (\mathcal{M}_1, \dots, \mathcal{M}_{n-1}) (\mathcal{T}_1, \dots, \mathcal{T}_n) \\ \mathcal{T}_1 &= D_1 (), & \dots & \mathcal{T}_n = D_n (\mathcal{M}_1, \dots, \mathcal{M}_{n-1}) \end{aligned}$$

Here \mathcal{C} are the method bodies of the methods C_1, \dots, C_n with all self-calls recursively resolved with \mathcal{C} . The statements \mathcal{D} represent $dm ((\mu cm) \downarrow (R \times Id))$, where each \mathcal{D}_i is a method body of the method D_i with all self-calls resolved with properly coerced \mathcal{C} . The statements \mathcal{L} represent the least fixed point of (L **upcalls** C) methods. Note how \mathcal{M} and \mathcal{T} jointly represent the least fixed point of (L **mod** D) methods. The statements \mathcal{M} stands for the methods of the modifier L with calls to *self* resolved with \mathcal{M} themselves and calls to *super* resolved with \mathcal{T} . On the other hand, the statements \mathcal{T} represent methods of the revision D with calls to *self* resolved with \mathcal{M} .

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

$$(\mathcal{C}_1, \dots, \mathcal{C}_n) \sqsubseteq_{R \times Id} (\mathcal{D}_1, \dots, \mathcal{D}_n) \wedge (\mathcal{C}_1, \dots, \mathcal{C}_n) \sqsubseteq (\mathcal{L}_1, \dots, \mathcal{L}_n) \Rightarrow (\mathcal{L}_1, \dots, \mathcal{L}_n) \sqsubseteq_{R \times Id} (\mathcal{M}_1, \dots, \mathcal{M}_n) \wedge (\mathcal{D}_1, \dots, \mathcal{D}_n) \sqsubseteq (\mathcal{T}_1, \dots, \mathcal{T}_n)$$

from which the target goal follows by transitivity. For proving this stronger goal we need the *wrapping theorem*

$$L_i (\mathcal{M}_1, \dots, \mathcal{M}_{i-1}) \uparrow (R \times Id) (\mathcal{D}_1, \dots, \mathcal{D}_i) \uparrow (R \times Id) \sqsubseteq (L_i (\mathcal{M}_1, \dots, \mathcal{M}_{i-1}) (\mathcal{D}_1, \dots, \mathcal{D}_i)) \uparrow (R \times Id)$$

proved in Appendix.

We prove the goal by induction on the index of methods. Here we only present the inductive step of the proof, because the base step can be proved similarly. The inductive assumption for the inductive case 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:

$$(\mathcal{C}_1, \dots, \mathcal{C}_{n+1}) \sqsubseteq_{R \times Id} (\mathcal{D}_1, \dots, \mathcal{D}_{n+1}) \wedge (\mathcal{C}_1, \dots, \mathcal{C}_{n+1}) \sqsubseteq (\mathcal{L}_1, \dots, \mathcal{L}_{n+1}) \wedge (\mathcal{L}_1, \dots, \mathcal{L}_n) \sqsubseteq_{R \times Id} (\mathcal{M}_1, \dots, \mathcal{M}_n) \wedge (\mathcal{D}_1, \dots, \mathcal{D}_n) \sqsubseteq (\mathcal{T}_1, \dots, \mathcal{T}_n) \Rightarrow \mathcal{L}_{n+1} \sqsubseteq_{R \times Id} \mathcal{M}_{n+1} \wedge \mathcal{D}_{n+1} \sqsubseteq \mathcal{T}_{n+1}$$

We prove this goal as follows:

$$\begin{aligned} & \mathcal{L}_{n+1} \sqsubseteq_{R \times Id} \mathcal{M}_{n+1} \wedge \mathcal{D}_{n+1} \sqsubseteq \mathcal{T}_{n+1} \\ = & \{ \text{definitions, definition of data refinement} \} \\ & L_{n+1} (\mathcal{L}_1, \dots, \mathcal{L}_n) (\mathcal{C}_1, \dots, \mathcal{C}_{n+1}) \sqsubseteq \\ & (L_{n+1} (\mathcal{M}_1, \dots, \mathcal{M}_n) (\mathcal{T}_1, \dots, \mathcal{T}_{n+1})) \uparrow (R \times Id) \wedge \mathcal{D}_{n+1} \sqsubseteq \mathcal{T}_{n+1} \\ \Leftarrow & \left\{ \begin{array}{l} \text{monotonicity of } L_{n+1}, \text{ assumption } (\mathcal{L}_1, \dots, \mathcal{L}_n) \sqsubseteq (\mathcal{M}_1, \dots, \mathcal{M}_n) \uparrow (R \times Id), \\ \text{assumption } (\mathcal{C}_1, \dots, \mathcal{C}_{n+1}) \sqsubseteq (\mathcal{D}_1, \dots, \mathcal{D}_{n+1}) \uparrow (R \times Id), \text{ wrapping theorem} \end{array} \right\} \\ & L_{n+1} (\mathcal{M}_1, \dots, \mathcal{M}_n) \uparrow (R \times Id) (\mathcal{D}_1, \dots, \mathcal{D}_{n+1}) \uparrow (R \times Id) \sqsubseteq \\ & L_{n+1} (\mathcal{M}_1, \dots, \mathcal{M}_n) \uparrow (R \times Id) (\mathcal{T}_1, \dots, \mathcal{T}_{n+1}) \uparrow (R \times Id) \wedge \mathcal{D}_{n+1} \sqsubseteq \mathcal{T}_{n+1} \\ \Leftarrow & \{ \text{monotonicity of } L_{n+1} \} \\ & (\mathcal{D}_1, \dots, \mathcal{D}_{n+1}) \uparrow (R \times Id) \sqsubseteq (\mathcal{T}_1, \dots, \mathcal{T}_{n+1}) \uparrow (R \times Id) \wedge \mathcal{D}_{n+1} \sqsubseteq \mathcal{T}_{n+1} \end{aligned}$$

$$\begin{aligned}
&\Leftarrow \{ \textit{monotonicity of wrappers, assumption } (D_1, \dots, D_n) \sqsubseteq (\mathcal{T}_1, \dots, \mathcal{T}_n) \} \\
&\quad D_{n+1} \sqsubseteq \mathcal{T}_{n+1} \\
&= \{ \textit{definitions} \} \\
&\quad D_{n+1} (\mathcal{C}_1, \dots, \mathcal{C}_n) \downarrow (R \times Id) \sqsubseteq D_{n+1} (\mathcal{M}_1, \dots, \mathcal{M}_n) \\
&\Leftarrow \left\{ \begin{array}{l} \textit{monotonicity of } D_{n+1}, \textit{ assumption } (\mathcal{C}_1, \dots, \mathcal{C}_{n+1}) \sqsubseteq (\mathcal{L}_1, \dots, \mathcal{L}_{n+1}) \textit{ then} \\ \textit{ assumption } (\mathcal{L}_1, \dots, \mathcal{L}_n) \sqsubseteq (\mathcal{M}_1, \dots, \mathcal{M}_n) \uparrow (R \times Id) \end{array} \right\} \\
&\quad D_{n+1} (\mathcal{M}_1, \dots, \mathcal{M}_{n+1}) \uparrow (R \times Id) \downarrow (R \times Id) \sqsubseteq D_{n+1} (\mathcal{M}_1, \dots, \mathcal{M}_n) \\
&\Leftarrow \{ \textit{wrapping rule} \} \\
&\quad D_{n+1} (\mathcal{M}_1, \dots, \mathcal{M}_n) \sqsubseteq D_{n+1} (\mathcal{M}_1, \dots, \mathcal{M}_n) \\
&= T \\
&\square
\end{aligned}$$

5.2 Interpretation and Implications of Flexibility Theorem

Let us consider how the requirements of Sec. 3.6 are reflected in the flexibility theorem and what are the consequences of this theorem.

In our formalization the “no cycles” requirement is handled by the fact that we have rearranged methods in the participating classes and the modifier in linear order, and assigned the same index to the corresponding methods. Thus mutual recursion of methods cannot appear. In practice the unanticipated mutual recursion can be avoided if methods of a revision class and a modifier do not introduce self-calls to the methods which are not self-called in the corresponding methods of the base class.

The “no revision self-calling assumptions” requirement states that, while reasoning about the behavior of a revision class method, we should not assume the behavior of the methods it self-calls, but should consider instead the behavior described by the base class. This requirement together with the first assumption of the flexibility property is formalized as the first conjunct in the antecedent of the flexibility theorem, $\mu C \sqsubseteq_R (d_0, dm ((\mu cm) \downarrow (R \times Id)))$. As we have explained above, the application of dm to the properly coerced (μcm) returns a tuple of methods of D with all self-calls redirected to methods of C .

The “no base class down-calling assumptions” requirement states that, while reasoning about the behavior of modifier methods, we should not assume the behavior of those modifier methods to which self-calls of the base class can get redirected. Instead we should consider the corresponding methods of the base class. This requirement along with the second assumption of the flexibility property is handled by the second conjunct in the antecedent of the flexibility theorem, $\mu C \sqsubseteq \mu (L \textbf{upcalls } C)$. As we have explained earlier, if a modifier is applied to the base class using the **upcalls** operator, self-calls of the base class remain within the base class itself.

The “no direct access to the base class state” requirement is addressed by our formalization of modifiers. As we have stated above, methods of a modifier

always skip on the α component of the state space, thus excluding the possibility of direct access to the base class state.

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 applies to the first scenario as well.

6 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 [32, 15], since it has critical significance for component systems. Modification of 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 [15]. 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. This aspect of the problem was recognized by COM developers [32] and led them to the decision to abandon inheritance in favor of forwarding as the reuse mechanism. Although solving the problem, this approach comes at the cost of reduced flexibility. On the other hand, inheritance proved to be a convenient mechanism for code reuse. This consideration brought us to the following question: “How can we discipline code inheritance to avoid the fragile base class problem, but still retain a significant degree of flexibility in code reuse?”

We have encountered several research directions which are related to ours. The first direction combines research on semantics of object oriented languages with ensuring substitutability of objects. Hense in [13] gives a 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 [10]. However, a number of notions

crucial for our purposes, such as, e.g. refinement ordering on classes, are not defined. Various approaches to substitutability of objects are presented in [1, 17, 18]. However, they do not model self-calls in classes in presence of inheritance and dynamic binding.

The second direction of related research is oriented towards developing a methodology for specifying classes to make code reuse less error prone. The key idea of extending a class specification with specification of its *specialization interface* was presented first by Kiczales and Lamping in [16]. This idea was further developed by Steyaert et al. in [26]. In fact the second paper considers the fragile base class problem in our formulation (although they do not refer to it by this name). 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.

Our first example is, in fact, adopted from [26]. 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 $CountingBag$ relies on the invariant $n = |b|$ binding values of the instance variable n with the number of elements in the inherited instance variable b . This invariant is violated when Bag is substituted with Bag' and, therefore, the *cardinality* method of the resulting class returns the incorrect value. Apparently, this problem is just an instance of the “unjustified assumption of the binding invariant in modifier” problem presented in Sec. 3.5.

As a potential solution to this problem one can specify methods *add* and *addAll* as *consistent methods* as was first suggested in [16]. This means that the extension developers would be disallowed to override one without overriding the other. However, recommendations of Kiczales and Lamping are based only on empirical expertise, thus it is not clear whether they apply in the general case. We believe that such methodology should be grounded on a mathematical basis, and developing such methodology constitutes the ultimate goal of our research.

Stata and Guttag in [24, 25] elaborate the idea of specialization interfaces by providing a mathematical foundation in behavioral subtyping style [1, 17]. They introduce *class components*, which combine a substate of a class and a set of methods responsible for maintaining that state, as units of modularity for the specialization interface. Such class components are overridable, i.e. an extension class may provide its own implementation of an entire class component. Even though they do not consider substitution of a base class with a revision in presence of extensions, many of their findings are related to ours. However,

they specify methods in terms of pre and post conditions which in our opinion is not expressive enough to capture the intricacies of the example in Sec. 3.5. Furthermore, Stata in [25] states that class components are independent if “the instance variables of a component may be accessed only by the methods in the component”, and one component “depends only on the specification of the other component, not on its implementation”. It is possible to construct an example, similar to the one presented in Sec. 3.5, which has two class components independent according to the independence requirement above, yet overriding in a subclass one of these components, following its specification, leads to a crash.

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 “direct access to base class state” problem was known from the literature. The other problems were noticed 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.

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

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

is stronger than the flexibility property. If inheritance were monotonic, not only the fragile base class problem would be avoided, but also the extension $(M \text{ mod } C)$ would be fully substitutable with the new extension $(M \text{ mod } C')$, 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, Joakim von Wright, Anna Mikhajlova, Jim Grundy, Wolfgang Weck, Martin Büechi, and Linas Laibinis for a lot of valuable comments.

References

1. P. America. Designing an object-oriented programming language with behavioral 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 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.
6. R. J. R. Back, J. von Wright. Refinement Calculus I: Sequential Nondeterministic Programs. In *Stepwise Refinement of Distributed Systems*, pp. 42–66, Springer–Verlag, 1990.
7. R. J. R. Back, J. von Wright. Predicate Transformers and Higher Order Logic. In *REX Workshop on Semantics: Foundations and Applications*, the Netherlands, 1992.
8. R.J.R. Back, J. von Wright. Refinement Calculus, A Systematic Introduction. *Springer–Verlag*, April 1998.
9. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.
10. W. Cook, J. Palsberg. A denotational semantics of inheritance and its correctness. *OOPSLA '89 Proceedings*, volume 24 of SIGPLAN Notices, pp. 433–443, October 1989.
11. E.W.Dijkstra, A discipline of programming. *Prentice–Hall International*, 1976.
12. P.H.B. Gardiner and C.C. Morgan. Data refinement of predicate transformers. *Theoretical Computer science*, 87:143–162, 1991.
13. A. 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.
14. C.A.R. Hoare. Proofs of correctness of data representation. *Acta informatica*, 1(4), 1972.
15. IBM Corporation. IBM's System Object Model (SOM): Making reuse a reality. *IBM Corporation, Object Technology Products Group*, Austin, Texas.
16. G. Kiczales, J. Lamping. Issues in the design and specification of class libraries. *OOPSLA '92 Proceedings*, volume 27 of SIGPLAN Notices, pp. 435–451, October 1992.
17. B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

18. A. Mikhajlova, E. Sekerinski. Class refinement and interface refinement in object-oriented programs. *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of Lecture Notes in Computer Science, Springer-Verlag, 1997.
19. C. C.Morgan. Programming from specifications. *Prentice-Hall*, 1990.
20. J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9, 287–306, 1987.
21. C. Pfister, C. Szyperski. Oberon/F framework. Tutorial and reference. *Oberon microsystems, Inc.*, 1994.
22. D. Pountain, C. Szyperski. Extensible software systems. *Byte Magazine*, 19(5): 57–62, May 1994. <http://www.byte.com/art/9405/sec6/art1.html>.
23. A. Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. *OOPSLA '86 Proceedings*, volume 21 of SIGPLAN Notices, pp. 38–45, 1986.
24. R. Stata, J. V. Guttag. Modular reasoning in the presence of subtyping. *OOPSLA '95 Proceedings*, volume 30 of SIGPLAN Notices, pp. 200–214, 1995.
25. R. Stata. Modularity in the presence of subclassing. *SRC Research Report 145*, Digital Systems Research Center, 1997.
26. P. 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.
27. B. Stroustrup. The C++ programming language. *Addison-Wesley*, 1986.
28. C. Szyperski. Independently extensible systems. Software engineering potential and challenges. *Proceedings of the 19th Australasian Computer Science Conference* Melbourne, Australia, February 1996.
29. D. Taenzer, M. Gandi, S. Podar. Problems in object-oriented software reuse. *Proceedings ECOOP'89*, S. Cook (Ed.), Cambridge University Press Nottingham, July 10-14, 1989, pp. 25–38.
30. A. Tarski. A Lattice Theoretical Fixed Point Theorem and its Applications. volume 5 of *Pacific J. Mathematics*, pp. 285–309, 1955.
31. P. Wegner, S. 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.
32. S. Williams and C. Kinde. The Component Object Model: Technical Overview. *Dr. Dobbs Journal*, December 1994.

Appendix

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, \dots, m_n can be rewritten as

$$\mathbf{begin\ var\ } l \mid p \cdot \mathbf{do\ } g_1 \rightarrow m_1; S_1 \ [] \dots \ [] g_n \rightarrow m_n; S_n \mathbf{od\ end}$$

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

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

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 $S_1 \ \sqcap \dots \ \sqcap S_n$.

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

$$\begin{aligned}
m_i = & \mathbf{begin\ var\ } l \mid p \cdot \\
& \mathbf{while\ } g_1 \vee \dots \vee g_{i-1} \vee g'_1 \vee \dots \vee g'_i \mathbf{ do} \\
& \quad [g_j]; x_j; S_j \mathbf{ for\ } j = 1..i - 1 \sqcap \\
& \quad [g'_j]; y_j; S'_j \mathbf{ for\ } j = 1..i \\
& \mathbf{ od} \\
& \mathbf{ end}
\end{aligned}$$

for some local variables 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 [8]:

$$\begin{aligned}
& \text{; rule : } (S_1; S_2) \downarrow R \sqsubseteq (S_1 \downarrow R); (S_2 \downarrow R) \\
& \sqcap \text{ rule : } (S_1 \sqcap S_2) \downarrow R \sqsubseteq (S_1 \downarrow R \sqcap S_2 \downarrow R) \\
& \text{while rule : } \mathbf{while\ } g \mathbf{ do\ } S \mathbf{ od} \downarrow R \sqsubseteq \mathbf{while\ } \{R\} g \mathbf{ do\ } [[R] g]; (S \downarrow R) \mathbf{ od}
\end{aligned}$$

We define an *indifferent statement* [8] 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:

indifference rule : $S \downarrow R \sqsubseteq S$, 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:

indifferent while rule : $\mathbf{while\ } g \mathbf{ do\ } S \mathbf{ od} \downarrow R \sqsubseteq \mathbf{while\ } g \mathbf{ do\ } S \downarrow R \mathbf{ od}$

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

$$\begin{aligned}
L_i (\mathcal{M}_1, \dots, \mathcal{M}_{i-1}) \uparrow (R \times Id) (\mathcal{T}_1, \dots, \mathcal{T}_i) \uparrow (R \times Id) \sqsubseteq \\
(L_i (\mathcal{M}_1, \dots, \mathcal{M}_{i-1}) (\mathcal{T}_1, \dots, \mathcal{T}_i)) \uparrow (R \times Id)
\end{aligned}$$

Proof If we denote $(R \times Id)$ by P , we can rewrite our goal as was described above:

$$\left(\begin{array}{l} \mathbf{begin\ var\ } l \mid p \\ \mathbf{while\ } g_1 \vee \dots \vee g_{i-1} \\ \quad \vee g'_1 \vee \dots \vee g'_i \mathbf{ do} \\ \quad [g_j]; \mathcal{M}_j \uparrow P; S_j \mathbf{ for\ } j = 1..i - 1 \sqcap \\ \quad [g'_j]; \mathcal{T}_j \uparrow P; S'_j \mathbf{ for\ } j = 1..i \\ \mathbf{ od} \\ \mathbf{end} \downarrow P \end{array} \right) \sqsubseteq \left(\begin{array}{l} \mathbf{begin\ var\ } l \mid p \\ \mathbf{while\ } g_1 \vee \dots \vee g_{i-1} \\ \quad \vee g'_1 \vee \dots \vee g'_i \mathbf{ do} \\ \quad [g_j]; \mathcal{M}_j; S_j \mathbf{ for\ } j = 1..i - 1 \sqcap \\ \quad [g'_j]; \mathcal{T}_j; S'_j \mathbf{ for\ } j = 1..i \\ \mathbf{ od} \\ \mathbf{end} \end{array} \right)$$

Consider the typing of the participating constructs. If the type of the local variable l is Δ , then state predicates p , g_j , g'_j and statements S_j are working on the state space $\Delta \times \Sigma \times \Delta$, but they skip on the state component of type Σ . Thus, 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{aligned}
& \text{begin var } l \mid p \\
& \quad \text{while } g_1 \vee \dots \vee g_{i-1} \vee g'_1 \vee \dots \vee g'_i \text{ do} \\
& \quad \quad [g_j]; \mathcal{M}_j \uparrow P; S_j \text{ for } j = 1..i - 1 \sqcap \\
& \quad \quad [g'_j]; \mathcal{T}_j \uparrow P; S'_j \text{ for } j = 1..i \\
& \quad \text{od} \\
& \text{end } \downarrow P \\
= & \{ \text{definition of block} \} \\
& \text{(enter } l \mid p; \\
& \quad \text{while } g_1 \vee \dots \vee g_{i-1} \vee g'_1 \vee \dots \vee g'_i \text{ do} \\
& \quad \quad [g_j]; \mathcal{M}_j \uparrow P; S_j \text{ for } j = 1..i - 1 \sqcap \\
& \quad \quad [g'_j]; \mathcal{T}_j \uparrow P; S'_j \text{ for } j = 1..i \\
& \quad \text{od;} \\
& \text{exit } l) \downarrow P \\
\sqsubseteq & \{ ; \text{ rule} \} \\
& \text{(enter } l \mid p) \downarrow P; \\
& \quad \text{while } g_1 \vee \dots \vee g_{i-1} \vee g'_1 \vee \dots \vee g'_i \text{ do} \\
& \quad \quad [g_j]; \mathcal{M}_j \uparrow P; S_j \text{ for } j = 1..i - 1 \sqcap \\
& \quad \quad [g'_j]; \mathcal{T}_j \uparrow P; S'_j \text{ for } j = 1..i \\
& \quad \text{od } \downarrow P; \\
& \text{(exit } l) \downarrow P \\
\sqsubseteq & \{ \text{enter and exit are conjunctive and indifferent to } P, \text{ thus} \\
& \quad \text{indifference rule, indifferent while rule} \} \\
& \text{enter } l \mid p; \\
& \quad \text{while } g_1 \vee \dots \vee g_{i-1} \vee g'_1 \vee \dots \vee g'_i \text{ do} \\
& \quad \quad ([g_j]; \mathcal{M}_j \uparrow P; S_j \text{ for } j = 1..i - 1 \sqcap \\
& \quad \quad [g'_j]; \mathcal{T}_j \uparrow P; S'_j \text{ for } j = 1..i) \downarrow P \\
& \quad \text{od;} \\
& \text{exit } l \\
\sqsubseteq & \{ \sqcap \text{ rule, } ; \text{ rule} \} \\
& \text{enter } l \mid p; \\
& \quad \text{while } g_1 \vee \dots \vee g_{i-1} \vee g'_1 \vee \dots \vee g'_i \text{ do} \\
& \quad \quad [g_j] \downarrow P; \mathcal{M}_j \uparrow P \downarrow P; S_j \downarrow P \text{ for } j = 1..i - 1 \sqcap \\
& \quad \quad [g'_j] \downarrow P; \mathcal{T}_j \uparrow P \downarrow P; S'_j \downarrow P \text{ for } j = 1..i \\
& \quad \text{od;} \\
& \text{exit } l
\end{aligned}$$

\sqsubseteq $\{[g_j]$ are indifferent to P , and S_j are conjunctive,
 thus indifference rule; wrapping rule $\}$
enter $l \mid p$;
while $g_1 \vee \dots \vee g_{i-1} \vee g'_1 \vee \dots \vee g'_i$ **do**
 $[g_j]; \mathcal{M}_j; S_j$ **for** $j = 1..i - 1 \sqcap$
 $[g'_j]; \mathcal{T}_j; S'_j$ **for** $j = 1..i$
od;
exit l
 $=$ $\{$ definition of block $\}$
begin var $l \mid p$.
 while $g_1 \vee \dots \vee g_{i-1} \vee g'_1 \vee \dots \vee g'_i$ **do**
 $[g_j]; \mathcal{M}_j; S_j$ **for** $j = 1..i - 1 \sqcap$
 $[g'_j]; \mathcal{T}_j; S'_j$ **for** $j = 1..i$
 od
end

□