Software Reuse Mechanisms and Techniques: Safety Versus Flexibility

by

Leonid Mikhajlov

Turku Centre for Computer Science

TUCS Dissertations

No 21, December 1999

Software Reuse Mechanisms and Techniques: Safety Versus Flexibility

Leonid Mikhajlov

To be presented – with the permission of the Faculty of Mathematics and Natural Sciences at Åbo Akademi University – for public criticism, in Auditorium 3102 at the Department of Computer Science at Åbo Akademi University, on December 17th, 1999, at 14.00.

Department of Computer Science
Åbo Akademi University

ISBN 951-29-1581-2
ISSN 1239-1883

Painosalama Oy

To my parents

Я сразу смазал карту будня,
плеснувши краску из стакана;
я показал на блюде студня
косые скулы океана.
На чешуе жестяной рыбы
прочел я зовы новых губ.
А вы
ноктюрн сыграть
могли бы
на флейте водосточных труб?

Владимир Маяковский

Acknowledgements

I would like to express my gratitude to my supervisors Professor Ralph-Johan Back and Professor Joakim von Wright for setting one of the highest standards of quality of research and results presentation that exist in this scientific field. Learning how to match these standards was challenging but beneficial in the long run. To a great extent, it is the strength and clarity of their theory, which is used as a theoretical basis in this dissertation, that makes this thesis as interesting as it is. I would also like to thank them for giving me the freedom to study what I wanted and waiting patiently for the results.

Professor Eric Hehner of the University of Toronto, Canada, and Dr. David Naumann of Stevens Institute of Technology, USA, kindly agreed to review this dissertation and proposed several improvements. Their comments and suggestions are gratefully acknowledged.

Special thanks are due to Dr. Emil Sekerinski (currently at McMaster University, Canada) who has introduced me to the field of formal studies of problems in object-oriented programming. Some of the most important results presented in this dissertation were developed in cooperation with Emil. Apart from scientific help, Emil's friendly encouraging advise and kind concern helped me to cope with the feeling of "academic frustration". For this I am much obliged.

I would like to thank the board of Turku Centre for Computer Science for accepting me to study at TUCS and granting me generous financial support during the period of my PhD studies. Not only did I get the chance to extend my knowledge of computer science, but also an opportunity to get immersed into a different culture, which, I believe, was beneficial for me. I am thankful to my colleagues in the department and fellow students at TUCS for practical and scientific help. With Linas Laibinis, who co-authored one of the papers used in this dissertation, I had particularly inspiring and illuminating discussions.

I would like to thank all professors and lecturers at Taganrog State University of Radio-Engineering, where I studied for my Engineer's Diploma, who contributed to my scientific development. Especially, I am grateful to D.P. Kalachev for getting me interested in software engineering. I also would like to thank the rector of the University, professor V.G. Zakharevich, and professor A.N. Melikhov, for giving me recommendations for applying to TUCS.

Last but not least, I would like to thank my friends and family. Especially, Tamara Ivanovna for her love and emotional support. My grandmother Sosja Ihilevna for creating the “need to study” script in my family. My father Leonid Samojlovich for showing me what it means and takes to be a scientist and my mother Aleksandra Aleksejevna for letting me feel that, regardless of anything, I am loved and needed. To Anna, I am grateful for her help, encouragement, enthusiasm, infinite supply of excellent ideas, and the never-ending striving for the very best.

Leonid Mikhajlov
November, 1999

Contents

Chapter 1

Introduction

It is widely recognized that we are experiencing a so-called software crisis. While computer hardware is getting cheaper and more powerful, software remains almost as expensive as it was 30 years ago [46]. In part this can be attributed to the fact that software is reinvented almost every time when something new is needed. To resolve the crisis, it is necessary to foster the reusability of software with the ultimate goal of creating a market for software components composable by end users analogous to the market of computer hardware components.

The oldest software reuse technique of all is the source code Copy&Paste technique. It is inappropriate for numerous reasons, such as space, efficiency, and legacy. To alleviate at least some of these problems, a technique based on the notorious **goto** statement was used. Unfortunately, this led to an absolutely unmanageable and incomprehensible spaghetti-like code. The need for better reusability initiated research in the programming language community on development of software reuse mechanisms and techniques, which are language-specific constructs and their applications intended to facilitate software reuse.

The work on new software reuse mechanisms and techniques is still very active. Practically every new programming language comes with new mechanisms and may introduce new techniques. Sometimes a technique or a mechanism can gain overwhelming popularity, while a systematic study of its properties may not be concluded yet or has not even been attempted. The lack of solid foundations can lead to the incorrect use of a particular mechanism or application of a particular technique in an inappropriate context. The effect of these kinds of errors can be very difficult to assess as it can be obscure. For example, the application of certain code reuse mechanisms may hamper the ability of developers to maintain the system.

In this dissertation, we study the properties of a number of popular software reuse mechanisms and techniques. Of all the possible properties of these mechanisms and techniques, in this work we concentrate on two of key importance: safety and flexibility. We say that a code reuse mechanism or technique is safe if there exists a modular reasoning method associated with it. To verify that a client program reusing a particular predefined piece of code (packaged with a specific code reuse mechanism) is correct, it should be sufficient to establish that the code candidate for reuse correctly implements the functionality expected by the client. In this case, we say that a code reuse mechanism permits modular reasoning. We provide an extensive discussion of the importance of the safety of reuse mechanisms and techniques in practical programming.

While for some of the considered mechanisms, modular reasoning is well understood, for others it remains a research topic. Often ad-hoc modular reasoning appeals to the developers' intuition, but under careful analysis fails to live up to the promise. In order to be able to analyze reuse techniques and mechanisms rigorously, we develop customized formal models capturing essential features of these mechanisms and techniques, relevant to the discussion of their safety. The models of the reuse mechanisms and techniques are based on the refinement calculus [12, 14, 13]. We formulate a modular reasoning property for each mechanism or technique and analyze whether this property holds. In the cases when the property does not hold, we either formulate requirements restricting the reuse mechanism or identify additional verification obligations for developers. After that, we formally prove that the modular reasoning property augmented with the formulated restrictions and the additional verification conditions holds. Application of formal methods is beneficial for this kind of analysis, as it permits us to meticulously study the artifacts of programming languages and to illuminate the features of the reuse mechanisms and techniques that dramatically influence their applicability. The choice of the refinement calculus in its higher-order logic formalization is not incidental – this logical framework permits us to analyze very different software reuse mechanisms and techniques in a uniform manner.

In general, a piece of software can rarely be reused in a new context without prior adaptation. Thus the flexibility of a code reuse mechanism, i.e., the ability of a developer to adopt a candidate code for reuse in a new context, is a very important property. All software reuse mechanisms can be split into two categories, those that allow for inline adaptation of reused code and those that do not. A reuse mechanism permits inline adaptation of the reused code if developers can customize this code in a manner that was not preplanned by its developers. Code inheritance is one example of a reuse mechanism permitting inline adaptation. In the absence of inline adaptation, a code fragment can only be reused in its entirety. While inline adaptation clearly provides extra flexibility, its virtues are a topic of ongoing discussion [73, 82]. In our opinion, most of the problems associated with inline adaptation can be traced back to the lack of safety in the existing mechanisms supporting such inline adaptation.

With the software reuse mechanisms that do not permit inline adaptation, the degree of reuse that can be achieved depends on the ability of reusable code developers to anticipate the context in which the possibility to reuse the code might appear. In many cases, we can speak about the flexibility of one or another reuse technique, as there can be several techniques based on the same reuse mechanism. The essence of any kind of the code adaptation is in keeping parts of code that satisfy certain requirements, while substituting the irrelevant parts with new ones. Hence the reusability of a reuse technique largely depends on the granularity of reusable code components. In general, making components of smaller size generates more opportunities for reuse. However, this can decrease the efficiency of the resulting system, as communication through interfaces is more computationally expensive than communication through common variables. Thus finding the right balance between the size of code components and the general performance of the resulting systems constitutes the primary challenge for the developers of the reusable code. In this work, we discuss different issues related to flexibility of various mechanisms and techniques that we consider and provide examples illustrating practical application of these mechanisms and techniques facilitated by their flexibility.

1.1 Layout of the Dissertation

We consider the following software reuse mechanisms and techniques: Copy&Paste; functions and procedures; modules; objects and two reuse techniques relying on objects: forwarding and delegation; classes and inheritance. We model the software reuse mechanisms and techniques on a very abstract level, yet we believe that the presented models correctly capture the features of the code reuse techniques essential for discussing their safety. The refinement calculus in the higher-order logic formulation, which we use for modeling reuse mechanisms and techniques, permits modeling these mechanisms and techniques in a uniform manner. In the next section, we present a brief introduction to the refinement calculus which serves as the underlying formalism for the models that we present. We conclude the introductory chapter by considering the most basic technique, Copy&Paste of source code, and argue that monotonicity of constructs in a programming language is critical for the use of any reuse mechanism or technique used in this language.

The main body of the dissertation begins with the analysis of functions and procedures. We present a simplified model of procedures and programs using procedures, the sole intent of which is to introduce a general idea of the models presented in this dissertation. Regardless of its apparent simplicity, the model permits us to show that procedures permit reasoning about the programs using them in a modular manner. We describe the flexibility of procedures and discuss the consequences of procedure application in practical programming. This chapter sets up a general layout which we follow throughout this dissertation.

In chapter 3, we discuss another basic mechanism – modules. Like for procedures, we present a model of modules and of a client program using modules. We formulate a modular reasoning property for modules and prove that it holds. We discuss the flexibility of modules and discuss the applicability of modules in different software systems.

The next two chapters, 4 and 5, are closely related as they discuss two software reuse techniques based on objects. In chapter 4, we study objects and the reuse technique known as forwarding. We argue that objects, unlike modules, rarely operate in isolation. From a mathematical standpoint, an object which is not invoking methods on other objects is no different from a module. We present a model of an interactive object and a model of a forwarding object composition, with which the structure of object references can only be acyclic. Then we formulate a modular reasoning property for forwarding objects and prove that it holds. Next, we discuss the flexibility of objects and the forwarding technique and conclude by considering the applicability of forwarding.

In chapter 5, we consider the reuse technique known as delegation, which permits composing objects in recursive patterns, that is the structure of object references can be cyclic. We develop a fixed point model of a delegating object composition. We formulate a modular reasoning property for delegating objects and argue that its meaning strongly depends on the definition of object refinement. We consider a simple definition of object refinement and show that, while it is definitely safe, it is too inflexible to be used in practice. Next, we consider an ad-hoc definition of refinement that is often used in informal reasoning. We present a component re-entrance problem induced by this definition. Analyzing aspects of the component re-entrance problem, we formulate two requirements that should be taken into account in order for the modular reasoning property to hold. We formulate context object refinement accounting for one of the requirements and reformulate the modular reasoning property in terms of context object refinement and accounting for the second requirement. Then we formally prove that this property holds and present a check-list for informally verifying correctness of a delegating object against its specification. Next, we discuss the flexibility of delegation and show how delegation is used in the implementation of distributed component platforms. We conclude the chapter by reviewing related work and discussing the applicability of delegation to different systems. The content of this chapter is roughly based on a joint work with Emil Sekerinski and Linas Laibinis as presented in [50]. While [50] focuses on mutually dependent components, here we use a similar model to reason about delegating objects. The solution to the component re-entrance problem for delegating objects presented here is also similar to that presented in [50]. The check-list for verifying delegating objects and the discussion of flexibility of delegation are original to this dissertation.

Chapter 6 is devoted to classes and inheritance. We present a model of classes and inheritance and define refinement on classes. Then we discuss the safety of inheritance and present the semantic fragile base class problem which plagues the maintenance of object-oriented systems employing code inheritance as the reuse mechanism. We present a detailed analysis of the semantic fragile base class problem, discuss how code inheritance can be disciplined to become a safe software reuse mechanism, and prove the corresponding modular reasoning property. Next we illustrate the flexibility of code inheritance showing an architectural pattern that inheritance facilitates. Further, we discuss the degree of flexibility attainable with different variants of inheritance such as interface inheritance, code inheritance, and the disciplined inheritance. In the concluding sections, we relate to the work of other researchers and consider the applicability of classes and inheritance for different kinds of software systems, emphasizing a balance between the flexibility and the safety. The semantic fragile base class problem was first studied in a joint work with Emil Sekerinski in [49]. The formal model of classes and inheritance and the analysis of the problem presented here follows those in [49]. However the main result of this chapter – the Modular Reasoning Theorem for Disciplined Inheritance – is original. The other contributions original to this chapter include a proposal for implementing disciplined inheritance as a language mechanism, a check-list for verifying disciplined inheritance in practice, and discussions of safety and flexibility of different inheritance mechanisms and techniques.

In the concluding chapter, we briefly review the contributions of this dissertation, discuss the applicability of various reuse mechanisms and techniques to various systems and outline directions of future research.

1.2 Mathematical Background

The *refinement calculus* is a logical framework for reasoning about correctness and refinement of imperative programs developed by Ralph Back, Caroll Morgan, Joe Morris, Joakim von Wright, and others [6, 54, 55, 85, 9, 26, 56]. In this section, we present the necessary definitions and rules of the refinement calculus. The models of software reuse mechanisms and techniques that we present in the following chapters are developed as extensions of the basic refinement calculus. The material of this section is based on the work by Back and von Wright as presented in [12, 10, 5, 14, 13].

1.2.1 Predicates, Relations, and Predicate Transformers

A program state with n components is modeled by a tuple of values, and a set of states (type) Σ is a product space, $\Sigma = \Sigma_1 \times \ldots \times \Sigma_n$.

A *predicate* over Σ is a boolean function $p : \Sigma \to Bool$ which assigns a truth value to each state. The set of predicates on Σ is denoted $\mathcal{P}\Sigma$. The predicates *true* and *false* over Σ map every $\sigma : \Sigma$ to the boolean values T and F, respectively. The *entailment ordering* on predicates is defined by pointwise extension, so that for $p, q : \mathcal{P}\Sigma$,

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

Conjunction $p \cap q$, disjunction $p \cup q$, and negation $\neg p$ of (similarly-typed) predicates are defined by pointwise extension of the corresponding operations on *Bool*. For $p, q : \mathcal{P}\Sigma$,

$$\begin{array}{rcl} p \cap q & \widehat{=} & (\forall \sigma : \Sigma \bullet p.\sigma \ \wedge \ q.\sigma) \\ p \cup q & \widehat{=} & (\forall \sigma : \Sigma \bullet p.\sigma \ \vee \ q.\sigma) \\ \neg p & \widehat{=} & (\forall \sigma : \Sigma \bullet \neg(p.\sigma)) \end{array}$$

A *relation* from Σ to Γ is a function of type $\Sigma \to \mathcal{P}\Gamma$ that maps each state σ to a predicate on Γ. We write $\Sigma \leftrightarrow \Gamma$ to denote a set of all relations from Σ to Γ. This view of relations is isomorphic to viewing them as predicates on the cartesian space $\Sigma \times \Gamma$. For functions $f : \Sigma \to \Gamma$, $g : \Gamma \to \Delta$, and relations $P : \Sigma \leftrightarrow \Gamma$, $Q : \Gamma \leftrightarrow \Delta$, functional and relational compositions are defined in the usual manner:

$$\begin{array}{rcl} (f \circ g).\sigma & \widehat{=} & g.(f.\sigma) \\ (P;Q).\sigma.\delta & \widehat{=} & \exists \gamma \bullet P.\sigma.\gamma \ \wedge \ Q.\gamma.\delta \end{array}$$

Repeated function application f^n is defined inductively as follows:

$$\begin{array}{rcl} f^0.x & = & x \\ f^{n+1}.x & = & f^n.(f.x) \end{array}$$

A *predicate transformer* is a function $S : \mathcal{P}\Gamma \to \mathcal{P}\Sigma$ from predicates to predicates. We write $\Sigma \mapsto \Gamma$ to denote a set of all predicate transformers from Σ to Γ (i.e., $\mathcal{P}\Gamma \to \mathcal{P}\Sigma$). We write $Ptran(\Sigma)$ for the case when initial and final state spaces are the same. In this dissertation, we often work with tuples of predicate transformers operating on certain state spaces. We write $\Pi^n(Ptran(\Sigma))$ to denote the type of a tuple of n predicate transformers operating on the state space Σ. We also write $\Phi^{m,n}(Ptran(\Sigma))$ to denote the type of functions $\Pi^m(Ptran(\Sigma)) \to \Pi^n(Ptran(\Sigma))$.

The *refinement ordering* on predicate transformers is defined by pointwise extension from predicates. For $S, T : \Sigma \mapsto \Gamma$,

$$S \sqsubseteq T \ \widehat{=} \ (\forall q : \mathcal{P}\Gamma \bullet S.q \subseteq T.q)$$

The refinement ordering on tuple of predicate transformers is defined elementwise:

$$(S_1, ..., S_n) \sqsubseteq (T_1, ..., T_n) \ \widehat{=} \ S_1 \sqsubseteq T_1 \ \wedge \ ... \ \wedge \ S_n \sqsubseteq T_n$$

Product operators combine predicates, relations, and predicate transformers by forming cartesian products of their state spaces. For predicates $p : \mathcal{P}\Sigma$ and $q : \mathcal{P}\Gamma$, their product $p \times q$ is a predicate of type $\mathcal{P}(\Sigma \times \Gamma)$ defined by

$$(p \times q).(\sigma, \gamma) \ \ \widehat{=} \ \ p.\sigma \ \wedge \ q.\gamma$$

A product $f \times g$ of two functions $f : \Sigma_1 \to \Gamma_1$ and $g : \Sigma_2 \to \Gamma_2$ is a function of type $(\Sigma_1 \times \Sigma_2) \to (\Gamma_1 \times \Gamma_2)$ defined by

$$(f \times g).(\sigma_1, \sigma_2) \ \ \widehat{=} \ \ (f.\sigma_1, g.\sigma_2)$$

A product $P \times Q$ of two relations $P : \Sigma_1 \leftrightarrow \Gamma_1$ and $Q : \Sigma_2 \leftrightarrow \Gamma_2$ is a relation of type $(\Sigma_1 \times \Sigma_2) \leftrightarrow (\Gamma_1 \times \Gamma_2)$ defined by

$$(P \times Q).(\sigma_1, \sigma_2).(\gamma_1, \gamma_2) \ \ \widehat{=} \ \ P.\sigma_1.\gamma_1 \ \wedge \ Q.\sigma_2.\gamma_2$$

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:

$$(S_1 \times S_2).q \ \ \widehat{=} \ \ (\cup q_1, q_2 \,|\, q_1 \times q_2 \subseteq q \bullet S_1.q_1 \times S_2.q_2)$$

The product operator is right associative, i.e. $p \times q \times r = p \times (q \times r)$, for predicates p, q, and r. While the product $p \times (q \times r)$ is not always equal to $(p \times q) \times r$, products with components associated in different ways are isomorphic to each other with respect to the properties that we consider in this dissertation. Accordingly, we disregard the associativity of the product operators; explicitly accounting for this associativity would only clutter the presentation.

1.2.2 Statements of Refinement Calculus

The language used in the refinement calculus includes executable statements along with (abstract) specification statements. Every statement has a precise mathematical meaning as a monotonic predicate transformer. A statement with initial state in Σ and final state in Γ determines a monotonic predicate transformer $S : \Sigma \mapsto \Gamma$ that maps any postcondition $q : \mathcal{P}\Gamma$ to the weakest precondition $p : \mathcal{P}\Sigma$ such that the statement is guaranteed to terminate in a final state satisfying q whenever the initial state satisfies p. In the refinement calculus, statements are identified with the monotonic predicate transformers that they determine in this manner.

Predicate transformers form a complete lattice under the refinement ordering. The bottom element is the **abort** statement, which does not guarantee any outcome or termination, therefore, it maps every postcondition to *false*. The top element is the statement **magic** which is *miraculous*, since it is always guaranteed to establish any postcondition. As such, **magic** is the opposite of the abortion and is not considered to be an error. For any predicate $q : \mathcal{P}\Sigma$,

$$\begin{array}{rcl} \textbf{abort}.\, q & \mathrel{\widehat{=}} & \mathit{false} \\ \textbf{magic}.\, q & \mathrel{\widehat{=}} & \mathit{true} \end{array}$$

Conjunction $\sqcap$ and disjunction $\sqcup$ of (similarly-typed) predicate transformers are defined pointwise:

$$\begin{array}{rcl} (\sqcap\, i \in I \bullet S_i).\, q & \mathrel{\widehat{=}} & (\cap\, i \in I \bullet S_i.\, q) \\ (\sqcup\, i \in I \bullet S_i).\, q & \mathrel{\widehat{=}} & (\cup\, i \in I \bullet S_i.\, q) \end{array}$$

Both conjunction and disjunction of predicate transformers model *nondeterministic choice* among executing either of S_i. Conjunction models *demonic* nondeterministic choice in the sense that nondeterminism is uncontrollable and each alternative must establish the postcondition. Disjunction, on the other hand, models *angelic* nondeterminism, where the choice between alternatives is aimed at establishing the postcondition.

Sequential composition of program statements is modeled by functional composition of predicate transformers. The program statement **skip** is modeled by the identity predicate transformer and its execution has no effect on the program state. For $S : \Sigma \mapsto \Gamma$, $T : \Gamma \mapsto \Delta$ and $q : \mathcal{P}\Delta$,

$$\begin{array}{rcl} (S;T).\, q & \mathrel{\widehat{=}} & S.\, (T.\, q) \\ \textbf{skip}.\, q & \mathrel{\widehat{=}} & q \end{array}$$

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

$$\begin{array}{rcl} \{p\}.\, q & \mathrel{\widehat{=}} & p \,\cap\, q \\ [p].\, q & \mathrel{\widehat{=}} & p \,\subseteq\, q \end{array}$$

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

$$\begin{array}{rcl} \langle f \rangle .\, q .\, \sigma & \mathrel{\widehat{=}} & q .\, (f .\, \sigma) \\ \{P\} .\, q .\, \sigma & \mathrel{\widehat{=}} & (\exists \gamma : \Gamma \bullet (P .\, \sigma .\, \gamma) \ \wedge \ (q .\, \gamma)) \\ [P] .\, q .\, \sigma & \mathrel{\widehat{=}} & (\forall \gamma : \Gamma \bullet (P .\, \sigma .\, \gamma) \ \Rightarrow \ (q .\, \gamma)) \end{array}$$

The functional update applies the function f to the state σ to yield the new state $f .\, \sigma$. When started in a state σ, $\{P\}$ angelically chooses a new state γ such that $P .\, \sigma .\, \gamma$ holds, while $[P]$ demonically chooses a new state γ such that $P .\, \sigma .\, \gamma$ holds. If no such state exists, then $\{P\}$ aborts, whereas $[P]$ behaves as **magic**. For the identity function *id* and the identity relation *Id*, all of $\langle id \rangle$, $\{Id\}$, and $[Id]$ behave as **skip**.

The *conditional* statement is defined by the demonic choice of guarded alternatives:

$$\textbf{if } g \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \ \mathrel{\widehat{=}} \ [g]; S_1 \ \sqcap \ [\neg g]; S_2$$

Iteration is defined as the least fixpoint of a function on predicate transformers with respect to refinement ordering:

$$\textbf{while } g \textbf{ do } S \textbf{ od} \ \mathrel{\widehat{=}} \ (\mu\, X \bullet \textbf{if } g \textbf{ then } S; X \textbf{ else skip fi})$$

A variant of iteration, the *iterative choice* introduced in [12], allows the user to choose repeatedly an alternative that is enabled and have it executed until the user decides to stop:

$$\begin{array}{l} \textbf{do } g_1 :: S_1 \between \ldots \between g_n :: S_n \textbf{ od} \ \mathrel{\widehat{=}} \\ \qquad (\mu\, X \bullet \{g_1\}; S_1; X \sqcup \ldots \sqcup \{g_n\}; S_n; X \sqcup \textbf{skip}) \end{array}$$

We will abbreviate $g_1 :: S_1 \between \ldots \between g_n :: S_n$ by $\between_{i=1}^{n} g_i :: S_i$.

Following [11], we use the notions of assignments, program variables, and variable declarations based on a simple syntactic extension to the typed lambda calculus. For a function $(\lambda u \bullet t)$ which replaces the old state u with the new state t, changing some components $x_1, \ldots, x_m$ of u, while leaving the others unchanged, the *functional assignment* describing such a state change is defined by

$$(\lambda u \bullet x_1, \ldots, x_m := t_1, \ldots, t_m) \ \mathrel{\widehat{=}} \ (\lambda u \bullet u[x_1, \ldots, x_m := t_1, \ldots, t_m])$$

For a relation $(\lambda u \bullet \lambda u' \bullet b)$, which using the set notation could also be written as $(\lambda u \bullet \{u' \mid b\})$, changing a component x of state u to some x' related to x via a boolean expression b, the *relational assignment* is defined by

$$(\lambda u \bullet x := x' \mid b) \ \mathrel{\widehat{=}} \ (\lambda u \bullet \{u[x := x'] \mid b\})$$

As such, the notation for both functional and relational assignments is a convenient syntactic abbreviation for the corresponding lambda term describing a certain state change. Lambda terms, unfortunately, do not maintain consistent naming of state components, due to the possibility of α-conversion of bound variables. To enforce the clearly desirable naming consistency, we use the program variable notation, writing, e.g., $(\mathbf{var}\ x, y \bullet (x := x+y); (y := 0))$ to express that each function term is to be understood as a lambda abstraction over the bound variables x, y:

$$\begin{array}{l}(\mathbf{var}\ x, y \bullet (x := x + y); (y := 0)) \ = \\ \qquad (\lambda x, y \bullet x := x + y); (\lambda x, y \bullet y := 0)\end{array}$$

Ordinary program statements can be modeled using the basic predicate transformers and operators presented above, using the program variable notation. For example, the (multiple) assignment statement can be modeled by the functional update:

$$(\mathbf{var}\ u \bullet x_1, \ldots, x_m := t_1, \ldots, t_m) \ \ \widehat{=}\ \ \langle \lambda u \bullet x_1, \ldots, x_m := t_1, \ldots, t_m \rangle$$

Our specification language includes specification statements. The *demonic assignment* and the *angelic assignment* are modeled by the demonic and the angelic updates respectively:

$$\begin{array}{lcl}[\mathbf{var}\ u \bullet x := x' \mid b] & \widehat{=} & [\lambda u \bullet x := x' \mid b] \\ \{\mathbf{var}\ u \bullet x := x' \mid b\} & \widehat{=} & \{\lambda u \bullet x := x' \mid b\}\end{array}$$

Intuitively, the demonic assignment expresses an uncontrollable nondeterministic choice in selecting a new value x' satisfying b, whereas the angelic assignment expresses a controllable choice. The angelic assignment can, e.g., be understood as a request to the user to supply a new value. The program variable declaration can be propagated outside statements and distributed through sequential composition. When the variable declaration is clear from the context, we will omit it.

A pre- and postcondition specification $(\mathbf{pre}\ p, \mathbf{post}\ q)$, where p is a boolean expression on initial values of program variables x, and q is a boolean expression on initial and final values of x, can be expressed in the refinement calculus as follows:

$$(\mathbf{pre}\ p, \mathbf{post}\ q) \ \ \widehat{=}\ \ \{p\}; [x := x' \mid q]$$

The language also supports blocks with *local variables*. Block beginning and end are modeled by demonic and functional updates respectively:

$$\begin{array}{lcl}\textbf{begin}\ (\textbf{var}\ x, u \bullet b) & \mathrel{\widehat{=}} & [\lambda u \bullet \lambda(x, u') \bullet b \ \wedge \ u = u'] \\ \textbf{end} & \mathrel{\widehat{=}} & \langle \lambda(x, u) \bullet u \rangle\end{array}$$

The **begin** statement introduces a new variable x which is initialized according to the boolean expression b. The global variable u can occur in b, but it must retain its value as indicated by the conjunct $u = u'$. When the variable declaration is clear from the context, we will, for simplicity, write just **begin var** $x \bullet b; S;$ **end**.

All statements in the language of refinement calculus are monotonic and all statement constructors preserve monotonicity.

1.2.3 Data Refinement

Data refinement, as introduced by C.A.R. Hoare [32], is a general technique by which one can systematically change a state space in a refinement. Data refinement has been the subject of many detailed studies. A good overview of different methods and theories of data refinement can be found in [22].

For statements $S : \mathit{Ptran}(\Sigma)$ and $S' : \mathit{Ptran}(\Sigma')$, let $R : \Sigma' \leftrightarrow \Sigma$ be a relation between the state spaces Σ and Σ'. According to [8], the statement S is said to be data refined by S' via R, denoted $S \sqsubseteq_R S'$, if

$$\{R\}; S \sqsubseteq S'; \{R\}$$

This method of data refinement is often referred to as forward data refinement or downward simulation, and is the one most widely used. Although it was shown to be incomplete, this method of proving data refinement is sufficient for most cases in practical program development. Alternative and equivalent characterizations of data refinement using the inverse relation R^{-1} are then

$$S; [R^{-1}] \sqsubseteq [R^{-1}]; S' \qquad S \sqsubseteq [R^{-1}]; S'; \{R\} \qquad \{R\}; S; [R^{-1}] \sqsubseteq S'$$

These characterizations follow from the fact that $\{R\}$ and $[R^{-1}]$ form a Galois connection, in the sense that $\{R\}; [R^{-1}] \sqsubseteq$ **skip** and **skip** $\sqsubseteq [R^{-1}]; \{R\}$.

In general, different state spaces can be coerced using *encoding* and *decoding* operators [13, 33, 70]. 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 $[R^{-1}] : \Sigma \mapsto \Sigma'$. These statements are used to define encoding $\downarrow$ and decoding $\uparrow$ operators as follows:

$$\begin{array}{lcl} S{\downarrow}R & \widehat{=} & \{R\}; S; [R^{-1}] \\ S'{\uparrow}R & \widehat{=} & [R^{-1}]; S'; \{R\} \end{array}$$

Thus, we have that the statements $S{\downarrow}R$ and $S'{\uparrow}R$ operate on the state spaces Σ' and Σ respectively. Encoding and decoding operators can be extended to work with tuples of statements:

$$\begin{array}{lcl} (S_1, ..., S_n){\downarrow}R & \widehat{=} & (S_1{\downarrow}R, ..., S_n{\downarrow}R) \\ (S'_1, ..., S'_n){\uparrow}R & \widehat{=} & (S'_1{\uparrow}R, ..., S'_n{\uparrow}R) \end{array}$$

The encoding and decoding operators are left associative and have higher precedence than function application.

The refinement calculus provides rules for transforming more abstract program structures into more concrete ones based on the notion of refinement of predicate transformers presented above. A large collection of algorithmic and data refinement rules is given, for instance, in [12, 54]. Let us briefly present a few of these rules which will be later used in proofs.

Further on, we make use of the following rules:

$$S{\uparrow}R{\downarrow}R \sqsubseteq S \tag{1.1}$$
$$S' \sqsubseteq S'{\downarrow}R{\uparrow}R \tag{1.2}$$

The *sequential composition rule* states that the data refinement of a sequential composition is refined by a sequential composition of the data refined components:

$$(S_1; S_2){\downarrow}R \sqsubseteq (S_1{\downarrow}R); (S_2{\downarrow}R) \tag{1.3}$$

The *demonic update rule* states that data refinement of demonic updates follows from inclusion of the coerced corresponding relations:

$$R^{-1}; Q' \subseteq Q; R^{-1} \;\Rightarrow\; [Q] \sqsubseteq_R [Q'] \tag{1.4}$$

Demonic assignments are subject to the following *demonic assignment rule*:

$$(b' \;\Rightarrow\; b) \;\Rightarrow\; [x := x' \,|\, b] \sqsubseteq [x := x' \,|\, b'] \tag{1.5}$$

The *iterative choice rule* states that for state predicates $g_1 : \mathcal{P}\Sigma, ..., g_n : \mathcal{P}\Sigma$, statements $S_1 : Ptran(\Sigma), ..., S_n : Ptran(\Sigma)$ and a relation $R : \Sigma' \leftrightarrow \Sigma$ the following holds:

$$\mathbf{do}\ \lozenge_{i=1}^{n} q_i :: S_i\ \mathbf{od}{\downarrow}R \sqsubseteq\ \mathbf{do}\ \lozenge_{i=1}^{n} (\{R\}.\, q_i) :: S_i{\downarrow}R\ \mathbf{od} \tag{1.6}$$

We say that a statement is *indifferent* to data refinement if the statement effectively operates only on the part of the state not affected by a state change. Indifferent statements are subject to the following rules:

$$\begin{aligned} (\mathbf{skip} \times S){\downarrow}(R \times Id) &\sqsubseteq \mathbf{skip} \times S & (1.7)\\ (S \times \mathbf{skip}){\downarrow}(Id \times R) &\sqsubseteq S \times \mathbf{skip} & (1.8)\\ (\mathbf{skip} \times S') &\sqsubseteq (\mathbf{skip} \times S'){\uparrow}(R \times Id) & (1.9)\\ (S' \times \mathbf{skip}) &\sqsubseteq (S' \times \mathbf{skip}){\uparrow}(Id \times R) & (1.10) \end{aligned}$$

In the proofs presented in this dissertation, we will use the following easily verifiable rules. For a statement $S : Ptran(\Sigma \times \Gamma)$ and relations $R : \Sigma' \leftrightarrow \Sigma$ and $P : \Gamma' \leftrightarrow \Gamma$,

$$\begin{aligned} S{\downarrow}(R \times P) &= S{\downarrow}(Id \times P){\downarrow}(R \times Id) & (1.11)\\ S{\downarrow}(R \times Id){\downarrow}(Id \times P) &= S{\downarrow}(Id \times P){\downarrow}(R \times Id) & (1.12) \end{aligned}$$

For a statement $S : \alpha \times \Sigma \times \gamma \times \delta$, a relation $R : \Sigma' \leftrightarrow \Sigma$, and the relation $P = \lambda(x', y', z', u') \bullet \lambda(z, y, u) \bullet (y' = y\ \wedge\ z' = z\ \wedge\ u' = u)$, we have the following rule:

$$S{\uparrow}(Id \times R \times Id \times Id){\uparrow}P = S{\uparrow}P{\uparrow}(Id \times R \times Id) \tag{1.13}$$

When a predicate effectively operates only on the part of the state not affected by a state change then this predicate remains unchanged, as expressed by the rule:

$$\{Id \times R\}.\,(g \times true) \subseteq (g \times true) \tag{1.14}$$

1.3 Copy&Paste as a Software Reuse Technique

As we mentioned above, Copy&Paste is the most primitive and the oldest known software reuse technique. While it is unsatisfactory as far as space, efficiency, and legacy are concerned, from practical experience it is well-known that copying and pasting source code is safe. This empirical knowledge is backed up by the fact that in theory the following monotonicity property holds:

$$(\exists R \bullet S \sqsubseteq_R S') \ \Rightarrow \ (\exists P \bullet T|[S\,]| \sqsubseteq_P T|[S'\,]|)$$

Here $T|[S\,]|$ represents a statement T with at least one occurrence of the statement S. The statement S represents a specification of a desired functionality at a certain point in the context T. This specification can be substituted with the statement S' under the condition that S' refines S. Data refinement is necessary, because S, S', and T can operate on different state spaces. Note that this formula models a "smart" Copy&Paste, i.e. if the pasted statement uses a variable that would be captured in the new context, it should be renamed first. This property is essentially monotonicity of the statement T with respect to data refinement. If all statements of a given programming language are monotonic and all statement constructors preserve monotonicity then applying Copy&Paste as a software reuse technique is safe in this language.

We believe that the monotonicity of constructs in a programming language is of paramount importance for the safety of software reuse mechanisms and techniques that can be used in this language. It is important to note that almost all programming languages contain some non-monotonic constructs. If non-monotonic constructs of a programming language are used, then even such a basic reuse technique as Copy&Paste is not guaranteed to always deliver the expected behavior. In the discussions of different software reuse mechanisms and techniques we always assume that all constructs other than the one under consideration are monotonic, because we model them in the refinement calculus whose constructs are monotonic.

Chapter 2

Functions and Procedures

Functions and procedures are two of the most basic software reuse mechanisms. They were introduced in the programming language Fortran [36]. These code reuse mechanisms separate code intended for reuse into immediately executable sequences of statements. Code reuse is achieved through invoking a function or a procedure from the client code. Information can be passed in and out of a function or a procedure through input and output parameters and, in the case of procedures, through global variables existing in the scope the procedure is invoked in. From the mathematical standpoint, the ability of procedures to access and modify global variables constitutes the primary difference between functions and procedures. In some imperative programming languages, these two mechanisms are unified into procedures that can access and modify global variables, but have functional syntax, i.e., can participate in expressions of the form $a := proc(b)$, where a and b are some variables (b can also be an expression) and *proc* is the name of the procedure. Accordingly, further on in this chapter, we speak about procedures as this is a more general concept.

The arrival of procedures dramatically changed software architecture and the way software was constructed. The technical ability to separate monolithic code into clearly defined sections immediately stimulated a more logical hierarchical organization of programs. Procedures promoted a so-called structural programming style, focused on the functionality of the system and organized around global data structures. In the presence of procedures, the **goto** statement, which was often used as a primary code reuse mechanism, became obsolete. This greatly improved the clarity of the programs, indirectly contributing to their correctness. The emergence of a procedural architecture has dramatically changed the process of software development as well. With a system separated into logically independent parts, it became possible to split an entire software project among independent development teams, enabling them to cope with much larger projects.

Many languages provide for separate compilation capabilities, with a procedure declared in two stages, first declaring a signature of the procedure, i.e., the name and the list of input and output parameters, and declaring the body of the procedure implementing this signature. With separate compilation to compile client code invoking a procedure, it is only necessary that the header of the procedure be within the scope of the client code. After the compilation, the linker will bind invocations of all procedures to their implementations in the client code.

Separate compilation permitted creation of large procedure libraries which until now remain one of the most often used ways of distributing software for reuse. In procedure libraries, code is stored in binary form, thus permitting code developers to protect their intellectual property.

2.1 Modeling Procedures and Procedure Invocation

Procedures have been extensively studied by the formal methods community [31, 7, 54, 53, 12, 56]. The purpose of our simple abstract model of procedures is to establish a general modeling framework which we follow throughout the dissertation.

For simplicity, we model parameters of procedures with global variables. We say that for every input and output procedure parameter, there exists a distinct global variable in the scope of procedure invocation through which parameters are passed in and out of the procedure. We assume that a procedure does not operate on any other global variables, except those which represent procedure parameters. A non-recursive procedure can be modeled as a statement:

$$P \;\widehat{=}\; (\mathbf{skip} \times S) : \mathit{Ptran}(\alpha \times \Delta \times \Gamma)$$

As the procedure can be executed in different contexts, the type of the global variables is not known in advance and therefore is represented with the type variable α. The statement S operates on the procedure parameters, modeled with global variables with the type Δ and on its local state of the type Γ.

Let us now consider an operator modeling application of the procedure reuse mechanism. Code reuse with procedures is achieved through procedure invocation. In general, a program can invoke different procedures. However, for our purposes, it is sufficient to consider one procedure, as the same modeling principles would apply in the case of many procedures.

A program using a procedure can be thought of as a context in which this procedure can be used. Such a context, with an arbitrary but finite number of occurrences of a call to the procedure, can be modeled as a function of the procedure, returning a program in which that procedure is used:

$$C \ \widehat{=}\ \lambda P \bullet \mathit{program}$$

We assume that C is a monotonic function. In the case when the procedure is invoked in a program operating on the state space Σ, the function C has the type $\mathit{Ptran}(\Sigma \times \Delta \times \beta) \to \mathit{Ptran}(\Sigma \times \Delta \times \beta)$. As the type of the internal state of a procedure cannot be known in advance, it is represented with a type variable β.

To model procedure invocation, we use an infix operator **call** which takes two parameters: a context C in which a procedure P is invoked and the procedure itself, and returns a program resulting from invoking P in C:

$$C \ \mathbf{call}\ P \ \widehat{=}\ C.P$$

Thus, **call** is a function of the type $(\mathit{Ptran}(\Sigma \times \Delta \times \beta) \to \mathit{Ptran}(\Sigma \times \Delta \times \beta)) \to \mathit{Ptran}(\alpha \times \Delta \times \Gamma) \to \mathit{Ptran}(\Sigma \times \Delta \times \Gamma)$.

2.2 Are Procedures Safe?

As we stipulated in the introduction chapter, a safe code reuse mechanism is accompanied by a modular verification method. Namely, to prove the correctness of a program invoking a procedure, it is sufficient to prove that the procedure correctly implements the expected functionality.

Suppose that at some point in a client program it is necessary to execute a sequence of statements whose cumulative effect on the state of the program can be described by a specification P. Suppose also that at our disposal there is a procedure P', which appears to implement the specification P correctly. This can be established by verifying that P' refines P. For procedures P :

$Ptran(\alpha \times \Delta \times \Gamma)$ and $P' : Ptran(\alpha \times \Delta \times \Gamma')$, the *refinement on procedures* can be defined as follows:

$$P \sqsubseteq P' \;\widehat{=}\; \exists R : \Gamma' \leftrightarrow \Gamma \bullet P \sqsubseteq_{Id \times Id \times R} P'$$

If procedures are a safe code reuse mechanism, to establish that client code can safely invoke P' in place of P, it should be sufficient to verify that P' is a correct refinement of P. Mathematically, it is necessary to show that the operator **call** is monotonic in its second argument, i.e.,

$$P \sqsubseteq P' \Rightarrow (C \textbf{ call } P) \sqsubseteq (C \textbf{ call } P')$$

Modular Reasoning Theorem for Procedures. *For procedures* $P : Ptran(\alpha \times \Delta \times \Gamma)$ *and* $P' : Ptran(\alpha \times \Delta \times \Gamma')$ *and the context* $C : Ptran(\Sigma \times \Delta \times \beta) \to Ptran(\Sigma \times \Delta \times \beta)$*, the following holds*:

$$P \sqsubseteq P' \Rightarrow \exists R : \Gamma' \leftrightarrow \Gamma \bullet (C \textbf{ call } P) \sqsubseteq_{Id \times Id \times R} (C \textbf{ call } P')$$

Proof The proof follows directly from monotonicity of programming language constructs and monotonicity of the function C. $\square$

This result has been presented earlier by other researchers in the formal methods community, perhaps in a slightly different formulation. Based on the modular reasoning theorem for procedures, we conclude that *procedures are a safe software reuse mechanism.*

2.3 Are Procedures Flexible?

Procedures do not support inline adaptation, yet there exist a number of software reuse techniques relying on procedures which allow for significant degree of flexibility. As with other code reuse techniques, the size of procedures is a major factor. A standard recommendation is that a procedure should encapsulate a piece of code that constitutes a certain semantic entity and should not exceed one page of code. On the one hand, making procedures to implement smaller units of functionality increases the chances of reuse. On the other hand, communication by sending and receiving method parameters is more computationally expensive then communication through common variables. Having an excessive number of procedures in a program might be undesirable for performance reasons. Accordingly, finding the right balance between the reusability of procedures and the efficiency of the programs in which they are used poses the main challenge for procedure developers.

2.4 Discussion

Procedures are one of the basic and the most established code reuse mechanisms. The methods for modular reasoning about procedures have been studied extensively in the literature [31, 7, 54, 53, 12, 16, 56, 66, 60]. Modular reasoning is only possible in programming languages consisting of monotonic constructs. However, not all constructs of contemporary programming languages are monotonic.

A number of imperative programming languages such as Pascal [88] and C [67] support so-called procedure variables. A procedure variable is a variable which can store a procedure as a value. Procedure variables allow for achieving an even higher degree of flexibility as compared to simple variables. The added flexibility comes from the fact that the resulting software becomes dynamically configurable. Imagine that a client program has a variable *a* which is a procedure variable. Which procedure will be called from the client depends on the value the variable *a* has at the moment of invocation. Thus it becomes possible to alter the behavior of a program dynamically during its execution. Obviously, in compiled languages relying exclusively on usual procedures such a degree of flexibility is unachievable, because every time developers substitute the called procedure in a client, it is necessary to recompile it. Methods of reasoning about the programs employing procedure variables were extensively studied by David Naumann [56, 58, 57].

In some languages, it is possible to compare two procedure variables for equality (usually only memory addresses of the corresponding procedures are compared). Although rather useful, a statement expressing such a comparison is not monotonic, as refining procedures does not imply that the client using the procedure variable comparison gets refined. As an example, consider the following program fragment in an Oberon-like syntax.

```
TYPE ReadingProcedure = PROCEDURE(VAR x : CHAR);

PROCEDURE ReadFromFile1(VAR Next : CHAR);
...
END ReadFromFile1;
PROCEDURE ReadFromFile2(VAR Next : CHAR);
...
END ReadFromFile2;
```

```
PROCEDURE Read(p : ReadingProcedure)
BEGIN
    IF p = ReadFromFile1 THEN
        S1;
    ELSE
        S2;
    END;
END Read;
```

Suppose that the procedure `ReadFromFile1` is refined by the procedure `ReadFromFile2`, e.g., it can be an alternative more efficient implementation. A call to the procedure `Read` with the argument `ReadFromFile1` results in an execution of the statement `S1`, while calling this procedure with the argument `ReadFromFile2` results in an execution of the statement `S2`. Obviously, in general, `S1` is not refined by `S2`. Therefore, `Read(ReadFromFile1)` is not refined by `Read(ReadFromFile2)`. Accordingly, the extra flexibility achieved with the use of procedure variables can undermine safety.

Chapter 3

Modules

Procedures promote the structural programming style. Programs developed following this style have clear separation between algorithms and the data structures that these algorithms operate on. Data structures are usually defined in the global scope of a program. Since different procedures usually operate on different global variables, it gradually became clear that grouping variables and procedures operating on them together can lead to better structured programs.

In 1972, David Parnas published an influential work on modularization [62], in which he demonstrated the value of decomposing a system into a collection of modules. To promote a modular programming style, a language support for modules limiting access to the internal representation from outside of the module is clearly desirable.

Protection of the module's internal representation from external access (i.e., encapsulation) can be achieved through type abstraction or procedural data abstraction [18]. For example, in SML [63] a construct `structure` allows for defining modules in an algebraic manner. With an algebraic approach, every module's function is defined in terms of the constructors of a type implementing the module's internal representation. By default, constructors of the internal module representation type are exposed to clients of the module. This permits a client program to tamper with the internal representation of the module, which might lead to invalidation of the module invariant. To avoid such an exposure, a module can be constrained by an *opaque signature constraint*, which makes only module constructors visible from outside, while constructors of the internal representation type remain completely hidden. Thus internal representation of the module is protected by abstracting from the type of internal representation.

Alternatively, access to the module's internal representation can be restricted using an *export facility*, as, for example, is done in Oberon [89]. Variables and procedures declared in a module can be marked with an asterisk to indicate which declarations are visible from outside the module. An attempt to access a module's attribute not marked with an asterisk from outside of the module is a compiler error. A programmer can protect the module's internal representation by not exporting it and providing access to it exclusively through module's procedures. Thus a client can abstract from particular module's implementation by viewing it exclusively through its procedural interface.

An important characteristic feature of modules is that they are usually not first-class values of a programming language. This means that modules cannot be passed as arguments or returned as values to and from procedures.

Modules can invoke procedures on other modules. Usually there exists a restriction stipulating that a graph of procedure invocations on modules be acyclic, i.e., if a module A invokes procedures on another module B, the latter should not invoke procedures on the former. Depending on a particular implementation language, this restriction can be enforced in various ways. In languages in which a module is a compilation unit (like in Oberon), this restriction can be enforced by prohibiting a cyclic importing structure. We believe that this restriction constitutes a major distinguishing feature of modules as compared to other software reuse mechanisms concerned with grouping of data and algorithms.

3.1 Modeling Modules

As the structure of procedure invocations on modules is acyclic, it is always possible to flatten the structure of procedure calls by first extending the state space of a module with state spaces of the called modules and then substituting the bodies of invoked procedures for procedure calls. Therefore, a module M with an internal state and n non-recursive procedures can be represented by

$$\mathsf{M} \mathrel{\widehat{=}} (m_0, Mp),$$

where $m_0 : \Gamma$ is an initial value of the internal state, and Mp is a tuple of procedures $(P_1, ..., P_n)$ modifying this internal state. For simplicity, we model parameters of module procedures with global variables. We say that for every input and output procedure parameter, there exists a distinct global variable in the scope of procedure invocation through which parameters are passed in and out of the procedure. As, in general, a module can be used by an arbitrary client program, we assume that a module does not refer to global variables. Therefore, the tuple of procedures Mp is of type $\Pi^n(\alpha \times \Delta \times \Gamma)$, where α is a type variable representing the state space of the client program and Δ is the state space component representing all the parameters of all procedures.

Methods of refining modules were thoroughly studied in the literature [32, 8, 27]. We make a simplification that a refining module has the same number of methods as the refined one. We can define *refinement on modules* as follows: the module $\mathsf{M} = (m_0 : \Gamma, Mp : \Pi^n(Ptran(\alpha \times \Delta \times \Gamma)))$ is refined by a module $\mathsf{M}' = (m'_0 : \Gamma', Mp' : \Pi^n(Ptran(\alpha \times \Delta \times \Gamma')))$, if there exists a relation $R : \Gamma' \leftrightarrow \Gamma$ connecting the initial values m'_0 and m_0 and it is possible to show that all procedures in Mp are data refined by the corresponding procedures in Mp' via the relation $Id \times Id \times R$. Formally,

$$\mathsf{M} \sqsubseteq \mathsf{M}' \;\widehat{=}\; \exists R \bullet R.\, m'_0.\, m_0 \;\wedge\; Mp \sqsubseteq_{Id \times Id \times R} Mp'$$

A program using a module can be modeled using the iterative choice statement as proposed in [5]. Every time the program has the choice as to which procedure to choose for execution. In general, each option is preceded with an assertion which determines whether the option is enabled in a particular state. While at least one of the assertions holds, the program may repeatedly choose a particular option which is enabled and have it executed. The program decides on its own when it is willing to stop choosing options. Such an iterative choice of procedure calls, followed by arbitrary statements operating only on a local state of the program, describes all possible interactions the program might have with the module:

$$Program \;\widehat{=}\; \left(\begin{array}{l} \textbf{var } l, d, m \bullet \\ \qquad [l, d, m := \underline{l}, \underline{d}, \underline{m} \mid p \;\wedge\; \underline{m} = m_0]; \\ \qquad\quad \textbf{do } \widehat{q_1} :: P_1; \widehat{T_1} \between \ldots \between \widehat{q_n} :: P_n; \widehat{T_n} \textbf{ od} \end{array}\right)$$

Variables $l : \Sigma$ are some variables local to the program, while variables $d : \Delta$ represent procedure arguments. The variable m holds the internal state of the used module. The variables l and d are initialized according to the condition p expressed in terms of $\underline{l}, \underline{d}$, which permits to specify the arguments passed to procedures. Before the module can be used, its internal state must be initialized. The condition $\underline{m} = m_0$ of the nondeterministic assignment, where m_0 is the initial state of the module, models this initialization. The predicates $\widehat{q} : \mathcal{P}(\Sigma \times \Delta \times \Gamma)$ are the asserted conditions on the program state and the procedure arguments. As the state of the module is encapsulated, the predicates $\widehat{q}$ do not refer to the module's internal variables, i.e., $\widehat{q} = q \times \mathit{true}$. We assume that procedures P do not refer to the program's global variables. Accordingly, as defined in Chapter 2, procedures have a structure $\mathbf{skip} \times S$, where S has a type $\mathit{Ptran}(\Delta \times \Gamma)$. Statements $\widehat{T}$ operate on the state space $\Sigma \times \Delta$ and model the arbitrary actions a program can do on its local state and procedure parameters in between procedure calls. Accordingly, $\widehat{T} = (T \times \mathbf{skip})$. Opening up all the abbreviations, the program can be rewritten as follows:

$$\mathit{Program} = \left(\begin{array}{l} \mathbf{var}\ l, d, m \bullet \\ \qquad [l, d, m := \underline{l}, \underline{d}, \underline{m} \mid p \ \wedge\ \underline{m} = m_0]; \\ \qquad \mathbf{do}\ \between_{i=1}^{n} (q_i \times \mathit{true}) :: (\mathbf{skip} \times S_i); (T_i \times \mathbf{skip})\ \mathbf{od} \end{array} \right)$$

A program can use any module under the condition that signatures of module's procedures match those expected by the program. Accordingly, a context in which a module is used can be described as follows:

$$\mathit{Context}\ \widehat{=}\ \lambda(X_1, ..., X_n) \bullet \left(\begin{array}{l} \mathbf{var}\ l, d, m \bullet \\ \qquad [l, d, m := \underline{l}, \underline{d}, \underline{m} \mid p \ \wedge\ \underline{m} = m_0]; \\ \qquad \mathbf{do}\ \between_{i=1}^{n} \widehat{q}_i :: X_i; \widehat{T}_i\ \mathbf{od} \end{array} \right)$$

Note that as the type of the internal state of the module is not yet known, it is modeled with the type variable β. On a more abstract level, a context can be modeled as a function of module procedures, returning a program where that module is used:

$$\mathit{Context} = \lambda Mp \bullet \mathit{Program}$$

Note that *Context* is a monotonic function of the type $\Pi^n(\mathit{Ptran}(\Sigma \times \Delta \times \beta)) \to \mathit{Ptran}(\Sigma \times \Delta \times \beta)$.

Now a program calling module procedures can be described using an infix operator **uses** which takes two parameters, a context C in which invocations of the procedures of the module M occur an arbitrary but finite number of times and the module itself, and returns a program which invokes procedures of M from the context C:

$$(C\ \mathbf{uses}\ \mathsf{M})\ \widehat{=}\ C.(\mathit{snd}.\mathsf{M})$$

During the function application all type variables get instantiated with the corresponding types, and therefore the resulting program has the type $Ptran(\Sigma \times \Delta \times \Gamma)$.

3.2 Are Modules Safe?

As was already mentioned above, a code reuse mechanism or technique permits modular reasoning if it is sufficient to verify that the code candidate for reuse correctly implements the functionality expected by the client. For modules, we can express the modular reasoning property as the following theorem:

Modular Reasoning Theorem for Modules. *For modules* M, M' *defined by*

$$\begin{array}{rcl} \mathsf{M} & = & (m_0 : \Gamma, Mp : \Pi^n(Ptran(\alpha \times \Delta \times \Gamma))) \\ \mathsf{M}' & = & (m'_0 : \Gamma', Mp' : \Pi^n(Ptran(\alpha \times \Delta \times \Gamma'))) \end{array}$$

and a context C *of type* $\Pi^n(Ptran(\Sigma \times \Delta \times \beta)) \to Ptran(\Sigma \times \Delta \times \beta)$*, the following holds:*

$$\mathsf{M} \sqsubseteq \mathsf{M}' \Rightarrow \exists R \bullet (C \textbf{ uses } \mathsf{M}) \sqsubseteq_{Id \times Id \times R} (C \textbf{ uses } \mathsf{M}')$$

Proof The goal can be rewritten as follows:

$$\begin{array}{l} (\exists R \bullet R.\, m'_0.\, m_0 \ \wedge \ (P_1, ..., P_n) \sqsubseteq_{Id \times Id \times R} (P'_1, ..., P'_n)) \ \Rightarrow \\ \quad \exists R \bullet \quad (\textbf{var } \ l, d, m \bullet [l, d, m := \underline{l}, \underline{d}, \underline{m} \mid p \ \wedge \ \underline{m} = m_0]; \\ \quad\quad\quad \textbf{do } \lozenge_{i=1}^n \widehat{q_i} :: P_i; \widehat{S_i} \textbf{ od}) \\ \quad\quad\quad \sqsubseteq_{Id \times Id \times R} \\ \quad\quad\quad (\textbf{var } \ l, d, m' \bullet [l, d, m' := \underline{l}, \underline{d}, \underline{m'} \mid p \ \wedge \ \underline{m'} = m'_0]; \\ \quad\quad\quad \textbf{do } \lozenge_{i=1}^n \widehat{q_i} :: P'_i; \widehat{S_i} \textbf{ od}) \end{array} \tag{3.1}$$

The proof of this goal is similar to the proof of Theorem 1 in [10]. To prove the goal, we need the following lemma:

Lemma. *For a relation* $R : \Gamma' \leftrightarrow \Gamma$,

$$\begin{array}{l} (\textbf{var } \ l, d, m \bullet [l, d, m := \underline{l}, \underline{d}, \underline{m} \mid p \ \wedge \ \underline{m} = m_0]) \sqsubseteq_{Id \times Id \times R} \\ \quad\quad (\textbf{var } \ l, d, m' \bullet [l, d, m' := \underline{l}, \underline{d}, \underline{m'} \mid p \ \wedge \ R.\, \underline{m'}.\, m_0]) \end{array}$$

Proof Using the demonic update rule 1.4, this goal can be reduced as follows:

$$\begin{array}{l}(Id \times Id \times R)^{-1};\\ (\lambda\, l', d', m' \bullet \lambda\, \underline{l'}, \underline{d'}, \underline{m'} \bullet p[l, d, \underline{l}, \underline{d} := l', d', \underline{l'}, \underline{d'}] \ \wedge \ R.\, \underline{m'} m_0) \subseteq\\ \qquad (\lambda\, l, d, m \bullet \lambda\, \underline{l}, \underline{d}, \underline{m} \bullet p \ \wedge \ \underline{m} = m_0);(Id \times Id \times R)^{-1}\end{array}$$

Applying the definition of relational inclusion, this can further be reduced to:

$$\begin{array}{l}((Id \times Id \times R)^{-1};\\ (\lambda\, l', d', m' \bullet \lambda\, \underline{l'}, \underline{d'}, \underline{m'} \bullet p[\underline{l}, \underline{d} := \underline{l'}, \underline{d'}] \ \wedge \ R.\, \underline{m'}.\, m_0)).\\ \qquad (l, d, m).\, (\underline{l'}, \underline{d'}, \underline{m'}) \ \Rightarrow\\ \qquad\qquad ((\lambda\, l, d, m \bullet \lambda\, \underline{l}, \underline{d}, \underline{m} \bullet p \ \wedge \ \underline{m} = m_0);\\ \qquad\qquad (Id \times Id \times R)^{-1}).\, (l, d, m).\, (\underline{l'}, \underline{d'}, \underline{m'})\end{array}$$

Using the definition of relational composition and simple logical transformations, this goal can be reduced to true. $\square$

To prove our main goal 3.1, we first discharge the existential quantification and assume the antecedent. Now the goal looks as follows:

$$\begin{array}{l}\exists R \bullet (\textbf{var}\ \ l, d, m \bullet \ [l, d, m := \underline{l}, \underline{d}, \underline{m} \mid p \ \wedge \ \underline{m} = m_0];\\ \qquad\qquad \textbf{do}\ \between_{i=1}^{n} \widehat{q_i} :: P_i; \widehat{S_i}\ \textbf{od}){\downarrow}(Id \times Id \times R) \sqsubseteq\\ \qquad (\textbf{var}\ \ l, d, m' \bullet \ [l, d, m' := \underline{l}, \underline{d}, \underline{m'} \mid p \ \wedge \ \underline{m'} = m_0'];\\ \qquad\qquad \textbf{do}\ \between_{i=1}^{n} \widehat{q_i} :: P_i'; \widehat{S_i}\ \textbf{od})\end{array}$$

Choosing the quantified relation to be the same as the relation in the assumption, we start with the left-hand side and reduce it to the right-hand side as follows:

$$\begin{array}{ll} & (\textbf{var}\ \ l, d, m \bullet \ [l, d, m := \underline{l}, \underline{d}, \underline{m} \mid p \ \wedge \ \underline{m} = m_0];\\ & \qquad \textbf{do}\ \between_{i=1}^{n} \widehat{q_i} :: P_i; \widehat{S_i}\ \textbf{od}){\downarrow}(Id \times Id \times R)\\ \sqsubseteq & \{\textit{sequential composition rule 1.3}\ \}\\ & (\textbf{var}\ \ l, d, m \bullet \ [l, d, m := \underline{l}, \underline{d}, \underline{m} \mid p \ \wedge \ \underline{m} = m_0]){\downarrow}(Id \times Id \times R);\\ & (\textbf{var}\ \ l, d, m \bullet \ \ \textbf{do}\ \between_{i=1}^{n} \widehat{q_i} :: P_i; \widehat{S_i}\ \textbf{od}){\downarrow}(Id \times Id \times R)\\ \sqsubseteq & \{\textit{lemma above}\ \}\\ & (\textbf{var}\ \ l, d, m' \bullet \ [l, d, m' := \underline{l}, \underline{d}, \underline{m'} \mid p \ \wedge \ R.\, \underline{m'}.\, m_0]);\\ & (\textbf{var}\ \ l, d, m \bullet \ \ \textbf{do}\ \between_{i=1}^{n} \widehat{q_i} :: P_i; \widehat{S_i}\ \textbf{od}){\downarrow}(Id \times Id \times R)\\ \sqsubseteq & \{\textit{iterative choice rule 1.6}\ \}\end{array}$$

$$
\begin{array}{ll}
& (\mathbf{var}\ l, d, m' \bullet [l, d, m' := \underline{l}, \underline{d}, \underline{m'} \mid p \ \wedge\ R.\,\underline{m'}.\,m_0]); \\
& (\mathbf{var}\ l, d, m' \bullet \\
& \qquad \mathbf{do}\ \between_{i=1}^{n}(\{Id \times Id \times R\}.\,\widehat{q_i}) :: (P_i; \widehat{S_i}){\downarrow}(Id \times Id \times R)\ \mathbf{od}) \\
\sqsubseteq & \{\mathit{rules\ 1.14,\ 1.3}\} \\
& (\mathbf{var}\ l, d, m' \bullet [l, d, m' := \underline{l}, \underline{d}, \underline{m'} \mid p \ \wedge\ R.\,\underline{m'}.\,m_0]); \\
& (\mathbf{var}\ l, d, m' \bullet \\
& \qquad \mathbf{do}\ \between_{i=1}^{n}\widehat{q_i} :: P_i{\downarrow}(Id \times Id \times R); \widehat{S_i}{\downarrow}(Id \times Id \times R)\ \mathbf{od}) \\
\sqsubseteq & \{\mathit{rules\ for\ indifferent\ statements\ 1.7}\} \\
& (\mathbf{var}\ l, d, m' \bullet [l, d, m' := \underline{l}, \underline{d}, \underline{m'} \mid p \ \wedge\ R.\,\underline{m'}.\,m_0]); \\
& (\mathbf{var}\ l, d, m' \bullet\ \mathbf{do}\ \between_{i=1}^{n}\widehat{q_i} :: P_i{\downarrow}(Id \times Id \times R); \widehat{S_i}\ \mathbf{od}) \\
\sqsubseteq & \{\mathit{demonic\ assignment\ rule\ 1.5\ and\ assumption}\ (R.\,m'_0.\,m_0), \\
& \ \mathit{assumption}\ (P_1, ..., P_n) \sqsubseteq_{Id \times Id \times R} (P'_1, ..., P'_n)\} \\
& (\mathbf{var}\ l, d, m' \bullet [l, d, m' := \underline{l}, \underline{d}, \underline{m'} \mid p \ \wedge\ \underline{m'} = m'_0]; \\
& \qquad \mathbf{do}\ \between_{i=1}^{n}\widehat{q_i} :: P'_i; \widehat{S_i}\ \mathbf{od})
\end{array}
$$

□

A similar result, in a slightly different formulation, was shown in [54]. Accordingly, we conclude that *modules are a safe software reuse mechanism.*

3.3 Are Modules Flexible?

Modules do not allow inline code modification, meaning that a module can only be reused as is. A client of a module cannot customize the module, unless facilities for module customization were incorporated in its design. However, modules have many useful properties contributing to their flexibility.

Often module interfaces are specified separately from module implementations. Different modules can therefore be coded by different members of a project team. The compiler can check that each module meets its syntactic interface specification. In many programming languages and systems, a module is the smallest compilation unit and separate compilation of modules is supported. This helps to reduce compilation time, as only recently modified modules have to be recompiled.

Modules are easily combined with other software reuse mechanisms. A module may contain pieces of code packaged for reuse employing, e.g., procedures, objects, or classes. Usually, modules export definitions of procedures so that they can be used from outside the exporting modules. When modules are combined with object-oriented software reuse mechanisms, classes and objects can also be exported from the incorporating modules.

To a large extent, the flexibility of modules is determined by their granularity. The smaller the modules are the higher is the possibility of their reuse. However, communication overhead has its costs, as communicating through procedure invocations is more computationally expensive than communicating through shared variables. Finding the right balance between the granularity of modules and the efficiency of programs that use these modules is a primary challenge for module designers.

3.4 Discussion

Before the arrival of modules, programs were structured into "subprograms" that were procedures or functions. It is arguable that procedures and functions are too fine-grained structuring units. According to Lawrence Paulson [63], structuring programs into procedures and functions "is like regarding [a] bicycle as composed of thousands of metal shapes". The procedural style of programming allows building programs in a hierarchical manner, starting with fine-grained low-level procedures, which are called by procedures on the next layer of the hierarchy and so forth. While modules do not change the style of programming, they permit extra structuring of programs which is orthogonal to the style of programming. This extra structuring is achieved through composition of procedures into logically connected units. The resulting modules usually incorporate program fragments with high cohesion, i.e., a high degree of coupling between its parts.

Not only can modules help better structuring of monolithic programs, but they also can be used for creating module libraries. Some facility can be so useful that it is worthwhile extracting it into a general purpose solution applicable in other programs as well. Such a solution is a software component and it can be packaged using the module mechanism.

Partitioning programs into modules, in addition to the benefits listed above, allows reasoning about program properties, including correctness, in a modular fashion. Therefore, modules can also be seen as a mechanism of structuring proofs about program properties.

Chapter 4

Objects and Forwarding

Although Parnas recognized that modules with compatible interfaces can be used interchangeably, he did not develop this possibility. As a result, modules cannot be passed as arguments or returned as values, so they are not first-class values in programming languages.

There are two kinds of module-like constructs that are the first-class values in modern programming languages. Like modules, an abstraction barrier between a client and data can be achieved in these constructs via type abstraction or procedural abstraction. The language constructs are referred to respectively as *objects* and *ADTs*.

Objects were introduced in the programming language Simula of Ole-Johan Dahl and Kristen Nygaard [21]. Another early reference is [90]. In this paper, Stephen Zilles showed "how procedures can be used to represent another class of system components, *data objects*, which are not normally expressed as programs". Objects in Smalltalk exemplify the first kind of internal representation's protection via procedural abstraction. The built-in language mechanism hides the internal representation of an object permitting access to it only through object methods.

Protecting internal representation by type abstraction was first discussed by Barbara Liskov and Stephen Zilles in [45]. The construct `abstype` in the programming language SML [63] can serve as the example. It allows creating a new programming type, which permits instantiating values of this type. This type is abstract, because internal representation of the type's values can be observed or modified from the outside only by accessing the type's constructors and operations. The mechanism of achieving this is analogous to that of SML's `structure` construct, as described in Chapter 3.

While the difference in achieving encapsulation of internal representation is traditionally considered to be the main feature distinguishing between objects, ADTs, and modules, we maintain that the ability to be passed and received as procedure arguments is the main differentiating factor. This point of view determines the structure of the dissertation, namely, we consider objects and ADTs together and separately from modules. As most of the arguments and results that we present here apply to both objects and ADTs, further on, when speaking about objects, we mean both objects and ADTs.

The main characteristic feature of objects is their dynamic nature. Objects can be substituted for other objects dynamically, at run-time, which greatly increases the flexibility of programs. Dynamic object substitutability allows one to reconfigure a program without having to recompile it.

As modules are usually a supplementary construct in programming languages, helping to group data structures and procedures operating on them, introduction of modules did not lead to changes in the programming style. The introduction of objects, on the other hand, changed the entire programming paradigm. Instead of focusing on the procedures modifying passive data, the new programming style focused on the objects encapsulating relevant data structures and communicating with other objects by invoking their procedures (or methods, in object-oriented terms).

From the formal standpoint, an object differs from a module because an object rarely operates in isolation. Usually, at run-time, an OO system can be seen as a network of communicating objects which cooperate to achieve an overall functionality of the system. Thus an object is usually dependent on other objects.

In object-oriented programming (and especially in the Smalltalk programming jargon), invoking a method on an object is often referred to as *sending a message* to the object. The well-known error "message is not understood" means, in fact, that the object does not have the method that client of the object tries to invoke. It is interesting to note that this error usually occurs when programming with untyped languages lacking static typechecking, e.g., Smalltalk; with statically-typed languages such an error in most cases would be detected by a compiler.

Object-oriented programming languages can be divided into two broad categories – those that employ classes and those relying on prototypical objects. This differentiation, as it appears, stems from two fundamental knowledge representation models, as studied in artificial intelligence [43].

The first category uses a syntactic notion of a class, which is a stencil describing a particular type of objects created by instantiating this class. Classes and related software reuse mechanisms and techniques are studied in detail in Chapter 6. The second category of programming languages uses prototypical objects, i.e., objects representing a stereotype of a particular behavior. When this stereotype behavior is insufficient for describing some subtleties, an *extension* object providing the additional behavior can be created and linked to the prototypical object. This approach to sharing knowledge in object-oriented systems first appeared in *actor* languages, and several Lisp-based object-oriented systems such as Director [37], T [65], Orbit [76], and others. When the extension object receives a message, it first attempts to respond to the message using its own behavior. If the object's characteristics are not relevant for answering the message, the object sends the message on to its prototype(s) to see if one can respond to the message. In the programming language community, this message sending is referred to as being carried through *forwarding* or *delegation.* The names of these reuse techniques can be interpreted as follows: when an object receives a message that it does not know how to process, it forwards the message or delegates the task of responding to this message to another object which knows how to handle it.

The terms forwarding and delegation are often used interchangeably, as synonyms. However, as noted by Clemens Szyperski in [81], "the difference is of such a fundamental nature that the price for this imprecision is a resulting lack of understanding". While in rare discussions of the differences between these two object-based software reuse techniques the treatment of self references was elevated as the major differentiation factor, we believe that this issue is secondary. Forwarding and delegation reflect the two reoccurring patterns of object communication through message sending. Forwarding corresponds to the *push* pattern – all the data that might be needed for another object is passed as parameters. Delegation corresponds to the *pull* pattern, with which a reference to the original object is established, e.g., by passing the reference to this object as one of method parameters, and the other object pulls all the data it needs from the original object by calling back its methods. We consider the difference in object communication patterns to be the main factor differentiating between these reuse techniques. Namely, with forwarding, the structure of object method invocations is acyclic, i.e., associations between communicating objects represent a directed acyclic graph. With delegation, objects can refer to each other, i.e., associations between communicating objects represent an undirected graph. Most of the results

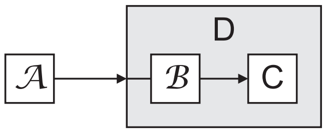

Figure 4.1: Three objects participating in forwarding

and discussions in this and the following chapters apply not only to software reuse techniques alone but also to the corresponding communication patterns.

In this chapter, we present a model of objects and a model of forwarding. Then we focus on the safety and flexibility of this software reuse technique and finish with drawing some conclusions and discussing related work.

4.1 Modeling Objects

Without loss of generality, we can consider the case when only two objects participate in the forwarding relation. For justification of this simplification consider figure 4.1. In this figure, the program consisting of objects $\mathcal{A}$, $\mathcal{B}$, and C such that $\mathcal{A}$ forwards to $\mathcal{B}$ which in turn forwards to C can be considered as consisting of only two objects $\mathcal{A}$ and D such that D is the object resulting from forwarding messages from $\mathcal{B}$ to C. The case when an object forwards messages to several other objects can be modeled in a similar manner.

Consider two objects $\mathcal{A}$ and B such that $\mathcal{A}$ forwards to B. The definition of methods in $\mathcal{A}$ depends on the definition of methods in B that they invoke. As B does not call back on methods of $\mathcal{A}$, from the mathematical standpoint B is no different from a module and therefore can be modeled by a pair of the initial state value and the tuple of procedures.[1]

Let us now consider a model of a *dependable object*, such as $\mathcal{A}$. The object $\mathcal{A}$ communicates with B by invoking its methods and passing parameters. Here we only consider objects which do not have recursive and mutually recursive methods. For simplicity, we model method parameters by global variables that methods of both $\mathcal{A}$ and B can access. For every formal parameter of a method, we introduce a separate global variable which is used for passing values in and out of objects. It is easy to see that parameter passing

[1]This explains the difference in typesetting the object names.

by value and by reference can be modeled in this way.

As, the type of the internal state of the other object is not known due to encapsulation, we say that the body of a method of $\mathcal{A}$ has the type $Ptran(\Sigma \times \Delta \times \beta)$, where Σ is the type of $\mathcal{A}$'s internal state, Δ is the type of global variables modeling method parameters, and β is the type variable to be instantiated with the type of the internal state of the other object during composition. As the internal state of the other object is not accessible, we assume that methods of $\mathcal{A}$ operate only on their internal state and the state representing method parameters, and are therefore of the form $S \times$ **skip**. Similarly, methods of B have bodies that are of the form **skip** $\times S$ and of the type $Ptran(\alpha \times \Delta \times \Gamma)$, where α is a type variable.

The behavior of an object's method depends on the behavior of the methods it invokes. We can model a method a_i of the object $\mathcal{A}$ as a function of a list of method bodies returning a method body: [2]

$$a_i \ \widehat{=}\ \lambda Bb \bullet ab_i$$

The type of a_i can be written out as $\Pi^m(Ptran(\Sigma \times \Delta \times \beta)) \rightarrow Ptran(\Sigma \times \Delta \times \beta)$, where m is the number of methods of B.

We assume that every method is monotonic in its argument. Accordingly, we can collectively describe all methods $a_1, ..., a_n$ of $\mathcal{A}$ as a function A given as follows:

$$A \ \widehat{=}\ (\lambda Bb \bullet (ab_1, ..., ab_n)) : \Phi^{m,n}(Ptran(\Sigma \times \Delta \times \beta))$$

Therefore, the object $\mathcal{A}$ can be defined by

$$\mathcal{A} \ \widehat{=}\ (a_0, A),$$

where $a_0 : \Sigma$ is an initial value of the internal state and A is the function defined above.

4.2 Modeling Forwarding

As we already mentioned, the structure of method invocations between the objects participating in forwarding is acyclic. For the case of two objects

[2]We accept the following scheme for naming variables: a variable starting with a capital letter represents a tuple of variables; the second letter b in the name of a variable means that it represents a method body (statement) or a tuple of method bodies.

involved in forwarding, this means that $\mathcal{A}$ invokes methods on B which does not call-back on $\mathcal{A}$. From a formal perspective, an object which does not invoke methods on any other object is no different from a module.

Suppose that the object B is defined as follows:

$$\mathsf{B} \mathrel{\widehat{=}} (b_0 : \Gamma, Bp : \Pi^m(Ptran(\alpha \times \Delta \times \Gamma)))$$

An infix operator **forw** is defined according to the following formula:

$$(\mathcal{A} \ \mathbf{forw} \ \mathsf{B}) \mathrel{\widehat{=}} ((a_0, b_0), (A. (snd. \mathsf{B})))$$

It is easy to see that from a formal standpoint ($\mathcal{A}$ **forw** B) is a module, as it is a pair of an initial value and a tuple of procedures.

4.3 Is Forwarding Safe?

As we already explained, a code reuse technique is safe if there is a modular reasoning method associated with it. A reasoning method is modular if it allows for establishing the correctness of a client program reusing a piece of code only through verifying that the code candidate for reuse correctly implements the functionality expected by the client. Since an object that does not invoke methods of other objects is essentially a module, we can abstract the functionality expected by the client as a module.

In the previous chapter, we have shown that by refining a module we refine a client program using this module. In particular, if we refine a module D to a module D', then a client program C using D is refined to C' through using D' instead of D. Suppose now that D results from forwarding from an object $\mathcal{A}$ to an object B, and D' from forwarding from the object $\mathcal{A}$ to an object B'. A natural question arises whether forwarding supports modular reasoning. In other words, is it possible by refining B to B' to establish refinement between the modules ($\mathcal{A}$ **forw** B) and ($\mathcal{A}$ **forw** B')? Obviously, this is a desirable property, because by combining it with the modular reasoning property for modules we get, by transitivity, that for establishing the refinement between the clients C and C' it is sufficient to prove the refinement between the objects B and B'. Fortunately, this property holds as proved by the following theorem.

Modular Reasoning Theorem for Forwarding. *For an object $\mathcal{A}$ and modules B, and B' defined by*

$$\begin{array}{lcl}
\mathcal{A} & = & (a_0 : \Sigma, A : \Phi^{m,n}(\mathit{Ptran}(\Sigma \times \Delta \times \beta))) \\
\mathsf{B} & = & (b_0 : \Gamma, \mathit{Bp} : \Pi^{m}(\mathit{Ptran}(\alpha \times \Delta \times \Gamma))) \\
\mathsf{B}' & = & (b_0' : \Gamma', \mathit{Bp} : \Pi^{m}(\mathit{Ptran}(\alpha \times \Delta \times \Gamma')))
\end{array}$$

the following holds:

$$\mathsf{B} \sqsubseteq \mathsf{B}' \Rightarrow (\mathcal{A}\ \mathbf{forw}\ \mathsf{B}) \sqsubseteq (\mathcal{A}\ \mathbf{forw}\ \mathsf{B}')$$

Proof To prove the theorem, we need to prove the following two goals:

$$(\exists Q \bullet Q.\, b_0'.\, b_0) \ \Rightarrow\ \exists Q \bullet (\mathit{Id} \times Q).\, (a_0, b_0').\, (a_0, b_0) \tag{4.1}$$

$$\begin{array}{l}(\exists Q \bullet \mathit{Bp}{\downarrow}(\mathit{Id} \times \mathit{Id} \times Q) \sqsubseteq \mathit{Bp}') \ \Rightarrow \\ \qquad \exists Q \bullet (A.\, \mathit{Bp}){\downarrow}(\mathit{Id} \times \mathit{Id} \times Q) \sqsubseteq A.\, \mathit{Bp}'\end{array} \tag{4.2}$$

The goal 4.1 is obviously true. Let us consider a proof of the goal 4.2. Applying the function A, representing methods of the object $\mathcal{A}$, to the tuple of method bodies Bp of another object, results in a tuple of method bodies of $\mathcal{A}$ with the method bodies in Bp substituted for the corresponding method calls. Accordingly, $A.\, \mathit{Bp}$ can be modeled in a manner similar to that used for modeling module clients as presented in Chapter 3.

$$A.\, \mathit{Bp} = \left(\begin{array}{l}
\mathbf{var}\ l^1, d^1, k^1 \bullet \\
\qquad [l^1, d^1, k^1 := \underline{l^1}, \underline{d^1}, \underline{k^1} \mid p \ \wedge\ \underline{k^1} = k_0^1]; \\
\qquad \mathbf{do}\ \lozenge_{i=1}^{m} \widehat{q}_i^1 :: P_i; \widehat{T}_i^1\ \mathbf{od}, \\
\qquad\qquad \vdots, \\
\mathbf{var}\ l^n, d^n, k^n \bullet \\
\qquad [l^n, d^n, k^n := \underline{l^n}, \underline{d^n}, \underline{k^n} \mid p \ \wedge\ \underline{k^n} = k_0^n]; \\
\qquad \mathbf{do}\ \lozenge_{i=1}^{m} \widehat{q}_i^n :: P_i; \widehat{T}_i^n\ \mathbf{od}
\end{array} \right)$$

Therefore, after discharging the existential quantifications, the goal 4.2 can be rewritten as follows:

$$(P_1, ..., P_n){\downarrow}(\mathit{Id} \times \mathit{Id} \times Q) \sqsubseteq (P_1', ..., P_n') \ \Rightarrow$$

$$
\left(\begin{array}{l}
\textbf{var } l^1, d^1, k^1 \bullet \\
\qquad [l^1, d^1, k^1 := \underline{l^1}, \underline{d^1}, \underline{k^1} \mid p \ \wedge \ \underline{k^1} = k_0^1]; \\
\qquad \textbf{do } \Diamond_{i=1}^{m} \widehat{q}_i^1 :: P_i; \widehat{T}_i^1 \textbf{ od}, \\
\qquad\qquad \vdots, \\
\textbf{var } l^n, d^n, k^n \bullet \\
\qquad [l^n, d^n, k^n := \underline{l^n}, \underline{d^n}, \underline{k^n} \mid p \ \wedge \ \underline{k^n} = k_0^n]; \\
\qquad \textbf{do } \Diamond_{i=1}^{m} \widehat{q}_i^n :: P_i; \widehat{T}_i^n \textbf{ od}
\end{array}\right) \downarrow (Id \times Id \times Q) \sqsubseteq
$$
$$
\left(\begin{array}{l}
\textbf{var } l^1, d^1, k^{1\prime} \bullet \\
\qquad [l^1, d^1, k^{1\prime} := \underline{l^1}, \underline{d^1}, \underline{k^{1\prime}} \mid p \ \wedge \ \underline{k^{1\prime}} = k_0^{1\prime}]; \\
\qquad \textbf{do } \Diamond_{i=1}^{m} \widehat{q}_i^1 :: P_i'; \widehat{T}_i^1 \textbf{ od}, \\
\qquad\qquad \vdots, \\
\textbf{var } l^n, d^n, k^{n\prime} \bullet \\
\qquad [l^n, d^n, k^{n\prime} := \underline{l^n}, \underline{d^n}, \underline{k^{n\prime}} \mid p \ \wedge \ \underline{k^{n\prime}} = k_0^{n\prime}]; \\
\qquad \textbf{do } \Diamond_{i=1}^{m} \widehat{q}_i^n :: P_i'; \widehat{T}_i^n \textbf{ od}
\end{array}\right)
$$

Applying the definitions of the encoding operator and refinement on tuples of statements, the goal can be reduced to n subgoals of the form:

$$
(P_1, ..., P_n) \downarrow (Id \times Id \times Q) \sqsubseteq (P_1', ..., P_n') \ \Rightarrow
$$
$$
\left(\begin{array}{l}
\textbf{var } l^j, d^j, k^j \bullet \\
\qquad [l^j, d^j, k^j := \underline{l^j}, \underline{d^j}, \underline{k^j} \mid p \ \wedge \ \underline{k^j} = k_0^j]; \\
\qquad \textbf{do } \Diamond_{i=1}^{m} \widehat{q}_i^j :: P_i; \widehat{T}_i^j \textbf{ od}
\end{array}\right) \downarrow (Id \times Id \times Q) \sqsubseteq
$$
$$
\left(\begin{array}{l}
\textbf{var } l^j, d^j, k^j \bullet \\
\qquad [l^j, d^j, k^{j\prime} := \underline{l^j}, \underline{d^j}, \underline{k^{j\prime}} \mid p \ \wedge \ \underline{k^{j\prime}} = k_0^{j\prime}]; \\
\qquad \textbf{do } \Diamond_{i=1}^{m} \widehat{q}_i^j :: P_i'; \widehat{T}_i^j \textbf{ od}
\end{array}\right)
$$

Proofs of these subgoals are similar to the proof of the Modular Reasoning Theorem for Modules in Chapter 3.[3] □

4.4 Is Forwarding Flexible?

The purpose of a software reuse technique is to permit multiple applications of a particular fragment of software packaged with the corresponding code reuse mechanism. As with most software reuse techniques, to a large extent,

[3]Note that this theorem could have been proved in terms of the theorem for modules, but we believe that this would introduce more confusion than clarity.

flexibility depends on the level of granularity. As compared to the technique based on modules, forwarding is more flexible, because it permits finer granularity of the code packaged for reuse. To illustrate this idea, let us consider the setting shown on figure 4.2(a). In this setting, the object $\mathcal{A}$ forwards to the object B and the client program C uses the resulting module. Suppose that a developer wants to alter the behavior of the module used by C so that the part B of this module behaves like B' instead. With modules, the developer would need to reimplement the entire module used by C, whereas with forwarding it will only be necessary to reimplement B'.

Figure 4.2(b) shows how forwarding can aid with reuse in another situation. Suppose that the module D captures the functionality expected by the client program C, and there exists an object B which almost conforms to this specification, but has a slightly different interface or lacks a minor part of the functionality. With modules, it would be impossible to reuse B, but forwarding permits a developer to construct the object $\mathcal{A}$ adjusting the interface or complementing the missing functionality. Objects such as $\mathcal{A}$ are known as *wrappers* and are pervasively used in object-oriented programming.

We do not intend to enumerate here all useful applications of the forwarding technique. However, there is one application which deserves to be mentioned, as it clearly illustrates the possibilities of programming with objects and the use of forwarding. This application of the forwarding technique is listed in the catalog of object-oriented design patterns [25] as the *Decorator* pattern.

Imagine that we are designing a graphical user interface toolkit in which properties like different borders and services like scrolling should all be freely attachable to any user interface component. One possible implementation

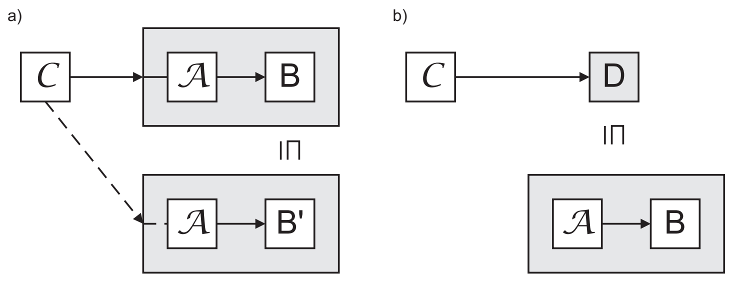

Figure 4.2: Flexibility of forwarding

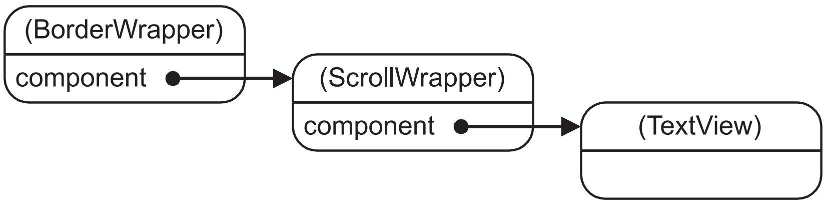

Figure 4.3: Object references in a program implementing Decorator

is to have all possible combinations of all user interface components and all borders and scrolling facilities, etc; but this might lead to a combinatorial explosion of the number of resulting objects.

There exists a more flexible way of achieving the goal. The user interface component represented by an object can be decorated with a border, a shadow, or a scroll bar by enclosing this object in another object that adds the corresponding decoration. The enclosing object (a wrapper) must be transparent for users of the object and should therefore support the original interface of the graphical user interface object. The transparency of wrappers permits constructing complex user interface objects that can have arbitrary combinations of borders, shadows, zooming and scrolling functionalities. Such a wrapper in the context of this pattern is referred to as *Decorator*. Figure 4.3 illustrates the composition of a TextView with a BorderWrapper and a ScrollWrapper to produce a bordered scrollable TextView.

4.5 Discussion and Related Work

The popularity of objects can be explained by the fact that they permit conceptualizing real world abstractions in a natural manner. This permits designing software systems uniformly, starting from a very abstract conceptual model of the system and gradually proceeding to its implementation, while adding new abstractions introducing additional functionality. New abstractions emerging during design do not necessarily directly correspond to real world entities, but may represent concepts existing only in the virtual world of a software system. Objects help focus a design on system structure rather than algorithms. The procedural style of programming follows a passive view of system design, in which passive data structures are modified by algorithms. The object-oriented style of programming promotes an active view, in which the functionality of the system is achieved through communication of active objects. Objects are naturally organized into hierarchical structures during analysis, design, and implementation. This hierarchical organization encourages the reuse of methods and data that are located higher in the hierarchy.

The need to better understand objects and their complex collaborations has initiated active research on foundations of object-oriented languages and systems. Most of the research concentrated on developing type theories for reasoning about correct syntactic compatibility of objects. An influential book [1] by Martín Abadi and Luca Cardelli systematically develops a type-theoretic model of objects and various object-oriented constructs and mechanisms. In general, the type-theoretic research significantly contributed to improving correctness of object-oriented software, by permitting compilers to find syntactic incompatibilities of objects. Ensuring syntactic compatibility is a prerequisite for establishing the complete compatibility of objects which, in turn, is the key issue in verifying correctness of object-oriented systems.

The forwarding technique is pervasive in object-oriented programming. In languages using prototypical objects forwarding is implemented as one of the main reuse mechanisms [15, 43]. In the component framework of BlackBox Component Builder by Oberon Microsystems [59] forwarding is used as a primary software reuse technique. Developers of Microsoft COM [68] went one step further and implemented forwarding as a reuse mechanism known as *containment*. Our study of the safety and flexibility of forwarding shows that the balance between the safety and flexibility of forwarding leans towards the safety. This, perhaps, explains the wide acceptance of forwarding as a primary reuse technique in open systems, such as object-oriented frameworks and component platforms.

Forwarding is also used for dealing with different kinds of problems that might obstruct reuse. For example, as we discussed above, an object with a slightly inappropriate interface can be used by a client employing forwarding. A special wrapper object, providing the appropriate interface and forwarding client messages to the reused object, should be created. In fact, it is even possible to slightly adjust the behavior of the reused object adopting it for particular client needs. In the formal literature on this topic, a similar technique is referred to as *interface refinement* [52] and it permits verifying behavioral conformance in such an interface transformation.

Chapter 5

Objects and Delegation

In this chapter we study *delegation* – a widely used software reuse technique relying on objects. The discussion and the model of objects presented in the previous chapter remain unchanged in this chapter.

Objects in real-life programs are composed via both delegation and forwarding. We present an example of this setting in section 5.8. As was already mentioned in the previous chapter, delegation is essentially similar to forwarding: when an object receives a message that it does not know how to process, it delegates the task of responding to this message to another object that knows how to handle it. Unlike in the case of forwarding, however, the object processing the message can send a message back to the original object, if the need arises. In other words, with delegation the structure of method calls on objects can be cyclic.

Delegation corresponds to the *pull* object communication pattern, with which a backward reference to an object initiating communication is established, e.g., by passing the reference to this object as one of method arguments, when invoking a method on another object. This other object then pulls all the data it needs from the original object by calling back its methods. Thus, in this chapter, we consider delegation in a broader sense, i.e., not only as a software reuse mechanism, but also as a pervasive pattern of object composition. In our opinion, the recursive structure of object references distinctly differentiates delegation from forwarding and, as we discuss in this chapter, dramatically influences its safety and flexibility.

This chapter is organized as follows. First, we present a model of delegation. We formulate a modular reasoning property for delegation and study whether it can be established in the general case in a manner feasible from

$$\begin{tikzcd}\boxed{\mathcal{C}} \arrow[r] & \boxed{\mathcal{A}} \arrow[r] & \boxed{\mathcal{B}} \arrow[ll, bend left]\end{tikzcd}$$

Figure 5.1: Delegation between three objects.

the perspective of practical programming. Clearly, definition of the modular reasoning property depends on the definition of refinement on objects.

First, we consider a simple definition of object refinement and prove that in theory this definition permits reasoning about delegating objects in a modular manner. We argue that this definition of object refinement is too restrictive to be of practical interest.

Next, we show that the intuitive definition of refinement used for ad-hoc reasoning about programs using delegation is not modular. The effects of such non-modular definitions of object refinement are known in practice to cause the *component re-entrance problem* [50]. We study the essence of the component re-entrance problem and formulate two requirements that should be respected to make the property hold. The contents of sections 5.1, 5.2, 5.4, and 5.5 are based on [50].

We give a definition of object refinement in context which accounts for our first requirement, and prove that if assumptions of the modular reasoning property are reinforced according to the second requirement, the property holds.

We then discuss the safety of delegation, consider how our results can be applied in practice for informal reasoning about delegation, and review the related literature.

In section 5.8, we discuss the flexibility of delegation and, finally, in section 5.9, we present some concluding remarks pertaining to delegation.

5.1 Modeling Delegation

Without loss of generality, we can consider the case when only two objects delegate messages to each other. To justify this statement, consider figure 5.1. All three objects $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ have the form as described in the previous chapter, e.g., $\mathcal{B} = (b_0, B)$, where b_0 is an initial value of the object's state and B is a function representing its methods. The object $\mathcal{A}$ invokes methods on $\mathcal{B}$, which invokes methods on $\mathcal{C}$, which, in turn, calls back on $\mathcal{A}$. We can define an operator **comb** that given two dependable objects returns a dependable object resulting from combining the two in the following manner:

$$(\mathcal{A}\ \mathbf{comb}\ \mathcal{B})\ \widehat{=}\ ((a_0, b_0), (A \circ B))$$

Thus, instead of considering delegation between the objects $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ we can consider delegation between the objects $(\mathcal{A}\ \mathbf{comb}\ \mathcal{B})$ and $\mathcal{C}$ that invoke each other's methods.

Let us now consider the case when two objects $\mathcal{A}$ and $\mathcal{B}$ participate in the delegation relation. Unlike in the case of forwarding, object $\mathcal{B}$ can invoke methods back on $\mathcal{A}$. Therefore, the objects $\mathcal{A}$ and $\mathcal{B}$ are defined in a manner similar to the definition of a dependable object in the previous chapter:

$$\begin{aligned}\mathcal{A}\ &\widehat{=}\ (a_0, A)\\ \mathcal{B}\ &\widehat{=}\ (b_0, B)\end{aligned}$$

where $a_0 : \Sigma$ and $b_0 : \Gamma$ are initial values of the internal states, and A and B are functions defined as follows:

$$\begin{aligned}A\ &\widehat{=}\ (\lambda Bb \bullet (ab_1, ..., ab_n)) : \Phi^{m,n}(Ptran(\Sigma \times \Delta \times \beta))\\ B\ &\widehat{=}\ (\lambda Ab \bullet (bb_1, ..., bb_m)) : \Phi^{n,m}(Ptran(\alpha \times \Delta \times \Gamma))\end{aligned}$$

Delegating messages from the object $\mathcal{A}$ to the object $\mathcal{B}$, from a mathematical standpoint, results in a module $(\mathcal{A}\ \mathbf{deleg}\ \mathcal{B})$ having all methods of $\mathcal{A}$ and $\mathcal{B}$ with all method calls to and from the corresponding counterpart object resolved. In other words, methods in $(\mathcal{A}\ \mathbf{deleg}\ \mathcal{B})$ do not contain method calls, as all the calls are substituted with the bodies of the called methods. The methods of $\mathcal{A}$ in the module resulting from delegating from $\mathcal{A}$ to $\mathcal{B}$ can be approximated by $A.\,B.\,\mathbf{Abort}$, where **Abort** is a tuple of **abort** statements. Using functional composition, this can be rewritten as $(A \circ B).\,\mathbf{Abort}$. The methods in such an approximation behave as the methods of $\mathcal{A}$ with all external calls redirected to $\mathcal{B}$, but with external calls of $\mathcal{B}$ aborting rather than going back to $\mathcal{A}$. Hence, a better approximation of the methods of $\mathcal{A}$ in the composed system would be $(A \circ B \circ A \circ B).\,\mathbf{Abort}$, and yet a better one $(A \circ B \circ A \circ B \circ A \circ B).\,\mathbf{Abort}$, etc. The desired result is then the limit of this sequence. This limit can be expressed as the least fixed point $(\mu\ A \circ B)$, which is the least Xb with respect to the refinement ordering on tuples of statements such that $Xb = (A \circ B).\,Xb$. Choosing the least fixed point means that a non-terminating sequence of calls from $\mathcal{A}$ to $\mathcal{B}$ and back is equivalent to **abort**, which is the meaning of a non-terminating loop.

According to the theorem of Knaster-Tarski [84], 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 function $A \circ B$ is monotonic in its argument, therefore, $(\mu\ A \circ B)$ exists and is unique. The methods of $\mathcal{B}$ in $(\mathcal{A}\ \mathbf{deleg}\ \mathcal{B})$ can be described in a similar manner.

The module resulting from the composition of the object $\mathcal{A}$ with the object $\mathcal{B}$ through delegation can be defined as follows:

$$(\mathcal{A}\,\mathbf{deleg}\,\mathcal{B})\ \widehat{=}\ ((a_0, b_0), (\mu\ A \circ B,\ \ \mu\ B \circ A))$$

Delegation is symmetric in the sense that delegating from $\mathcal{A}$ to $\mathcal{B}$ and delegating from $\mathcal{B}$ to $\mathcal{A}$ results in the same module.

Note that during composition, the type variables α and β, representing unknown state spaces of the objects $\mathcal{B}$ and $\mathcal{A}$, get instantiated with Σ and Γ respectively, so that the composed system has methods operating on the state space $\Sigma \times \Delta \times \Gamma$.

5.2 Safety of Delegation

Let us consider how we can formulate the corresponding modular reasoning property for delegating objects. Suppose that we have two delegating objects $\mathcal{A}$ and $\mathcal{B}$, invoking each other's methods. Ultimately, developers of revised objects $\mathcal{A}'$ and $\mathcal{B}'$ would like to achieve that the module resulting from composing these objects via delegation be a refinement of the composition of the original objects $\mathcal{A}$ and $\mathcal{B}$, namely,

$$(\mathcal{A}\,\mathbf{deleg}\,\mathcal{B})\ \textit{is refined by}\ (\mathcal{A}'\,\mathbf{deleg}\,\mathcal{B}')$$

To provide for modular reasoning, developers of the revised objects $\mathcal{A}'$ and $\mathcal{B}'$ should be able to establish this goal only by considering the original objects $\mathcal{A}$ and $\mathcal{B}$, as illustrated in figure 5.2. Therefore, for the case of delegation, the *modular reasoning property* looks as follows:

$$\begin{array}{ll} \textit{if} & \mathcal{A}\ \textit{is refined by}\ \mathcal{A}'\ \textit{and} \\ & \mathcal{B}\ \textit{is refined by}\ \mathcal{B}' \\ \textit{then} & (\mathcal{A}\,\mathbf{deleg}\,\mathcal{B})\ \textit{is refined by}\ (\mathcal{A}'\,\mathbf{deleg}\,\mathcal{B}') \end{array}$$

Obviously, the correctness of the formula above (and therefore of the safety of delegation) depends on the definition of refinement on objects

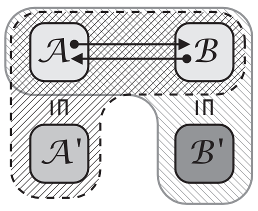

Figure 5.2: Modular reasoning for delegating objects.

and modules. As we mentioned in the previous section, composing objects through delegation returns a module. The refinement on modules was defined in section 3.1. We can slightly adjust the definition of module refinement to account for the fact that modules are created by the composition of objects. We say that the module ($\mathcal{A}\,\textbf{deleg}\,\mathcal{B}$) is refined by the module ($\mathcal{A}'\,\textbf{deleg}\,\mathcal{B}'$), if there exist such relations R and P that initial values of these modules are related via the relation $R \times P$ and lists of method bodies are related via the relation $R \times Id \times P$. Formally, we have:

$$\begin{array}{l}
(\mathcal{A}\,\textbf{deleg}\,\mathcal{B}) \sqsubseteq (\mathcal{A}'\,\textbf{deleg}\,\mathcal{B}') \;\widehat{=} \\
\quad (\exists R, P \bullet (R \times P).\,(a_0', b_0').\,(a_0, b_0) \;\wedge \\
\qquad (\mu\; A \circ B){\downarrow}(R \times Id \times P) \sqsubseteq (\mu\; A' \circ B') \;\wedge \\
\qquad (\mu\; B \circ A){\downarrow}(R \times Id \times P) \sqsubseteq (\mu\; B' \circ A'))
\end{array}$$

In order to be able to study delegation mathematically, we make a number of restrictions on the object model. In particular, we have assumed that objects do not have self-calls and revised objects do not introduce new methods. We would like to note, that while these restrictions of the object model are important from the perspective of practical programming, they do not reduce the generality of our results. Relaxing these restrictions would not invalidate any of our conclusions, but would only introduce additional verification obligations.

We proceed by considering different definitions of refinement on dependable objects and study theoretical and practical consequences of these definitions. First, we consider a simple definition of object refinement.

5.3 Safety: A Simple Definition of Object Refinement

Refinement on objects can be defined in a straightforward manner. We say that the object $\mathcal{A} = (a_0 : \Sigma, A : \Phi^{m,n}(\mathit{Ptran}(\Sigma \times \Delta \times \beta)))$ is refined by the object $\mathcal{A}' = (a'_0 : \Sigma', A' : \Phi^{m,n}(\mathit{Ptran}(\Sigma' \times \Delta \times \beta)))$, if there exists a relation $R : \Sigma' \leftrightarrow \Sigma$ such that this relation holds between the initial values, and data refinement via the relation $R \times \mathit{Id} \times \mathit{Id}$ holds between the lists of method bodies constructed by applying A and A' to the list of method bodies of an arbitrary object $\mathcal{X}$. Formally,

$$\begin{array}{l} \mathcal{A} \sqsubseteq \mathcal{A}' \;\widehat{=}\; \exists R \cdot (R.\, a'_0.\, a_0) \;\wedge \\ \qquad \forall Xb \bullet (A.\, Xb){\downarrow}(R \times \mathit{Id} \times \mathit{Id}) \sqsubseteq A'.\, Xb{\downarrow}(R \times \mathit{Id} \times \mathit{Id}) \end{array}$$

5.3.1 Modular Reasoning

Let us now consider how the modular reasoning property can be formalized and proved using the simple definition of object refinement. In the proof of the theorem, we use the following lemma.

Encoding Propagation Lemma. *For object methods $A : \Phi^{m,n}(\mathit{Ptran}(\Sigma \times \Delta \times \beta))$, $B : \Phi^{n,m}(\mathit{Ptran}(\alpha \times \Delta \times \Gamma))$, $A' : \Phi^{m,n}(\mathit{Ptran}(\Sigma' \times \Delta \times \beta))$, and $B' : \Phi^{n,m}(\mathit{Ptran}(\alpha \times \Delta \times \Gamma'))$ and relations $R : \Sigma' \leftrightarrow \Sigma$ and $P : \Gamma' \leftrightarrow \Gamma$, the following holds*:

$$\begin{array}{rcl} (A.\, Xb){\downarrow}(\mathit{Id} \times \mathit{Id} \times P) & \sqsubseteq & A.\, Xb{\downarrow}(\mathit{Id} \times \mathit{Id} \times P) \\ (B.\, Yb){\downarrow}(R \times \mathit{Id} \times \mathit{Id}) & \sqsubseteq & B.\, Yb{\downarrow}(R \times \mathit{Id} \times \mathit{Id}) \\ (A'.\, Xb){\downarrow}(\mathit{Id} \times \mathit{Id} \times P) & \sqsubseteq & A'.\, Xb{\downarrow}(\mathit{Id} \times \mathit{Id} \times P) \\ (B'.\, Yb){\downarrow}(R \times \mathit{Id} \times \mathit{Id}) & \sqsubseteq & B'.\, Yb{\downarrow}(R \times \mathit{Id} \times \mathit{Id}) \end{array}$$

Proof The proof of this lemma is similar to the proof of Modular Reasoning Theorem for Forwarding. □

Theorem. *Let objects $\mathcal{A}$, $\mathcal{B}$, $\mathcal{A}'$,* and *$\mathcal{B}'$ be given as follows*:

$$\begin{array}{rcl} \mathcal{A} & = & (a_0 : \Sigma, A : \Phi^{m,n}(\mathit{Ptran}(\Sigma \times \Delta \times \beta))) \\ \mathcal{B} & = & (b_0 : \Gamma, B : \Phi^{n,m}(\mathit{Ptran}(\alpha \times \Delta \times \Gamma))) \\ \mathcal{A}' & = & (a'_0 : \Sigma', A' : \Phi^{m,n}(\mathit{Ptran}(\Sigma' \times \Delta \times \beta))) \\ \mathcal{B}' & = & (b'_0 : \Gamma', B' : \Phi^{n,m}(\mathit{Ptran}(\alpha \times \Delta \times \Gamma'))) \end{array}$$

Then we have:

$$\mathcal{A} \sqsubseteq \mathcal{A}' \;\wedge\; \mathcal{B} \sqsubseteq \mathcal{B}' \;\Rightarrow\; (\mathcal{A}\,\mathbf{deleg}\,\mathcal{B}) \sqsubseteq (\mathcal{A}'\,\mathbf{deleg}\,\mathcal{B}')$$

Proof According to the definitions of refinement on objects and modules, our goal can be rewritten as follows:

$$\begin{array}{l}
(\exists R \cdot (R.\, a_0'.\, a_0) \;\wedge \\
\quad \forall Xb \bullet (A.\, Xb){\downarrow}(R \times Id \times Id) \sqsubseteq A'.\, Xb{\downarrow}(R \times Id \times Id)) \;\wedge \\
(\exists P \cdot (P.\, b_0'.\, b_0) \;\wedge \\
\quad \forall Yb \bullet (B.\, Yb){\downarrow}(Id \times Id \times P) \sqsubseteq B'.\, Yb{\downarrow}(Id \times Id \times P)) \;\Rightarrow \\
\qquad (\exists R, P \cdot \;\; (R \times P).\, (a_0', b_0').\, (a_0, b_0) \;\wedge \\
\qquad\qquad (\mu\; A\circ B){\downarrow}(R \times Id \times P) \sqsubseteq (\mu\; A'\circ B') \;\wedge \\
\qquad\qquad (\mu\; B\circ A){\downarrow}(R \times Id \times P) \sqsubseteq (\mu\; B'\circ A'))
\end{array}$$

After some logical transformations, we have two subgoals to prove. The first one states that, under the assumptions that the initial values of $\mathcal{A}$ and $\mathcal{B}$ are connected by arbitrary relations R and P, initial values of the resulting modules will be connected by the compositions of these relations, namely,

$$R.\, a_0'.\, a_0 \;\wedge\; P.\, b_0'.\, b_0 \;\Rightarrow\; (R \times P).\, (a_0', b_0').\, (a_0, b_0)$$

The proof of this subgoal follows directly from the definition of relational product.

The second subgoal states that, under the assumptions that methods of $\mathcal{A}$ are data refined by methods of $\mathcal{A}'$ via the relation $(R \times Id \times Id)$ and methods of $\mathcal{B}$ are data refined by methods of $\mathcal{B}'$ via the relation $(Id \times Id \times P)$, methods of the resulting modules are data refined through the relation $(R \times Id \times P)$. The goal we have to prove is as follows:

$$\begin{array}{l}
\forall Xb \bullet (A.\, Xb){\downarrow}(R \times Id \times Id) \sqsubseteq A'.\, Xb{\downarrow}(R \times Id \times Id) \;\wedge \\
\forall Yb \bullet (B.\, Yb){\downarrow}(Id \times Id \times P) \sqsubseteq B'.\, Yb{\downarrow}(Id \times Id \times P) \;\Rightarrow \\
\quad (\mu\; A\circ B){\downarrow}(R \times Id \times P) \sqsubseteq (\mu\; A'\circ B') \;\wedge \\
\quad (\mu\; B\circ A){\downarrow}(R \times Id \times P) \sqsubseteq (\mu\; B'\circ A')
\end{array}$$

According to the classical fixed point theory [84], if two functions on a complete lattice are in the partial order relation, their fixed points preserve this partial order. Therefore, we need to prove that under the assumptions as in the goal above functions $A\circ B$ and $B\circ A$ are data refined with respect to the relation $R \times Id \times P$, i.e.,

$$\begin{array}{l}
\forall Xb \bullet (A.\,Xb)\!\downarrow\!(R \times Id \times Id) \sqsubseteq A'.\,Xb\!\downarrow\!(R \times Id \times Id) \;\wedge \\
\forall Yb \bullet (B.\,Yb)\!\downarrow\!(Id \times Id \times P) \sqsubseteq B'.\,Yb\!\downarrow\!(Id \times Id \times P) \;\Rightarrow \\
\quad \forall Yb \bullet ((A \circ B).\,Yb)\!\downarrow\!(R \times Id \times P) \sqsubseteq \\
\qquad (A' \circ B').\,Yb\!\downarrow\!(R \times Id \times P)
\end{array}$$

$$\begin{array}{l}
\forall Xb \bullet (A.\,Xb)\!\downarrow\!(R \times Id \times Id) \sqsubseteq A'.\,Xb\!\downarrow\!(R \times Id \times Id) \;\wedge \\
\forall Yb \bullet (B.\,Yb)\!\downarrow\!(Id \times Id \times P) \sqsubseteq B'.\,Yb\!\downarrow\!(Id \times Id \times P) \;\Rightarrow \\
\quad \forall Xb \bullet ((B \circ A).\,Xb)\!\downarrow\!(R \times Id \times P) \sqsubseteq \\
\qquad (B' \circ A').\,Xb\!\downarrow\!(R \times Id \times P)
\end{array}$$

To prove the first of these subgoals, we assume its antecedent and prove the consequent. For arbitrary Ab, the consequent can be rewritten to

$$((A \circ B).\,Ab)\!\downarrow\!(R \times Id \times P) \sqsubseteq (A' \circ B').\,Ab\!\downarrow\!(R \times Id \times P)$$

We start with the left-hand side of the consequent and refine it to the right-hand side:

$$\begin{array}{ll}
 & ((A \circ B).\,Ab)\!\downarrow\!(R \times Id \times P) \\
= & \{\textit{definition of composition of functions, rule 1.11}\} \\
 & (A.\,(B.\,Ab))\!\downarrow\!(Id \times Id \times P)\!\downarrow\!(R \times Id \times Id) \\
\sqsubseteq & \{\textit{encoding propagation lemma}\} \\
 & (A.\,(B.\,Ab)\!\downarrow\!(Id \times Id \times P))\!\downarrow\!(R \times Id \times Id) \\
\sqsubseteq & \{\textit{second assumption}\} \\
 & (A.\,(B'.\,Ab\!\downarrow\!(Id \times Id \times P)))\!\downarrow\!(R \times Id \times Id) \\
\sqsubseteq & \{\textit{first assumption}\} \\
 & A'.\,(B'.\,Ab\!\downarrow\!(Id \times Id \times P))\!\downarrow\!(R \times Id \times Id) \\
\sqsubseteq & \{\textit{encoding propagation lemma}\} \\
 & A'.\,B'.\,Ab\!\downarrow\!(Id \times Id \times P)\!\downarrow\!(R \times Id \times Id) \\
= & \{\textit{definition of composition of functions, rule 1.11}\} \\
 & (A' \circ B').\,Ab\!\downarrow\!(R \times Id \times P)
\end{array}$$

The proof of the second subgoal is analogous. $\Box$

5.3.2 Discussion

Unfortunately, this definition of refinement is too restrictive to be used in practice. According to this definition, a developer of a revised object is not allowed to look at the definitions of other objects, neither the revised definitions nor even the original ones. In fact, the developers can only refine bodies of methods around method invocations, without being able either to remove or introduce method invocations, or to assume anything about the called methods.

For the definition of object refinement to be useful in practice, it should allow for making assumptions about the context in which the revised object will operate. When an object $\mathcal{A}$ delegates to an object $\mathcal{B}$, the latter can be seen as a specification of the context in which the revised object $\mathcal{A}'$ will operate. Since delegation is symmetric, $\mathcal{A}$ can also be seen as a specification of the context for the revised object $\mathcal{B}'$. Accordingly, a practically useful *modular reasoning property* can be formulated as follows:

$$\begin{array}{ll} \textit{if} & \mathcal{A} \textit{ is refined by } \mathcal{A}' \textit{ in context of } \mathcal{B} \textit{ and} \\ & \mathcal{B} \textit{ is refined by } \mathcal{B}' \textit{ in context of } \mathcal{A} \\ \textit{then} & (\mathcal{A}\,\mathbf{deleg}\,\mathcal{B}) \textit{ is refined by } (\mathcal{A}'\,\mathbf{deleg}\,\mathcal{B}') \end{array}$$

In the following section, we consider an ad-hoc definition of object refinement which is often used in practical programming while reasoning informally about delegating objects. First, we consider an example illustrating such ad-hoc reasoning and the component re-entrance problem [50] that it can induce. Then we analyze this problem and formulate requirements which should be taken into account while reasoning about delegating objects.

5.4 Safety: An Ad-hoc Definition of Object Refinement

To illustrate the way informal reasoning is done in practice, let us consider the following example.

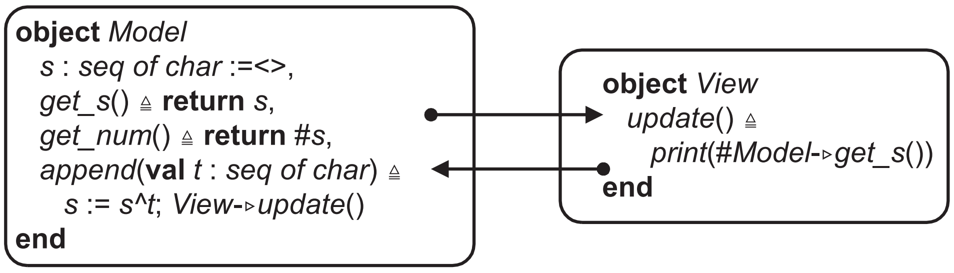

Figure 5.3: Specification of the Model-View object system. The operator # returns the length of a sequence and the operator $\frown$ concatenates two sequences.

5.4.1 An Example of Ad-hoc Informal Reasoning Used in Practice

The example in figure 5.3 follows the Observer pattern [25], which allows separating the presentational aspects of the user interface from the underlying application data, by defining two objects *Model* and *View*. The objects *Model* and *View* delegate messages to each other. Note that we deliberately abstract away from the mechanism by which mutual reference between objects can be achieved, because a particular reference mechanism is irrelevant for the presentation of the problem. For example, such mutual reference can be achieved by passing pointers to objects as method parameters.

The object *Model* maintains a string s, represented by a sequence of characters and initialized with an empty sequence. Every time a new string is appended to the string in *Model*, the method *update* of *View* is called. In turn, *update* calls back *Model*'s *get_s*() method and prints out the number of elements in the received string.

Suppose that a developer decides to improve the efficiency of the object *Model*. To avoid counting characters in the method *get_num*, the developer introduces an integer attribute n to represent the number of characters in the sequence. Accordingly, an object *Model′* is implemented as follows:

```
**object** $Model'$
    $s : \ seq \ of \ char := \langle\rangle,$
    $n : int := 0,$
    $get\_s() \ \widehat{=} \ \mathbf{return} \ s,$
```

```
        $get\_num() \;\widehat{=}\; \textbf{return}\; n,$
        $append(\textbf{val}\; t : \; seq\ of\ char) \;\widehat{=}$
            $s := s ^\frown t;\ View \text{-}\!\triangleright update(); n := n + \#t$
    end
```

Note that the developer made an assumption about the context in which *Model′* will be used. In particular, in *append* the developer assumed that to retrieve the number of characters in s the object *View* will invoke the method *get_s* back on *Model′*.[1] This implementation of the method *append* appears to be perfectly valid. In fact, updating the screen as early as possible is a reasonable policy.[2]

Now suppose that the developer decides to revise the object *View* to construct an object *View′*. To make such a revision in a modular manner, the developer can make assumptions about the future context of *View′* based only on the original definition of *Model*. To avoid passing a sequence of characters as a parameter, the developer implements the method *update* to invoke the method *get_num* of *Model*:

```
    object $View'$
        $update() \;\widehat{=}\; print(Model \text{-}\!\triangleright get\_num())$
    end
```

Here the developer faces a problem. Even though the revised objects *Model′* and *View′* appear to reimplement the original objects correctly, the composition of the revised objects behaves incorrectly: the number of elements in the string s that *update* prints out is wrong. Further on we refer to this problem as *the component re-entrance problem.* This name can be justified by the following observation: the problem occurs when a thread of control leaves an object (with the method invocation on another object) and then re-enters the initial object (with the call-back on this object). Traditionally, the source of the problem is attributed to the fact that an intermediary state of the initial object can be observed by others through re-entrance before it reaches consistency.

[1]In the absence of this assumption the implementation of *append* in *Model′* is rather meaningless, as the other option of retrieving the number of characters in s is by calling *get_num*.

[2]A discussion of the well-known recommendation *to always re-establish an object's invariant before invoking an external method* follows in section 5.6.

5.4.2 The Essence of The Component Re-entrance Problem

As we have already mentioned, when reasoning about the conformance of the object $\mathcal{A}'$ to the object $\mathcal{A}$, the developers need to make assumptions about the behavior of the object(s) to which $\mathcal{A}$ delegates messages. The ad-hoc method for taking such assumptions into account is to reason about the refinement between the results of composing $\mathcal{A}$ with its context $\mathcal{B}$ and the revised object $\mathcal{A}'$ with the same context. Therefore, the intuitive definition of object refinement in context that developers informally used in the example is

$$\mathcal{A} \overset{\mathcal{B}}{\preccurlyeq} \mathcal{A}' \;\widehat{=}\; (\mathcal{A}\,\mathbf{deleg}\,\mathcal{B}) \sqsubseteq (\mathcal{A}'\,\mathbf{deleg}\,\mathcal{B}),$$

where $\mathcal{B}$ is a known object. This definition permits making assumptions about the entire possible context of an object while refining it.

Unfortunately, this definition does not support modular reasoning, as was demonstrated by the previous example. In other words, the modular reasoning property does not hold, i.e.,

$$\mathcal{A} \overset{\mathcal{B}}{\preccurlyeq} \mathcal{A}' \;\wedge\; \mathcal{B} \overset{\mathcal{A}}{\preccurlyeq} \mathcal{B}' \;\not\Rightarrow\; (\mathcal{A}\,\mathbf{deleg}\,\mathcal{B}) \sqsubseteq (\mathcal{A}'\,\mathbf{deleg}\,\mathcal{B}')$$

We believe that this fact constitutes the essence of the component re-entrance problem. The problem occurs due to the conflict of assumptions the developers of objects make about the behavior of other objects in the system. In the previous example, the developer of the object *View'* assumed that at the moment when the method *update* was called the invariant of the implementation of *Model* would have held. Similarly, when developing *Model'*, the developer assumed that it was not necessary to establish the invariant before invoking *update*, because the definition of this method in *Model* did not rely on it. These conflicting assumptions led to the problem after composition.

This consideration brings us to the question how, while developing an object, one can make assumptions about the behavior of the other objects participating in delegation in a consistent manner.

5.4.3 Restricting Assumptions About Context

In order to establish the modular reasoning property, it is necessary to restrict the assumptions that object developers can make about the context in which the object is going to operate.

No Call-Back Assumptions Requirement

To identify the first restriction that should be imposed on the assumptions about the object context, let us consider a counter example invalidating the modular reasoning property.

$$\begin{array}{ll}
\begin{array}{l}
\textbf{object } \mathcal{A}_1 \\
\quad m_1(\textbf{valres } x : \mathit{int}) \mathrel{\widehat{=}} \{x > 5\}; x := 5, \\
\quad m_2(\textbf{valres } x : \mathit{int}) \mathrel{\widehat{=}} \{x > 0\}; x := 5 \\
\textbf{end}
\end{array}
&
\begin{array}{l}
\textbf{object } \mathcal{B}_1 \\
\quad n(\textbf{valres } x : \mathit{int}) \mathrel{\widehat{=}} \\
\qquad \mathcal{A} \rightarrowtriangle m_1(x) \\
\textbf{end}
\end{array}
\\
\begin{array}{l}
\textbf{object } \mathcal{A}_2 \\
\quad m_1(\textbf{valres } x : \mathit{int}) \mathrel{\widehat{=}} \{x > 0\}; x := 5, \\
\quad m_2(\textbf{valres } x : \mathit{int}) \mathrel{\widehat{=}} \mathcal{B} \rightarrowtriangle n(x) \\
\textbf{end}
\end{array}
&
\begin{array}{l}
\textbf{object } \mathcal{B}_2 \\
\quad n(\textbf{valres } x : \mathit{int}) \mathrel{\widehat{=}} \\
\qquad \{x > 5\}; x := 5 \\
\textbf{end}
\end{array}
\end{array}$$

If we expand the bodies of the method m_2 in the composed systems, then we have:

$$\begin{array}{ll}
(\mathcal{A}_1 \,\textbf{deleg}\, \mathcal{B}_1) :: m_2 = & (\mathcal{A}_2 \,\textbf{deleg}\, \mathcal{B}_1) :: m_2 = \\
\quad \{x > 0\}; x := 5 & \quad \{x > 0\}; x := 5 \\
(\mathcal{A}_1 \,\textbf{deleg}\, \mathcal{B}_2) :: m_2 = & (\mathcal{A}_2 \,\textbf{deleg}\, \mathcal{B}_2) :: m_2 = \\
\quad \{x > 0\}; x := 5 & \quad \{x > 5\}; x := 5
\end{array}$$

where :: selects a method of the corresponding module. Therefore, we have:

$$\begin{array}{lcl}
(\mathcal{A}_1 \,\textbf{deleg}\, \mathcal{B}_1) :: m_2 & \sqsubseteq & (\mathcal{A}_2 \,\textbf{deleg}\, \mathcal{B}_1) :: m_2 \text{ and} \\
(\mathcal{A}_1 \,\textbf{deleg}\, \mathcal{B}_1) :: m_2 & \sqsubseteq & (\mathcal{A}_1 \,\textbf{deleg}\, \mathcal{B}_2) :: m_2
\end{array}$$

However, it is not the case that

$$(\mathcal{A}_1 \,\textbf{deleg}\, \mathcal{B}_1) :: m_2 \sqsubseteq (\mathcal{A}_2 \,\textbf{deleg}\, \mathcal{B}_2) :: m_2$$

Due to the presence of assertions, the precondition $x > 5$ of $(\mathcal{A}_2 \,\textbf{deleg}\, \mathcal{B}_2) :: m_2$ is stronger than the precondition $x > 0$ of $(\mathcal{A}_1 \,\textbf{deleg}\, \mathcal{B}_1) :: m_2$, while to preserve refinement, preconditions can only be weakened.

This example motivates us to formulate the following requirement.

"No call-back assumptions":

> *While developing a revised implementation of a method, revised implementations of other methods in the same object cannot be assumed; their definitions in the original object should be considered instead.*

As the behavior of the object serving as a context depends on the behavior of the object under consideration, assuming that the context object is going to call back on the refined object would implicitly modify the specification.

No Infinite Recursion Requirement

However, there exists another aspect of the component re-entrance problem that cannot be handled by simply restricting the context for refinement. The following rather trivial example illustrates this aspect of the problem:

object $\mathcal{A}_1$
  $m(\mathbf{res}\ r : int)\ \widehat{=}\ r := 5$
end

object $\mathcal{A}_2$
  $m(\mathbf{res}\ r : int)\ \widehat{=}\ \mathcal{B}{\rightarrowtriangle}n(r)$
end

object $\mathcal{B}_1$
  $n(\mathbf{res}\ r : int)\ \widehat{=}\ r := 5$
end

object $\mathcal{B}_2$
  $n(\mathbf{res}\ r : int)\ \widehat{=}\ \mathcal{A}{\rightarrowtriangle}m(r)$
end

It is easy to see that a call to the method m in the object $\mathcal{A}_2\ \mathbf{deleg}\,\mathcal{B}_2$ leads to a never terminating recursion of method invocations. Obviously, such a behavior does not refine the behavior of the original system. In fact, a similar problem was described by Carroll Morgan in [54]. He mentions that in the case of mutually dependent modules, their independent refinements can accidentally introduce mutual recursion. Based on the example, we formulate the following requirement:

"No infinite recursion":

> *The invocation of a method on the revised object participating in delegation should not lead to an infinite recursion of method invocations between the revised delegating objects.*

5.5 Safety: A Definition of Context Object Refinement

Let us now consider how we can define object refinement in context, taking into account the "no call-back assumptions" requirement. As stipulated by this requirement, when refining the object $\mathcal{A}$ to $\mathcal{A}'$, we should not assume that the object $\mathcal{B}$ calls back methods of $\mathcal{A}'$, because in doing so we would implicitly modify the definition of $\mathcal{B}$. The method bodies of $\mathcal{A}$ are mathematically defined by $(\mu\ A \circ B)$, whereas the method bodies of $\mathcal{B}$ are defined by $(\mu\ B \circ A)$. Relying on the fact that $\mathcal{B}$ calls back methods of $\mathcal{A}$ amounts to considering A' applied to the method bodies $(\mu\ B \circ A)$ rather than considering A' applied to $(\mu\ B \circ A')$. Taking into account the "no call-back assumptions" requirement, refinement between the methods of $\mathcal{A}$ and the methods of $\mathcal{A}'$ in context of $\mathcal{B}$ can then be defined in terms of the refinement between $(\mu\ A \circ B)$ and A' applied to $(\mu\ B \circ A)$.

Let $A : \Phi^{m,n}(\mathit{Ptran}(\Sigma \times \Delta \times \beta))$ and $A' : \Phi^{m,n}(\mathit{Ptran}(\Sigma' \times \Delta \times \beta))$ be methods of objects $\mathcal{A}$ and $\mathcal{A}'$, respectively. Data refinement between A and A' in the context of $\mathcal{B}$ via a relation $R : \Sigma' \leftrightarrow \Sigma$ is denoted by $A \overset{\mathcal{B}}{\sqsubseteq}_R A'$ and defined as follows:

$$A \overset{\mathcal{B}}{\sqsubseteq}_R A' \ \widehat{=}\ (\mu\ A \circ B){\downarrow}(R \times \mathit{Id} \times \mathit{Id}) \sqsubseteq A'.(\mu\ B \circ A){\downarrow}(R \times \mathit{Id} \times \mathit{Id})$$

Here the encoding operators are necessary for adjusting the state spaces of the participating objects. For methods of objects $\mathcal{B}$ and $\mathcal{B}'$, we have a similar definition but via a relation $\mathit{Id} \times \mathit{Id} \times P$, with $P : \Gamma' \leftrightarrow \Gamma$.

We say that $\mathcal{A} = (a_0 : \Sigma, A : \Phi^{m,n}(\mathit{Ptran}(\Sigma \times \Delta \times \beta)))$ is refined by $\mathcal{A}' = (a_0' : \Sigma', A' : \Phi^{m,n}(\mathit{Ptran}(\Sigma' \times \Delta \times \beta)))$ in the context of $\mathcal{B}$, if there exists a relation $R : \Sigma' \leftrightarrow \Sigma$ such that this relation holds between the initial values, and methods of $\mathcal{A}$ are data refined by methods of $\mathcal{A}'$ in the context of $\mathcal{B}$ via the relation $R \times \mathit{Id} \times \mathit{Id}$. Formally,

$$\mathcal{A} \overset{\mathcal{B}}{\sqsubseteq} \mathcal{A}' \ \widehat{=}\ (\exists R \bullet (R.\,a_0'.\,a_0) \ \wedge\ A \overset{\mathcal{B}}{\sqsubseteq}_R A')$$

For the objects $\mathcal{B} = (b_0 : \Gamma, B : \Phi^{n,m}(\mathit{Ptran}(\alpha \times \Delta \times \Gamma))$ and $\mathcal{B}' = (b_0' : \Gamma', B' : \Phi^{n,m}(\mathit{Ptran}(\alpha \times \Delta \times \Gamma'))$, the definition of refinement is similar, only that the initial values are connected via a relation $P : \Gamma' \leftrightarrow \Gamma$ and the lists of method bodies are connected via $\mathit{Id} \times \mathit{Id} \times P$.

5.5.1 Modular Reasoning Theorem

Our objective is to prove that the modular reasoning property holds for objects delegating messages to each other if the "no call-back assumptions" and "no infinite recursion" requirements are satisfied.

As we have just demonstrated the "no call-back assumptions" requirement is captured in the definition of context object refinement. To be able to take into account the "no infinite recursion" requirement, we need to strengthen assumptions of the modular reasoning property with additional assumptions. This requirement can be captured with the following formula:

$$\exists n \bullet \forall Yb \bullet (A' \circ B')^n .\ Yb \sqsubseteq (\mu\ A' \circ B')$$

In this formula, $(A' \circ B')^n$ is the function resulting from composing the function $(A' \circ B')$ with itself $n-1$ times. The intuition behind this formula is as follows: if the result of applying the function $(A' \circ B')$ to an arbitrary list of method bodies a finite number of times is refined by the result of the complete unfolding of method invocations between $\mathcal{A}'$ and $\mathcal{B}'$, then the bodies of methods of A' are completely defined. This can only be achieved if the unfolding terminates, i.e., there is no infinite mutual recursion.

Now we are all set for formulating and proving the modular reasoning property for delegating objects.

Modular Reasoning Theorem for Delegation. *Let objects $\mathcal{A}$, $\mathcal{B}$, $\mathcal{A}'$, and $\mathcal{B}'$ be given as follows*:

$$\begin{array}{rcl} \mathcal{A} & = & (a_0 : \Sigma, A : \Phi^{m,n}(Ptran(\Sigma \times \Delta \times \beta))) \\ \mathcal{B} & = & (b_0 : \Gamma, B : \Phi^{n,m}(Ptran(\alpha \times \Delta \times \Gamma))) \\ \mathcal{A}' & = & (a_0' : \Sigma', A' : \Phi^{m,n}(Ptran(\Sigma' \times \Delta \times \beta))) \\ \mathcal{B}' & = & (b_0' : \Gamma', B' : \Phi^{n,m}(Ptran(\alpha \times \Delta \times \Gamma'))) \end{array}$$

Then we have:

$$\begin{array}{lr} \mathcal{A} \overset{\mathcal{B}}{\sqsubseteq} \mathcal{A}' \ \wedge & (a) \\ \mathcal{B} \overset{\mathcal{A}}{\sqsubseteq} \mathcal{B}' \ \wedge & (b) \\ (\exists k \bullet \forall Yb \bullet (A' \circ B')^k .\ Yb \sqsubseteq (\mu\ A' \circ B')) \ \wedge & (c) \\ (\exists l \bullet \forall Xb \bullet (B' \circ A')^l .\ Xb \sqsubseteq (\mu\ B' \circ A')) \ \Rightarrow & (d) \\ \quad (\mathcal{A}\, \mathbf{deleg}\, \mathcal{B}) \sqsubseteq (\mathcal{A}'\, \mathbf{deleg}\, \mathcal{B}') & \end{array}$$

Proof Expanding the definitions and making simple logical transformations, we get three subgoals:

$$\begin{array}{l} (R.\, a_0'.\, a_0) \ \wedge \ (P.\, b_0'.\, b_0) \ \Rightarrow \\ \qquad (R \times P).\, (a_0', b_0').\, (a_0, b_0) \ \wedge \ (P \times R).\, (b_0', a_0').\, (b_0, a_0) \end{array} \tag{5.1}$$

$$\begin{array}{l} A \overset{\mathcal{B}}{\sqsubseteq}_R A' \ \wedge \ B \overset{\mathcal{A}}{\sqsubseteq}_P B' \ \wedge \ (c) \ \wedge \ (d) \Rightarrow \\ \qquad (\mu\ A \circ B){\downarrow}(R \times Id \times P) \sqsubseteq (\mu\ A' \circ B') \end{array} \tag{5.2}$$

$$\begin{array}{l} A \overset{\mathcal{B}}{\sqsubseteq}_R A' \ \wedge \ B \overset{\mathcal{A}}{\sqsubseteq}_P B' \ \wedge \ (c) \ \wedge \ (d) \Rightarrow \\ \qquad (\mu\ B \circ A){\downarrow}(R \times Id \times P) \sqsubseteq (\mu\ B' \circ A') \end{array} \tag{5.3}$$

where R and P are fixed but arbitrary relations. The first subgoal is obviously true. To prove the second and the third subgoals, we first prove the following lemma.

Lemma. *For functions $A : \Phi^{m,n}(Ptran(\Sigma \times \Delta \times \beta))$, $B : \Phi^{n,m}(Ptran(\alpha \times \Delta \times \Gamma))$, $A' : \Phi^{m,n}(Ptran(\Sigma' \times \Delta \times \beta))$, and $B' : \Phi^{n,m}(Ptran(\alpha \times \Delta \times \Gamma'))$ defined as above, relations $R : \Sigma' \leftrightarrow \Sigma$ and $P : \Gamma' \leftrightarrow \Gamma$, and any natural number k, we have*:

$$\begin{array}{l} A \overset{\mathcal{B}}{\sqsubseteq}_R A' \ \wedge \ B \overset{\mathcal{A}}{\sqsubseteq}_P B' \ \Rightarrow \\ \qquad (\mu\ A \circ B){\downarrow}(R \times Id \times P) \sqsubseteq (A' \circ B')^k.\,(\mu\ A \circ B){\downarrow}(R \times Id \times P) \end{array}$$

Proof We prove this lemma by induction over k.
Base case:

$$\begin{array}{ll} & (A' \circ B')^0.\,(\mu\ A \circ B){\downarrow}(R \times Id \times P) \\ = & \{\textit{definition of } f^0\} \\ & (\mu\ A \circ B){\downarrow}(R \times Id \times P) \end{array}$$

Inductive case:
The goal for the inductive case is as follows:

$$\begin{array}{l} A \overset{\mathcal{B}}{\sqsubseteq}_R A' \ \wedge \ B \overset{\mathcal{A}}{\sqsubseteq}_P B' \ \wedge \\ (\mu\ A \circ B){\downarrow}(R \times Id \times P) \sqsubseteq (A' \circ B')^k.\,(\mu\ A \circ B){\downarrow}(R \times Id \times P) \ \Rightarrow \\ \qquad (\mu\ A \circ B){\downarrow}(R \times Id \times P) \sqsubseteq (A' \circ B')^{k+1}.\,(\mu\ A \circ B){\downarrow}(R \times Id \times P) \end{array}$$

Assuming the antecedent, we calculate:

$$\begin{array}{ll} & (\mu\ A \circ B){\downarrow}(R \times Id \times P) \\ \sqsubseteq & \{\textit{induction assumption}\} \\ & (A' \circ B')^k.\,(\mu\ A \circ B){\downarrow}(R \times Id \times P) \\ = & \{\textit{rule 1.11}\} \\ & (A' \circ B')^k.\,(\mu\ A \circ B){\downarrow}(R \times Id \times Id){\downarrow}(Id \times Id \times P) \\ \sqsubseteq & \{\textit{assumption of } A \overset{\mathcal{B}}{\sqsubseteq}_R A'\} \end{array}$$

$$
\begin{array}{ll}
 & (A' \circ B')^k . (A' . (\mu\ B \circ A) \downarrow (R \times Id \times Id)) \downarrow (Id \times Id \times P) \\
\sqsubseteq & \{\textit{encoding propagation lemma}\ \} \\
 & (A' \circ B')^k . A' . (\mu\ B \circ A) \downarrow (R \times Id \times Id) \downarrow (Id \times Id \times P) \\
= & \{\textit{rule 1.12}\} \\
 & (A' \circ B')^k . A' . (\mu\ B \circ A) \downarrow (Id \times Id \times P) \downarrow (R \times Id \times Id) \\
\sqsubseteq & \{\textit{assumption of } B \overset{\mathcal{A}}{\sqsubseteq}_P B'\} \\
 & (A' \circ B')^k . A' . (B' . (\mu\ A \circ B) \downarrow (Id \times Id \times P)) \downarrow (R \times Id \times Id) \\
\sqsubseteq & \{\textit{encoding propagation lemma}\ \} \\
 & (A' \circ B')^k . A' . B' . (\mu\ A \circ B) \downarrow (Id \times Id \times P) \downarrow (R \times Id \times Id) \\
= & \{\textit{rule 1.11}\} \\
 & (A' \circ B')^k . A' . B' . (\mu\ A \circ B) \downarrow (R \times Id \times P) \\
= & \{f^{k+1} . x = f^k . (f . x),\ \textit{definition of composition}\} \\
 & (A' \circ B')^{k+1} . (\mu\ A \circ B) \downarrow (R \times Id \times P)\ \Box
\end{array}
$$

Using this lemma, we can now prove the subgoal 5.2. Assume $A \overset{\mathcal{B}}{\sqsubseteq}_R A'$, $B \overset{\mathcal{A}}{\sqsubseteq}_P B'$, and $\forall Yb \bullet (A' \circ B')^k . Yb \sqsubseteq (\mu\ A' \circ B')$, for fixed but arbitrary k. The conclusion is then proved as follows:

$$
\begin{array}{ll}
 & (\mu\ A \circ B) \downarrow (R \times Id \times P) \\
\sqsubseteq & \{\textit{Lemma}\} \\
 & (A' \circ B')^k . (\mu\ A \circ B) \downarrow (R \times Id \times P) \\
= & \{\textit{the third assumption, instantiating Yb with} \\
 & \quad (\mu\ A \circ B) \downarrow (R \times Id \times P)\} \\
 & (\mu\ A' \circ B')
\end{array}
$$

The proof of the third subgoal is similar. □

5.6 Discussion of Safety of Delegation

As we have stipulated above, the software reuse technique is safe if it is possible to establish the corresponding modular reasoning property. We have demonstrated that in theory the property can be established if the objects are refined according to the simple definition of refinement, as presented in section 5.3. Unfortunately, this definition is too restrictive to be interesting in practice, as it does not permit making assumptions about the context of an object under refinement. To circumvent this limitation of the simple definition, we defined the context object refinement. However, the context object refinement is insufficient for establishing the modular reasoning property, and the assumptions of the property should be strengthened with the additional assumptions (c) and (d) according to the "no infinite recursion of method invocations" requirement. Unfortunately, this requirement is non-modular, in the sense that it requires that the module resulting from the composition of two objects by delegation have no infinite mutual recursion, but does not state how this can be achieved by looking exclusively at the specifications of these objects. The simplicity of the "no infinite recursion" requirement hints at the fact that it is a necessary requirement, in the sense that if assumptions of the modular reasoning property are not strengthened according to this requirement, the property cannot be established. From this we make a conclusion that *in the general case unrestricted delegation does not allow for modular reasoning and thus, as a software reuse technique, is unsafe.*

However, we envision several approaches to satisfying the "no infinite recursion" requirement in a modular manner, by restricting the delegation technique. For example, object methods in the original specification can be marked as atomic if they do not call other methods. While refining the object, atomic methods must remain atomic and non-atomic ones can introduce new calls only to the atomic methods. Although restrictive, this approach guarantees the absence of accidental mutual recursion in the refined composed system. With another approach, we can assign an index to every method which indicates the maximal depth of method calls that this method is allowed to make. This approach only works if the original specification does not have mutually recursive method calls. For example, a method m that does not invoke any other method will have index 0, whereas a method n invoking m will have index 1. If a method invokes several methods with different indices, it is assigned the maximal of these indices plus one. With the original specification annotated in this manner, we can require that, while refining a method, calls to methods with indices higher than the indices of the methods that were called before cannot be introduced. However, the detailed analysis of different methods for establishing the "no infinite recursion of method invocations" requirement in a modular manner is outside the scope of this dissertation. Our conclusion is therefore that *delegation restricted to account for the "no infinite recursion" requirement allows for modular reasoning and* *is a safe code reuse technique.*

The common recommendation [81] for ensuring safe composition through delegation of the refined objects is to always establish an object's invariant before invoking any external method. It is interesting to note that this recommendation does not follow from the specification and is rather motivated by empirical expertise. As we argued in section 5.4, the informal reasoning which is usually done while developing an object is captured as the ad-hoc definition of object refinement. As we demonstrated by the counter examples introducing the requirements, this definition of refinement is simply inappropriate to ensure the desired conclusion. In these examples, participating objects do not have internal state, thus re-establishing invariant cannot help. However, the problem persists. We believe that the recommendation to always establish an object's invariant before invoking any external method, although being necessary, is not sufficient in the general case.

In fact, our definition of context object refinement takes this recommendation into account. Let us reconsider our first example. According to the Modular Reasoning Theorem for Delegation, to demonstrate that $Model'$ is a valid implementation of $Model$ in the context of $View$, we would need to show that every method of $Model'$ calling methods of $View$ composed with methods of $Model$ refines the corresponding methods of $Model$ composed with methods of $View$. Since $Model$ and $Model'$ operate on different attributes, to express, for example, in the method *append* of $Model'$ the behavior of a call to $View.update$, which calls *get_s* of $Model$, we need to coerce this call using an abstraction relation. Such an abstraction relation usually includes component invariants, and in this case includes the object invariant $n = \#s$ of $Model'$, i.e., $R.(s', n').s \;\widehat{=}\; s' = s \wedge n' = \#s'$. Note that in the definition of R, the attributes of $Model'$ are primed in order to distinguish them from the attributes of $Model$. According to the definition of refinement in context, the proof obligation for the method *append* after expansion and simplification is

$$(s := s \frown t; print(\#s)){\downarrow}R \sqsubseteq s := s \frown t; (print(\#s)){\downarrow}R; n := n + \#t$$

The right hand side can be expanded to $s := s \frown t; \{R\}; print(\#s); [R^{-1}]; n := n + \#t$. The abstraction statement preceding the invocation of *print* aborts, because it tries to find an abstract value of a sequence s satisfying the invariant $\#s = n$, which obviously does not hold at this point. Certainly, an aborting method is not a refinement of a non-aborting one, and therefore $Model'$ fails to correctly implement $Model$ in the context of $View$, breaching our requirement.

5.7 Check-List for Verifying Delegating Objects

Based on the Modular Reasoning Theorem for Delegation, we can formulate a check-list for informally verifying correctness of an object implementation against its specification. A developer of an object participating in delegation should verify that the following conditions are satisfied:

- The initial values of the implementation instance variables correspond to the initial values of the instance variables of the object's specification.
- Before every external method invocation the object's invariant is established.
- For every method, the precondition p of the method's specification as expressed on the instance variables of its implementation is stronger than or equal to the precondition p' of the method's implementation, i.e. $p \subseteq p'$.
- For every method, one must consider the postcondition q of the method's specification as expressed on the instance variables of the method's implementation, the instance variables of the specifications of objects called from the method's specification, and the result and value-result method parameters. One must also consider the postcondition q' of the method body constructed by substituting in the corresponding method's implementation the bodies of the call-backed methods of the same object with their specifications expressed on the implementation instance variables. Then, q' must be stronger than or equal to q, i.e., $q' \subseteq q$.
- For every method, one must make certain that after composing refined objects an invocation of the method would not result in infinite recursion of method invocations between objects participating in delegation.

The last item in this list can be established by either applying an approach to specifying objects similar to those discussed above or, if possible, by verifying the corresponding condition in the module resulting from the composition of objects.

5.8 Is Delegation Flexible?

The high degree of flexibility of delegation determines its wide acceptance and application in practice. During the development period, a system is constantly evolving. In early stages of system development, it is usually rather difficult to predict various changes that the system will undergo. Therefore, developers usually try to build a very general foundation of the system permitting the widest range of possible changes. In this respect, delegation has proved to be very useful as it allows for the creation of very general object interfaces. With delegation, an object can pass a reference to itself as an argument to a method of another object, which can then establish a back reference to the original object and request all of the needed data by querying its interface. The same degree of flexibility can be achieved with forwarding only if an object would pass all possible data as arguments when invoking a method on another object. Clearly, this is not desirable.

Let us now consider an example of applying delegation in practice. In most of the distributed object platforms such as CORBA [61], Java RMI [79], and COM/DCOM [69] communication between distributed objects is arranged according to a composition of two object-oriented design patterns [64], Broker and Proxy [25].

In distributed environments, objects reside in separate address spaces and their methods can be subject to *remote method calls* (a remote method call is issued in an address space different from the address space where the target object resides). By convention, the object issuing a call is referred to as the *client*; the target object is referred to as the *server*. For remote method calls to be possible, a platform should support remote object referencing. A remote reference identifies an object over the network and the particular interface that this object implements. To bridge the conceptual gap between the remote and local style of references, both in the client and in the server code, the actual manipulation with remote references is typically encapsulated in wrapper-like objects, known as client-side and server-side proxies. The key idea behind proxies [71] is that the client-side proxy can be considered as the local representative of the corresponding remote object and the server-side proxy can be considered as the representative of all potential clients of the remote object. The client-side proxy and the server-side proxy communicate with each other to transmit requests and responses. In general, clients and severs do not necessarily know about each other at compile time, and an intermediary, the *Broker* [25], is needed to dynamically relate the two

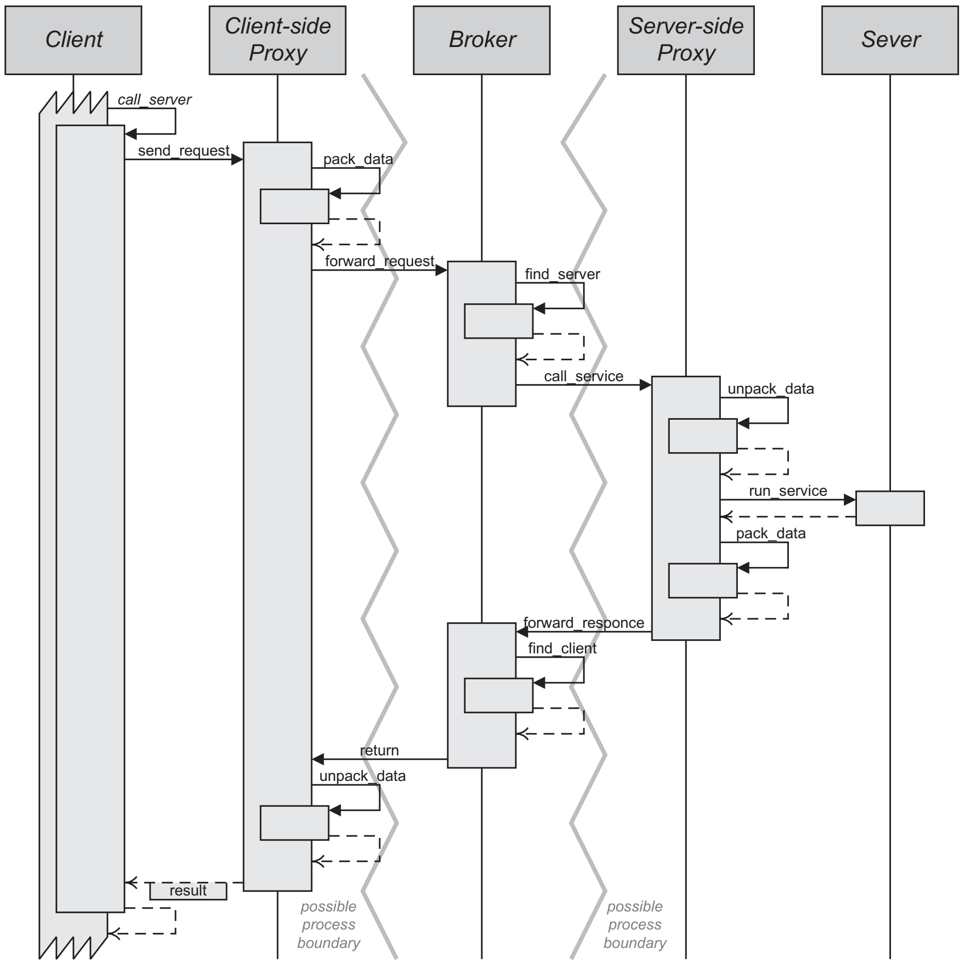

Figure 5.4: Request delivery scenario.

parties.

Figure 5.4 demonstrates a typical scenario of object interaction for delivering a request and receiving a response in a distributed setting on a message sequence chart. Note that solid arrows represent method invocation, while dashed arrows stand for return of control. When *Client* wants to obtain a certain service, it sends a request to *Client-side Proxy* which packs the request data for transferring it over the network and then forwards the request to *Broker*. *Broker* finds a server which has registered with it the service requested by *Client* and calls the service. The request for the service goes to

Server-side Proxy. The latter unpacks the data, runs the service on *Server*, packs the response data for transmitting over the network, and then forwards the packed response data to *Broker*. *Broker* finds *Client* who requested the service in the first place, returns the response data to its *Client-side Proxy* which, in turn, unpacks the data and returns the results to *Client*.

Analyzing the diagram, one can see that *Client-side Proxy* and *Broker*, as well as *Broker* and *Server-side Proxy* invoke methods on each other and are, therefore, composed via delegation. *Client* and *Client-side Proxy*, as well as *Server-side Proxy* and *Server* are composed via forwarding, as solid arrows go only in one direction. The delegation technique permits us to dynamically register services with a broker and to dynamically find servers providing the services requested by clients.

5.9 Discussion, Conclusions, and Related Work

Even though delegation in this chapter is presented as a software reuse technique which relies on objects as an ultimate software reuse mechanism, most of the results and discussions presented here concern a very general layout of composition of software components permitting mutual reference between components.

Based on our discussion of the safety and flexibility of delegation, we can now consider the advantages and disadvantages of using this software reuse technique in various kinds of systems.

Delegation is heavily used in open distributed component systems. One of the characteristic features of distributed component systems and platforms is the fact that components can be developed by independent developers and an integration phase is either completely absent or minimized. When the integration phase is missing, as in the case of CI Labs OpenDoc [24], components are composed by end users. When the integration phase is postponed, as in the case of Sun Java Beans [80] and Microsoft COM [68], components are composed by application developers. Due to the missing or postponed integration phase, it is impossible to analyze semantic integrity of the resulting composed system. Therefore, a specification and verification method for distributed component systems needs to be modular, i.e., verifying that participating components meet their contractual obligations should be sufficient to guarantee that the composed system operates correctly. As was argued above, unrestricted delegation is not safe in general, thus special measures (as we discussed) should be taken for restricting delegation and in this way avoiding an infinite recursion.

The delegation technique is, of course, often used for developing ordinary closed systems. In a closed environment, the final composed system can be verified to correctly implement the specification. However, in a large project application of modular reasoning is of crucial importance, as the complexity of the system can quickly become overwhelming. It seems that an incidental introduction of infinite recursion is unlikely to occur. Thus the last item in the check-list can be verified after composing the components.

Problems with re-entrance are also often discussed in the context of concurrent programming. In a multithreaded environment, several instances of the same procedure modifying global variables can be executed simultaneously. One thread of control can enter the procedure and, before the end of the procedure is reached, a second thread of control can re-enter the same procedure. Obviously, such a situation is problematic, because the second instance of the procedure might observe the global variables in an inconsistent state, or it can modify these global variables and then the first instance will observe them in an inconsistent state. The problem that we consider is sufficiently different from the re-entrance problem as known in concurrent programming to deserve a separate name, the "component re-entrance problem". There are two scenarios in which this problem can occur; firstly, when components are independently developed from specifications and, secondly, during independent maintenance of components.

One of the recommendations in concurrent programming is to circumvent the re-entrance problem by avoiding the re-entrant setting, which can be achieved using various locking mechanisms. In object-oriented and component-based programming, the re-entrant setting can be avoided by refusing to use delegation and instead relying exclusively on forwarding. However, as we discussed above, forwarding is less flexible than delegation, thus the design decision to refrain from applying delegation significantly reduces design options.

Problems occurring during maintenance of mutually dependent components similar to the component re-entrance problem have been mentioned by several researchers, e.g., Bertrand Meyer in [47] and Clemens Szyperski in [81]. Meyer considers the setting with two mutually dependent classes whose invariants include each other's attributes. His method for verification of conformance between two implementations of one class requires that the new implementation respect the invariant of the original implementation. He notices that this requirement alone is not sufficient for establishing correctness of the composed system and refers to this problem as “indirect invariant effect”. He then makes a conjecture that mirroring such interclass invariants in the participating classes would be sufficient to avoid the problem. Although we disagree with the practice of stating interclass invariants, it seems that the problem considered by Meyer is just a special case of the component re-entrance problem as formulated in this paper. As our examples demonstrate, preserving invariants, taken alone, does not eliminate the problem.

Szyperski describes a similar problem, but sees it rather as an instance of the re-entrance problem as occurring in concurrent systems. He reiterates the common recommendation for avoiding the problem, which recommends establishing a component invariant before invoking any external method. Interestingly enough, the recommendation to re-establish the invariant before all external method calls does not follow from the specification and is rather motivated by empirical expertise. As demonstrated by our examples, this recommendation, although necessary, is insufficient.

The requirement to re-establish a component invariant before all external calls is rather restrictive, because re-establishing the invariant might require a sequence of method calls to this and other components. Besides, it is not always necessary to establish the entire component invariant before external calls, because clients of the component can depend on some parts of the component invariant while being indifferent to the other parts. In [81], Szyperski proposes to “weaken invariants conditionally and make the conditions available to clients through test functions”. In a way, he proposes to make assumptions that component developers make about other components more explicit. This idea can be elaborated through augmenting specifications of components with require/ensure statements stipulating assumptions and guarantees that the components make. To avoid a conflict of assumptions, a component specification can make explicit the information the component relies on and provides to other components. For instance, every method can begin with a require condition and end with an ensure condition. Also every method invocation can be surrounded by an ensure/require couple. Then, while implementing a method, the developer can assume the information as stipulated in the require condition and ought to establish the ensure condition. Such an explicit statement of mutual assumptions and guarantees between components would reduce the need to unfold method invocations when verifying refinement in context. Note that the theoretical underpinning of such an approach to specifying component systems is an interpretation of the results presented in this paper, as the refinement calculus includes constructs for expressing the require/ensure statements.

A specification and verification method for component systems based on such an approach should additionally provide for satisfying the “no accidental mutual recursion” requirement in a modular manner. The detailed elaboration of such a method represents the subject of current research.

As was already mentioned, we have made a number of simplifications in the component model. In particular, we have assumed that components do not have self-calls and component implementations do not introduce new methods. Relaxing these restrictions on the component model is the subject of future work.

Chapter 6

Classes and Inheritance

As we already mentioned, object-oriented programming languages can be divided into two broad categories – those employing classes and those relying on prototypical objects. The first category includes all mainstream object-oriented languages most widely used in practice, C++ [78], Smalltalk [28], and Java [29], to name just a few. A *class* is a language construct representing a syntactic stencil that describes a particular kind of objects.

In many strongly-typed object-oriented languages, the class construct is tightly bound with the notion of an *object type* – instances of the same class usually have the same type.[1] An object type describes a syntactic interface of the object including signatures of object methods and sometimes also constructors. The object type is used for typechecking programs which, in the case of compiled languages, is done by a compiler. Typechecking is very useful as it permits finding mechanically certain kinds of programming errors, e.g., the "message not understood" error occasionally encountered in programs written in the untyped programming language Smalltalk. Further on we do not discuss the syntactic compatibility of object types, as we consider it a prerequisite for semantic compatibility of the corresponding objects.

A user can create an object by instantiating a class or can extend the functionality of the class by creating an *extension class.* The latter can be achieved through *inheritance*, which is a software reuse mechanism present in most class-based object-oriented languages. Not only can the extension class inherit attributes, i.e., instance variables and methods of the *base class*, it can also modify the inherited behavior and provide new attributes. The high

[1]In the programming language Java a type of an object can be declared separately.

degree of flexibility of inheritance explains its wide acceptance in practical programming.

In this chapter, we present a model of classes and inheritance and define refinement on classes. Then we discuss the safety of inheritance and present the semantic fragile base class problem that plagues maintenance of object systems relying on inheritance. We present a detailed analysis of the semantic fragile base class problem, discuss how code inheritance can be disciplined to become a safe software reuse mechanism, and prove the corresponding modular reasoning property. Next we illustrate the flexibility of code inheritance, showing an architectural pattern that inheritance facilitates. Further, we discuss the degree of flexibility attainable with different variants of inheritance, such as interface inheritance, code inheritance, and the disciplined inheritance. In the concluding sections we relate to the work of other researchers and consider the applicability of classes and inheritance for various kinds of software systems, emphasizing the balance between the flexibility and the safety.

6.1 Modeling Classes

We model classes as self-referential structures, as proposed by William Cook and Jens Palsberg in [17]. However, unlike in their model, classes can have instance variables in our formalization. A similar but more restricted model of classes and inheritance was first developed in [49].

We make the following simplifications in the model of classes. We consider only the classes that do not invoke methods on other classes. Modeling this feature can be done in a straightforward manner, but would only introduce an unnecessary complication, as this feature is irrelevant for considering the safety and flexibility of inheritance. We also consider only the classes that do not have recursive and mutually recursive methods. We model method parameters by global variables that both methods of a class and its clients can access. For every formal parameter of a method, we introduce a separate global variable used for passing values in and out of objects. It is easy to see that parameter passing by value and by reference can be modeled in this way. We mark 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 mark a formal parameter with a keyword **res** to indicate that the method returns a value in this parameter.

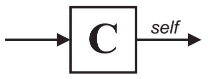

Figure 6.1: Illustration of a class.

In practice, a call to a method m_j from 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 the definition of m_j in an extension class. Accordingly, we model the method m_i as a function of the called method. As, in general, a class method may invoke all other methods of the same class, in the case of n methods we have

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

where $x_1, ..., x_n$ represent bodies of the methods of the class, and c_i is a statement representing the body of the method m_i. Accordingly, methods of a class **C** can be defined by

$$C = \lambda \mathit{self} \bullet (c_1, ..., c_n)$$

where *self* is an abbreviation for the tuple $(x_1, ..., x_n)$. We assume that C is monotonic in the *self* parameter with respect to the refinement relation.

A class **C** with an initial value of the internal state $c_0 : \Sigma$ and methods C can be given by

$$\mathbf{C} = (c_0, C)$$

and declared as follows:

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

A class can be depicted as shown in figure 6.1. The incoming arrow represents external calls to the class **C**, the outgoing arrow stands for self-calls of **C**.

As we have stated before, some global variables $d_1 : \Delta_1, ..., d_k : \Delta_k$ are used for parameter passing. Due to dynamic binding, methods of **C** operate not only on the state space $\Sigma \times \Delta$, where $\Delta = \Delta_1 \times ... \times \Delta_k$, but also on the state space of some modifier whose type is unknown until the modifier is applied. Thus C has the type $\Phi^{n,n}(\mathit{Ptran}(\alpha \times \Sigma \times \Delta))$, where α is a type variable which is instantiated with the type of modifier instance variables at modifier application. We assume that methods of the base class **C** have the

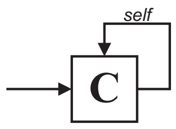

Figure 6.2: Illustration of creating an instance of **C**.

structure **skip** $\times$ S, where **skip** is executed on the component α of the state and S is executed on the component $\Sigma \times \Delta$.

In our model, classes are used as templates for creating objects. Objects have all self-calls resolved with methods of the same object. Modeling this formally amounts to taking the least fixed point of the function C. Statement tuples form a complete lattice with the refinement ordering. Also C is required to be monotonic in its *self* argument. These two conditions are sufficient to guarantee that the least fixed point of the function C exists and is unique. As we argued in Chapter 4, from the mathematical standpoint, an object that does not invoke methods on other objects is similar to a module, i.e., it is a pair consisting of the initial state of its instance variables and the tuple of statements representing its methods. We can define an operation of creating an object from its class as follows:

$$\mathit{create}\ \mathbf{C} \ \ \widehat{=}\ \ (c_0 : \Sigma, (\mu\ C) : \Pi^n(\mathit{Ptran}(\alpha \times \Sigma \times \Delta)))$$

Figure 6.2 illustrates creating an instance of the class **C**.

6.2 Modeling Inheritance

Imagine that there exists a class **C** which implements a certain functionality. With the use of inheritance, **C** can be adjusted and extended into an extension class **E**. Not only the extension class can inherit attributes, i.e., instance variables and methods of the base class **C**, but also it can modify the inherited behavior and provide new attributes. Modification of inherited behavior is achieved through overriding inherited methods with the new definitions in the extension class. For such method overriding to take place, not only the external method invocations should be redirected to the new definitions of the methods, but also all internal calls, known as *self-calls*, in the base class should be resolved similarly. The most often referred definition of inheritance is the "method lookup" algorithm of Smalltalk [28]:

"When a message is sent, the methods in the receiver's class are searched for one with a matching selector. If none is found, the methods in that class's superclass are searched next. The search continues up the superclass chain until a matching method is found. [...]

When a method contains a message whose receiver is *self*, the search for the method for this message begins in the instance's class, regardless of which class contains the method containing *self*. [...]

When a message is sent to *super*, the search for a method [...] begins in the superclass of the class containing the method. The use of *super* allows a method to access methods defined in a superclass even if the methods have been overridden in the subclasses."

Unfortunately, such an operational definition does not aid the intuitive understanding. For modeling single inheritance, we adopt the notion of modifiers as proposed by Cook and Palsberg in [19].[2] This model of inheritance was proved to correspond to the form of inheritance used in object-oriented systems.

Assume that we have a base class **C** and an extension class **E** inheriting from it. We say that **E** is equivalent to ($\boldsymbol{L}$ **mod C**)[3], where $\boldsymbol{L}$ corresponds to the extending part of the definition of **E**, and the operator **mod** combines $\boldsymbol{L}$ with the inherited part **C**. We refer to $\boldsymbol{L}$ as a *modifier* [86].

We make a number of restrictions on inheritance. We consider only single inheritance, and an extension class created through inheritance from a base class cannot have additional methods. Moreover, methods in a modifier cannot be recursive and mutually recursive. Modeling classes and inheritance without making these restrictions is possible along the same lines, however it would significantly complicate the presentation. It suffices to consider only those modifiers that 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 declared by

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

is modeled by a pair

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

[2]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.

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

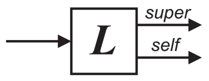

Figure 6.3: Illustration of modifiers.

Here $l_0 : \Gamma$ are initial values of new instance variables l, L is a function representing methods of the modifier, the bounded variables *self* and *super* are abbreviations of the tuples $(x_1, ..., x_n)$ and $(y_1, ..., y_n)$ respectively, and $(l_1, ..., l_n)$ are the bodies of the overriding methods. We assume that L is monotonic in both arguments. See figure 6.3 for an illustration of modifiers. As with the class diagram, the incoming arrow represents external calls to methods of the modifier, whereas outgoing arrows stand for self and super-calls of the modifier.

Under the condition that signatures of overriding methods in the modifier match the corresponding method signatures in the base class, the modifier can be applied to an arbitrary base class. We make a restriction that modifier methods are not allowed to access the instance variables of the base class directly, but only by making super-calls. As was pointed out by Alan Snyder in [72], "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 general, accessing the base class state from the modifier directly is recognized as a poor programming practice.

As the state space of the base class is unknown until modifier application, we say that the methods L of the modifier $\boldsymbol{L}$ operate on the state space $\alpha' \times \beta \times \Gamma \times \Delta$, where β is a type variable to be instantiated with the type of base class instance variables while modifier application, and Δ is the type of the state component representing all parameters of all modifier methods. The role of the type variable α' will become clear after we define the creation of new classes by applying modifiers. Hence, the type of the function L representing the methods of the modifier is as follows:

$$\Pi^n(Ptran(\alpha' \times \beta \times \Gamma \times \Delta)) \to \Phi^{n,n}(Ptran(\alpha' \times \beta \times \Gamma \times \Delta))$$

We assume that the methods of the modifier $\boldsymbol{L}$ have the structure **skip** $\times\, S$, where **skip** is executed on the component $\alpha' \times \beta$ of the state and S is executed on the component $\Gamma \times \Delta$.

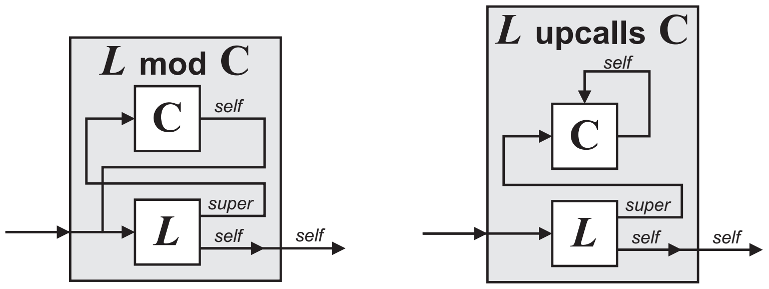

Figure 6.4: Illustration of the operators **mod** and **upcalls**.

Inheritance can be modeled by means of the operator **mod** which applies a modifier $\boldsymbol{L} = (l_0, L)$ to a base class $\mathbf{C} = (c_0, C)$ in the following manner:

$$(\boldsymbol{L}\ \mathbf{mod}\ \mathbf{C}) \ \widehat{=}\ ((c_0, l_0),\ \lambda \mathit{self} \bullet L.\, \mathit{self}.\, (C.\, \mathit{self}{\downarrow}Q){\uparrow}Q)$$

Here Q is a state rearranging relation swapping the second and the third elements of the state:

$$Q.\,(x', y', z', u').\,(x, z, y, u) \ = \ x' = x \ \wedge\ y' = y \ \wedge\ z' = z \ \wedge\ u' = u$$

Figure 6.4 illustrates application of the modifier $\boldsymbol{L}$ to the base class $\mathbf{C}$ to construct the class $(\boldsymbol{L}\ \mathbf{mod}\ \mathbf{C})$. Note that in the resulting class self-calls of $\mathbf{C}$ are redirected to the methods of the modifier $\boldsymbol{L}$ due to dynamic binding. Both external and internal method calls are thus resolved dynamically.

Let us now consider a restricted form of inheritance in which external method calls are resolved dynamically whereas internal calls are resolved statically. This form of inheritance can be modeled using an operator **upcalls** which applies the modifier $\boldsymbol{L} = (l_0, L)$ to the base class $\mathbf{C} = (c_0, C)$ as follows:

$$(\boldsymbol{L}\ \mathbf{upcalls}\ \mathbf{C}) \ \widehat{=}\ ((c_0, l_0),\ \lambda \mathit{self} \bullet L.\, \mathit{self}.\, (\mu\ C){\downarrow}P)$$

Here the relation P discards the first element of the state and swaps the second and the third elements of the state:

$$P.\,(x', y', z', u').\,(z, y, u) \ = \ y' = y \ \wedge\ z' = z \ \wedge\ u' = u$$

See figure 6.4 for an illustration of modifier application with the operator **upcalls**. Note that in the resulting class self-calls of $\mathbf{C}$ remain in $\mathbf{C}$ ignoring dynamic binding.

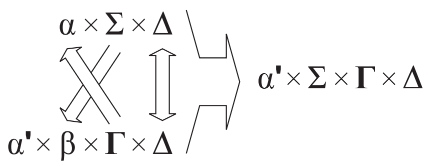

Figure 6.5: Illustration of type variables instantiation.

Application of the modifier ***L*** using both **mod** and **upcalls** instantiates its type variable β with the type Σ of the base class **C**. Simultaneously, the type variable α of **C** is instantiated with the type Γ of the modifier ***L***. Hence, after the corresponding state space rearrangement, the constructed classes (***L*** **mod C**) and (***L*** **upcalls C**) have methods operating on $\alpha' \times \Sigma \times \Gamma \times \Delta$, where α' is a type variable to be instantiated in the next modifier application, $\Sigma \times \Gamma$ is the type of instance variables of the resulting classes, and Δ is the type of parameters of all methods of the resulting classes. Instantiation of type variables is illustrated in figure 6.5.

Further on we say that an *up-call* occurs when an extension class invokes a base class method; when a base class invokes a method of a class derived from it, we refer to such an invocation as a *down-call*. Figure 6.6 justifies these terms.

6.3 Refinement on Classes

We consider only refinement between classes with identical interfaces. As was mentioned before, from a mathematical perspective, an object that does not invoke methods on other objects is similar to a module. Before defining

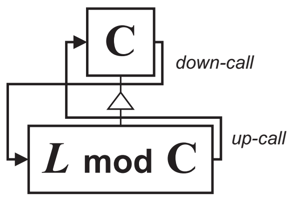

Figure 6.6: Up-calls and down-calls.

refinement on classes, we need to slightly adjust the definition of module refinement (which was first presented in section 3.1) to take into account a different layout of state spaces in class instances. For class instances $\mathsf{C} = (c_0 : \Sigma, Cp : \Pi^n(\mathit{Ptran}(\alpha \times \Sigma \times \Delta)))$ and $\mathsf{D} = (d_0 : \Sigma', Dp' : \Pi^n(\mathit{Ptran}(\alpha \times \Sigma' \times \Delta)))$ refinement is defined as follows:

$$\mathsf{C} \sqsubseteq \mathsf{D} \;\;\widehat{=}\;\; \exists R \bullet R.\, d_0.\, c_0 \;\wedge\; Cp \sqsubseteq_{\widehat{R}} Dp$$

While it is possible to give different definitions of refinement on classes, we believe that the following definition captures the intuition used by programmers when reasoning informally about behavioral conformance between the objects these classes instantiate. This definition strongly relates to the notion of behavioral subtyping [2, 3, 44, 40, 23]. The paper "A behavioral notion of subtyping" by Barbara Liskov and Jeannette Wing [44] starts with the following informal definition of behavioral subtyping: "What does it mean for one type to be a subtype of another? We argue that this is a semantic question having to do with the behavior of the objects of the two types: the objects of the subtype ought to behave the same as those of the supertype as far as anyone or any program using supertype objects can tell." Following [51], we prefer to separate the decidable syntactic descriptions of objects, as described by object types, and the undecidable semantic specifications of objects, as described by classes instantiating these objects. Accordingly, the above informal definition can be rewritten as follows: a class **C** is refined by another class **D**, if all objects instantiated by **C** are substitutable with the objects instantiated by **D** in any context. Note that after creating an instance of a class all self-calls in the methods of this instance are already resolved, i.e. they necessarily refer to the methods of the same instance. Let $\mathbf{C} = (c_0, C)$, where $C = \lambda \mathit{self} \bullet (c_1, ..., c_n)$, and $\mathbf{D} = (d_0, D)$, where $D = \lambda \mathit{self} \bullet (d_1, ..., d_n)$, be classes, then refinement on these classes can be expressed through refinement on modules as follows:

$$\mathbf{C} \sqsubseteq \mathbf{D} \;\;\widehat{=}\;\; (\mathit{create}\ \mathbf{C}) \sqsubseteq (\mathit{create}\ \mathbf{D})$$

This notion of class refinement is very general. The class **D** can be an extension of the class **C** or be completely independent. If **D** is an extension of **C**, instance variables of **D** can extend those of **C** or be completely different. The refinement relation can be also applied to pairs of abstract and concrete classes.

6.4 Safety of Inheritance

Most of the work concerning the safety of inheritance concentrated around the notion of behavioral subtyping. The research was focusing on the question as to how one can establish that an object of a subtype is safely substitutable for an object of the supertype in any client. In these works an object's behavior is captured in the object's type. As was already mentioned, we believe that object behavior should be captured in the object's class. Thus in our terms the question can be reformulated as to how one can establish that an object is safely substitutable for another one in any client by considering only the classes of the objects. This question can be rewritten as the following property:

$$\mathbf{C} \sqsubseteq \mathbf{D} \;\Rightarrow\; (P \; \mathit{uses} \; \mathbf{C}) \sqsubseteq (P \; \mathit{uses} \; \mathbf{D})$$

where P is a client program, and **C** and **D** are classes. The property of this kind has been extensively studied in works on behavioral subtyping and now also in [51]. In our abstract formalization, this property can be easily established, as it directly translates into the corresponding property formulated and proved for modules.

We, however, believe that other factors also contribute to the safety of inheritance. In particular, while maintaining a software system built using code inheritance, it is natural for users to expect that if they use an improved version of the base class instead of the original class, the resulting extension class will also become better. Consider the following situation: imagine that there exists a class **C** supplied as a part of a class library and that a developer decides to reuse it. To adjust **C** for the use of a particular client, the developer creates a modifier ***L***, applying which to **C** results in an extension class (***L*** **mod C**). After some time, developers of the class library decide to release a new version of the library containing a new version **D** of the class **C**. As extensions of the library are not available to its developers (the extensions are usually developed by an independent party), in order to avoid invalidating the existing extensions, verifying that **D** is a refinement of **C** should be sufficient for ensuring that the extension class (***L*** **mod C**) is refined by the extension class resulting from substituting **C** with **D**. In other words, the *modular reasoning property for inheritance* looks as follows:

$$\mathbf{C} \sqsubseteq \mathbf{D} \;\Rightarrow\; (\boldsymbol{L} \; \mathbf{mod} \; \mathbf{C}) \sqsubseteq (\boldsymbol{L} \; \mathbf{mod} \; \mathbf{D})$$

Unfortunately, this property does not hold because of the so-called *semantic fragile base class problem*, which was first pointed out in [87] and systematically studied in [49]. To gain an intuitive understanding of this problem and see how disguised it can be, let us first consider an example. The contents of sections 6.4.1-6.4.3 are based on [49].

6.4.1 An Example of the Fragile Base Class Problem

Consider the example presented in figure 6.7.[4] Suppose that a class **Bag** is provided by some object-oriented system, e.g., an extensible container framework. This class has an instance variable $b : bag\ of\ char$ initialized with an empty bag, and 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 develops a modifier ***Counting*** that introduces an instance variable n, and overrides *add* to increment n every time a new element is added to the bag. As the extension class **CountingBag** resulting from applying the modifier ***Counting*** to the base class **Bag** will maintain an invariant $n = |b|$, the user overrides the method *cardinality* to return the value of n, as shown in figure 6.8.

After some time, framework developers decide to improve the efficiency of the class **Bag** and release a new version of the system. An "improved" **Bag**$'$ implements *addAll* without invoking *add*. Naturally, the framework developers claim that the new version of the system is fully compatible with the previous one. It definitely appears to be so if considered separately from the extensions. However, when trying to use **Bag**$'$ instead of **Bag** as the base class, the framework user suddenly discovers that the resulting class **CountingBag**$'$ returns the incorrect number of elements in the bag (see figure 6.8) for the resulting definition of **CountingBag**$'$). This happens because the new implementation of the method *addAll* in **Bag**$'$ does not invoke *add* and, therefore, n does not get increased. Here we face the semantic fragile base class problem. Any system employing code inheritance and self-recursion is vulnerable to this problem.

The framework developers relied on the following property, which precisely matches our formulation of the modular reasoning property for inher-

[4]This example is adopted from [77].

```
**Bag** =
**class**
    $b : bag\ of\ char := \llbracket\ \rrbracket$,
    $cardinality(\mathbf{res}\ r : int)\ \widehat{=}$
        $r := |b|$,
    $add(\mathbf{val}\ x : char)\ \widehat{=}$
        $b := b\ \cup\ \llbracket x \rrbracket$,
    $addAll(\mathbf{val}\ bs : bag\ of\ char)\ \widehat{=}$
        **while** $bs \neq \llbracket\ \rrbracket$ **do**
            **begin var** $y \bullet y \in bs$;
                $self \triangleright add(y)$;
                $bs := bs - \llbracket y \rrbracket$;
            **end**
        **od**
**end**
```

```
**Bag′** =
**class**
    $b : bag\ of\ char := \llbracket\ \rrbracket$,
    $cardinality(\mathbf{res}\ r : int)\ \widehat{=}$
        $r := |b|$,
    $add(\mathbf{val}\ x : char)\ \widehat{=}$
        $b := b\ \cup\ \llbracket x \rrbracket$,
    $addAll(\mathbf{val}\ bs : bag\ of\ char)\ \widehat{=}$
        $b := b\ \cup\ bs$
**end**
```

```
***Counting*** = **modifier**
        $n : int := 0$,
        $cardinality(\mathbf{res}\ r : int)\ \widehat{=}\ r := n$,
        $add(x : char)\ \widehat{=}\ n := n + 1; super \triangleright add(x)$,
        $addAll(\mathbf{val}\ bs : bag\ of\ char)\ \widehat{=}\ super \triangleright addAll(bs)$
    **end**
```

Figure 6.7: Example of the fragile base class problem.

itance:

$$\mathbf{Bag} \sqsubseteq \mathbf{Bag}' \ \Rightarrow\ (\boldsymbol{Counting}\ \mathbf{mod\ Bag}) \sqsubseteq (\boldsymbol{Counting}\ \mathbf{mod\ Bag}')$$

Unfortunately, as demonstrated by the above example, this property does not hold and, therefore, neither does the modular reasoning property for inheritance. This leaves us with the question as to whether it is possible to restrict code inheritance, so that it would be possible to reason about it in a modular manner.

```
**CountingBag** =
class
    $b : bag\ of\ char := \llbracket\ \rrbracket$,
    $n : int := 0$,
    $cardinality($**res** $r : int) \mathrel{\widehat{=}}$
        $r := n$,
    $add($**val** $x : char) \mathrel{\widehat{=}}$
        $n := n + 1; b := b \cup \llbracket x \rrbracket$,
    $addAll($**val** $bs : bag\ of\ char) \mathrel{\widehat{=}}$
        while $bs \neq \llbracket\ \rrbracket$ do
            begin var $y \bullet y \in bs$;
                $self \twoheadrightarrow add(y)$;
                $bs := bs - \llbracket y \rrbracket$;
            end
        od
end
```

```
**CountingBag′** =
class
    $b : bag\ of\ char := \llbracket\ \rrbracket$,
    $n : int := 0$,
    $cardinality($**res** $r : int) \mathrel{\widehat{=}}$
        $r := n$,
    $add($**val** $x : char) \mathrel{\widehat{=}}$
        $n := n + 1; b := b \cup \llbracket x \rrbracket$,
    $addAll($**val** $bs : bag\ of\ char) \mathrel{\widehat{=}}$
        $b := b \cup bs$
end
```

Figure 6.8: **CountingBag** and **CountingBag′**.

6.4.2 Aspects of the Problem

Let us now consider five examples invalidating the modular reasoning property and illuminating the shortcomings of inheritance. The examples are orthogonal to each other, meaning that all of them illustrate different aspects of the problem.

Direct Access to the Base Class State

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

A base class **C** represents its state by an integer variable x and declares two methods m and n increasing x by 1 and 2 respectively. A modifier ***L*** 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

x by 2.

$$
\begin{array}{rll}
\mathbf{C} = & \mathbf{class} & \\
 & \quad x : \mathit{int} := 0, & \\
 & \quad m() \mathrel{\widehat{=}} x := x + 1, & \\
 & \quad n() \mathrel{\widehat{=}} x := x + 2 & \\
 & \mathbf{end} &
\end{array}
\qquad
\begin{array}{rll}
\boldsymbol{L} = & \mathbf{modifier} & \\
 & & \\
 & \quad m() \mathrel{\widehat{=}} \mathit{super} \rightarrowtriangle m(), & \\
 & \quad n() \mathrel{\widehat{=}} x := x + 2 & \\
 & \mathbf{end} &
\end{array}
$$

A revision **D** introduces an extra instance variable y, initializing it to 0. The methods m and n increase x and y by 1 and by 2, but indirectly via y. Therefore, the methods of **D** implicitly maintain an invariant $x = y$.

$$
\begin{array}{rll}
\mathbf{D} = & \mathbf{class} & \\
 & \quad x : \mathit{int} := 0; y : \mathit{int} := 0, & \\
 & \quad m() \mathrel{\widehat{=}} y := y + 1; x := y, & \\
 & \quad n() \mathrel{\widehat{=}} y := y + 2; x := y & \\
 & \mathbf{end} &
\end{array}
$$

Now, if we consider an object *obj* which is an instance of class ($\boldsymbol{L}$ **mod D**), obtained by substituting **D** for **C**, and the sequence of method calls $obj \rightarrowtriangle n(); obj \rightarrowtriangle m()$, 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. Therefore,

$$\mathbf{C} \sqsubseteq \mathbf{D} \;\not\Rightarrow\; (\boldsymbol{L}\ \mathbf{mod}\ \mathbf{C}) \sqsubseteq (\boldsymbol{L}\ \mathbf{mod}\ \mathbf{D})$$

An analogous problem was described by Alan Snyder in [72]. 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 $\boldsymbol{L}$ is allowed to modify the instance variables inherited from **C** directly, it becomes impossible to change the data representation in **D**.

Unanticipated Mutual Recursion

Suppose that **C** describes a class with an instance variable x initialized to 0 and two methods m and n both incrementing x by 1. A modifier $\boldsymbol{L}$ overrides n so that it calls m. Now, if a revision **D** reimplements m by calling the method n, which has an implementation exactly as it had before, we run into the problem:

```
C =                                L =                                     D =
class                              modifier                                class
    $x : int := 0$,                                                            $x : int := 0$,
    $m() \mathrel{\widehat=} x := x + 1$,     $m() \mathrel{\widehat=} super \triangleright m()$,     $m() \mathrel{\widehat=} self \triangleright n()$,
    $n() \mathrel{\widehat=} x := x + 1$      $n() \mathrel{\widehat=} self \triangleright m()$       $n() \mathrel{\widehat=} x := x + 1$
end                                end                                     end
```

When the modifier $\boldsymbol{L}$ is applied to **D**, the methods m and n of the resulting class ($\boldsymbol{L}$ **mod D**) become mutually recursive. Clearly, a call to either one leads to a never terminating loop. Therefore,

$$\mathbf{C} \sqsubseteq \mathbf{D} \;\not\Rightarrow\; (\boldsymbol{L}\ \mathbf{mod}\ \mathbf{C}) \sqsubseteq (\boldsymbol{L}\ \mathbf{mod}\ \mathbf{D})$$

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

Unjustified Assumptions in Revision Class

To illustrate the next shortcoming of inheritance, it is sufficient to provide only a 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 postconditions which state that, given a non-negative real number x, methods m and n of **C** will find such r that its power of two and four respectively equal x.

A modifier $\boldsymbol{L}$ overrides the method m so that it would return a negative value.[5] Such an implementation of m is a refinement of the original

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

specification, because it decreases nondeterminism.

```
C =                                          L =
class                                        modifier
    $m(\textbf{val } x : real, \textbf{res } r : real) \mathrel{\widehat{=}}$    $m(\textbf{val } x : real, \textbf{res } r : real) \mathrel{\widehat{=}}$
        ( pre $x \geq 0$,                            $r := -\sqrt{x}$,
          post $r^2 = x$ ),
    $n(\textbf{val } x : real, \textbf{res } r : real) \mathrel{\widehat{=}}$    $n(\textbf{val } x : real, \textbf{res } r : real) \mathrel{\widehat{=}}$
        ( pre $x \geq 0$,                            $super \twoheadrightarrow n()$
          post $r^4 = x$ )
                                             end
```

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

```
D = class
        $m(\textbf{val } x : real, \textbf{res } r : real) \mathrel{\widehat{=}}$
            $r := \sqrt{x}$,
        $n(\textbf{val } x : real, \textbf{res } r : real) \mathrel{\widehat{=}}$
            $self \twoheadrightarrow m(x, r); self \twoheadrightarrow m(r, r)$
    end
```

Suppose now that we have an instance of a class (***L*** **mod D**). 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. Therefore,

$$\mathbf{C} \sqsubseteq \mathbf{D} \;\not\Rightarrow\; (\boldsymbol{L}\ \mathbf{mod}\ \mathbf{C}) \sqsubseteq (\boldsymbol{L}\ \mathbf{mod}\ \mathbf{D})$$

This example demonstrates that the problem might occur if developers of a revision class assume that, when self-calling a method, the body of a method as defined in the revision class is guaranteed to be executed. Due to inheritance and dynamic binding such an assumption is unjustified.

Unjustified Assumptions in Modifier

To illustrate the next aspect of the semantic fragile base class problem, let us consider the following example:

$$\begin{array}{ll}
\mathbf{C}\ = & \boldsymbol{L}\ = \\
\mathbf{class} & \mathbf{modifier} \\
\quad l(\mathbf{val}\ v : \mathit{int}) \ \widehat{=}\ \{v \geq 5\}, & \quad l(\mathbf{val}\ v : \mathit{int}) \ \widehat{=}\ \mathbf{skip}, \\
\quad m(\mathbf{val}\ v : \mathit{int}) \ \widehat{=}\ \mathit{self} \mathbin{-\!\!\triangleright} l(v), & \quad m(\mathbf{val}\ v : \mathit{int}) \ \widehat{=}\ \mathit{super} \mathbin{-\!\!\triangleright} m(v), \\
\quad n(\mathbf{val}\ v : \mathit{int}) \ \widehat{=}\ \mathbf{skip} & \quad n(\mathbf{val}\ v : \mathit{int}) \ \widehat{=}\ \mathit{self} \mathbin{-\!\!\triangleright} m(v) \\
\mathbf{end} & \mathbf{end}
\end{array}$$

$$\begin{array}{ll}
\mathbf{D}\ = & \mathbf{class} \\
& \quad l(\mathbf{val}\ v : \mathit{int}) \ \widehat{=}\ \{v \geq 5\}, \\
& \quad m(\mathbf{val}\ v : \mathit{int}) \ \widehat{=}\ \{v \geq 5\}; \mathit{self} \mathbin{-\!\!\triangleright} l(v), \\
& \quad n(\mathbf{val}\ v : \mathit{int}) \ \widehat{=}\ \mathbf{skip} \\
& \mathbf{end}
\end{array}$$

Let us compute full definitions of the classes ($\boldsymbol{L}$ **mod C**) and ($\boldsymbol{L}$ **mod D**):

$$\begin{array}{ll}
(\boldsymbol{L}\ \mathbf{mod}\ \mathbf{C})\ = & (\boldsymbol{L}\ \mathbf{mod}\ \mathbf{D})\ = \\
\mathbf{class} & \mathbf{class} \\
\quad l(\mathbf{val}\ v : \mathit{int}) \ \widehat{=}\ \mathbf{skip}, & \quad l(\mathbf{val}\ v : \mathit{int}) \ \widehat{=}\ \mathbf{skip}, \\
\quad m(\mathbf{val}\ v : \mathit{int}) \ \widehat{=}\ \mathit{self} \mathbin{-\!\!\triangleright} l(v), & \quad m(\mathbf{val}\ v : \mathit{int}) \ \widehat{=}\ \{v \geq 5\}; \mathit{self} \mathbin{-\!\!\triangleright} l(v), \\
\quad n(\mathbf{val}\ v : \mathit{int}) \ \widehat{=}\ \mathit{self} \mathbin{-\!\!\triangleright} l(v) & \quad n(\mathbf{val}\ v : \mathit{int}) \ \widehat{=}\ \{v \geq 5\}; \mathit{self} \mathbin{-\!\!\triangleright} l(v) \\
\mathbf{end} & \mathbf{end}
\end{array}$$

It is easy to see that, while **C** is refined by **D**, the class ($\boldsymbol{L}$ **mod C**) is not refined by ($\boldsymbol{L}$ **mod D**). Due to the presence of the assertion $\{v \geq 5\}$ in the methods m and n of ($\boldsymbol{L}$ **mod D**), their preconditions are stronger than those of the corresponding methods in ($\boldsymbol{L}$ **mod C**), while to preserve refinement their preconditions could have only been weakened. Therefore,

$$\mathbf{C} \sqsubseteq \mathbf{D} \ \not\Rightarrow\ (\boldsymbol{L}\ \mathbf{mod}\ \mathbf{C}) \sqsubseteq (\boldsymbol{L}\ \mathbf{mod}\ \mathbf{D})$$

To summarize, the problem might occur due to the 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.

Unjustified Assumption of Binding Invariant in Modifier

A class **C** has an instance variable x. A modifier $\boldsymbol{L}$ 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.

```
$\mathbf{C}$ =
class
    $x : \mathit{int} := 0,$

    $l(\mathbf{res}\ r : \mathit{int}) \mathrel{\widehat{=}} r := x,$
    $m() \mathrel{\widehat{=}} x := x + 1; \mathit{self} \rightarrow\!\!\triangleright n(),$
    $n() \mathrel{\widehat{=}}$ skip
end
```

```
$\boldsymbol{L}$ =
modifier
    $y : \mathit{int} := 0,$

    $l(\mathbf{res}\ r : \mathit{int}) \mathrel{\widehat{=}} \mathit{super} \rightarrow\!\!\triangleright l(r),$
    $m() \mathrel{\widehat{=}} y := y + 1; \mathit{super} \rightarrow\!\!\triangleright m(),$
    $n() \mathrel{\widehat{=}}$ begin var $r \bullet T;$
                  $\mathit{super} \rightarrow\!\!\triangleright l(r);$
                  $\{r = y\};$
              end
end
```

It is easy to see that before and after execution of any method of ($\boldsymbol{L}$ **mod C**) the value of x is equal to the value of y. We can say that ($\boldsymbol{L}$ **mod C**) maintains the invariant $(x = y)$. The full definition of the method m in an instance of the class ($\boldsymbol{L}$ **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 **D** reimplements m by first self-calling n and then incrementing x as illustrated below, we run into the problem.

```
$\mathbf{D}$ = class
        $x : \mathit{int} := 0,$

        $l(\mathbf{res}\ r : \mathit{int}) \mathrel{\widehat{=}} r := x,$
        $m() \mathrel{\widehat{=}} \mathit{self} \rightarrow\!\!\triangleright n(); x := x + 1,$
        $n() \mathrel{\widehat{=}}$ skip
    end
```

The body of the method m in an instance of the class ($\boldsymbol{L}$ **mod D**) is effectively of the form $y := y + 1; \{x = y\}; x := x + 1$, and, naturally, it aborts. Therefore,

$$\mathbf{C} \sqsubseteq \mathbf{D} \;\not\Rightarrow\; (\boldsymbol{L}\ \mathbf{mod}\ \mathbf{C}) \sqsubseteq (\boldsymbol{L}\ \mathbf{mod}\ \mathbf{D})$$

When creating a modifier, its developer usually intends it for a particular base class. A common practice is introducing 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, even if the actual modification in the base class code appears as harmless as a change in the order of statements. If methods of the modifier rely on the presence of such an invariant, a crash might occur.

6.4.3 Conflict Between Safety and Flexibility of Inheritance

The presented examples demonstrate different aspects of the semantic 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 key aspects there are some that were overlooked by other researchers, as we discuss in the conclusions of this chapter.

The orthogonality of the considered examples suggests that it might be possible to formulate requirements that would allow circumventing the aspects of the problem illustrated by the examples. These requirements can be of two kinds, those that can be taken into account by additional verification obligations (extra conjuncts in the antecedent of the modular reasoning property), and those that can only be addressed by restricting the code inheritance mechanism. Clearly, the first kind is preferable, as restricting the inheritance mechanism sacrifices some of its flexibility in favor of safety. Let us consider what such restrictions can look like and whether they would enable modular reasoning about inheritance.

As direct access to the base class state is clearly harmful, we can straightforwardly formulate the following requirement:

"No direct access to the base class state":

> *An extension class should not access the state of its base class directly, but only through calling base class methods.*

This requirement can only be addressed by a restriction on the code inheritance mechanism. The internal state of the base class should be made inaccessible to extension classes. For example, in C++ this can be achieved by declaring instance variables as `private`.

To capture the unanticipated mutual recursion aspect of the problem we formulate the following requirement:

"No cycles":

> *A base class revision and a modifier should not jointly introduce new cyclic method dependencies.*

This requirement is similar to the "no infinite recursion" requirement of Chapter 5. Accordingly, it can also be handled in a modular manner by indexing the methods according to the depth of possible method invocations, and requiring that both extension and revision developers preserve the index order. Thus imposing this requirement amounts to introducing an additional verification obligation.

The aspect concerning the unjustified assumptions in revision classes can be captured by the following requirement:

"No revision self-calling assumptions":

> *When verifying a method of the revision class, its developers should not make any additional assumptions about the behavior of the other methods in the revision class. Only the behavior described in the base class may be taken into consideration.*

Taking this requirement into account amounts to introducing an extra verification obligation that an instance of the base class **C** is refined by an instance of the revision class **D** with all self-calls substituted with the bodies of the corresponding methods of **C**.

The aspect concerning the unjustified assumptions in modifiers leads us to formulating the following requirement:

"No base class down-calling assumptions":

> *When constructing a method of the modifier, its developers should not make additional assumptions about the behavior of the other methods in the modifier that can get invoked due to dynamic binding of the base class self-calls. Bodies of the corresponding methods in the base class should be considered instead.*

This requirement can be addressed by introducing a verification obligation stipulating that the base class is refined by the class resulting from applying the modifier to the base class with the operator **upcalls**.

The example illustrating unjustified assumption of binding invariant in a modifier is the most surprising of all. This example demonstrates that an extension class can be invalidated by such an innocent-looking change in the base class as a modification in the order of statements. This aspect of the problem is clearly induced by the fact that the developers of the modifier intended the modifier for a particular base class, assuming the order in which the base class instance variable is updated and the self-invocation occurs. Based on this assumption, they created an invariant binding instance variables of the modifier with those of the base class.

One can think of two proposals for dealing with the problem in this case. The revision class developers can be blamed for the problem, as they have changed the order in which the internal state of the base class is updated and methods on self are invoked. The first proposal then implies that the order of state changes and method invocations on self cannot be changed. Obviously, this proposal is infeasible in practice, as classes in real programs work with complex data structures and have nontrivial self-invocation patterns. The essence of the second proposal is that modifier developers should not intentionally create and rely on the presence of the invariant that appears in the extension class resulting from applying the modifier to the base class. However, this requirement in combination with the "no direct access to the base class state" requirement would effectively defeat the flexibility of code inheritance, as extension developers would be only allowed to either access the modifier's own instance variables or make up-calls to the base class.

We believe that it is impossible to formulate verification obligations that would allow for reasoning about code inheritance in a modular manner and without defeating its flexibility. To enable modular reasoning, code inheritance should be disciplined. The study described above has led us to the observation that a significant part of the problem can be traced back to dynamic binding of self-calls in a base class. If an extension class overrides a particular method of the base class, a self-call to this method in the base class gets redirected to the overriding definition, due to dynamic binding. In the following sections we study disciplined inheritance that restricts code inheritance by prohibiting dynamic binding of self-calls.

6.4.4 Disciplining Inheritance

In section 6.2 we defined the operator **upcalls** which models a restricted form of inheritance in which all external calls to a class are resolved dynamically, while internal self-calls are resolved statically. We refer to inheritance restricted in this manner as *disciplined inheritance*. The *modular reasoning property for the disciplined inheritance* can be formulated as follows:

$$\mathbf{C} \sqsubseteq \mathbf{D} \;\Rightarrow\; (\boldsymbol{L}\ \mathbf{upcalls}\ \mathbf{C}) \sqsubseteq (\boldsymbol{L}\ \mathbf{upcalls}\ \mathbf{D})$$

Testing this property on the examples presented above, shows that to provide for modular reasoning, we still need to strengthen the assumptions according to the "no direct access to the base class state" and "no revision self-calling assumptions" requirements. With the disciplined inheritance the other aspects of the semantic fragile base class problem do not appear. As was mentioned in section 6.2, our formalization of inheritance does not permit a direct access to instance variables of the base class from the modifier methods. According to the "no revision self-calling assumptions" requirement, while reasoning about the behavior of a revision class method, its developer should not assume the behavior of the methods it self-calls, but should consider the behavior described by the base class. The application of a function D representing methods of the revision class $\mathbf{D}$ to properly encoded method bodies $(\mu\, C)$ of the base class $\mathbf{C}$ returns a tuple of methods of $\mathbf{D}$ with all self-calls redirected to the methods of $\mathbf{C}$. Therefore imposing this requirement amounts to verifying that:

$$\exists R \bullet (R.\,d_0.\,c_0) \;\wedge\; (\mu\, C) \sqsubseteq_{\widehat{R}} D.\,(\mu\, C)\!\downarrow\!\widehat{R}$$

where $\widehat{R} = (Id \times R \times Id)$. Accordingly, the modular reasoning property for disciplined inheritance acquires the following form:

$$\begin{array}{l}(\exists R \bullet (R.\,d_0.\,c_0) \;\wedge\; (\mu\, C) \sqsubseteq_{\widehat{R}} D.\,(\mu\, C)\!\downarrow\!\widehat{R}) \;\Rightarrow \\ \qquad (\boldsymbol{L}\ \mathbf{upcalls}\ \mathbf{C}) \sqsubseteq (\boldsymbol{L}\ \mathbf{upcalls}\ \mathbf{D})\end{array}$$

Now we can formulate the *modular reasoning theorem for disciplined inheritance*:

Modular Reasoning Theorem for Disciplined Inheritance. *For classes* $\boldsymbol{C}$, $\boldsymbol{D}$, *and a modifier* $\boldsymbol{L}$ *given by*

$$\begin{array}{rcl} \boldsymbol{C} & = & (c_0 : \Sigma, C : \Phi^{n,n}(Ptran(\alpha \times \Sigma \times \Delta))) \\ \boldsymbol{D} & = & (d_0 : \Sigma', D : \Phi^{n,n}(Ptran(\alpha \times \Sigma' \times \Delta))) \\ \boldsymbol{L} & = & (l_0 : \Gamma, L : \Pi^n(Ptran(\alpha' \times \beta \times \Gamma \times \Delta)) \to \\ & & \qquad \Phi^{n,n}(Ptran(\alpha' \times \beta \times \Gamma \times \Delta))) \end{array}$$

the following holds:

$$\begin{array}{l} (\exists R \bullet (R.\, d_0.\, c_0) \ \wedge \ (\mu\ C) \sqsubseteq_{\widehat{R}} (D.\, (\mu\ C){\downarrow}\widehat{R})) \ \Rightarrow \\ \qquad (\boldsymbol{L}\ \mathbf{upcalls}\ \mathbf{C}) \sqsubseteq (\boldsymbol{L}\ \mathbf{upcalls}\ \mathbf{D}) \end{array}$$

Proof In accordance with the definition of abstract data type refinement, we first need to show that $R.\, d_0.\, c_0 \ \Rightarrow \ (R \times Id).\, (d_0, l_0).\, (c_0, l_0)$, which is trivially true.

Next we need to show the following goal:

$$\begin{array}{l} (\mu\ C) \sqsubseteq_{\widehat{R}} D.\, (\mu\ C){\downarrow}\widehat{R} \ \Rightarrow \\ \qquad (\mu\ \lambda self \bullet L.\, self.\, (\mu\ C){\downarrow}P) \sqsubseteq_{\widehat{R} \times Id} \\ \qquad\qquad (\mu\ \lambda self \bullet L.\, self.\, (\mu\ D){\downarrow}P) \end{array} \tag{6.1}$$

Recall that the relation P discards the first element of the state and swaps the second and the third elements of the state.

As was already mentioned, we assume that the classes and the modifier do not have recursive and mutually recursive methods. Therefore, it is always possible to rearrange their methods in the linear order in the following manner. We can assign an index to each method in **C**, **D**, and ***L***, according to the depth of the call graph. That is if a method does not self-call any other methods, it is assigned index 1. If a method invokes a method with index i, it receives the index $i+1$. If a method invokes several methods with different indexes, its index becomes the maximum of these indexes plus one. An example presented in figure 6.9 illustrates the assignment of indexes to methods. Note that implementations of a certain method in **C**, **D**, and ***L*** can receive different indexes, as they can introduce or remove self-calls. In this case, the corresponding methods in **C**, **D**, and ***L*** are re-assigned the

$$
\begin{array}{ll}
\begin{array}{l}
\mathbf{C}\ = \\
\mathbf{class} \\
\quad k^1() \ \widehat{=}\ S, \\
\quad l^2() \ \widehat{=}\ \mathit{self}\rightarrowtriangle m(); \mathit{self}\rightarrowtriangle k(), \\
\quad m^1() \ \widehat{=}\ T, \\
\quad n^3() \ \widehat{=}\ \mathit{self}\rightarrowtriangle l() \\
\mathbf{end}
\end{array}
&
\begin{array}{l}
\boldsymbol{L}\ = \\
\mathbf{modifier} \\
\quad k^1() \ \widehat{=}\ \mathit{super}\rightarrowtriangle k(); W, \\
\quad l^2() \ \widehat{=}\ \mathit{self}\rightarrowtriangle m(); \mathit{self}\rightarrowtriangle k(), \\
\quad m^1() \ \widehat{=}\ \mathit{super}\rightarrowtriangle m(), \\
\quad n^1() \ \widehat{=}\ \mathit{super}\rightarrowtriangle n() \\
\mathbf{end}
\end{array}
\end{array}
$$

$$
\begin{array}{ll}
\mathbf{D}\ = & \mathbf{class} \\
& \quad k^2() \ \widehat{=}\ \mathit{self}\rightarrowtriangle m(), \\
& \quad l^2() \ \widehat{=}\ \mathit{self}\rightarrowtriangle m(); U, \\
& \quad m^1() \ \widehat{=}\ V, \\
& \quad n^2() \ \widehat{=}\ \mathit{self}\rightarrowtriangle m() \\
& \mathbf{end}
\end{array}
$$

Figure 6.9: Example of method indexing

index which is the maximum of the indexes these methods received in the previous step. Accordingly, in our example, the methods k, l, m, and n of **C**, **D**, and ***L*** receive indexes 2, 2, 1, and 3 respectively. Next the methods of **C**, **D**, and ***L*** are sorted according to the received indexes. Without loss of generality, we can consider the case when for every distinct index there is only one method and the indexes increase sequentially. We represent the methods by functions of the methods they invoke:

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

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

$$C = \lambda \mathit{self} \bullet (C_1.\,(), C_2.\,(x_1), ..., C_n.\,(x_1, ..., x_{n-1}))$$

Note that in goal 6.1 the data refinement relations connect tuples of predicate transformers that 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

$$\begin{array}{l}(\mathcal{C}_1,\ldots,\mathcal{C}_n) \sqsubseteq_{\widehat{R}} (\mathcal{D}_1,\ldots,\mathcal{D}_n) \;\Rightarrow \\ \quad ((\mathcal{L}_1,\ldots,\mathcal{L}_n) \sqsubseteq_{\widehat{R}\times Id} (\mathcal{M}_1,\ldots,\mathcal{M}_n) \;\wedge \\ \quad (\mathcal{C}_1,\ldots,\mathcal{C}_n) \sqsubseteq_{\widehat{R}} (\mathcal{T}_1,\ldots,\mathcal{T}_n)),\end{array} \tag{6.2}$$

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

$$\begin{array}{lll}\mathcal{C}_1 = C_1.\,(), & \ldots & \mathcal{C}_n = C_n.\,(\mathcal{C}_1,...,\mathcal{C}_{n-1}) \\ \mathcal{D}_1 = D_1.\,()\downarrow\widehat{R}, & \ldots & \mathcal{D}_n = D_n.\,(\mathcal{C}_1,...,\mathcal{C}_{n-1})\downarrow\widehat{R} \\ \mathcal{L}_1 = L_1.\,().\,\mathcal{C}_1\downarrow P, & \ldots & \mathcal{L}_n = L_n.\,(\mathcal{L}_1,...,\mathcal{L}_{n-1}).\,(\mathcal{C}_1,...,\mathcal{C}_n)\downarrow P \\ \mathcal{M}_1 = L_1.\,().\,\mathcal{T}_1\downarrow P, & \ldots & \mathcal{M}_n = L_n.\,(\mathcal{M}_1,...,\mathcal{M}_{n-1}).\,(\mathcal{T}_1,...,\mathcal{T}_n)\downarrow P \\ \mathcal{T}_1 = D_1.\,(), & \ldots & \mathcal{T}_n = D_n.\,(\mathcal{T}_1,...,\mathcal{T}_{n-1})\end{array}$$

Here $\mathcal{C}$ are the method bodies of the methods $C_1,...,C_n$ with all self-calls recursively resolved with $\mathcal{C}$, similarly $\mathcal{T}$ represents the method bodies $(\mu\ D)$. The statements $\mathcal{D}$ represent $(D.\,(\mu\ C)\downarrow\widehat{R})$, where each $\mathcal{D}_i$ is a method body of the method D_i with all self-calls resolved with properly coerced $\mathcal{C}$. Note how $\mathcal{L}$ and $\mathcal{C}$ jointly represent the least fixed point of methods of (***L*** **upcalls C**). The statements $\mathcal{L}$ stand for the methods of the modifier ***L*** with calls to *self* resolved with $\mathcal{L}$ themselves and calls to *super* resolved with $\mathcal{C}$. Similarly, $\mathcal{L}$ and $\mathcal{T}$ jointly represent the least fixed point of methods of (***L*** **upcalls D**).

In the proof of the theorem, we need the following lemma.

Decoding Propagation Lemma. *For a method* $L_n : \Pi^{n-1}(Ptran(\alpha'\times\beta\times\Gamma\times\Delta) \rightarrow \Phi^{n,n}(Ptran(\alpha'\times\beta\times\Gamma\times\Delta))$*, a tuple of statements* $(\mathcal{M}_1,...,\mathcal{M}_{n-1}) : \Pi^{n-1}(Ptran(\alpha'\times\Sigma\times\Gamma\times\Delta)$*, a tuple of statements* $(\mathcal{T}_1,...,\mathcal{T}_n) : \Pi^n(Ptran(\alpha\times\Sigma'\times\Delta))$*, a relation* $R : \Sigma' \leftrightarrow \Sigma$*, and a relation* $P = \lambda(x',y',z',u')\bullet\lambda(z,y,u)\bullet(y'=y \;\wedge\; z'=z \;\wedge\; u'=u)$*, the following holds*:

$$\begin{array}{l}L_n.\,(\mathcal{M}_1,\ldots,\mathcal{M}_{n-1})\uparrow(\widehat{R}\times Id).\,(\mathcal{T}_1,\ldots,\mathcal{T}_n)\downarrow P\uparrow(\widehat{R}\times Id) \;\sqsubseteq \\ \quad (L_n.\,(\mathcal{M}_1,\ldots,\mathcal{M}_{n-1}).\,(\mathcal{T}_1,\ldots,\mathcal{T}_n)\downarrow P)\uparrow(\widehat{R}\times Id)\end{array}$$

Proof According to the definition of data refinement, the goal can be rewritten as follows:

$$\begin{array}{l}(L_n.\,(\mathcal{M}_1,\ldots,\mathcal{M}_{n-1})\uparrow(\widehat{R}\times Id).\,(\mathcal{T}_1,\ldots,\mathcal{T}_n)\downarrow P\uparrow(\widehat{R}\times Id))\downarrow(\widehat{R}\times Id) \;\sqsubseteq \\ \quad L_n.\,(\mathcal{M}_1,\ldots,\mathcal{M}_{n-1}).\,(\mathcal{T}_1,\ldots,\mathcal{T}_n)\downarrow P\end{array}$$

A body of the method L_n with bodies of the corresponding methods substituted for the self and super-called methods can be modeled similarly to modeling module clients, as presented in Chapter 3. Accordingly, the goal acquires the following form:

$$\left(\begin{array}{l} \textbf{var}\ x, m, l, d \bullet (\mathit{skip} \times [l, d := \underline{l}, \underline{d} \mid \underline{l} = l_0 \ \wedge\ p]); \\ \quad \textbf{do} \quad \lozenge_{i=1}^{n-1} \widehat{g}_i :: \mathcal{M}_i {\uparrow} (\widehat{R} \times \mathit{Id}); \widehat{S}_i \ \lozenge \\ \quad\qquad\ \lozenge_{j=1}^{n} \widehat{q}_j :: \mathcal{T}_j {\downarrow} P {\uparrow} (\widehat{R} \times \mathit{Id}); \widehat{K}_j \\ \quad \textbf{od} \end{array}\right) {\downarrow} (\widehat{R} \times \mathit{Id}) \sqsubseteq$$
$$\left(\begin{array}{l} \textbf{var}\ x, m', l, d \bullet (\mathit{skip} \times [l, d := \underline{l}, \underline{d} \mid \underline{l} = l_0 \ \wedge\ p]); \\ \quad \textbf{do} \quad \lozenge_{i=1}^{n-1} \widehat{g}_i :: \mathcal{M}_i; \widehat{S}_i \ \lozenge \\ \quad\qquad\ \lozenge_{j=1}^{n} \widehat{q}_j :: \mathcal{T}_j {\downarrow} P; \widehat{K}_j \\ \quad \textbf{od} \end{array}\right)$$

The variables $x : \alpha'$ are the "place holders" to be substituted with real instance variables of a modifier in the next modifier application. The variables $m : \Sigma$ are the instance variables of the base class, the variables $l : \Gamma$ are the instance variables of the modifier, while the variables $d : \Delta$ represent method parameters. Prior to making self and super-calls, variables representing method parameters can be assigned values to model parameter passing, as modeled by the statement $(\mathit{skip} \times [l, d := \underline{l}, \underline{d} \mid \underline{l} = l_0 \ \wedge\ p])$. Note that according to the "no direct access to the base class state" requirement, this statement skips on the instance variables of the base class (and on the variables x). The predicates $\widehat{q} : \mathcal{P}(\alpha' \times \beta \times \Gamma \times \Delta)$ are the asserted conditions on the method arguments and the instance variables of the modifier. As the state of the base class is encapsulated, the predicates $\widehat{q}$ do not refer to the instance variables of the base class, i.e., $\widehat{q} = (\mathit{true} \times q)$. The statements $\widehat{S} = (\textbf{skip} \times S)$ and $\widehat{K} = (\textbf{skip} \times K)$, with S and K operating on the state $\Gamma \times \Delta$, model arbitrary actions the modifier can perform on its instance variables and method parameters between the method calls.

We prove the goal by starting with the left-hand side and refining it to the right-hand side as follows:

$$\left(\begin{array}{l} \textbf{var}\ x, m, l, d \bullet (\mathit{skip} \times [l, d := \underline{l}, \underline{d} \mid \underline{l} = l_0 \ \wedge\ p]); \\ \quad \textbf{do} \quad \lozenge_{i=1}^{n-1} \widehat{g}_i :: \mathcal{M}_i {\uparrow} (\widehat{R} \times \mathit{Id}); \widehat{S}_i \ \lozenge \\ \quad\qquad\ \lozenge_{j=1}^{n} \widehat{q}_j :: \mathcal{T}_j {\downarrow} P {\uparrow} (\widehat{R} \times \mathit{Id}); \widehat{K}_j \\ \quad \textbf{od} \end{array}\right) {\downarrow} (\widehat{R} \times \mathit{Id})$$
$$\sqsubseteq \quad \{\textit{sequential composition rule 1.3}\,\}$$

$$\begin{array}{ll}
& \textbf{var}\ x, m', l, d \bullet (skip \times [l, d := \underline{l}, \underline{d} \mid \underline{l} = l_0\ \wedge\ p])\!\downarrow\!(\widehat{R} \times Id); \\
& \left(\begin{array}{ll} \textbf{do} & \lozenge_{i=1}^{n-1} \widehat{g}_i :: \mathcal{M}_i\!\uparrow\!(\widehat{R} \times Id); \widehat{S}_i\ \lozenge \\ & \lozenge_{j=1}^{n} \widehat{q}_j :: \mathcal{T}_j\!\downarrow\! P\!\uparrow\!(\widehat{R} \times Id); \widehat{K}_j \\ \textbf{od} & \end{array}\right)\!\downarrow\!(\widehat{R} \times Id) \\
\sqsubseteq & \left\{\begin{array}{l} \textit{properties of indifferent statements 1.7,} \\ \textit{iterative choice rule 1.6, sequential composition rule 1.3} \end{array}\right\} \\
& \textbf{var}\ x, m', l, d \bullet (skip \times [l, d := \underline{l}, \underline{d} \mid \underline{l} = l_0\ \wedge\ p]); \\
& \quad \textbf{do} \\
& \quad\ \lozenge_{i=1}^{n-1}(\{\widehat{R} \times Id\}.\,\widehat{g}_i) :: \mathcal{M}_i\!\uparrow\!(\widehat{R} \times Id)\!\downarrow\!(\widehat{R} \times Id); \widehat{S}_i\!\downarrow\!(\widehat{R} \times Id) \\
& \quad\ \lozenge \\
& \quad\ \lozenge_{j=1}^{n}(\{\widehat{R} \times Id\}.\,\widehat{q}_j) :: \mathcal{T}_j\!\downarrow\! P\!\uparrow\!(\widehat{R} \times Id)\!\downarrow\!(\widehat{R} \times Id); \widehat{K}_j\!\downarrow\!(\widehat{R} \times Id) \\
& \quad \textbf{od} \\
\sqsubseteq & \left\{\begin{array}{l} \textit{rules 1.14, 1.1,} \\ \textit{properties of indifferent statements 1.7} \end{array}\right\} \\
& \textbf{var}\ x, m', l, d \bullet (skip \times [l, d := \underline{l}, \underline{d} \mid \underline{l} = l_0\ \wedge\ p]); \\
& \quad\quad \begin{array}{ll} \textbf{do} & \lozenge_{i=1}^{n-1} \widehat{g}_i :: \mathcal{M}_i; \widehat{S}_i\ \lozenge \\ & \lozenge_{j=1}^{n} \widehat{q}_j :: \mathcal{T}_j\!\downarrow\! P; \widehat{K}_j \\ \textbf{od} & \end{array}
\end{array}$$

□

We prove goal 6.2 by induction on the index of methods. Consider first the base step, when a method in **C**, **D**, and ***L*** does not self-call other methods. The proof obligation in this case is as follows:

$$\mathcal{C}_1 \sqsubseteq_{\widehat{R}} \mathcal{D}_1\ \Rightarrow\ (\mathcal{L}_1 \sqsubseteq_{\widehat{R} \times Id} \mathcal{M}_1\ \wedge\ \mathcal{C}_1 \sqsubseteq_{\widehat{R}} \mathcal{T}_1)$$

Assuming the antecedent, we prove the consequent of this goal as follows:

$$\begin{array}{ll}
& \mathcal{L}_1 \sqsubseteq_{\widehat{R} \times Id} \mathcal{M}_1\ \wedge\ \mathcal{C}_1 \sqsubseteq_{\widehat{R}} \mathcal{T}_1 \\
= & \{\textit{definitions, definition of data refinement}\} \\
& L_1.\,().\,\mathcal{C}_1\!\downarrow\! P \sqsubseteq (L_1.\,().\,\mathcal{T}_1\!\downarrow\! P)\!\uparrow\!(\widehat{R} \times Id)\ \wedge\ \mathcal{C}_1 \sqsubseteq \mathcal{T}_1\!\uparrow\!\widehat{R} \\
\Leftarrow & \{\textit{assumption, decoding propagation lemma}\} \\
& L_1.\,().\,\mathcal{C}_1\!\downarrow\! P \sqsubseteq L_1.\,()\!\uparrow\!(\widehat{R} \times Id).\,\mathcal{T}_1\!\downarrow\! P\!\uparrow\!(\widehat{R} \times Id)\ \wedge\ \mathcal{C}_1 \sqsubseteq \mathcal{T}_1\!\uparrow\!\widehat{R} \\
\Leftarrow & \{\textit{monotonicity of } L_1\textit{, definition of data refinement}\} \\
& \mathcal{C}_1 \sqsubseteq \mathcal{T}_1\!\downarrow\! P\!\uparrow\!(\widehat{R} \times Id)\!\uparrow\! P\ \wedge\ \mathcal{C}_1 \sqsubseteq \mathcal{T}_1\!\uparrow\!\widehat{R}
\end{array}$$

$$\begin{array}{ll}
\Leftarrow & \{\textit{rule 1.13, rule 1.2}\} \\
& \mathcal{C}_1 \sqsubseteq \mathcal{T}_1{\uparrow}\widehat{R} \\
\Leftarrow & \{\textit{definitions}\ \} \\
& C_1.\,() \sqsubseteq (D_1.\,()){\uparrow}\widehat{R} \\
\Leftarrow & \{\textit{assumption, definition of data refinement, definitions}\} \\
& T
\end{array}$$

Consider now the inductive step. The inductive assumption for the inductive case states that the goal holds for n methods in the participating entities. After simple logic transformations, our proof obligation for $n+1$ methods is:

$$\begin{array}{ll}
(\mathcal{C}_1, \ldots, \mathcal{C}_{n+1}) \sqsubseteq_{\widehat{R}} (\mathcal{D}_1, \ldots, \mathcal{D}_{n+1}) \ \wedge & (a) \\
(\mathcal{L}_1, \ldots, \mathcal{L}_n) \sqsubseteq_{\widehat{R}\times Id} (\mathcal{M}_1, \ldots, \mathcal{M}_n) \ \wedge & (b) \\
(\mathcal{C}_1, \ldots, \mathcal{C}_n) \sqsubseteq_{\widehat{R}} (\mathcal{T}_1, \ldots, \mathcal{T}_n) \ \Rightarrow & (c) \\
\quad \mathcal{L}_{n+1} \sqsubseteq_{\widehat{R}\times Id} \mathcal{M}_{n+1} \ \wedge \ \mathcal{C}_{n+1} \sqsubseteq_{\widehat{R}} \mathcal{T}_{n+1} &
\end{array}$$

Assuming the antecedent, we prove the consequent of this goal as follows:

$$\begin{array}{ll}
& \mathcal{L}_{n+1} \sqsubseteq_{\widehat{R}\times Id} \mathcal{M}_{n+1} \ \wedge \ \mathcal{C}_{n+1} \sqsubseteq_{\widehat{R}} \mathcal{T}_{n+1} \\
= & \{\textit{definitions, definition of data refinement}\} \\
& L_{n+1}.\,(\mathcal{L}_1, \ldots, \mathcal{L}_n).\,(\mathcal{C}_1, \ldots, \mathcal{C}_{n+1}){\downarrow}P \sqsubseteq \\
& \qquad (L_{n+1}.\,(\mathcal{M}_1, \ldots, \mathcal{M}_n).\,(\mathcal{T}_1, \ldots, \mathcal{T}_{n+1}){\downarrow}P){\uparrow}(\widehat{R} \times Id) \ \wedge \\
& \qquad\quad \mathcal{C}_{n+1} \sqsubseteq (\mathcal{T}_{n+1}){\uparrow}\widehat{R} \\
\Leftarrow & \left\{ \begin{array}{l} \textit{monotonicity of } L_{n+1}, \textit{assumption } (b), \\ \textit{decoding propagation lemma} \end{array} \right\} \\
& L_{n+1}.\,(\mathcal{M}_1, \ldots, \mathcal{M}_n){\uparrow}(\widehat{R} \times Id).\,(\mathcal{C}_1, \ldots, \mathcal{C}_{n+1}){\downarrow}P \sqsubseteq \\
& \qquad L_{n+1}.\,(\mathcal{M}_1, \ldots, \mathcal{M}_n){\uparrow}(\widehat{R} \times Id).\,(\mathcal{T}_1, \ldots, \mathcal{T}_{n+1}){\downarrow}P{\uparrow}(\widehat{R} \times Id) \ \wedge \\
& \qquad\quad \mathcal{C}_{n+1} \sqsubseteq \mathcal{T}_{n+1}{\uparrow}\widehat{R} \\
\Leftarrow & \{\textit{monotonicity of } L_{n+1}, \textit{definition of data refinement}\} \\
& (\mathcal{C}_1, \ldots, \mathcal{C}_{n+1}) \sqsubseteq (\mathcal{T}_1, \ldots, \mathcal{T}_{n+1}){\downarrow}P{\uparrow}(\widehat{R} \times Id){\uparrow}P \ \wedge \\
& \qquad\quad \mathcal{C}_{n+1} \sqsubseteq \mathcal{T}_{n+1}{\uparrow}\widehat{R} \\
\Leftarrow & \{\textit{rule 1.13, rule 1.2}\} \\
& (\mathcal{C}_1, \ldots, \mathcal{C}_{n+1}) \sqsubseteq (\mathcal{T}_1, \ldots, \mathcal{T}_{n+1}){\uparrow}\widehat{R} \ \wedge \ \mathcal{C}_{n+1} \sqsubseteq \mathcal{T}_{n+1}{\uparrow}\widehat{R} \\
\Leftarrow & \{\textit{assumption } (c)\}
\end{array}$$

$$
\begin{array}{ll}
 & \mathcal{C}_{n+1} \sqsubseteq \mathcal{T}_{n+1}{\uparrow}\widehat{R} \;\wedge\; \mathcal{C}_{n+1} \sqsubseteq \mathcal{T}_{n+1}{\uparrow}\widehat{R} \\
\Leftarrow & \{\textit{assumption (a)}\} \\
 & \mathcal{D}_{n+1}{\uparrow}\widehat{R} \sqsubseteq \mathcal{T}_{n+1}{\uparrow}\widehat{R} \\
\Leftarrow & \{\textit{definitions, monotonicity of encoding}\} \\
 & D_{n+1}.\,(\mathcal{C}_1,\ldots,\mathcal{C}_n){\downarrow}\widehat{R} \sqsubseteq D_{n+1}.\,(\mathcal{T}_1,\ldots,\mathcal{T}_n) \\
\Leftarrow & \left\{\begin{array}{l}\textit{monotonicity of } D_{n+1}\textit{, assumption (c),} \\ \textit{definition of data refinement}\end{array}\right\} \\
 & T
\end{array}
$$

□

6.4.5 Discussion of Safety of Inheritance

The study presented in sections 6.4.1-6.4.3 demonstrates that reasoning about unrestricted inheritance in a modular fashion is infeasible. From this we conclude that *in general, code inheritance is an unsafe software reuse mechanism.* The analysis of the fragile base class problem presented above demonstrates how tightly code inheritance binds the new code of the modifier with the inherited code. Even in a very restricted setting, it is impossible to substitute the base class with its revision without invalidating extensions.

The problems with the safety of code inheritance were recognized by many researchers [72, 83, 20] and various solutions were proposed. Most notably, *interface inheritance* is widely accepted as a less flexible, but still very useful software reuse technique [4, 3, 34]. Interface inheritance and code inheritance are closely related. The names of these reuse techniques provide good intuition on their essence, as they explicitly state which part of a base class can be inherited by an extension, the interface or the implementation. With code inheritance, the extension class inherits the implementation of the base class, i.e., its instance variables and its method definitions. With interface inheritance the extension inherits exclusively the interface from its base class. Interface inheritance is primarily used for establishing syntactic compatibility between objects. This, in turn, permits creating polymorphic programs that can operate on objects of different but compatible types.

No explicit language support is required to support interface inheritance, provided that code inheritance is supported. Programs employing interface inheritance usually define so-called *abstract classes* whose sole purpose is to identify interfaces to be inherited by their extensions. Definitions of methods in such abstract classes usually contain only `halt` statements, which helps during debugging, should the abstract class accidentally be instantiated. Some object-oriented languages provide explicit support for the interface inheritance technique, while others go even further and incorporate it as a software reuse mechanism. For example, in C++ one can define an abstract class by marking all of its methods with '=0'. An attempt to create an object of this class will be caught by a compiler as an error. In Java the type that a class implements is declared separately using the keyword `interface`. An interface is, in fact, nothing other than an abstract class.

The model of classes and inheritance presented in this chapter applies equally well to abstract classes and interface inheritance. An abstract class is a tuple $(arb, \lambda self \bullet (\mathbf{abort}_1, ..., \mathbf{abort}_n))$, where arb is an arbitrary value initializing a "ghost" instance variable of the abstract class. The function $\lambda self \bullet (\mathbf{abort}_1, ..., \mathbf{abort}_n)$ represents methods of the abstract class with the method bodies modeled by the statement **abort**. Interface inheritance then is modeled by the application of a modifier to such an abstract class using the operator **mod**. The developers of the modifier should define an internal state and implementations of every method. Obviously, method implementations in the modifier should not contain super-calls, as otherwise the resulting methods will also be aborting.

As with interface inheritance it is only the interface of the base class that gets inherited, the semantic fragile base class problem cannot arise. The mathematical justification for this observation is that for interface inheritance the modular reasoning property holds trivially. From this we conclude that *interface inheritance is a safe software reuse technique.*

In section 6.4.4 we considered a restricted version of code inheritance – disciplined inheritance – and proved that it can be reasoned about in a modular manner. From this we conclude that at least under the restrictions we imposed in the formal model, *the disciplined inheritance is a safe software reuse mechanism.*

6.4.6 Check-List for Verifying Disciplined Inheritance

Based on the modular reasoning theorem for disciplined inheritance, we can formulate a check-list for informally verifying correctness of a revision class with respect to the original base class, which preserves correctness of extensions created through disciplined inheritance. Under the assumption that the original class does not have recursive and mutually recursive methods, and overriding methods of the extensions do not access the state of its base class directly, a developer of the revision should verify that the following conditions are satisfied:

- The initial values of the revision instance variables correspond to the initial values of the instance variables of the base class.
- For every method, the precondition p of the method definition as given in the base class when expressed on the instance variables of the revision class is stronger than or equal to the precondition p' of this method definition as given in the revision class, i.e., $p \subseteq p'$.
- For every method, one must consider the postcondition q of the base class method definition as expressed on the instance variables of the revision class and the result and value-result method parameters. One must also consider the postcondition q' of the method definition constructed by substituting self-calls in the corresponding definition of the revision method with the definitions of the corresponding methods of the base class expressed on the revision instance variables. Then, q' must be stronger than or equal to q, i.e., $q' \subseteq q$.

In the following sections we discuss a possible implementation of the disciplined inheritance. We also consider the flexibility of code inheritance, interface inheritance, and disciplined inheritance and discuss the trade-offs between the safety and flexibility offered by these reuse techniques.

6.5 Is Inheritance Flexible?

The flexibility of inheritance is the main reason for its popularity. Inheritance supports inline modification of code, i.e., unlike other software reuse mechanisms and techniques, not only a developer can add functionality to an existing system but also can modify the existing functionality. These possibilities promoted the construction of semi-finished programs, known as object-oriented frameworks, providing the core functionality of an application and expecting user extensions to add the “interesting” parts.

Studies of object-oriented frameworks allowed to extract a number of recurring class composition patterns [25]. One of these patterns, known as the “Template Method”, can serve as a good illustration of flexibility of inheritance. Essentially, this pattern is used for defining the skeleton

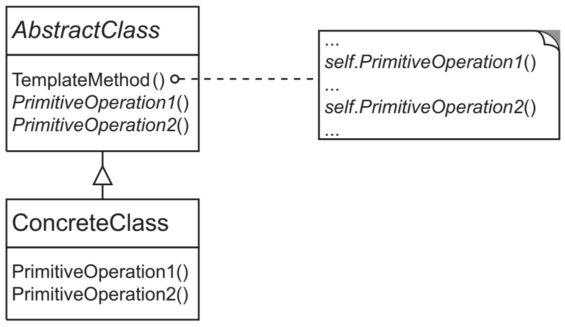

Figure 6.10: Class diagram of the Template Method pattern.

of an algorithm in an operation, deferring some steps to subclasses. The following description of this pattern is based on [25]. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure, as illustrated in figure 6.10. The class *AbstractClass* implements a template method defining the skeleton of an algorithm, and defines abstract primitive operations that concrete subclasses must define to implement steps of the algorithm. The template method of *AbstractClass* calls primitive operations as well as other operations defined in *AbstractClass*. The class *ConcreteClass* implements the primitive operations to carry out subclass-specific steps of the algorithm. As rightfully noted by the authors of [25], "Template methods are a fundamental technique for code reuse. They are particularly important in class libraries, because they are the means for factoring out common behavior in library classes."

Part of the flexibility of inheritance comes from the fact that an extension class can access the base class state, thus permitting to reuse the data structure of the base class. In the example above, the deference of a part of the algorithm implementation to subclasses is possible exactly because the subclasses inherit the data structure of their base class and can define operations on this data structure. Furthermore, the dynamic binding of self-calls in the base class allows for a fine degree of adjustment of the inherited behavior, contributing to the flexibility of inheritance. Unfortunately, as we demonstrated above, exactly these features appear to be the most troublesome with respect to safety. Flexibility is at odds with safety.

Interface inheritance is the basis for subtype polymorphism. Subtype polymorphism allows for combining the flexibility offered by untyped pro-

```
A =                                              B =
class                                            class inherits A
               $m() \mathrel{\widehat{=}} S_1; \mathit{self}\rightarrowtriangle n(),$            $m() \mathrel{\widehat{=}} T_1; \mathit{super}\rightarrowtriangle m(),$
 selfbound     $n() \mathrel{\widehat{=}} S_2; \mathit{self}\rightarrowtriangle l(),$            $n() \mathrel{\widehat{=}} T_2; \mathit{self}\rightarrowtriangle l(),$
               $l() \mathrel{\widehat{=}} S_3$                          $l() \mathrel{\widehat{=}} T_3$
end                                              end
```

Figure 6.11: Example of the method qualifier `selfbound`.

gramming languages with the advantages of strong typing offered by typed languages. The flexibility is achieved through permitting client code use objects of subtypes the way it uses objects of supertypes. In a way, interface inheritance complements code inheritance in that it allows for reusing client code, whereas code inheritance allows for reusing class code. Not only is interface inheritance flexible, but also, as we argued above, it is safe.

Obviously, it would be beneficial to obtain a safe code reuse mechanism having the flexibility of interface inheritance and providing as much flexibility of code inheritance as possible. In section 6.4.4 we showed that resolving self-calls of base classes statically permits one to reason about code inheritance disciplined in this way in a modular fashion. Unlike the proposals of other researchers that we discuss in section 6.6, our approach to disciplining inheritance can be easily implemented as a slight modification of the standard code inheritance mechanism. Disciplined inheritance can be easily superimposed on the implementation of ordinary code inheritance, by introducing an additional class qualifier `selfbound`. The meaning of this qualifier is that if a class is marked as `selfbound`, all self-calls within this class are always resolved statically, i.e., disregarding the dynamic type of the corresponding object. Methods of a class declared as `selfbound` can be overridden in a subclass and external calls to these methods are resolved dynamically. This proposal can be further extended by qualifying individual methods rather than entire classes as `selfbound`. Qualifying a method as `selfbound` would mean that all self-calls to this method are resolved statically, i.e., always remain within the same class. A class would then be selfbound if all of its methods are selfbound. Consider figure 6.11 demonstrating the application of the method qualifier `selfbound`. Suppose we have an object b which is an instance of the class **B**. The method call $b{\rightarrowtriangle}m()$ results in the sequence of statements $T_1; S_1; S_2; T_3$. However, if we assign b to a variable a declared to hold instances of the class **A**, an external call to n on a is resolved dynamically, resulting in the sequence of statements $T_2; T_3$.

An immediate question arises whether this functionality can be implemented with the currently available facilities of existing programming languages. A method in a C++ class can be qualified as `virtual`, which means that all calls to this method are resolved in accordance with the dynamic type of the corresponding object. If a method is not qualified as virtual, it is statically bound, i.e., regardless of whether it is a self-call or an external call, the dynamic type of the object is disregarded, and only its static type determines which method is invoked. A subclass can redefine a method of the base class declared as "non-virtual", but the new definition cannot be accessed through subsumption. Thus in C++ there is no explicit method or class qualifier that would have the functionality of the qualifier `selfbound`. We are not aware of a class-based code reuse mechanism in any object-oriented language that would be similar to the one defined by the **upcalls** operator.

In C++, however, when invoking a method on an object, it is possible to explicitly indicate from which class this method should be called: from the class of the object or one of its superclasses. When invoking a method on `this` (which is the counterpart of *self* in C++), one can indicate that the method of the current class must be invoked, for example as `this->C::m()`. Hence, the functionality modeled by the operator **upcalls** can be directly implemented in C++. The difference between using the qualifier `selfbound` and using the mechanism available in C++ is that the former permits qualification per method, while the latter per method call. We believe that qualifying methods is advantageous, as it promotes a particular design of classes.

Functionality similar to that of the **upcalls** operator can be achieved using object composition as done in the Microsoft COM code reuse mechanisms *containment* and *aggregation* [87]. It is argued by COM users and developers that significant software reuse can be achieved through using these mechanisms without having to deal with pitfalls of full code inheritance. With containment, functionality of a COM component can be reused essentially according to the forwarding technique: a forwarding component, called an *outer* component, holds an exclusive reference to the reused component, called an *inner* component, and forwards requests for method calls to this inner object transparently for the user. In the case when the reused component exhibits exactly the required functionality, in order to avoid a forwarding overhead, the aggregation mechanism can be used. With this mechanism, the interface of the reused component can be exposed as a part of the interface of the resulting component. Being essentially object composition mechanisms, containment and aggregation are implemented through intricate interconnections of references. Mechanism of memory deallocation in COM is based on reference counting and, as such, is error-prone and burdensome for users. We believe that implementing the same functionality as offered by containment and aggregation as a class composition mechanism using the qualifier `selfbound` would drastically simplify resulting programs.

6.6 Related Work

Inheritance is extremely powerful but problematic. Understanding the behavior of a class in a class hierarchy is complicated, since this behavior can depend on the behavior of any other class situated above the considered class in the hierarchy. When subtyping is unified with subclassing, in the sense that code inheritance forms the basis for subtype polymorphism, some form of behavioral compatibility has to be established between subclasses and their superclasses to permit correct subsumption. Moreover, maintenance and evolution of class hierarchies proved to be a major concern for the object-oriented community. These problems were recognized by many researchers who made different proposals for coping with them. The proposals range from informal recommendations based on empirical expertise to complete theories supporting formal reasoning.

A major part of research concentrated on improving the correctness of class hierarchies. Gregor Kiczales and John Lamping in [38] proposed to consider a special interface between a class and its subclasses, referring to it as the *specialization interface*. Lamping in [39] observed that information about the dependence of methods on other methods of the same class is crucial for developing correct extensions. He proposed to declare such dependencies statically and include this information in the type of *specialization interface* of a class for possible verification by a compiler. In the case of acyclic method dependencies, methods can be arranged in layers. When methods are recursively dependent, they form a group. Developers of an extension class are then required to always override the entire group, should the need arise to override one method of this group. An extension class can redefine dependencies for overridden or new methods, as it offers a fresh specialization interface to its subclasses. A subclass has to propagate unmodified parts of the specialization interface where inherited methods are used. If extension developers want to alter the layering of methods, they have to redefine all layers above those effected by the change. Although beneficial for documentation and suitable for compiler typechecking, this proposal was never developed into a practically applicable method supported by a type checker.

The most well-known approach to establishing behavioral conformance along with syntactic one is known as *behavioral subtyping* and extensively studied by Pierre America, Barbara Liskov, Jeannette Wing, Gary Leavens, and others [2, 3, 44, 40, 23]. The essence of behavioral subtyping is to associate behavior of objects with their types (interfaces) and to identify subtypes that conform to their supertypes not only syntactically, but also semantically. The behavior is specified in terms of pre- and postconditions which is a well known and popular approach to formal specification originating from work of C.A.R. Hoare [30]. Although well-suited for specification of imperative programs, the approach using pre- and postconditions is less suitable for specifying object-oriented programs, as pointed out by many researchers. First of all, specifications in terms of pre- and postconditions fail to capture subtle interdependencies which arise due to a specific order of method invocations, especially in the presence of self-referential method calls that get redirected to subclasses of the class that originated the call. This is one of the main reasons why this approach could not have been used for the studies of the kind we have carried out in this dissertation. Our model of classes and inheritance permits reasoning about method calls and can capture interdependencies of method invocations. No less important, specifications of object-oriented programs in terms of pre- and postconditions have only semi-formal semantics, which would disallow carrying out the proofs with the necessary level of detail.

In the original works on behavioral subtyping [2, 3, 44, 40, 23], code inheritance was not explicitly considered, as it is separated from subtyping and behavioral conformance is determined between subtypes and supertypes rather than subclasses and superclasses. Raymie Stata and John Guttag in [75, 74] elaborate the idea of specialization interfaces of Kiczales and Lamping [38] by providing a mathematical foundation in the style of behavioral subtyping. They introduce *class components* that combine a substate of a class and a set of methods directly accessing this state, as units of modularity

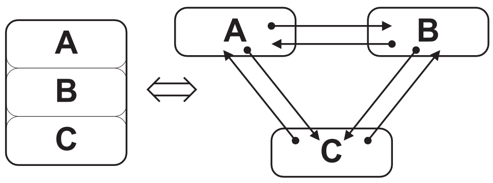

Figure 6.12: An instance of a class with several mutually dependent components corresponds to a composition of delegating objects.

for the specialization interface. Methods in a class component cannot access state in other class components directly, but only by invoking their methods. Such class components constitute a unit of overriding, i.e., if a developer of an extension class needs to override a method of a particular group, the developer is obliged to override the entire group. Furthermore, Stata and Guttag state that, to establish that an extension class is correctly substitutable for the base class, it is sufficient to ensure that a class component of the extension conforms to the specification of the corresponding base class component. While verifying an extension class component one can only take into account the specifications of the other components of the base class. It is claimed in [75, 74] that in this way one can establish the correctness of substituting the extensions for their base classes.

In the general case, class components of Stata and Guttag can be mutually dependent, referring to each other. It is easy to see that at run time, an instance of a class with several mutually dependent components can be seen as a composition of delegating objects, as illustrated in figure 6.12. In fact, this observation was also made by Szyperski in [81]. Accordingly, all our findings concerning the refinement of delegating objects, as presented in section 5, apply. Namely, while developing a method of the overriding class component in the extension, other methods in the same class component cannot be assumed and their specifications should be used instead. Moreover, special measures should be taken to prevent problems that might arise due to the accidental introduction of infinite recursion between the class components in the resulting extension class. We believe that these requirements were overlooked, because the behavioral subtyping approach to specification of object-oriented languages employs pre- and postconditions and is not expressive enough to capture complex dependencies of method invocations.

The main idea behind the approach to verifying correctness of class hierarchies introduced in [52] and further developed in [51] is essentially the same as in the behavioral subtyping, but has a number of fundamental differences. As syntactic subtyping is decidable and can be checked by a computer, while behavior-preserving subtyping is undecidable, the approach reported in [52, 51] chooses to associate the specifications of behavior with classes rather than with types, separating in this way decidable properties from undecidable ones. Classes are considered to be the carriers of behavior and compared for behavioral compatibility. The notion of *class refinement* is first defined in [52], and the later works [51] relate this notion to correctness of substitutability of subclass instances for superclass instances in clients. It is formally proved that when a class C' refines a class C, substituting instances of C' for instances of C is refinement for the clients. The model of object-oriented programs developed in [52, 51] is rather general, as the approach focuses on correctness of code inheritance when base classes are not intended to be changed. Our model of classes and inheritance is custom-tailored and more suited for studying the safety of inheritance.

So far, we have considered the related work on the subject of verification of class hierarchies disregarding the possible evolution of base classes in these hierarchies. As we have argued above, class hierarchies are subject to evolution, and therefore, while considering the safety of inheritance, it is necessary to take into account the possibility of base class changes. We have encountered several different interpretations of the fragile base class 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 [35]. While being obviously 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 [87]. Only in this case, can the extension class 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 but modifying the base class still can invalidate the extensions. 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.

Rustan Leino in [41] proposes a way of indicating in a specification of a base class "which program variables [of an extension class] are allowed to be changed by which methods [of the extension class]". The proposed specification construct is intended for use with the static program checker developed at Compaq Systems Research Center. In [42] Leino and Stata introduce into their specification notation several access control modifiers for class instance variables, with the intention to aid the program checker with verifying object invariants. While the proposed specification constructs and the corresponding restrictions on inheritance can aid to circumvent certain aspects of the semantic fragile base class problem, they are insufficient to resolve the problem, even if considered in the restricted setting as presented in this dissertation. In fact, it is easy to construct an example following the one presented in the section "Unjustified Assumption of Binding Invariant in Modifier" that would use all the constructs described by Leino and Stata, but still invalidate the modular reasoning property for inheritance.

Patrick Steyaert et al. in [77] consider 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.

The example in section 6.4.1 is, in fact, adopted from [77]. The authors 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. Clearly, this problem is just an instance of the "unjustified assumption of the binding invariant in modifier" problem presented in section 6.4.2.

As a potential solution to this problem, one can specify methods *add* and *addAll* as *consistent methods*, as was first suggested in [38]. This means that the extension developers would be disallowed to overriding one without overriding the other. However, the 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.

6.7 Conclusions

In this chapter we have considered classes and inheritance, evaluated the relationship between the safety and flexibility of inheritance, studied the fragile base class problem, and suggested the way of disciplining inheritance grounded on this study.

Based on our discussion of the safety and flexibility of inheritance, we can now consider suitability of inheritance for various kinds of object-oriented systems. In general, software systems can be divided into two broad categories – open and closed ones. As suggested by their names, open systems are open to user extensions during their life cycles, while closed systems are delivered as completed entities and cannot be extended or modified by the users.

Software component platforms and frameworks, which are the focus of much attention nowadays, are inherently open. Such systems are characterized by the late integration stage, i.e., components are composed by end users rather than their developers. Therefore, the ability to reason in a modular manner about properties of composed systems is of critical importance. Unfortunately, the fragile base class problem interferes with the modular reasoning.

The name fragile base class problem was introduced while discussing component standards [87, 35], since it has critical significance for component systems. Modification of components by their developers should not affect the component extensions of their users in any respect. Firstly, recompilation of derived classes should be avoided if possible [35]. This issue constitutes a syntactic aspect of the problem. While being important, that problem is only a technical issue. Even if recompilation is not necessary, component developers can make inconsistent modifications. The semantic aspect of the problem was recognized by COM developers [87] and led them to the decision to abandon code inheritance in favor of the object composition mechanisms, containment and delegation.[6]

[6]Surprisingly, code inheritance is reintroduced in the new release COM++. Perhaps

Based on our study of the fragile base class problem, we conclude that unrestricted code inheritance across component boundaries should not be supported by component standards, as it excludes the possibility of modular reasoning about component-based systems. However, as we argued in [48], disciplined inheritance across component boundaries can still be used without undermining safety.

In contrast with open systems, verification of closed systems can be accomplished after the final composition. Although it is not necessary to reason about classes in a modular manner, the ability to do so is obviously advantageous, as systems can be very large and sophisticated. Unrestricted code inheritance can be used in closed systems, however, maintaining such systems becomes more complicated as, after modifying a class, not only this class needs to be verified, but also all of its extensions.

Practical applicability of the disciplined inheritance represents an interesting research issue. Investigating a balance between the flexibility and the safety of the approach based on qualifying as selfbound individual methods rather than classes represents another promising research direction.

the users' demand has overweighed the developers' caution.

Chapter 7

Conclusions

In this dissertation, we studied the safety and flexibility of different software reuse mechanisms and techniques. We considered various mechanisms and techniques, starting with the most trivial – Copy&Paste of source code – and proceeding with more advanced ones – procedures, modules, objects, and object-based software reuse techniques, forwarding and delegation, as well as classes and inheritance. Each mechanism and technique that we considered was presented in a separate chapter with the presentations structured similarly. We began by introducing a mechanism or technique, then studied its safety, and discussed flexibility, illustrating it with examples of known practical applications. We also analyzed trade-offs between the safety and flexibility achieved by different mechanisms and techniques, and discussed their applicability in various kinds of software systems. Finally, in every chapter we provided an extensive overview of related work.

We argue that a software reuse mechanism or technique is safe if it is possible to reason about it in a modular manner. The central mathematical property behind our study of the safety of different mechanisms and techniques is monotonicity. We believe that the monotonicity of constructs in a programming language is of paramount importance for the safety of software reuse mechanisms and techniques that can be used in this language. If a language includes non-monotonic constructs, then even such a trivial and basic reuse technique as Copy&Paste is not guaranteed to always deliver the expected behavior. For some older mechanisms and techniques, the ways to reason modularly are well-known and widely used, while for more recent ones, the methods for modular reasoning are not yet well established.

For each software reuse mechanism (technique) under consideration, we developed a formal model within a uniform logical framework. These formal models capture the safety-related features of the mechanisms and techniques under consideration, and permit reasoning about behavioral substitutability of code fragments packaged with the corresponding mechanisms. To formalize the ways the modular reasoning is usually done in practice, we formulated the modular reasoning properties and studied whether these properties held. When such a property held, we concluded that the corresponding mechanism or technique was safe. When this was not the case, we identified the problems that invalidated the property and attempted to formulate requirements eliminating these problems. When taken into account as verification obligations, these requirements would enable modular reasoning about the mechanism or technique under consideration. For delegation, we succeeded in formulating such requirements and showed that in their presence it was possible to reason about delegating objects in a modular manner. Unfortunately, this approach did not work for inheritance. The requirements necessary to establish the corresponding modular reasoning property defeat the flexibility of inheritance, i.e. safety conflicts with flexibility. From this, we conclude that inheritance is an unsafe mechanism. We proposed a way to discipline inheritance through disallowing dynamic binding of self-calls which, according to our analysis, was the source of the problems. Such a disciplined inheritance supports modular reasoning and is therefore safe.

To conduct the studies presented in this dissertation, we extended the refinement calculus of Back and von Wright [12, 14, 13]. The refinement calculus served as an underlying framework in which we uniformly and systematically studied and compared very different mechanisms and techniques. The refinement calculus in its higher-order logic formulation is perfectly suitable for expressing object-oriented constructs that are intrinsically higher-order. The use of higher-order logic permits expressing the required constructs very precisely, yet abstracting away irrelevant details. For example, a program invoking a procedure is formalized simply as a higher-order function of the procedure which can occur in the body of the function an arbitrary finite number of times. This enables us, on the one hand, to abstract away from a particular mechanism of procedure invocation and, on the other hand, to reason about procedure invocation completely within the logic. Our formalization of delegating objects can serve as another example illustrating the power of higher-order logic. In practice, mutual reference between objects can be achieved in various ways. For instance, an object can pass a reference to itself as a parameter while invoking a method on another object, or objects can maintain constant references to each other established on object construction. In our formalization of delegating objects, we abstracted away from a particular mechanism of establishing mutual references by modeling object methods as functions of methods of the other object. This simple (but sufficient for our purposes) model was facilitated by higher order logic. Apart from being very expressive, our formalization permitted conducting proofs in a calculational style (following the proof style of [12]), which increased the trustworthiness of our results.

Application of formal methods gave a particular taxonomy of the concepts we studied in this dissertation, determining the structuring of the material. Table 7.1 organizes different data-centered code reuse mechanisms according to two features of the corresponding entities: whether they are implemented as procedural or data type abstractions, and whether they are first class values in a programming language. In informal studies of reuse mechanisms, the difference between abstract data types and procedural data abstraction (objects) is usually emphasized [18], i.e., the table is partitioned horizontally. We perceive this difference as rather being an implementation issue and consider the ability to pass a code fragment, packaged with a certain reuse mechanism, as a parameter as being a more important differentiating factor. Therefore, we emphasize vertical partitioning of the table. The ability to treat a code fragment as a first class value changes the programming style, permitting focusing on functionality rather than algorithms. Also, from a formal standpoint, this ability permits the creation of cyclic reference patterns which significantly complicates analysis. The formal analysis clearly distinguishes between forwarding and delegation, a difference which is quite

		First Class Value of a Programming Language	
		Yes	No
Implementation	Type Abstraction	opaque types of Modula2 abstract types of SML	structures of SML
	Procedural Abstraction	objects in C++, Smalltalk, Java...	modules of Modular2, Oberon...

Table 7.1: Taxonomy of data-centered code reuse mechanisms.

difficult to grasp from the informal literature on the subject.

In contemporary programming languages, all of the mechanisms and techniques that we considered are present and can be used in a complementary manner. Developers are offered a range of different mechanisms and techniques to be used in the development process. Based on the studies presented here, we can make the following general recommendation. When developing an open system or a closed system potentially subject to extensive revisions, adaptations, and customizations, the developers should choose the mechanisms and techniques which we showed to be safe, namely, procedures, modules, and forwarding. In applying delegation, care should be exercised according to the requirements stated in Chapter 5. In particular, the developers should verify delegating objects according to the check-list developed in section 5.7. When developing a closed system, i.e., a system which is not extended by the system's users, code inheritance can be used, because the system can be verified in its entirety before delivery to the users.

As future work, we envision lifting the restrictions we imposed on the mechanisms and techniques that we considered. We also intend to study other mechanisms and techniques which recently appeared, such as the inner classes of Java. In another direction, checking the practical applicability of disciplined inheritance represents an interesting research topic.

Bibliography

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] P. America. Inheritance and subtyping in a parallel object-oriented language. In J. Bezivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *ECOOP'87: European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science 276, pages 234–242, Paris, France, 1987. Springer-Verlag.

[3] P. America. Designing an object-oriented programming language with behavioral subtyping. In J. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, Lecture Notes in Computer Science 489, pages 60–90, New York, N.Y., 1991. Springer-Verlag.

[4] P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. *ACM SIGPLAN Notices*, 25(10):161–168, Oct. 1990. *Proceedings of OOPSLA/ECOOP '90*, N. Meyrowitz (editor).

[5] R. Back, A. Mikhajlova, and J. von Wright. Reasoning about interactive systems. In J. M. Wing, J. Woodcock, and J. Davis, editors, *FM'99 – World Congress On Formal Methods*, LNCS 1709, pages 1460–1476. Springer-Verlag, Sept. 1999.

[6] R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.

[7] R. J. R. Back. Procedural abstraction in the refinement calculus. Technical Report 55, Åbo Akademi, Turku, Finland, 1987.

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

[9] R. J. R. Back and M. J. Butler. Exploring summation and product operators in the refinement calculus. In B. Möller, editor, *Mathematics of Program Construction, 1995*, volume 947. Springer-Verlag, 1995.

[10] R. J. R. Back, A. Mikhajlova, and J. von Wright. Class refinement as semantics of correct subclassing. Technical Report 147, Turku Centre for Computer Science, December 1997.

[11] R. J. R. Back and J. von Wright. Programs on product spaces. Technical Report 143, Turku Centre for Computer Science, November 1997.

[12] R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, April 1998.

[13] R. J. R. Back and J. von Wright. Encoding, decoding and data refinement. Technical Report TUCS-TR-236, Turku Centre for Computer Science, Mar. 1, 1999.

[14] R. J. R. Back and J. von Wright. Products in the refinement calculus. Technical Report TUCS-TR-235, Turku Centre for Computer Science, Feb. 11, 1999.

[15] A. H. Borning. Classes versus prototypes in object-oriented languages. In *Proceedings of the ACM-IEEE Fall Joint Computer Conference, Montvale (NJ), USA*, pages 36–39, 1986.

[16] A. L. C. Cavalcanti, A. Sampaio, and J. C. P. Woodcock. An inconsistency in procedures, parameters, and substitution in the refinement calculus. *Science of Computer Programming*, 33(1):87–96, 1999.

[17] W. Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[18] W. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker et al., editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, 1991.

[19] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings OOPSLA '89*, volume 24, pages 433–443. ACM SIGPLAN notices, Oct. 1989.

[20] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, CA, Jan. 1990.

[21] O.-J. Dahl and K. Nygaard. Simula Begin. Technical report, Norsk Regnesentral (Norwegian Computing Center), Oslo/N, 1967.

[22] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998. With the assistance of Jos Coenen, Karl-Heinz Buth, Paul Gardiner, Yassine Lakhnech, and Frank Stomp.

[23] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering*, pages 258–267, Berlin, Germany, 1996.

[24] J. Feiler and A. Meadow. *Essential OpenDoc.* Addison-Wesley, 1996.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[26] P. H. Gardiner, C. E. Martin, and O. de Moor. An algebraic construction of predicate transformers. *Science of Computer Programming*, 22:21–44, 1994.

[27] P. H. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87(1):143–162, 1991.

[28] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[29] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Sun Microsystems, Mountain View, 1996.

[30] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–583, 1969.

[31] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer Verlag, 1971.

[32] C. A. R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.

[33] C. A. R. Hoare, J. He, and A. C. A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30(8):701–739, Nov. 1993.

[34] W. L. Hursch. Should superclasses be abstract? In M. Tokoro and R. Pareschi, editors, *Proceedings of ECOOP '94*, volume 821 of *Lecture Notes in Computer Science*, pages 12–31. Springer-Verlag, New York, N.Y., July 1994.

[35] IBM Corporation, Object Technology Products Group. *The System Object Model (SOM) and the Component Object Model (COM): A comparison of the technologies from a developer's perspective. White Paper.* IBM Corporation, Austin, Texas, 1994.

[36] International Business Machines Corporation. Data Processing Division. *FORTRAN.* IBM Corporation, New York, NY, USA, corrected printing edition, 1961.

[37] K. M. Kahn. Creation of computer animation from story descriptions. Technical Report AITR-540, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Aug. 1979.

[38] G. Kiczales and J. Lamping. Issues in the design and documentation of class libraries. *ACM SIGPLAN Notices*, 27(10):435–451, Oct. 1992. *OOPSLA '92 Proceedings*, Andreas Paepcke (editor).

[39] J. Lamping. Typing the specialization interface. *ACM SIGPLAN Notices*, 28(10):201–214, Oct. 1993. *OOPSLA '93 Proceedings*, Andreas Paepcke (editor).

[40] G. T. Leavens and W. E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). In N. Meyrowitz, editor, *Proceedings of OOPSLA/ECOOP '90*, volume 25(10) of *ACM SIGPLAN Notices*, pages 212–223. ACM, Oct. 1990.

[41] K. R. M. Leino. Data groups: Specifying the modification of extended state. *ACM SIGPLAN Notices*, 33(10):144–153, Oct. 1998.

[42] K. R. M. Leino and R. Stata. Checking object invariants. Technical Report 007, Digital Systems Research Center, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, January 1997.

[43] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. *ACM SIGPLAN Notices*, 21(11):214–214, Nov. 1986.

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

[45] B. M. Liskov and S. Zilles. Programming with abstract data types. In *Very High Level Languages*, pages 50–59, Apr. 1974.

[46] M. D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Software Engineering*, volume 1, pages 138–150. NATO Science Committee, Jan. 1969.

[47] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, New York, N.Y., second edition, 1997.

[48] L. Mikhajlov and E. Sekerinski. The fragile base class problem and its impact on component systems. In J. Bosch and S. Mitchell, editors, *ECOOP Workshops 1997*, LNCS 1357, pages 353–358. Springer-Verlag, June 1998.

[49] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In E. Jul, editor, *ECOOP'98*, LNCS 1445, pages 355–382. Springer-Verlag, July 1998.

[50] L. Mikhajlov, E. Sekerinski, and L. Laibinis. Developing components in the presence of re-entrance. In J. M. Wing, J. Woodcock, and J. Davis, editors, *FM'99 – World Congress On Formal Methods*, LNCS 1709, pages 1301–1320. Springer-Verlag, Sept. 1999.

[51] A. Mikhajlova. *Ensuring Correctness of Object and Component Systems.* PhD thesis, Åbo Akademi University, October 1999.

[52] A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97*, LNCS 1313, pages 82–101. Springer, 1997.

[53] C. Morgan. Procedures, parameters and abstraction: separate concerns. In C. Morgan and T. Vickers, editors, *On the Refinement Calculus*, Formal approaches of computing and information technology series, pages 47–58. Springer-Verlag, New York, N.Y., 1994.

[54] C. C. Morgan. *Programming from Specifications.* Prentice–Hall, 1990.

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

[56] D. A. Naumann. Predicate transformer semantics of a higher order imperative language with record subtypes. *Science of Computer Programming.* To appear.

[57] D. A. Naumann. Data refinement, call by value, and higher order programs. *Formal Aspects of Computing*, 7:652–662, 1995.

[58] D. A. Naumann. Predicate transformers and higher order programs. *Theoretical Computer Science*, 150:111–159, 1995.

[59] Oberon microsystems, Inc. BlackBox Component Builder, 1997. http://www.oberon.ch/.

[60] P. W. O'Hearn and R. D. Tennent, editors. *Algol-like Languages.* Birkhaüser, Boston, 1997.

[61] OMG. *CORBA 2.2 Specification.* OMG, February 1998.

[62] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[63] L. Paulson. *ML for the Working Programmer.* Cambridge University Press, second edition, 1992.

[64] F. Plášil and M. Stal. An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM. *Software - Concepts and Tools*, 19(1):14–28, 1998.

[65] J. Rees. The T manual. Technical Report CS-94-166, Yale University, Jan. 1994.

[66] J. C. Reynolds. *The Craft of Programming.* Prentice-Hall International, London, 1981.

[67] D. M. Ritchie. The development of the C language. *ACM SIGPLAN Notices*, 28(3):201–208, Mar. 1993.

[68] D. Rogerson. *Inside COM: Microsoft's Component Object Model.* Microsoft Press, 1997.

[69] W. Rubin and M. Brain. *Understanding DCOM.* Prentice-Hall, 1999.

[70] A. Sampaio. *An Algebraic Approach to Compiler Design*, volume 4 of *AMAST Series in Computing.* World Scientific Publishing Company, April 1997.

[71] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the 6th International Conference on Distributed Computer Systems*, pages 198–205, Washington, 1986. IEEE Press.

[72] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. *ACM SIGPLAN Notices*, 21(11):38–45, Nov. 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[73] A. Snyder. Inheritance and the development of encapsulated software components. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 165–188. MIT Press, Cambridge, MA, 1987.

[74] R. Stata. Modularity in the presence of subclassing. Technical Report 145, Digital Equipment Corporation, Systems Research Center, Apr. 1997.

[75] R. Stata and J. V. Guttag. Modular reasoning in the presence of subclassing. In *Proceedings of OOPSLA'95*, pages 200–214. ACM SIGPLAN notices, Oct. 1995.

[76] L. Steels. ORBIT: An applicative view of object oriented programming. In P. Degano and E. Sandwall, editors, *Integrated Interactive Computing Systems - Proceedings of the European ECICS 82 Conference, Stresa*. Elsevier, 1983.

[77] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of OOPSLA '96*, pages 268–285. ACM Press, 1996.

[78] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Mass, 1986.

[79] Sun. Java remote method invocation specification. Technical report, Sun Microsystems, 1997. http://www.javasoft.com/products/jdk/1.1/docs.

[80] Sun Microsystems. *Java Beans(TM)*, July 1997. Graham Hamilton (ed.). Version 1.0.1.

[81] C. Szyperski. *Component Software – Beyond Object-Oriented Software*. Addison-Wesley, 1997.

[82] C. A. Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings of the 19th Australasian Computer Science Conference*, Melbourne, 1996.

[83] D. Taenzer, M. Ganti, and S. Podar. Problems in Object-Oriented Software Reuse. In S. Cook, editor, *Proceedings of the ECOOP '89 European Conference on Object-oriented Programming*, pages 25–38, Nottingham, July 1989. Cambridge University Press.

[84] A. Tarski. A lattice theoretical fixed point theorem and its applications. *Pacific J. Mathematics*, 5:285–309, 1955.

[85] J. von Wright. *A Lattice-theoretic Basis for Program Refinement*. PhD thesis, Åbo Akademi University, 1990.

[86] P. Wegner and S. Zdonik. Inheritance as an incremental modification mechanism. In *European Conference on Object Oriented Programming (ECOOP'88)*, pages 55–77. Springer, 1988. Lecture Notes in Computer Science, Volume 322.

[87] S. Williams and C. Kinde. The component object model: Technical overview. *Dr. Dobbs Journal*, December 1994.

[88] N. Wirth. The programming language Pascal. *Acta Informatica*, 1:35–63, 1971.

[89] N. Wirth. The programming language Oberon. *Software Practice and Experience*, 18(7), July 1988.

[90] S. N. Zilles. Procedural encapsulation: a linguistic protection technique. *ACM SIGPLAN Notices*, 8(9):142–146, Sept. 1973.