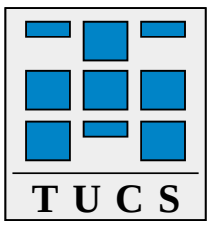
# Ensuring Correctness of Object and Component Systems

by

Anna Mikhajlova




Turku Centre for Computer Science

TUCS Dissertations

No 18, October 1999

# Ensuring Correctness of Object and Component Systems

**Anna Mikhajlova**

Department of Computer Science
Åbo Akademi University

# Ensuring Correctness of Object and Component Systems

**Anna Mikhajlova**

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 October 14th, 1999, at 12 noon.

Department of Computer Science
Åbo Akademi University

ISBN 952-12-0518-0
ISSN 1239-1883

Painosalama Oy

*To my mother,*
*for making this possible*

*Я и садовник, я же и цветок,*
*В темнице мира я не одинок.*
*На стекла вечности уже легло*
*Моё дыхание, моё тепло.*

*Осип Мандельштам*

## Acknowledgements

First, I would like to thank my supervisors, Professor Ralph-Johan Back and Professor Joakim von Wright, for establishing the highest standards of quality and for setting perfect examples of how to do science. It is their light that I reflect. I also want to express my gratitude to them for letting me do what I wanted.

Dr. Pierre America of Philips Research Laboratories, The Netherlands, and Dr. Michael Butler of the University of Southampton, United Kingdom, kindly agreed to review this dissertation. Their comments and suggestions on improving the introductory part and some of the papers are appreciated and gratefully acknowledged.

Dr. Emil Sekerinski, currently at McMaster University, Canada, contributed greatly by teaching me how to do PhD-level research. His friendly guidance and support helped me to take the first step which eventually led to this dissertation.

Dr. Jim Grundy, currently at The Australian National University, and Dr. John Harrison, currently at Intel Corporation, USA, have not only been senior colleagues, answering my numerous questions and carefully explaining difficult issues, but also good friends, providing emotional support and encouragement.

I would like to thank many people who taught me, at different stages of my life, not only the sciences but also important life values. My elementary school teacher A. T. Belokobylskaya, my mathematics teacher V. V. Skripnikova, and my English teacher G. V. Kutuzova have been remarkable teachers and excellent people who have had a profound influence on me. I am also grateful to professors and instructors at my alma mater, Taganrog State University of Radio-Engineering. Most notably, I would like to thank B. I. Orekhov, A. F. Olkhovoy, A. N. Garmash, Professor I. A. Tsaturova, Professor A. N. Melikhov, and Professor L. S. Berstein.

Also, I want to thank many wonderful people in the Department of Computing and Information Sciences at Kansas State University, where I studied in 1992-93 as an exchange student. Professor David Schmidt, Professor Elizabeth Unger, and Dr. Olivier Danvy, to name just a few, made my studies in the department interesting and fruitful, and my time in the United States eventful and full of pleasant memories. Olivier Danvy (currently at the University of Århus, Denmark) deserves special thanks for instilling in me the desire to be a scientist and making me believe that I was capable of becoming a Philosophiae Doctor in Computer Science.

I would like to thank the Department of Computer Science at Åbo

Akademi University and Turku Centre for Computer Science, where I carried out my PhD studies, for providing generous financial support and excellent working conditions. I am also grateful to my colleagues in the department for practical and scientific assistance.

And finally, I would like to thank my friends and family. Especially, Aleksandra Aleksejevna and Leonid Samojlovich for their love and patience. My mother, Tamara Ivanovna, to whom this dissertation is dedicated, for letting me become who I am. Ira, Yura, Sasha, and Andrei for bringing more joy and sunshine to my life. And Leonid for his help, encouragement, enthusiasm, infinite supply of excellent ideas, and the never-ending striving for the very best.

Åbo, August 1999
Anna Mikhajlova

# 1 Introduction

Object-oriented style of programming has become the prime technology of software development in the past few decades. The main reasons for the overwhelming popularity of this programming paradigm are better structuring and modularization as well as a high degree of polymorphic reuse that it supports. Component-based programming takes the benefits of object-orientation to an even higher level by allowing development of light-weight, highly-customized, independently extensible applications. Component-based systems consisting of loosely-bound highly configurable components and capable of accommodating continuous evolution can provide software developers and users with the same levels of plug-and-play interoperability that are available to manufacturers and consumers of electronic parts or custom integrated circuits. Establishing a software component market similar to the market of hardware components is the ultimate goal of the software industry.

The issue of correctness of object-oriented and component-based systems deserves close consideration in view of the present and ever-growing popularity of the corresponding programming styles and the necessity to enhance reliability of programs. Traditionally, correctness has been considered as a crucial requirement mainly for safety-critical systems, but nowadays, the need for ensuring correctness of object-oriented and component-based systems is becoming more widely recognized. In open systems, which are composed and extended by end users and characterized by a late integration phase, it is impossible to conduct a global integrity check. Therefore, it should be possible to guarantee error-free operation of resulting applications by establishing correctness of constituting components.

In general, correctness of a system $T$ is always determined with respect to another system $S$, which can be an original system whose functionality we want to improve in $T$, or a specification of the behavior we want to achieve in $T$. The ability to substitute more concrete, more specialized, and efficient entities for more abstract and general ones is the essence of polymorphic reuse, which is the defining feature of both the object-oriented and the component-based approaches. Correctness of such substitution determines correctness of the system using new entities revising and extending the functionality of the original entities. If an object or a component $c'$ preserves and improves the behavior of an object or a component $c$ then substituting $c'$ for $c$ in a system is guaranteed to be correct. Imposing semantics constraints on objects and components, requiring that they preserve the behavior of the entities they can be substituted for, allows us to build

systems that are correct by construction.

In this dissertation we develop a mathematical foundation for reasoning about correctness of object-oriented and component-based systems. The logical framework that we build is based on the refinement calculus [4, 17, 6], which is used for reasoning about correctness and refinement of imperative programs. The contribution of this dissertation is supported by the following publications.

## List of Publications

1. A. Mikhajlova and E. Sekerinski. Class Refinement and Interface Refinement in Object-Oriented Programs. In *Proceedings of the 4th International Formal Methods Europe Symposium (FME'97)*, edited by J. Fitzgerald, C. B. Jones, and P. Lucas, Lecture Notes in Computer Science 1313, Springer-Verlag, pp. 82–101, September 1997.

2. A. Mikhajlova. Refinement of Generic Classes as Semantics of Correct Polymorphic Reuse. In *Proceedings of the International Refinement Workshop and Formal Methods Pacific (IRW/FMP'98)*, edited by J. Grundy, M. Schwenke, and T. Vickers, Springer Series in Discrete Mathematics and Theoretical Computer Science, pp. 266–285, July 1998, Springer-Verlag.

3. R. J. R. Back, A. Mikhajlova, and J. von Wright. Reasoning About Interactive Systems. To appear in *Proceedings of the World Congress on Formal Methods (FM'99)*, Lecture Notes in Computer Science, September 1999, Springer-Verlag.

4. R. J. R. Back, A. Mikhajlova, and J. von Wright. Class Refinement as Semantics of Correct Object Substitutability. Extends Paper 1, previous version published as a technical report of Turku Centre for Computer Science, TUCS-TR-147, December 1997.
   Submitted to *Formal Aspects of Computing*.

5. A. Mikhajlova. Consistent Extension of Components in the Presence of Explicit Invariants. In *Proceedings of the 29th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 29)*, IEEE Computer Society Press, pp. 76–85, Nancy, France, June 1999.
   Previous version appeared in *Proceedings of the Third International Workshop on Component-Oriented Programming (WCOP'98) held in conjunction with ECOOP'98*, October 1998.

6. A. Mikhajlova and E. Sekerinski. Ensuring Correctness of Java Frameworks: A Formal Look at JCF. Technical Report of Turku Centre for Computer Science, TUCS-TR-250, March 1999.
A shorter version of this paper by Anna Mikhajlova appeared in *Proceedings of the 30th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 30)*, IEEE Computer Society Press, pp. 136–145, Santa Barbara, USA, August 1999.

## 2 Objects, Components, and Subtyping

In this section we present the basic concepts of the object-oriented and component-based paradigms, and explain which of the papers 1–6 present models of the corresponding constructs and mechanisms.

### 2.1 Objects, Components, and Interaction

*Objects* are entities that live in a program and have an identity, a local state, and methods. The methods can reveal the object state to the outside world and modify it. As objects do not exist in isolation, collaboration or interaction with other objects can also be encoded in the methods: objects interact with each other by calling each other's methods. The backbone of interaction between objects is their *interfaces*, i.e. the names of methods and the types of their parameters.[1]

The notion of a *component* does not have a standardized meaning and various researchers and practitioners understand fairly different things with components. A component can be a distributed object whose methods are subject to remote procedure calls, or an object having a graphical user interface that can be used to graphically compose it with other components in an application and graphically manipulate it.

On the conceptual level objects and components are rather similar: both encapsulate their local state and have methods visible to the environment through a public interface. It is the way of composing and using objects and components that distinguishes between them. An object is a unit of implementation which may not be a semantic entity on its own and may deliver certain functionality only in collaboration with other objects. A component, on the other hand, is an architectural building block, a unit of independent deployment subject to composition by users rather than developers. Many practitioners and researchers consider the heterogeneous nature of components to be their defining feature. Namely, components are often defined as entities that are not bound to a particular program, language, or implementation [19]. As such, components can be implemented as objects or compositions of collaborating objects, and packaged as independent pieces of code.

Both objects and components encapsulate their local state and behavior, exposing only interfaces to the outside world. For reasoning about the behavior of both objects and components, it is therefore sufficient to model

[1] Interfaces are often identified with object types, and we will follow this convention in this introduction, unless specifically stating the opposite.

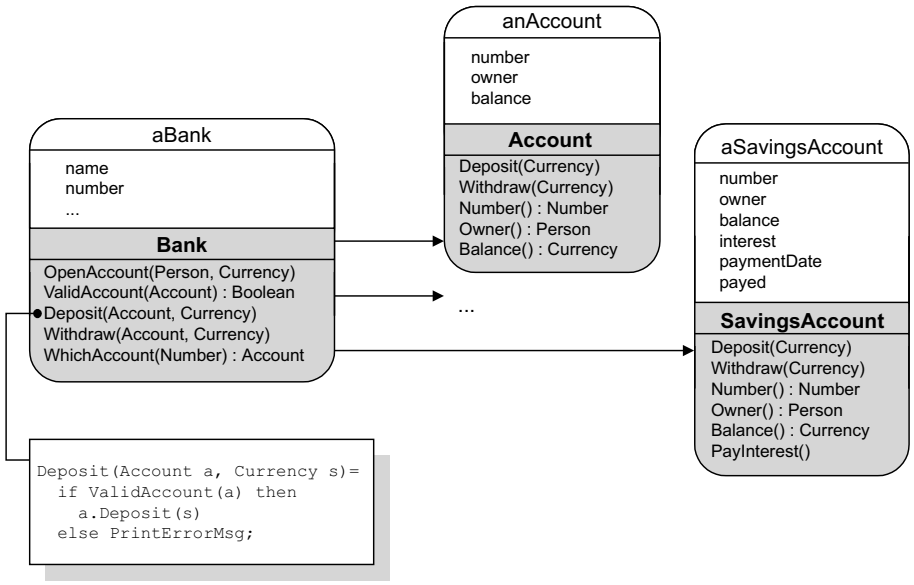



Figure 1: Objects, interfaces, and collaboration among objects

both as entities with local state represented by attributes, and methods accessible to the environment through interfaces.

Let us illustrate all the concepts that we present and discuss here with an example. Figure 1 illustrates a group of collaborating objects; in this and other diagrams objects are depicted as rounded boxes and method definitions are depicted as sharp-edged boxes with shadow. The object anAccount has a local state, represented by the attributes number, owner, and balance, and the methods Deposit, Withdraw, Number, Owner, and Balance. The first two methods allow clients of anAccount to modify its state, whereas the remaining methods let the clients find out the state of the respective attributes at a given point in time.[2] Another object in this figure, aBank, is a client of anAccount using it through invoking its methods. In the diagram, the interfaces of the objects are shown in grey, e.g., the object anAccount has the interface Account.

Note that anAccount and aBank can be implemented as distributed objects, with anAccount residing on the computer of its owner rather than in the same address space as aBank. Alternatively, these objects can be implemented as widgets in a financial organizer desktop application that

[2]The functionality of this bank account is somewhat simplistic, but is sufficient for illustrating required concepts.

can be graphically composed and manipulated by end users. Conceptually, anAccount and aBank can therefore be thought of as components that are developed and composed by independent parties. What is important for our purposes is that we can reason about their behavior the same way we reason about the behavior of objects.

We present a mathematical model of objects, their attributes and methods in Paper 1. We model these object-oriented constructs in the refinement calculus [4, 17, 6], which is a logical framework for reasoning about correctness and refinement of imperative programs in a rigorous, mathematically precise manner. In papers 3 and 5, we also present a mathematical model of components.

Just like objects, components do not operate in isolation but rather cooperate with other components by interacting with them. For each component all the other components it interacts with can be collectively considered as its *environment*. Paper 3 focuses on modelling component environments, and studies various embodiments of interaction and methods for reasoning about it in a formal framework. In particular, it is observed that on the conceptual level interaction between a component and a human user and interaction between a component and its environment work according to the same principles. Moreover, for components in a component-based system their environment can be transparent: they interact with this environment in the same way, regardless of whether it is another component or a human user. For example, the user can deposit and withdraw money from anAccount without going to aBank but directly, e.g., through the Internet. For anAccount it will then be transparent which environment deposits or withdraws the money from it, aBank or the user. In Paper 3, we describe a uniform logical framework in which one can reason about interactions of various kinds.

*Invariants* binding values of component attributes play an important rôle in maintaining correctness of component-based systems. Invariants can be *implicit* and *explicit*. The implicit, or the strongest, invariant characterizes exactly all reachable states of the component, whereas the explicit invariant restricts the values the component might have. The implicit invariant is established by component initialization, preserved by all its methods, and can be calculated from the component specification. As suggested by its name, the explicit invariant, on the other hand, is stated explicitly in the component specification, being part of the general description of the behavior the component promises to deliver to its clients. The model of components presented in Paper 5 includes explicit invariants, stating the properties that can be safely assumed about components' behavior during their entire lifetime.

## 2.2 Subtyping and Polymorphic Substitutability

As was already mentioned above, interaction between objects is carried through interfaces of these objects. Many account objects can have the same interface and clients can use all of them uniformly when trying to achieve the same functionality. For example, the method Deposit of aBank gets the account a to which the money is to be deposited as a parameter, and, regardless of which particular account it is, checks that it is a valid one, i.e. registered with this bank, and calls the method Deposit on this account, passing it the amount of money s to be deposited as an input argument.

Moreover, aBank can work uniformly not only with different accounts of one kind, it can work the same way with aSavingsAccount as well. As suggested by its name, aSavingsAccount is a special kind of account with interest on a deposited sum accumulating with time. As such, it has the interface SavingsAccount which is the same as Account extended with the new method signature specific to savings accounts, PayInterest(). In general, this kind of *interface extension*, which is also known as *subtyping* and *interface inheritance* and usually written as SavingsAccount <: Account, is the key mechanism supporting polymorphic substitutability of objects in clients. The essence of polymorphic substitutability is that objects with the extended interface (subtype) can be used by clients in the same way as objects with an original interface (supertype). Clients can work with objects with the extended interface as if they were objects with the original interface, because the objects "understand" all client requests for method invocation, having all the methods in the original interface which is known to the client.

In our example, aSavingsAccount can be passed as an argument to the method Deposit of aBank, and, since aSavingsAccount is a valid account, this will result in invoking the method Deposit defined in aSavingsAccount. For the client aBank it is only important that the object passed as an input argument has all the methods in the interface Account, including Deposit, which aSavingsAccount certainly does, having an interface that extends Account. In general, selecting a method for invocation based on the actual type of object rather than the declared type of the variable carrying this object is known as *dynamic binding*. The ability to substitute supertype objects with subtype objects in clients is referred to as *subtype polymorphism*.

The subtyping relation, in fact, does not have to be a simple extension, but can be more permissive in the sense that inherited method signatures can be modified in a subtype so that the types of method input parameters become *contravariant* and the types of method output parameters become *covariant*. Contravariance means that subtyping on the types of method

parameters is in the opposite direction from subtyping on the interfaces having these methods. Respectively, covariance means that subtyping on the types of method parameters is in the same direction as subtyping on the interfaces having these methods. Contravariance in input parameter types and covariance in output parameter types are the basic subtyping properties of function types [1]. As methods are essentially (object state modifying) functions of input parameters returning output parameters, they naturally have these properties as well.

Suppose that we have an interface Bank′ which has the same method signatures as Bank, but with the difference that the methods OpenAccount, ValidAccount, Deposit, and Withdraw have the input parameter of type SavingsAccount rather than Account, as shown in Fig. 2(a). As we know, SavingsAccount is a subtype of Account, written SavingsAccount <: Account, and hence a SavingsAccount object can be passed where an object of type Account is expected. In particular, passing it to the methods of Bank expecting an input parameter of type Account will be type-correct. According to the typing rules, Bank is therefore a subtype of Bank′, i.e. Bank <: Bank′. Covariance in output parameter types is illustrated in Fig. 2(b). Of the two

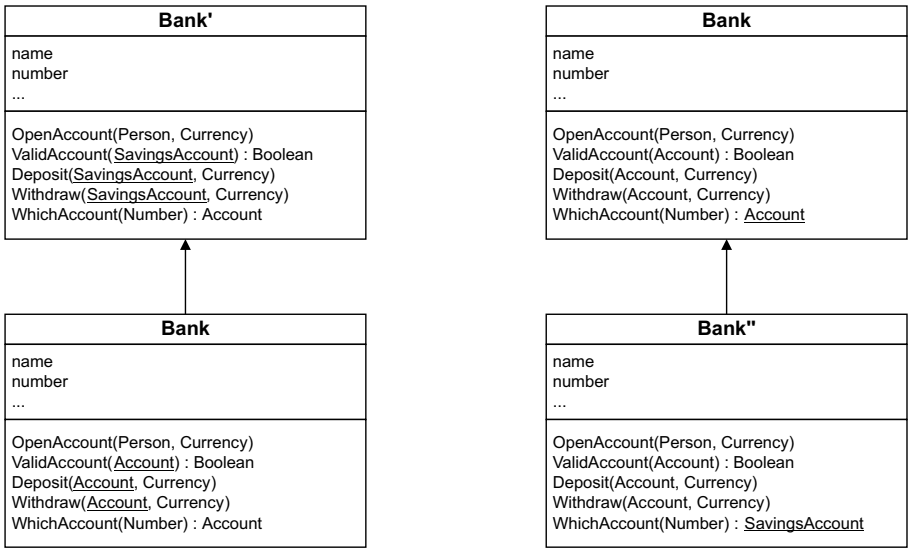



Figure 2: Contravariance in input type parameters a) and covariance in output type parameters b)

methods having output parameters, ValidAccount has the same output parameter type in both Bank and Bank″, whereas WhichAccount in Bank″ has the output parameter type which is a subtype of the corresponding output parameter type in Bank. Any output produced by Bank″ is a valid output of Bank, because SavingsAccount objects are Account objects as well. Therefore, Bank″ is a subtype of Bank, i.e. Bank″ <: Bank.

When one interface is the same as the other, except that it can specify new method signatures, redefine contravariantly value parameter types and covariantly result parameter types, this interface is said to *conform* to the original one. For example, Bank″ conforms to Bank which, in turn, conforms to Bank′.

In Paper 1, we show how to model within the refinement calculus subtyping and dynamic binding of method calls with coercions of contravariant and covariant arguments. A slightly updated version permitting, for example, subtype objects to have new methods appears in Paper 4.

# 3 Classes and Code Reuse

Class-based object-oriented languages constitute the mainstream in object-oriented programming. In these languages objects are instantiated by classes, as illustrated in Fig. 3; classes are depicted by sharp-edged boxes and a dashed arrow means "instantiates". Classes define the behavior of objects they instantiate by specifying their interface. As classes are templates for creating objects, apart from defining object attributes and methods, they provide *class constructors* specifying the way the objects are created and initializing their attributes. One class can have more than one constructor with the same name, which is usually the same as the name of the class, but different input parameters. Typically, class constructors are not part of the interface specified by their class, and hence different classes with different class constructors can still have conforming interfaces. For example, the interface SavingsAccount specified by SavingsAccountClass conforms to the interface Account specified by AccountClass, i.e. SavingsAccount <: Account, despite the fact that class constructors of AccountClass and SavingsAccountClass have different signatures. Finally, similar to components, classes can have explicit invariants stating the properties that can be safely assumed about the behavior of objects that these classes instantiate.

The strength of object technology comes not only from the better structuring and modularization that it provides, but also from the multitude of possibilities for code reuse that it offers. As we already mentioned, client

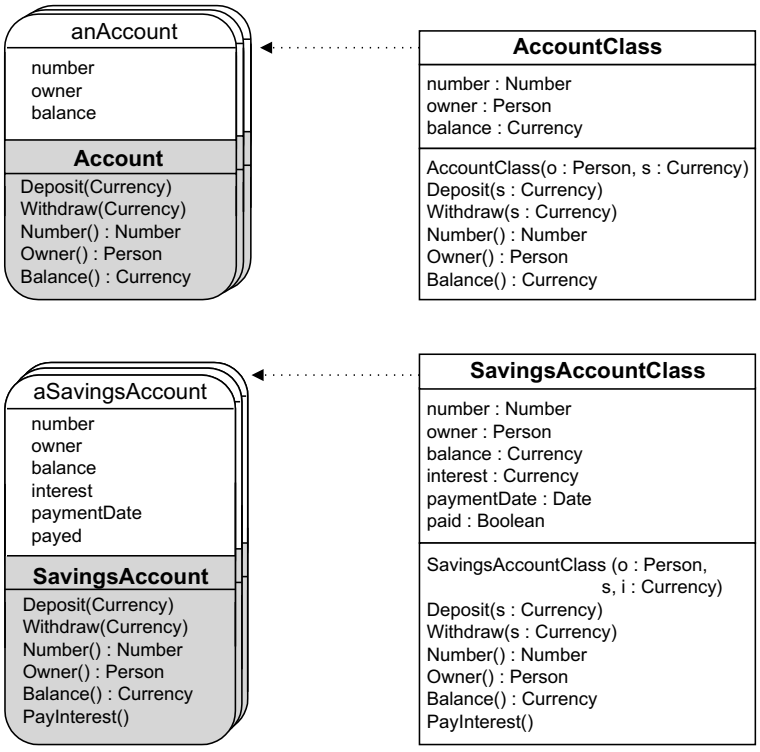



Figure 3: Classes and their instances

code reuse can be achieved through polymorphic substitution of subtype objects for supertype objects in clients. In addition, there are various techniques and mechanisms for reusing code in objects and classes, among them implementation inheritance (or subclassing) is the most well-known and widely used mechanism. We illustrate inheritance in Fig. 4 using the OMT notation. The classes AccountClass and SavingsAccountClass instantiate the corresponding account and savings account objects. Since most of the functionality specified in AccountClass remains the same in SavingsAccountClass, the latter inherits from the former, adding new parts, and overriding certain inherited parts customizing them to express the specifics of the savings account behavior. For example, SavingsAccountClass inherits all attributes of AccountClass while adding the attributes interest, paymentDate, and paid to hold the value of interest specific to this kind of a savings account, the date on which the interest is to be paid, and the interest payment status. The methods Number and Owner are inherited as a whole, the methods Deposit, Withdraw, and Balance are redefined to account for interest payment

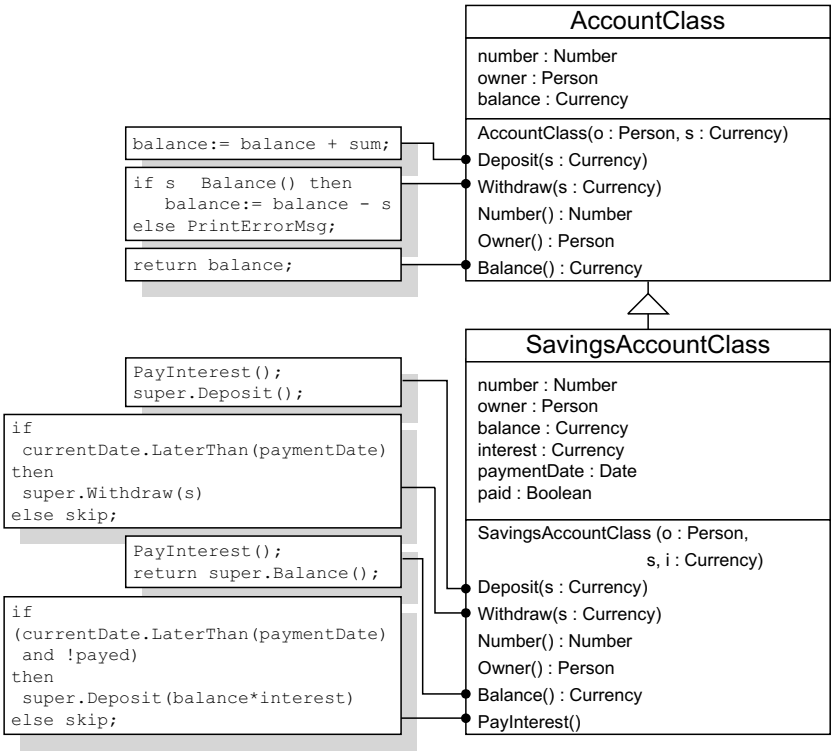



Figure 4: Inheritance

at the due time, and the method PayInterest defines new functionality.

Apart from inheriting methods as a whole, the inheriting class can reuse methods of its superclass through super-calling them from the overriding methods. For example, Deposit of SavingsAccountClass is specified to first pay the interest on the original balance if the contract requirements are satisfied and then super-call the method Deposit of AccountClass, delegating it the task of actual money depositing.

Inheritance is known to be a flexible reuse mechanism, because in combination with dynamic binding of self-referential methods it allows fine-tuning reused code for specific needs. Consider, for example, the definition of method Withdraw in SavingsAccountClass. Provided that the terms of the contract are met (the current date is past the interest payment date), it is possible to withdraw the money through super-calling the method Withdraw defined in AccountClass. The latter calls the method Balance to check whether the current balance is greater than the sum requested to be withdrawn. The effect of dynamic binding is that the method call to

Balance is redirected back to SavingsAccountClass rather than remaining in AccountClass. In other words, the body of method Balance as defined in SavingsAccountClass is executed when this method is called in Withdraw of AccountClass, which is in turn super-called from SavingsAccountClass. This kind of call-backs is instrumental in making fine dynamic adjustments in the code of methods defined in a superclass when super-calling them from a subclass. In the case of our example this is important, because calling Balance of SavingsAccountClass results in paying the interest, if the contract conditions are met, before returning the current balance value. If the call to Balance remained in AccountClass, only the sum not exceeding the balance could be withdrawn even after the interest payment date.

With the reuse mechanism known as *multiple inheritance* a class can inherit attributes and methods from more than one superclass. Multiple inheritance is a controversial reuse mechanism because on the one hand it permits more flexibility in code reuse and on the other hand suffers from various problems like potential name clashes, diamond structure, complex class libraries, and additional run-time costs [18]. This controversy is reflected in the fact that some programming languages, such as C++ and Eiffel, support multiple inheritance while others, e.g., Java and Oberon-2, choose to abandon it.

In Paper 1, we present a mathematical model of classes and inheritance. For simplicity, we do not consider multiple inheritance; yet we outline how it can be handled in the framework we present. An extended model of classes and (single) inheritance permitting, for example, subtype objects to have new methods appears in Paper 4. Also, in Paper 1 we model dynamic binding of only external method calls, while self-referential method calls are modeled as being *statically resolved*. In Paper 4, we present an improved model, where self-referential method invocations as well as external method invocations are dynamically-bound.

# 4 Ensuring Correctness of Object-Oriented and Component-Based Systems

## 4.1 Class Refinement

In most object-oriented languages, such as Simula, Eiffel, and C++, subclassing forms a basis for subtype polymorphism, i.e. signatures of subclass methods automatically conform to those of superclass methods, and syntactically subclass instances can be substituted for superclass instances. As the

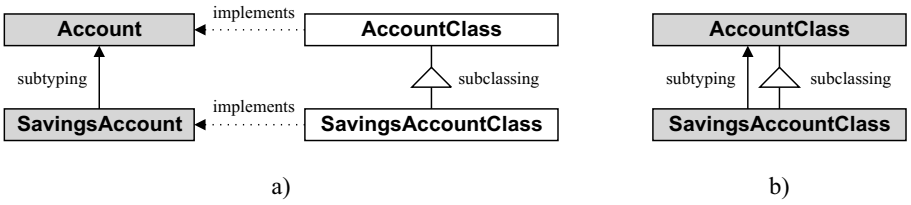



Figure 5: Subtyping and subclassing hierarchies: separate a) and unified b)

mechanism of polymorphic substitutability is, to a great extent, independent of the mechanism of implementation reuse, languages like Java and Sather separate the subtyping and subclassing hierarchies. These two approaches are illustrated in Fig. 5.

With both approaches, typechecking can be used to verify syntactic conformance of subtype objects to their supertype objects. Verifying syntactic conformance is necessary to ensure that, when substituting subtype objects for supertype objects in clients, all client requests for method invocations earlier directed to supertype objects "are understood" by subtype objects as well. However, it has been recognized that, while certainly very useful, typechecking alone is insufficient to guarantee correctness of object substitutability. Consider the example in Fig. 6. Here subclassing forms a basis for subtype polymorphism and instances of SavingsAccountClass′ can be substituted for instances of AccountClass in all clients. A customer using an ordinary account and having his balance increased whenever depositing money, can decide to open a savings account, expecting to get some interest on the savings. After depositing money on his savings account, the customer can become very disappointed when finding out that not only wouldn't he get any interest on the deposited sum, but also that the deposited money are missing on his savings account: they were transferred to a certain specialAccount instead.

To prevent this and similar kinds of errors, it is necessary to verify behavioral conformance between classes whose instances are intended for polymorphic substitution. Instances of one class are guaranteed to behave as expected from instances of another (usually more abstract) class if the more concrete class is a *refinement* of the more abstract. Refinement means preservation of observable behavior, while decreasing nondeterminism. In Paper 1, we give a definition of *class refinement* stating that a class $C$ is refined by a class $C'$, written $C \sqsubseteq C'$, if a constructor of $C$ is refined by a constructor of $C'$ and methods of $C$ are refined by the corresponding methods of $C'$. We prove that class refinement is a reflexive and transitive relation. In

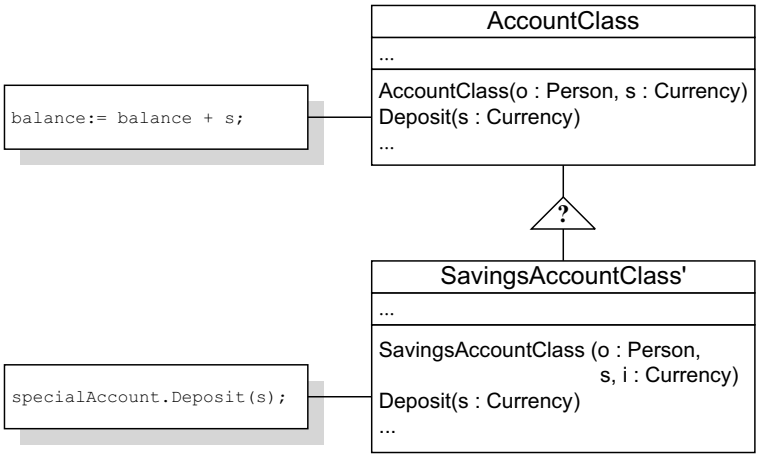



Figure 6: Illustration of the necessity to establish behavioral conformance between AccountClass and SavingsAccountClass′

Paper 4, we present the definition of class refinement in a slightly improved formulation and relate it to correctness of polymorphic object substitutability in clients. Modelling object clients as described in Paper 3 and using theoretical results developed in the same paper, allow us to reason about the effect that substituting instances of one class for instances of another class has on clients. In particular, we prove that when class $C'$ refines class $C$, substituting instances of $C'$ for instances of $C$ is refinement for the clients. This property allows us to regard class refinement as semantics of correct object substitutability.

Furthermore, in Paper 4 we extend the definition of class refinement to account for new methods added in a refined class. Formal treatment of behavioral conformance in the presence of new methods is a non-trivial issue because of inconsistencies possibly introduced by new methods and becoming apparent in the presence of subtype aliasing as well as in the general computational environment that allows sharing of objects by multiple users. Our treatment of new methods follows that of Barbara Liskov and Jeannette Wing, who originally identified this problem in [16], but is more formal and more permissive.

As we developed the theory of class refinement in the period between writing Paper 1 and Paper 4, our perception of this fundamental concept has evolved. If in Paper 1 we considered it to be the semantics of correct subclassing only, by the time Paper 4 has been written we realized that it is orthogonal to subclassing and is much more general. In particular, a class

and its subclass may not be in refinement, and two classes can be in refinement even if one of them is not declared to be a subclass of the other. With separate interface inheritance and implementation inheritance hierarchies a subclass may not even be intended for behavioral conformance with its superclass, as the substitution mechanism is completely independent of the reuse mechanism. Syntactic conformance of method signatures, however, is a prerequisite for class refinement, as it is meaningless to compare behavior of classes whose instances are not intended for substitution.

To illustrate application of our theory, we specify in Paper 6 the Java Collections Framework [7], which is a part of the standard distribution package of Java Development Kit 2.0. As interface inheritance is separated from implementation inheritance in Java, behavior of classes implementing a certain extended interface should conform to the behavior of classes implementing the original interface. Moreover, behavior of classes implementing some interface should conform to the "intended" behavior of this interface, as stated in its specification. In the Java Collections Framework, specifications of interface behavior are given informally. We provide formal specifications and highlight their advantages by comparing them to the informal descriptions, which are at times imprecise and ambiguous. We also demonstrate how one can prove that specifications of extended interfaces refine specifications of original interfaces.

### 4.2 Interface Refinement

In the process of system development or sometimes also maintenance, when feedback from the actual use of the system is taken into account, a change in user requirements or better understanding of these requirements by a developer may inflict an interface change in classes designed as refinements of other classes in the system. For example, suppose that the financial management application, whose part we described earlier, has been released on the market and gained substantial success. The owners of the software development company realized that they can expand the market, if they adjust the system to provide banking services to corporate customers as well as private ones. In response to these new requirements, developers can construct a class CorporateAccountClass with owner of type Company rather than Person and with the same (or improved) functionality as AccountClass, (see Fig. 7). However, aBank cannot handle both kinds of accounts in a uniform manner, because the interface of CorporateAccountClass does not conform to that of AccountClass: the result type returned by the method Owner is no longer Person but Company. If Company were a subtype of

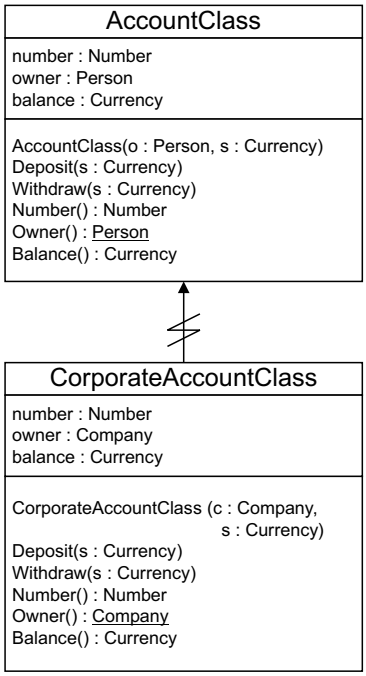



Figure 7: Illustration of interface refinement

Person, the problem would be avoided, as the type of method return parameters can be covariant in subtypes. But people are not just generalizations of companies, and the developers must find a way for aBank to use instances of CorporateAccountClass along with instances of AccountClass and SavingsAccountClass.

In Paper 1, we introduce a notion of *interface refinement*, capturing behavioral conformance between classes with similar yet slightly different interfaces, and provide a solution to the problem of mismatching interfaces based on a technique known as *forwarding*. Suppose we have two classes, OldClass and NewClass, such that NewClass refines the behavior of OldClass but has a slightly different interface. If we want to use instances of NewClass instead of instances of OldClass, we can introduce a class Wrapper aggregating an instance of NewClass and forwarding OldClass method calls to NewClass through this instance, as illustrated in Fig. 8. As Wrapper has an interface conforming to that of OldClass, clients of OldClass can work via Wrapper with NewClass the way they would work with OldClass. Apart from describing how clients such as aBank can benefit from the interface refinement of their server classes without having to be changed in any way, we

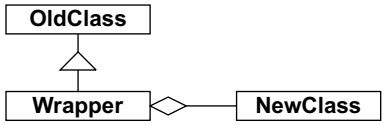


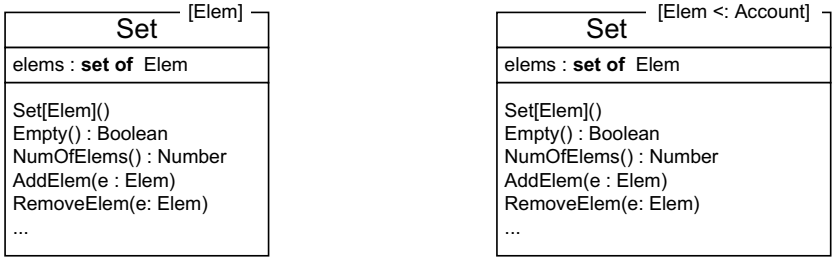

Figure 8: Illustration of forwarding

also establish conditions under which clients can be systematically changed to use the refined server classes.

In fact, forwarding, which was used in Paper 1 to resolve the problem of mismatching interfaces, is a well-known code reuse technique employed along with inheritance to create new classes from existing ones by reusing their functionality. As compared to inheritance, forwarding is less flexible, as it does not allow making fine adjustments to the inherited code but rather requires reusing code in its entirety. On the other hand, the flexibility of inheritance, despite the numerous benefits that it offers, can be rather problematic, especially in the presence of explicit invariants. In Paper 5, we study ways of ensuring behavioral consistency of extensions in component-based systems in the presence of explicit invariants. Our analysis indicates that ensuring correctness in this setting is easier with forwarding than with inheritance. We describe consistency problems that can occur when constructing an extension of a component having an explicit invariant, and formulate requirements that must be imposed on components and their extensions to avoid these problems.

### 4.3 Generic Class Refinement

To enhance reuse for different types and increase flexibility of class construction, *parametric polymorphism* is usually combined with object-oriented reuse mechanisms and techniques. Consider again our example of financial management application. To manage all accounts held by its customers, aBank should aggregate a collection of these accounts. Such a collection should be a set, as accounts cannot be duplicated. Moreover, aBank should also maintain a set of its customers. We can define a data structure general enough to represent both kinds of sets as shown in Fig. 9(a). The parameterized class Set [Elem] is a construct representing the collection of classes that can be constructed by substituting concrete types for the type parameter Elem. In various programming languages parameterized classes are referred to by different names, e.g., C++ calls it a template, while Eiffel – a generic class. We follow the terminology of Eiffel, referring to parameterized classes




a) b)

Figure 9: Generic classes: unbounded generic class a) and bounded generic class b)

as *generic classes*, and to classes produced from generic by substituting concrete types for formal parameters as *generically derived classes*.

By substituting concrete object types, such as Account, Person, and even Bank for Elem, we get the corresponding classes Set [Account], Set [Person], and Set [Bank] instantiating sets of accounts, sets of persons, and sets of banks respectively. In defining the behavior of methods in a generic class, it is often necessary to make sure that all concrete object types that can be substituted for the type parameter have certain methods, because for defining the functionality of the generic class it can be necessary to call these methods. For example, in the definition of AddElem we might want to invoke the method Equal on e, checking that e is not equal to one of the elements in elems. For this to work, we would like to ensure that any object type that can be substituted for Elem has at least the method Equal. To impose this kind of constraints on generic type parameters, one can employ a mechanism known as *bounded parameterization*. Essentially, a generic type parameter is restricted to be a subtype of some object type. For example, a generic class of account sets can be declared as shown in Fig. 9(b). In can be reasonable to construct this kind of a generic class if we want sets of accounts to have functionality different from that of sets of ordinary elements. For instance, before adding a new account e we may want to check that its number, returned by the method Number, is not the same as the number of some other account in the set of accounts elems.

In addition to generic derivation, type parameterization can be used to maximize code reuse through combining it with implementation inheritance. Consider an illustration of this combination in Fig. 10. In this diagram, the class SavingsAccountClass is constructed from the class AccountClass by sub-

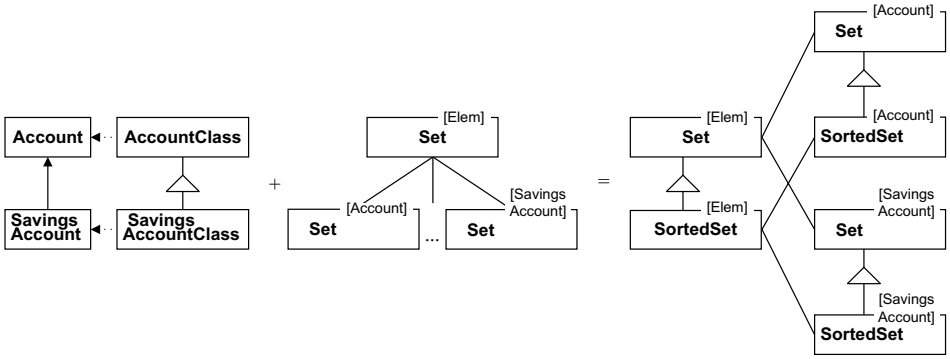



Figure 10: Combining subclassing with type parameterization

classing, whereas the classes Set [Account] and Set [SavingsAccount] are constructed from the generic class Set [Elem] by substituting the concrete types Account and SavingsAccount for the type parameter Elem. Combining the two mechanisms allows us to increase flexibility of class construction, producing the new generic class SortedSet [Elem] by (generic) subclassing from Set [Elem] and deriving concrete classes by instantiating the object types Account and SavingsAccount for the type parameter Elem in both Set [Elem] and SortedSet [Elem].

Paper 2 focuses on the correctness of a reuse mechanism based on the combination of subclassing and type parameterization. We give semantics of subclassing for generic classes, and study the conditions for correct generic subclassing based on the notion of class refinement. The results of this study reveal that relying on certain properties of generic classes, which may seem plausible, can lead to problems. Accordingly, we give a definition of *generic class refinement*, extending the definition of (ordinary) class refinement, to circumvent these problems and ensure that instances of a generically derived subclass will behave as expected from instances of the corresponding generically derived superclass. We also consider special cases in which class refinement is guaranteed to hold between generically derived classes.

# 5 Related Work

As practically every paper included in this dissertation contains comparison with related work, in this introduction we only focus on the specific features of our formalism and compare our approach to that of *behavioral subtyping*,

which is most closely related. The discussion presented here is a combination of ideas that were omitted from the papers 1–6 (mostly because of limited space considerations) but are nevertheless important, and of parts already presented in the papers but deserving to be included here for completeness of exposition.

## 5.1 Refinement Calculus and Higher-Order Logic Foundation

The logical framework for reasoning about object-oriented and component-based systems which is described in the papers constituting this dissertation is built as an extension of the *refinement calculus* of Ralph Back and Joakim von Wright [6]. We chose to extend an existing and profoundly developed logical framework used for reasoning about correctness and refinement of imperative programs, rather than building a new logic from scratch. This decision has many advantages and echoes that made by the programming language community in constructing object-oriented languages as extensions of imperative languages. The advantages in our case include inheriting all meta-logical properties of the underlying formalism such as soundness.

The choice of the refinement calculus as the underlying logical framework is justified by many factors. Primarily, it is particularly suited for describing object-oriented and component-based systems because it allows us to reason about the behavior of objects and components at various abstraction levels. Versatility of the specification language used in the refinement calculus permits treating specifications and implementations in a uniform manner considering implementations to be just deterministic specifications. The notion of an *abstract class*, specifying behavior common to its subclasses, can be fully elaborated in this formalization, since the state of class instances can be given using abstract mathematical structures, like sets and sequences, and class methods can be described in terms of nondeterministic statements, abstractly but precisely specifying the intended behavior.

The refinement calculus itself is formalized in higher-order logic. The expressiveness of the latter allows us to model complex method invocation mechanisms, such as dynamic binding, and define relations between classes, such as class refinement, entirely within the logic. Reasoning about such higher-order abstractions and relations can therefore be carried out completely formally, whereas formalizations based on first-order logic can only allow informal reasoning. The benefits of formalizing imperative programming languages in frameworks based on higher-order logic have been pointed out by many researchers, both in the areas of programming methodology and

logics. For example, Daniel Leivant in [15] notes: "Higher order functions, in the form of procedures, are at the core of higher level programming languages, whether imperative or applicative." Reasoning about data types and their elements is central to reasoning about imperative programs operating on these elements, and in higher-order logic one can define data types, including natural numbers, in a straightforward manner. Object-oriented languages emphasize data abstraction and procedural abstraction even more than ordinary imperative languages, and therefore using higher-order logic as the basis for formalizing object-oriented programs appears to be only natural.

Formalization of object-oriented constructs within a formal logic not only associates with them precise mathematical meanings but also opens a possibility for mechanized reasoning. A formal theory of correctness and refinement can also, in this case, be formalized within a theorem proving environment such as HOL [10] or PVS [20], permitting mechanized if not mechanical verification.

One of the important features of the refinement calculus is that it uses the same constructs for programming statements and mathematical entities (monotonic predicate transformers) that they determine. Identifying statements with their mathematical counterparts allows a purely algebraic treatment of statements, where the specific syntax chosen for expressing statements is irrelevant. No distinction needs to be made between syntax, semantics, and proof theory, as is traditional in programming logics. Being a conservative extension of higher-order logic, the refinement calculus is consistent and its proof theory is sound, because higher-order logic is consistent and sound with respect to a standard set-theoretic semantics. Our logical framework of reasoning about object-oriented and component-based systems inherits all these fundamental metalogical properties from the refinement calculus.

### 5.2 Relation to Behavioral Subtyping

The general idea behind our approach and the research direction known as *behavioral subtyping* is essentially the same – to develop a specification and verification methodology for reasoning about correctness of object-oriented programs. Our work has been to a great extent inspired by works of Pierre America, Barbara Liskov, Jeannette Wing, Gary Leavens, and others [2, 16, 14, 9]. However, our approach differs in a number of ways both on the conceptual and the technical level.

The essence of behavioral subtyping is to associate behavior with type

signatures and to identify subtypes that conform to their supertypes not only syntactically but also semantically. In our view subtyping is a mechanism for substituting objects with certain method signatures for other objects with conforming method signatures and, as such, is a purely syntactic concept. Behavior of objects, on the other hand, has little to do with their syntactic interfaces and is expressed in the specification of the objects' methods manipulating the objects' attributes. Most importantly, syntactic subtyping is decidable and can be checked by a computer, while behavior-preserving subtyping is undecidable. Hence, in our approach we separate syntactic subtyping from behavioral conformance of subtype objects to supertype objects. We use classes to express (at different abstraction levels) the behavior of objects and class refinement to express behavioral conformance. Our model of classes, subclassing, and subtyping, as well as the definition of class refinement, can be used to reason about the correctness of programs using both unified and separate subclassing and subtyping hierarchies.

When used in the context of separate subclassing and subtyping hierarchies, class refinement is very similar to behavioral subtyping. Consider a graphical representation of the corresponding settings in Fig. 11. In both cases $I$ and $I'$ are certain interfaces (types) such that $I'$ is a syntactic subtype of $I$. In the case of behavioral subtyping in Fig. 11 (a), the behavior of methods is specified in terms of pre- and post-conditions. To verify that $I'$ is a behavioral subtype of $I$, written $I' < I$, America, Liskov, and Wing require proving that every precondition $pre_i$ is stronger than the corresponding $pre'_i$ and every postcondition $post_i$ is weaker than the corresponding $post'_i$, while Dhara and Leavens in [9] weaken the requirement for postconditions. In addition to proving behavioral subtyping, one must also verify that the classes $C$ and $D$ claiming to implement the types $I$ and $I'$ respectively really

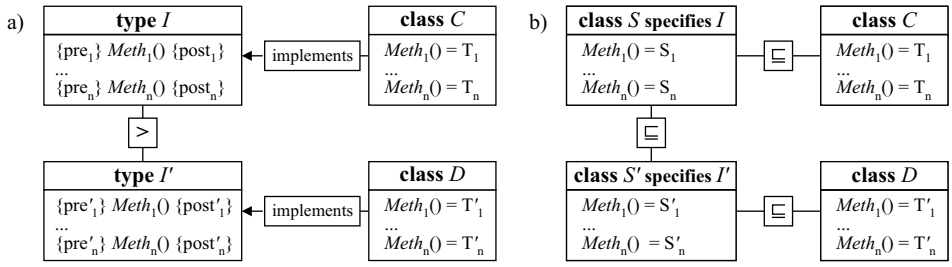



Figure 11: Behavioral subtyping a) and class refinement b) in the case of separate interface and implementation inheritance hierarchies

do so. America in [3] proposes a rigorous verification method that can be used for this purpose. For verifying, e.g., that $C$ implements $I$, he uses a representation function mapping concrete states of $C$ to the set of abstract states associated with $I$ as well as a representation invariant constraining the values of attributes in $C$, and requires proving that every method $T_i$ of $C$ preserves the representation invariant and establishes $post_i$ coerced to the state space of $C$ when $pre_i$ also coerced to the state space of $C$ holds. Since in [3] and other works on behavioral subtyping no formal semantics is given to implementation constructs and mechanisms, such as, e.g., super-calls or dynamic binding, this verification can only be done semi-formally.

Consider now the diagram (b) of Fig. 11 illustrating class refinement. First of all, we can reason about specification classes $S$ and $S'$ and implementation classes $C$ and $D$ in a uniform manner, and the behavioral conformance between the participating classes is the class refinement. Since class refinement is transitive, we get directly that $D$, implementing $I'$ by refining its specification $S'$, also refines the specification $S$ of $I$.

Class refinement can be used to verify correctness even if $D$ happens to be a subclass of $C$. Dynamic binding of self-referential methods, which becomes possible in this case, can be resolved as described in Paper 4, and then we can prove that, e.g., $S' \sqsubseteq D$ using the definition of class refinement. With behavioral subtyping, however, it is not clear how one can prove that a method satisfies certain pre- and postconditions in the presence of dynamic binding of self-referential methods.

When used for reasoning about systems with unified subclassing and subtyping, our methodology eliminates a significant amount of proof obligations as compared to behavioral subtyping. We do not need to prove separately that a class and its subclass implement the corresponding type and its behavioral subtype, all that needs to be proved is class refinement between the subclass and the superclass.

Several researchers have developed formal and semi-formal theories of behavioral compatibility based on behavioral subtyping. Raymie Stata and John Guttag in [21] use specialization specifications in the style of behavioral subtyping and require that specification of subclasses be constrained. They informally define correctness of a class with respect to its specialization specification, and require verification of correctness as well as verification of specification constraining, using standard simulation techniques. Stata and Guttag, following America in [3], maintain that declarative specifications are more abstract and easier to understand than operational ones capturing method invocation order. We, on the contrary, feel that the essence of object-oriented programs is in invoking methods on objects, and it might be

necessary to specify explicitly that a certain method calls other methods. As pointed out by Clemens Szyperski in [22], 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 *callbacks*, i.e. self-referential method calls that get redirected to subclasses of the class that originated the call. Our specification language includes method calls and, therefore, allows specifying callbacks as well as fixing an invocation of a method or a certain order of method invocations. Richard Helm et al. in [11], recognizing the need to express behavioral dependencies between co-operating objects, also include method calls in abstract specifications of contracts. No less important, as noted by several authors including Szyperski in [22], specifications of object-oriented programs in terms of pre- and postconditions have only semi-formal semantics, which excludes formal (ultimately, computer-assisted) verification. Finally, pre- and postconditions on methods specify only *partial correctness*, i.e. state the postcondition for a given precondition under the assumption that the method terminates. Our approach guarantees *total correctness*, i.e. no termination assumption has to be made. In general, we shift the focus from correctness reasoning to establishing refinements between methods.

Behavioral dependencies in the presence of subclassing have also been studied in various extensions of Z specification languages, e.g., [13, 8], but only between class specifications and not implementations. By having specification constructs as part of the (extended) programming language, this distinction becomes unnecessary.

# 6 Conclusions and Future Work

We hope that the method of reasoning about correctness of object-oriented and component-based systems supported by the mathematical foundation that we develop in this dissertation can be adopted by practitioners and be used to specify, document, and assist in the development of more reliable systems. As future work we envision development of a tool supporting mechanized verification of behavioral conformance between classes and components. A tool supporting verification of correctness and refinement of imperative programs and known as the Refinement Calculator [12] already exists, and extending it to handling object-oriented and component-based systems appears to be rather natural.

Another direction of developing this work includes extending our specification and verification method to handling concurrent systems. The re-

finement calculus supports reasoning about concurrent systems as reported in [5], and therefore extending it to handling object-oriented concurrency appears to be a challenging yet feasible task.

## References


[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 *Proceedings of ECOOP'87*, LNCS 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*, LNCS 489, pages 60–90, New York, N.Y., 1991. Springer-Verlag.

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

[5] R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

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

[7] J. Bloch. Java collections framework: Collections 1.2. http://java.sun.com/docs/books/tutorial/collections/index.html.

[8] E. Cusack. Inheritance in object-oriented Z. In P. America, editor, *Proceedings of ECOOP'91*, LNCS 512, pages 167–179, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.

[9] 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.

[10] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[11] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Proceedings of OOPSLA/ECOOP'90*, ACM SIGPLAN Notices, pages 169–180, Oct. 1990.

[12] T. Långbacka, R. Ruksenas, and J. von Wright. TkWinHOL: A tool for window inference in HOL. *Higher Order Logic Theorem Proving and its Applications: 8th International Workshop*, 971:245–260, September 1995.

[13] K. Lano and H. Haughton. Reasoning and refinement in object-oriented specification languages. In O. L. Madsen, editor, *Proceedings of ECOOP'92*, LNCS 615. Springer-Verlag, 1992.

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

[15] D. Leivant. Higher order logic. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1: Deduction Methodologies, pages 229–322. Oxford University Press, 1994.

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

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

[18] H. Mössenböck. *Object-Oriented Programming in Oberon-2*. Springer-Verlag, 1993.

[19] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons Inc., 1996.

[20] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[21] 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.

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

# Paper 1

## Class Refinement and Interface Refinement in Object-Oriented Programs

**A. Mikhajlova and E. Sekerinski**

Originally published in *Proceedings of the 4th International Formal Methods Europe Symposium (FME'97)*, edited by J. Fitzgerald, C. B. Jones, and P. Lucas, LNCS 1313, Springer-Verlag, pp. 82–101, September 1997.

# Class Refinement and Interface Refinement in Object-Oriented Programs

Anna Mikhajlova[1] and Emil Sekerinski[2]

[1] Turku Centre for Computer Science, Åbo Akademi University
Lemminkäisenkatu 14A, Turku 20520, Finland
[2] Dept. of Computer Science, Åbo Akademi University
Lemminkäisenkatu 14A, Turku 20520, Finland

**Abstract.** Constructing new classes from existing ones by inheritance or subclassing is a characteristic feature of object-oriented development. Imposing semantic constraints on subclassing allows us to ensure that the behaviour of superclasses is preserved or refined in their subclasses. This paper defines a class refinement relation which captures these semantic constraints. The class refinement relation is based on algorithmic and data refinement supported by Refinement Calculus. Class refinement is generalized to interface refinement, which takes place when a change in user requirements causes interface changes of classes designed as refinements of other classes. We formalize the interface refinement relation and present rules for refinement of clients of the classes involved in this relation.

## 1 Introduction

It has been widely recognized that design and development of object-oriented programs is difficult and intricate. The need for formal basis of object-oriented development was identified by many researchers. We demonstrate how formal methods, in particular, Refinement Calculus of Back, Morgan, and Morris [4, 20, 21], can be used for constructing more reliable object-oriented programs.

A characteristic feature of object-oriented program development is a uniform way of structuring all stages of the development by classes. The programming notation of Refinement Calculus is very convenient for describing object-oriented development because it allows us to specify classes at various abstraction levels. The specification language we use is based on monotonic predicate transformers, has class constructs, supports subclassing and subtype polymorphism. Besides usual imperative statements, the language includes specification statements which may appear in method bodies of classes leading to abstract classes. One of the main benefits offered by this language is that all development stages can be described in a uniform way starting with a simple abstract specification and resulting in a concrete program.

We build a logic of object-oriented programs as a conservative extension of (standard) higher-order logic, in the style of [7]. An alternative approach is undertaken by Abadi and Leino in [2]. They develop a logic of object-oriented

programs in the style of Hoare, prove its soundness, and discuss completeness issues. Naumann [22] defines the semantics of a simple Oberon-like programming language with similar specification constructs as here, also based on predicate transformers. Sekerinski [24, 25] defines a rich object-oriented programming and specification notation by using a type system with subtyping and type parameters, and also using predicate transformers. In both approaches, subtyping is based on extensions of record types. Here we use sum types instead, as suggested by Back and Butler in [6]. One motivation for moving to sum types is to avoid the complications in the typing and the logic when reasoning about record types: the simple typed lambda calculus as the formal basis is sufficient for our purposes. Another advantage of moving to sum types is that we can directly express whether an object is of exactly a certain type or of one of its subtypes (in the record approach, a type contains all the values of its subtypes). Using summations also allows us to model contravariance and covariance on method parameters in a simple way. Finally, to allow objects of a subclass to have different (private) attributes from those of the superclass, hiding by existential types was used in [24, 25]. It turned out that this leads to complications when reasoning about method calls, which are not present when using the model of sum types. In the latter, objects of a subclass can always have different attributes from those of the superclass.

Constructing new classes from existing classes by inheritance, or subclassing, is one of characteristic features of object-oriented program development. However, when a subclass overrides some methods of its superclass, there are no guarantees that its instances will deliver the same or refined behaviour as the instances of the superclass. We define a class refinement relation and relate the notion of subclassing to this relation. When a class $C'$ is constructed by subclassing from $C$ and class refinement holds between them, then it is guaranteed that any behaviour expected from $C$ will necessarily be delivered by $C'$.

Class refinement as defined here is based on data refinement [15, 14, 19, 5]. The definition generalizes that of Sekerinski [24] by allowing contravariance and covariance in the method parameters, and by considering constructor methods. Class refinement has also been studied in various extensions of the Z specification languages, e.g. [16, 17], but only between class specifications and not implementations. Other approaches on "behavioural subtyping" of classes [3, 18, 10] also make a distinction between the specification of a class and its implementation. By having specification constructs as part of the (extended) programming language, this distinction becomes unnecessary.

Subclassing requires that parameter types of a method be the same in the subclass and in the superclass or, at most, subject to contravariance and covariance rules, as described in [9, 1]. However, sometimes a change in user requirements causes interface changes of classes designed as refinements of other classes. We formalize the interface refinement relation as a generalization of class refinement, and present rules for refinement of clients of the classes involved in this relation. Interface refinement has also been considered by Broy in [8], but for networks of communicating components rather than for classes.

*Paper Outline:* In Section 2 we present the required concepts of the Refinement Calculus formalism. In Section 3 we explain our model of objects, classes, subclassing, and subtyping polymorphism. Section 4 defines the class refinement relation. In the following section we generalize class refinement to interface refinement, formalize implicit client refinement, and discuss explicit client refinement. Finally, we conclude with considering applications of our work.

## 2 Refinement Calculus

A *predicate* over a set of states $\Sigma$ is a boolean function $p : \Sigma \to Bool$ which assigns a truth value to each state. The set of predicates on $\Sigma$ is denoted $\mathcal{P}\Sigma$. The *entailment ordering* on predicates is defined by pointwise extension, so that for $p, q : \mathcal{P}\Sigma$,

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

A *relation* from $\Sigma$ to $\Gamma$ is a function $P : \Sigma \to \mathcal{P}\Gamma$ that maps each state $\sigma$ to a predicate on $\Gamma$. We write

$$\Sigma \leftrightarrow \Gamma \;\widehat{=}\; \Sigma \to \mathcal{P}\Gamma$$

to denote a set of all relations from $\Sigma$ to $\Gamma$. This view of relations is isomorphic to viewing them as predicates on the cartesian space $\Sigma \times \Gamma$. The *identity relation* and the *composition* of relations are defined as follows:

$$\begin{array}{rcl} Id\ x\ y & \widehat{=} & x = y \\ (P; Q)\ x\ z & \widehat{=} & (\exists y \;\cdot\; P\ x\ y \;\wedge\; Q\ y\ z) \end{array}$$

A *predicate transformer* is a function $S : \mathcal{P}\Gamma \to \mathcal{P}\Sigma$ from predicates to predicates. We write

$$\Sigma \mapsto \Gamma \;\widehat{=}\; \mathcal{P}\Gamma \to \mathcal{P}\Sigma$$

to denote a set of all predicate transformers from $\Sigma$ to $\Gamma$. Program statements in Refinement Calculus are identified with weakest-precondition monotonic predicate transformers that map a postcondition $q : \mathcal{P}\Gamma$ to the weakest precondition $p : \mathcal{P}\Sigma$ such that the program is guaranteed to terminate in a final state satisfying $q$ whenever the initial state satisfies $p$. A program statement $S$ need not have identical initial and final state spaces, though if it does, we write $S : \Xi(\Sigma)$ instead of $S : \Sigma \mapsto \Sigma$.

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

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

The refinement ordering on predicate transformers models the notion of total-correctness preserving program refinement. For statements $S$ and $T$, the relation $S \sqsubseteq T$ holds if and only if $T$ satisfies any specification satisfied by $S$.

The **abort** statement maps each postcondition to the identically false predicate *false*, and the **magic** statement maps each postcondition to the identically true predicate *true*. The **abort** statement is never guaranteed to terminate, while the **magic** statement is *miraculous* since it is always guaranteed to establish any postcondition.

Sequential composition of program statements is modeled by functional composition of predicate transformers. For $S : \Sigma \mapsto \Gamma$, $T : \Gamma \mapsto \Delta$ and $q : \mathcal{P}\Delta$,

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

The program statement $\mathbf{skip}_\Sigma$ is modeled by the identity predicate transformer on $\mathcal{P}\Sigma$.

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

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

When started in a state $\sigma$, $\{P\}$ angelically chooses a new state $\gamma$ such that $P\ \sigma\ \gamma$ holds, while $[P]$ demonically chooses a new state $\gamma$ such that $P\ \sigma\ \gamma$ holds. If no such state exists, then $\{P\}$ aborts, whereas $[P]$ behaves as **magic**, i.e. can establish any postcondition.

Ordinary program constructs may be modeled using the basic predicate transformers and operators presented above. For example, in a state space with two components $(x : T, y : S)$, an assignment statement may be modeled by the demonic update:

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

Our specification language includes specification statements. The *demonic specification statement* is written $[x := x' \cdot\ b]$, and the *angelic specification statement* is written $\{x\ :=\ x' \cdot\ \ b\}$, where $b$ is a boolean expression relating $x$ and $x'$. The program variable $x$ is assigned a value $x'$ satisfying $b$. These statements correspond to the demonic and the angelic updates respectively:

$$x := x' \cdot\ b \quad \widehat{=} \quad R,\ \text{where}\ R\ (x,y)(x',y')\ =\ b\ \wedge (y' = y)$$

We also have an *assertion*, written $\{p\}$, where $p$ is a predicate stating a condition on program variables. This assertion behaves as **skip** if $p$ is satisfied and as **abort** otherwise.

Finally, the language supports *local variables*. The construct $|[\ \mathbf{var}\ z\ \bullet\ S\ ]|$ states that the program variable $z$ is local to $S$:

$$|[\ \mathbf{var}\ z\ \bullet\ S\ ]| \quad \widehat{=} \quad [\mathit{Enter}_z]; S; [\mathit{Exit}_z],\ \text{where}$$

$$\begin{array}{lcl} \mathit{Enter}_z(x,y)(x',y',z') & \widehat{=} & (x' = x) \wedge (y' = y)\ \text{and} \\ \mathit{Exit}_z(x,y,z)(x',y') & \widehat{=} & (x' = x) \wedge (y' = y) \end{array}$$

The semantics of other ordinary program constructs, like multiple assignments, **if**-statements, and **do**-loops, is given, e.g. in [7].

Data refinement is a general technique by which one can change the state space in a refinement. For statements $S : \Xi(\Sigma)$ and $S' : \Xi(\Sigma')$, let $R : \Sigma' \leftrightarrow \Sigma$ be an *abstraction relation* between the state spaces $\Sigma$ and $\Sigma'$. The statement $S$ is said to be data refined by $S'$ via $R$, denoted $S \sqsubseteq_R S'$, if

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

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 $[R^{-1}]$ are each others inverses, in the sense that $\{R\};[R^{-1}] \ \sqsubseteq$ **skip** and **skip** $\sqsubseteq\ [R^{-1}];\{R\}$.

Refinement Calculus provides laws 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 laws is given in [7, 20, 12].

**Sum Types and Operators.** In our specification language we widely employ sum types for modeling subtyping polymorphism and dynamic binding. The sum or disjoint union of two types $\Sigma$ and $\Gamma$ is written $\Sigma + \Gamma$. The types $\Sigma$ and $\Gamma$ are called base types of the sum in this case. Associated with the sum types, are the injection relations[1] which map elements of the subsets to elements of the superset summation:

$$\iota_\Sigma : \Sigma \leftrightarrow \Sigma + \Gamma \qquad \iota_\Gamma : \Gamma \leftrightarrow \Sigma + \Gamma$$

and projection relations which relate elements of summation with elements of their subsets:

$$\pi_\Sigma : \Sigma + \Gamma \leftrightarrow \Sigma \qquad \pi_\Gamma : \Sigma + \Gamma \leftrightarrow \Gamma$$

In fact, the projection relation is an inverse of the injection relation for the corresponding subset of the summation.

We define the subtype relation as follows. The type $\Sigma$ is a subtype of $\Sigma'$, written $\Sigma <: \Sigma'$, if $\Sigma = \Sigma'$, or $\Sigma <: \Gamma$ or $\Sigma <: \Gamma'$, where $\Gamma + \Gamma' = \Sigma'$. For example, $\Gamma <: \Gamma + \Gamma'$ and, or course, $\Gamma + \Gamma' <: \Gamma + \Gamma'$. If $\Sigma$ is a subtype of $\Sigma'$, we can always construct the appropriate injection $\iota_\Sigma : \Sigma \leftrightarrow \Sigma'$ and projection $\pi_\Sigma : \Sigma' \leftrightarrow \Sigma$. The subtype relation is reflexive, transitive and antisymmetric.

A summation operator combines statements by forming the disjoint union of their state spaces. This operator is defined in [6] by extension from the

[1] In fact, the injections are functions rather than relations, but for our purposes it is more convenient to treat them as relations.

summation of types. For $S_1 : \Sigma_1 \mapsto \Gamma_1$ and $S_2 : \Sigma_2 \mapsto \Gamma_2$, the summation $S_1 + S_2 : \Sigma_1 + \Sigma_2 \mapsto \Gamma_1 + \Gamma_2$ is a predicate transformer such that the effect of executing it in some initial state $\sigma$ depends on the base type of $\sigma$. If $\sigma : \Sigma_1$ then $S_1$ is executed, while if it is of type $\Sigma_2$, then $S_2$ is executed.

The summation operator was shown to satisfy a number of useful properties. The one of interest to us is that it preserves refinement, allowing us to refine elements of the summation separately:

$$S_1 \sqsubseteq S_1' \ \wedge \ S_2 \sqsubseteq S_2' \ \Rightarrow \ (S_1 + S_2) \sqsubseteq (S_1' + S_2')$$

**Product Types and Operators.** The cartesian product of two types $\Sigma$ and $\Gamma$ is written $\Sigma \times \Gamma$. The product operator combines predicate transformers by forming the cartesian product of their state spaces. For $S_1 : \Sigma_1 \mapsto \Gamma_1$ 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 $S_2$.

In addition to many other useful properties, the product operator preserves refinement:

$$S_1 \sqsubseteq S_1' \ \wedge \ S_2 \sqsubseteq S_2' \ \Rightarrow \ (S_1 \times S_2) \sqsubseteq (S_1' \times S_2')$$

For $S : \Sigma \mapsto \Sigma$ we define lifting to a product predicate transformer of type $\Sigma \times \Gamma \mapsto \Sigma \times \Gamma$ as $S \times \mathbf{skip}_\Gamma$. When lifting is obvious from the context, we will simply write $S$ instead of $S \times \mathbf{skip}_\Gamma$.

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

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

## 3 Specifying Objects and Classes

Object-oriented systems are characterized by *objects*, which group together data, and operations for manipulating that data. The operations, called *methods*, can be invoked only by sending *messages* to the object. The complete set of messages that the object understands is characterized by the *interface* of the object. The interface represents the signatures of object methods, i.e. the name and the types of input and output parameters. As opposed to the interface, the *object type* is the type of object *attributes*. We consider all attributes as private or hidden, and all methods as public or visible to clients of the object. Accordingly, two objects with the same public part, i.e. the same interface, can differ in their private part, i.e. object types.

We focus on modeling class-based object-oriented languages, which form the mainstream of object-oriented programming. Accordingly, we take a view that objects are instantiated by classes. A class is a pattern used to describe objects with identical behaviour through specifying their interface. Specifically, a class

describes what attributes each object will have, the specification for each method, and the way the objects are created. We declare a class as follows:

$$
\begin{array}{l}
C \;=\; \textbf{class} \\
\quad attr_1 : \Sigma_1, \ldots, attr_m : \Sigma_m \\
\quad C\ (p : \Psi) \;=\; S, \\
\quad Meth_1\ (g_1 : \Gamma_1) : \Delta_1 \;=\; T_1, \\
\quad \ldots \\
\quad Meth_n\ (g_n : \Gamma_n) : \Delta_n \;=\; T_n \\
\textbf{end}
\end{array}
$$

Class attributes $(attr_1, \ldots, attr_m)$ abbreviated further on as $attr$ have the corresponding types $\Sigma_1$ through $\Sigma_m$. The type of $attr$ is then $\Sigma \;=\; \Sigma_1 \times \ldots \times \Sigma_m$[2]. A *class constructor* is used to instantiate objects and is distinguished by the same name as the class. Due to the fact that the constructor concerns object creation rather than object functionality, it is associated with the class rather than with the specified interface. We take a view that the constructor signature is not part of the interface specified by the class. The statement $S : \Xi(\Sigma \times \Psi)$ representing a body of the constructor initializes the attributes using input $p : \Psi$.

Methods $Meth_1$ through $Meth_n$ specified by bodies $T_1, \ldots, T_n$ operate on the attributes and realize the object functionality. Every statement $T_i$ is, in general, of type $\Xi(\Sigma \times \Gamma_i \times \Delta_i)$, where $\Sigma$ is the type of class attributes, $\Gamma_i$ and $\Delta_i$ are the types of input and output parameters respectively. A method may be parameterless with both $\Gamma_i$ and $\Delta_i$ the unit type (), have only input or only output parameters. When a method has an output parameter, a special variable $res : \Delta_i$ represents the result and assignment to this variable models returning a value in the output parameter. The signature of every method is part of the specified interface.

The object type specified by a class can always be extracted from the class and we do not need to declare it explicitly. We use $\tau(C)$ to denote the type of objects generated by the class $C$. Naturally, $\tau(C)$ is just another name for $\Sigma$.

Being declared as such, the class $C$ is modeled by a tuple $(K, M_1, \ldots, M_n)$, where

$$
\begin{array}{lcl}
K & = & [Enter_{attr}]; S; [Exit_p] \\
M_i & = & [Enter_{res}]; T_i; [Exit_{g_i}],\ \text{for i = 1,..., n.}
\end{array}
$$

Further on we will refer to $K$ as the constructor and to $M_1, \ldots, M_n$ as the methods, unless stated otherwise.

Instantiating a new variable of object type by class $C$ is modeled by invoking the corresponding class constructor:

$$c.C(e) \quad \widehat{=} \quad [Enter_p]; p := e; K; c := attr; [Exit_{attr}]$$

Naturally, a variable of object type can be local to a block:

$$|[\ \textbf{var}\ c : C(e)\ \bullet\ S\ ]| \quad \widehat{=} \quad [Enter_c]; c.C(e); S; [Exit_c]$$

[2] We impose a non-recursiveness restriction on $\Sigma$ so that none of $\Sigma_i$ is equal to $\Sigma$. This restriction allows us to stay within the simple-typed lambda calculus.

Often a class aggregates objects of another class, i.e. some attributes can be of object types. In this case the class declaration states the object types of these attributes, but only the constructor invocation actually introduces new objects into the state space and initializes them.

Invocation of a method $Meth_i\ (g_i : \Gamma_i) : \Delta_i$ on an object $c$ instantiated by class $C$ is modeled as follows:

$$\begin{aligned} d := c.Meth_i\ (g) \quad \widehat{=} \quad & [Enter_{attr}]; [Enter_{g_i}]; \\ & attr := c; g_i := g; M_i; c := attr; d := res; \\ & [Exit_{res}]; [Exit_{attr}] \end{aligned}$$

As an example of a class specification consider a class of bank accounts. An account should have an owner, and it should be possible to deposit and withdraw money in the currency of choice and check the current balance. We present the specification of the class *Account* in Fig. 1.

---

$Account\ =$ **class**
$owner : Name, balance : Currency$

$Account\ (name : Name, sum : Currency)\ =\ owner := name; balance := sum,$

$Deposit\ (sum : Currency, from : Name, when : Date)\ =$
$\{sum > 0\}; balance := balance + sum,$

$Withdraw\ (sum : Currency, to : Name, when : Date)\ =$
$\{sum > 0\ \wedge\ sum \leq balance\}; balance := balance - sum,$

$Owner\ () : Name\ =\ res := owner,$

$Balance\ () : Currency\ =\ res := balance$
**end**

---

**Fig. 1.** Specification of bank account

Obviously, this specification only demonstrates the most general behaviour of bank accounts. For example, when specifying *Deposit*, we only state that *balance* is increased by *sum* and leave the changes to the other input parameters unspecified. We would like to *subclass* from *Account* more concrete account classes. Let us consider specification of subclasses more closely.

### 3.1 Subclassing

Subclassing [3] is a mechanism for constructing new classes from existing ones by *inheriting* some or all of their attributes and methods, possibly *overriding* some attributes and methods, and adding extra methods. We limit our consideration

[3] We prefer the term *subclassing* to *implementation inheritance* because the latter literally means reuse of existing methods and does not, as such, suggest the possibility of method overriding.

of class construction to inheritance and overriding. Addition of extra methods is a non-trivial issue because of inconsistencies possibly introduced by extra methods which become apparent in presence of subtype aliasing, and is treated in another study.

We describe a subclass of class $C$ as follows:

$$
\begin{array}{l}
C' \;=\; \textbf{subclass of } C \\
\quad attr'_1 : \Sigma'_1, \ldots, attr'_p : \Sigma'_p \\
\quad C'\ (p : \Psi) \;=\; S', \\
\quad Meth_1\ (g_1 : \Gamma_1) : \Delta_1 \;=\; T'_1, \\
\quad \ldots \\
\quad Meth_k\ (g_k : \Gamma_k) : \Delta_k \;=\; T'_k \\
\textbf{end}
\end{array}
$$

Class attributes $attr'_1, \ldots, attr'_p$ have the corresponding types $\Sigma'_1$ through $\Sigma'_p$. Some of these attributes are inherited from the superclass $C$, others override attributes of $C$, and the other ones are new. The class $C'$ has its own class constructor without inheriting the one associated with the superclass. The bodies $T'_1, \ldots, T'_k$ override the corresponding $Meth_1, \ldots, Meth_k$ body definitions defined in $C$. The bodies of methods named $Meth_{k+1}, \ldots, Meth_n$ are inherited from the superclass $C$. The class $C'$ is modeled by a tuple $(K', M'_1, \ldots, M'_n)$, where the statements $K'$ and all $M'_i$ are related to $S', T'_1, \ldots, T'_n$ as described above.

We view subclassing as a syntactic relation on classes, since subclasses are distinguished by an appropriate declaration. Syntactic subclassing implies conformance of interfaces, in the sense that a subclass specifies an interface conforming to the one specified by its superclass. In the simple case the interface specified by a subclass is the same as that of the superclass. In the next section we explain how this requirement can be relaxed.

As an example of subclassing consider extending the class *Account* with a list of transactions, where every transaction has a sender, a receiver, an amount of money being transferred, and a date. We specify a record type representing transactions as follows:

$$
\begin{array}{l}
\textbf{type } Transaction \;=\; \textbf{record} \\
\quad from : Name,\ to : Name,\ amount : Currency,\ date : Date \\
\textbf{end}
\end{array}
$$

Here $Name$, $Currency$ and $Date$ are simple types. $Date$ is a type of six digit arrays for representing a day, a month, and a year, for example as '251296' for December 25, 1996.

Now we can specify in Fig. 2 a class of bank accounts based on sequences of transactions. Notice that we specify only the overriding methods, *Owner* and *Balance* are inherited from the superclass *Account*.

$AccountPlus$ = **subclass of** $Account$
$owner : Name, balance : Currency, transactions :$ **seq of** $Transaction$

$AccountPlus\ (name : Name, sum : Currency)$ =
$owner := name; balance := sum; transactions := \langle\rangle,$

$Deposit\ (sum : Currency, from : Name, when : Date)$ =
$\{sum > 0\}; |[$ **var** $t : Transaction \bullet t := (from, owner, sum, when);$
$transactions := transactions^\frown\langle t\rangle; balance := balance + sum\ ]|,$

$Withdraw\ (sum : Currency, to : Name, when : Date)$ =
$\{sum > 0 \ \wedge \ sum \leq balance\};$
$|[$ **var** $t : Transaction \bullet t := (owner, to, sum * (-1), when);$
$transactions := transactions^\frown\langle t\rangle; balance := balance - sum\ ]|$

**end**

**Fig. 2.** Specification of account based on transactions

### 3.2 Modeling Subtyping Polymorphism

To model subtyping polymorphism, we allow object types to be sum types. The idea is to group together an object type of a certain class and object types of all its subclasses, to form a polymorphic object type. A variable of such a sum type can be instantiated to any base type of the summation, in other words, to any object instantiated by a class whose object type is the base type of the summation. We will call the object types of only one class *ground* and summations of object types *polymorphic*. Since a ground object type uniquely identifies the class of objects, we can always tell whether a certain object is an instance of a certain class.

A sum of object types, denoted by $\tau(C)^+$ is defined to be such that its base types are $\tau(C)$ and all the object types of subclasses of $C$. For example, if $D$ is the only subclass of $C$ with the object type $\tau(D)$, then $\tau(C)^+ = \tau(C) + \tau(D)$. Naturally, we have that

$$\tau(C) <: \tau(C)^+ \text{ and } \tau(D) <: \tau(C)^+.$$

A variable $c : \tau(C)^+$ can be instantiated by either $C$ or $D$. The *subsumption* property holds of $c$, namely, if $c : \tau(C)$ and $\tau(C) <: \tau(C)^+$ then $c : \tau(C)^+$. This property is characteristic of subtype relations, it means that an object of type $\tau(C)$ can be viewed as an object of the supertype $\tau(C)^+$.

Suppose a method $Meth_i$ is specified in both $C$ and $D$ by statements $M_i$ and $M_i'$ respectively. An invocation of $Meth_i$ on an object $c$ of type $\tau(C)^+$ is modeled as follows:

$$c.Meth_i() \ \widehat{=} \ \left(\begin{array}{l} [Enter_{attr}]; \\ attr := c; M_i; c := attr; \\ [Exit_{attr}] \end{array}\right) + \left(\begin{array}{l} [Enter_{attr'}]; \\ attr' := c; M_i'; c := attr'; \\ [Exit_{attr'}] \end{array}\right)$$

where $attr : \Sigma$ and $attr' : \Sigma'$ are attributes of $C$ and $D$ respectively. Modeling an invocation of a method having input and output parameters is similar to method invocation on a non-polymorphic object.

Being equipped with subtyping polymorphism, we can allow overriding methods in a subclass to be generalized on the type of input parameters or specialized on the type of output parameters. In the first case this type redefinition is *contravariant* and in the second *covariant*[4]. When one interface is the same as the other, except that it can redefine contravariantly input parameter types and covariantly output parameter types, this interface conforms to the original one.

As an example of using polymorphic object types let us consider a client of the classes *Account* and *AccountPlus*, a bank which maintains a sequence of accounts and can transfer money from one account to another. The specification of the class *Bank* is presented in Fig. 3.

---

$Bank$ = **class**
  $accounts$ : **seq of** $\tau(Account)$

  $Transfer\ (from : \tau(Account), to : \tau(Account), s : Currency, d : Date)$ =
    $\{sum > 0\}$;
    |[ **var** $sender, receiver : Name$ •
    $sender := from.Owner(); receiver := to.Owner()$;
    $from.Withdraw(s, receiver, d); to.Deposit(s, sender, d)$ ]|
**end**

---

**Fig. 3.** Specification of bank using accounts

A subclass of *Bank* can redefine the method $Transfer$ with input parameters of types $\tau(Account)^+$ to meet the contravariant constraint. The new bank will be able to work with both *Account* and *AccountPlus* instances in this case, provided that the *accounts* attribute is redefined to be of type **seq of** $\tau(Account)^+$.

## 4 Class Refinement

When a subclass overrides some methods of its superclass, there are no guarantees that its instances will deliver the same or refined behaviour as the instances of the superclass. Unrestricted method overriding in a subclass can lead to an arbitrary behaviour of its instances. When used in a superclass context, such subclass instances may invalidate their clients. For example, the *Deposit* method of *Account* can be overridden so that the money is, in fact, withdrawn from the account instead of being deposited. Then the owner of the account will actually be at a loss.

Therefore, we would like to ensure that whenever $C'$ is subclassed from $C$, any behaviour expected from $C$ will necessarily be delivered by $C'$. For this purpose, we introduce the notion of class refinement between $C$ and $C'$.

Consider two classes $C = (K, M_1, \ldots, M_n)$ and $C' = (K', M'_1, \ldots, M'_n)$ such that $K : \Psi \ \mapsto \ \Sigma$ and $K' : \Psi' \ \mapsto \ \Sigma'$ are the corresponding class constructors, and all $M_i : \Sigma \times \Gamma_i \ \mapsto \ \Sigma \times \Delta_i$ and $M'_i : \Sigma' \times \Gamma'_i \ \mapsto \ \Sigma' \times \Delta'_i$ are the

[4] For a more extensive explanation of covariance and contravariance see, e.g. [1].

corresponding methods. The input parameter types of the constructors and the methods are either the same or contravariant, such that $\Psi <: \Psi'$ and $\Gamma_i <: \Gamma'_i$. The output parameter types of the methods are either the same or covariant, $\Delta'_i <: \Delta_i$.

We define the refinement of class constructors $K$ and $K'$ with respect to a relation $R$ as follows:

$$K \sqsubseteq_R K' \quad \widehat{=} \quad \{\pi_\Psi\};\ K \quad \sqsubseteq \quad K';\ \{R\} \tag{1}$$

where $R : \Sigma' \leftrightarrow \Sigma$ is an abstraction relation coercing attribute types of $C'$ to those of $C$, and $\pi_\Psi$ is the projection relation coercing $\Psi'$ to $\Psi$.

The refinement of all corresponding methods $M_i$ and $M'_i$ with respect to the relation $R$ is defined as

$$M_i \sqsubseteq_R M'_i \quad \widehat{=} \quad \{R \times \pi_{\Gamma_i}\};\ M_i \quad \sqsubseteq \quad M'_i;\ \{R \times \iota_{\Delta'_i}\} \tag{2}$$

Here $R$ is as above, $\pi_{\Gamma_i} : \Gamma'_i \leftrightarrow \Gamma_i$ projects the corresponding input parameters, and $\iota_{\Delta'_i} : \Delta'_i \leftrightarrow \Delta_i$ injects the corresponding output parameters. Obviously, when $\Gamma_i = \Gamma'_i$, the projection relation $\pi_{\Gamma_i}$ is taken to be the identity relation *Id*. The same holds when $\Delta_i = \Delta'_i$, namely, $\iota_{\Delta'_i} = Id$.

Now we can define the class refinement relation as follows.

**Definition 1 (Class refinement).** *The class $C$ is refined by the class $C'$, written $C \sqsubseteq C'$, if for some abstraction relation $R : \tau(C') \leftrightarrow \tau(C)$*

1. *The constructor of $C'$ refines the constructor of $C$ as defined in (1)*
2. *Every method of $C'$ refines the corresponding method of $C$ as defined in (2).*

The class refinement relation shares the properties of statement refinement and is, thus, reflexive and transitive.

**Theorem 1.** *Let $C, C'$ and $C''$ be classes. Then the following properties hold:*

1. $C \sqsubseteq C$
2. $C \sqsubseteq C' \ \wedge \ C' \sqsubseteq C'' \quad \Rightarrow \quad C \sqsubseteq C''$

Declaring one class as a subclass of another raises the proof obligation that the class refinement relation holds between these classes. This is a semantic constraint that we impose on subclassing to ensure that behaviour of subclasses conforms to the behaviour of their superclasses and, respectively, that the subclasses can be used in the superclass context.

As an example of class refinement consider the classes *Account* and *AccountPlus*. Since the latter is declared as a subclass of the former, we get a proof obligation $Account \sqsubseteq AccountPlus$. Under the abstraction relation $R\,(o', b', t')(o, b) \ = \ (o' = o) \wedge (b' = b)$, where $o, b$ correspond to $owner, balance$ of *Account* and $o', b', t'$ correspond to $owner, balance, transactions$ of *AccountPlus*, this proof obligation can be discharged, but we omit the proof for the lack of space.

## 5 Interface Refinement

Subclassing requires that parameter types of a method be the same in the subclass and in the superclass or, at most, subject to contravariance and covariance rules. However, sometimes, a change in user requirements causes interface changes of classes designed as refinements of other classes.

When the new interface is similar to the old one, we can identify abstraction relations coercing the new method parameters to the old ones. For every pair of corresponding methods we need to find two such relations, for input and output parameters. The rôle of these parameter abstraction relations is crucial for interface refinement of classes and for refinement of their clients. Let us first define the interface refinement relation between classes with respect to these relations.

Consider two classes $C = (K, M_1, \ldots, M_n)$ and $C' = (K', M'_1, \ldots, M'_n)$ with attribute types $\Sigma$ and $\Sigma'$ respectively, such that $K : \Psi \mapsto \Sigma$ and $K' : \Psi' \mapsto \Sigma'$ are the class constructors, and all $M_i : \Sigma \times \Gamma_i \mapsto \Sigma \times \Delta_i$ and $M'_i : \Sigma' \times \Gamma'_i \mapsto \Sigma' \times \Delta'_i$ are the corresponding methods.

Let $R : \Sigma' \leftrightarrow \Sigma$ be an abstraction relation coercing attribute types of $C'$ to those of $C$, and $I_0 : \Psi' \leftrightarrow \Psi$ an abstraction relation coercing the corresponding input parameter types. We define the refinement of class constructors $K$ and $K'$ through $R$ and $I_0$ as follows:

$$K \sqsubseteq_{R,I_0} K' \quad = \quad \{I_0\};\ K \quad \sqsubseteq \quad K';\ \{R\} \tag{3}$$

Obviously, (3) is a generalization of (1) with $I_0 = \pi_\Psi$ when the input types are contravariant.

Let $R : \Sigma' \leftrightarrow \Sigma$ be as before, and $I_i : \Gamma'_i \leftrightarrow \Gamma_i$ and $O_i : \Delta'_i \leftrightarrow \Delta_i$ be abstraction relations coercing the corresponding input and output parameter types. We define the refinement of corresponding methods $M_i$ and $M'_i$ through $R, I_i$ and $O_i$ as follows:

$$M_i \sqsubseteq_{R,I_i,O_i} M'_i \quad = \quad \{R \times I_i\};\ M_i \quad \sqsubseteq \quad M'_i;\ \{R \times O_i\} \tag{4}$$

Obviously, (4) is a generalization of (2) with $I_i = \pi_{\Gamma_i}$ when the inputs are contravariant, i.e. $\Gamma_i <: \Gamma'_i$, and with $O_i = \iota_{\Delta'_i}$ when the outputs are covariant, i.e. $\Delta'_i <: \Delta_i$.

**Definition 2 (Interface refinement).** *The class $C$ is interface refined by the class $C'$, written $C \sqsubseteq_{I,O} C'$, with respect to parameter abstraction relations $I = (I_0, I_1, \ldots, I_n)$ and $O = (O_1, \ldots, O_n)$ if for some abstraction relation $R : \tau(C') \leftrightarrow \tau(C)$*

1. *The constructor of $C'$ refines the constructor of $C$ as defined in (3)*
2. *Every method of $C'$ refines the corresponding method of $C$ as defined in (4).*

Being defined as such, interface refinement of classes is a generalization of class refinement. When every $I_i$ and $O_i$ is the identity relation or the projection and injection relations respectively, interface refinement is specialized to class refinement. The interface refinement relation has the basic properties required of a refinement relation, i.e. reflexivity and transitivity.

**Theorem 2.** *Let $C, C'$ and $C''$ be classes. Then the following properties hold:*

1. $C \sqsubseteq_{Id,Id} C$
2. $C \sqsubseteq_{I,O} C' \ \wedge \ C' \sqsubseteq_{I',O'} C'' \ \Rightarrow \ C \sqsubseteq_{I';I,O';O} C''$

*where the relational compositions $I'; I$ and $O'; O$ on tuples of relations are taken elementwise.*

*Proof.* The proof of (1) follows directly from reflexivity of statement refinement by taking the abstraction relation $R$ to be $Id$. To prove (2) we assume that $C \sqsubseteq_{I,O} C'$ and $C' \sqsubseteq_{I',O'} C''$ hold for abstraction relations $R$ and $R'$ respectively. We then show that methods $M_i, M_i'$ and $M_i''$ of the corresponding classes $C, C'$ and $C''$ have the property:

$$\{R \times I_i\}; M_i \ \sqsubseteq \ M_i'; \{R \times O_i\} \ \wedge \ \{R' \times I_i'\}; M_i' \ \sqsubseteq \ M_i''; \{R' \times O_i'\} \ \Rightarrow$$
$$\{(R'; R) \times (I_i'; I_i)\}; M_i \ \sqsubseteq \ M_i''; \{(R'; R) \times (O_i'; O_i)\}$$

The proof of the property is as follows:

$$\begin{array}{ll}
 & \{(R'; R) \times (I_i'; I_i)\}; M_i \ \sqsubseteq \ M_i''; \{(R'; R) \times (O_i'; O_i)\} \\
= & \text{lemma } (P; P') \times (Q; Q') \ = \ (P \times Q); (P' \times Q') \\
 & \{(R' \times I_i'); (R \times I_i)\}; M_i \ \sqsubseteq \ M_i''; \{(R' \times O_i'); (R \times O_i)\} \\
= & \text{homomorphism of angelic update statement } \{P\}; \{Q\} \ = \ \{P; Q\} \\
 & \{R' \times I_i'\}; \{R \times I_i\}; M_i \ \sqsubseteq \ M_i''; \{R' \times O_i'\}; \{R \times O_i\} \\
\Leftarrow & \text{assumption } \{R \times I_i\}; M_i \ \sqsubseteq \ M_i'; \{R \times O_i\} \\
 & \{R' \times I_i'\}; M_i'; \{R \times O_i\} \ \sqsubseteq \ M_i''; \{R' \times O_i'\}; \{R \times O_i\} \\
\Leftarrow & \text{assumption } \{R' \times I_i'\}; M_i' \ \sqsubseteq \ M_i''; \{R' \times O_i'\} \\
 & M_i''; \{R' \times O_i'\}; \{R \times O_i\} \ \sqsubseteq \ M_i''; \{R' \times O_i'\}; \{R \times O_i\} \\
= & \text{reflexivity of statement refinement} \\
 & \mathit{true}
\end{array}$$

The proof of the corresponding property for constructors is similar. □

Theorem 2 follows by specializing $I$ and $O$ appropriately.

As an example of interface refinement consider our previous specification of transactions, accounts and banks. Suppose that facing the start of the new century, we'd like to change the type of dates so that it's possible to specify a four-digit year:

$$\textbf{type} \ \ \mathit{NewDate} = \ \textbf{array} \ [1..8] \ \textbf{of} \ \mathit{Digit}$$

Accordingly, we define a new transaction record type $NewTran$ which is the same as $Transaction$ except that the *date* field is now of type $NewDate$. We construct a new class of accounts using $NewTran$ transactions as shown in Fig. 4. We omit specifications of *Owner* and *Balance* methods which are straightforward, and a specification of $Withdraw$, which is similar to that of *Account* with a local variable of type $NewTran$ rather than $Transaction$.

$NDAccount$ = **class**
$owner : Name, balance : Currency, transactions :$ **seq of** $NewTran$

$NDAccount\ (name : Name, initSum : Currency)$ =
$owner := name; balance := initSum; transactions := \langle\rangle,$

$Deposit\ (sum : Currency, from : Name, when : NewDate)$ =
$\{sum > 0\}; |[$ **var** $t : NewTran \bullet t := (from, owner, sum, when);$
$transactions := transactions^\frown\langle t\rangle; balance := balance + sum\ ]|,$
...
**end**

**Fig. 4.** Specification of account based on NewTran

It can be shown that $AccountPlus \sqsubseteq_{I,O} NDAccount$, where $I = (Id \times Id, Id \times Id \times D, Id \times Id \times D, Id, Id)$ and $O = (Id, Id, Id, Id)$. The abstraction relation $D : NewDate \leftrightarrow Date$ is defined so that for constants '1' and '9' of type $Digit$ and for any $d : Date$ and $d' : NewDate$:

$$D(d')(d) \quad = \quad (d'[1..4] = d[1..4]) \wedge (d'[5..6] = \text{`1``9`}) \wedge (d'[7..8] = d[5..6])$$

Now let us consider how parameter abstraction relations can be used for refinement of clients of the interface refined classes. Interface changes in class methods certainly affect clients of the class. Examining the ways the clients get affected allows us to discover the situations when the clients can benefit from the interface refinement of their server classes but need not be changed in any way. We can also establish conditions under which the clients can be systematically changed to use the refined server classes. For every $OldClass$ and $NewClass$, such that $NewClass$ is designed as a refinement of $OldClass$ but specifies a different interface, we distinguish two ways clients of $OldClass$ can be affected and changed.

### 5.1 Implicit Client Refinement

This kind of client refinement happens when it is impractical or impossible to redefine clients of $OldClass$, but is, however, desirable that they work with $NewClass$ which may offer a more efficient implementation or improved functionality to new clients, like in our example. We can implicitly refine clients by employing a so-called *forwarding scheme* illustrated in Fig. 5 using the OMT notation [23]. In this diagram the link with a triangle relates a superclass with a subclass with the superclass above. The link with a diamond shows an aggregation relation, i.e. that $Wrapper$ aggregates an instance of $NewClass$ in the attribute $impl$.

The idea behind such kind of forwarding is to introduce a subclass of $OldClass$, $Wrapper$, which aggregates an instance of $NewClass$ and forwards $OldClass$ method calls to $NewClass$ through this instance. This has also been identified as a reoccuring design pattern by Gamma et al. in [11]. Clients of

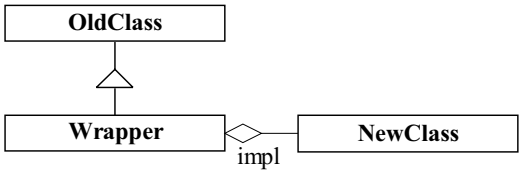



**Fig. 5.** Illustration of forwarding

$OldClass$ can work with $Wrapper$, which is a subclass of $OldClass$, but have all the benefits of working with $NewClass$ if

$$OldClass \sqsubseteq Wrapper \text{ and } Wrapper \sqsubseteq_{I,O} NewClass$$

Consider again our example. The client $Bank$ wants to use *NDAccount* but cannot do so since the latter specifies the interface different from that specified by $AccountPlus$. We can employ the forwarding scheme by introducing in Fig. 6 a new class $AccountWrapper$ which aggregates an instance of *NDAccount* and forwards $AccountPlus$ method calls to *NDAccount* via this instance. Specifications of $Withdraw$ and $Balance$ are straightforward and we omit them for brevity. The function $ToNewDate\ (old : Date) : NewDate$ converts dates from the old format to the new one. In fact, this function can be modeled by the statement $[D^{-1}]$, where $D : NewDate \leftrightarrow Date$ is as before. Provided that the necessary proof obligations are discharged, clients of $AccountPlus$, such as $Bank$, are implicitly refined to work with *NDAccount* via $AccountWrapper$.

Since wrapper classes are of a very specific form, proof obligations can be considerably simplified. Consider a typical wrapper class as given in Fig. 7. The demonic specification statements transform the input parameters of $OldClass$ to the input parameters of $NewClass$ using the corresponding parameter abstraction relations $I_i, i = 0, \ldots, n$. Similarly, the angelic specification statements transform the output parameters of $NewClass$ back to the output parameters of $OldClass$. For the class $Wrapper$ with such a structure, we have the following theorem.

---

$AccountWrapper\ =$ **subclass of** $AccountPlus$
  $impl : \tau(NDAccount)$

  $AccountWrapper\ (name : Name, initSum : Currency)\ =$
    $impl.NDAccount(name, initSum)$,

  $Deposit\ (sum : Currency, from : Name, when : Date)\ =$
    $\{sum > 0\}; |[$ **var** $d : NewDate \bullet$
    $d := ToNewDate(when); impl.Deposit(sum, from, d)\ ]|$,

  $Owner\ () : Name\ =\ res := impl.Owner()$,
  ...
**end**

---

**Fig. 6.** Specification of wrapper class for implicit interface refinement

$$
\begin{array}{l}
Wrapper \;=\; \textbf{subclass of}\ OldClass \\
\quad impl : \tau(NewClass) \\
\quad Wrapper\ (p : \Psi) \;=\; |[\ \textbf{var}\ e : \Psi' \ \bullet\ [e := e' \cdot\ I_0^{-1}\ p\ e']; impl.NewClass(e)\ ]|, \\
\quad Meth_i(g_i : \Gamma_i) : \Delta_i \;= \\
\qquad |[\ \textbf{var}\ c_i : \Gamma_i', d_i : \Delta_i' \ \bullet\ [c_i := c_i' \cdot\ I_i^{-1}\ g_i\ c_i']; \\
\qquad d_i := impl.Meth_i(c_i); \{res := res' \cdot\ O_i\ d_i\ res'\}\ ]|, \\
\quad \ldots \\
\textbf{end}
\end{array}
$$

**Fig. 7.** Schema of wrapper class for implicit interface refinement

**Theorem 3.** *For parameter abstraction relations* $I = (I_0, I_1, \ldots, I_n)$ *and* $O = (O_1, \ldots, O_n)$ *the following property holds:*

$$OldClass \sqsubseteq_{I,O} NewClass \;\Rightarrow\; OldClass \sqsubseteq Wrapper$$

The form of specification statements gives insight into suitable restrictions when choosing the parameter abstraction relations $I_i$ and $O_i$. If $I_i^{-1}$ is partial, then the corresponding specification statement can be **magic** and, thus, is not implementable. Hence, $I_i$ has to be surjective, i.e. relate all possible values of the old input parameters to some values of the new input parameters. Likewise, if $O_i$ is non-deterministic (not functional), then the result $res : \Delta_i$ is chosen angelically, and is, therefore, not implementable. Hence $O_i$ must be deterministic (functional), i.e. relate values of the new result parameters $d_i : \Delta_i'$ to at most one value of the old result parameters $res : \Delta_i$.

### 5.2 Explicit Client Refinement

This kind of client refinement happens quite often in the process of object-oriented development. After *NewClass* has been developed, using *OldClass* may become impractical and undesirable, and therefore, a client *OldClient* of *OldClass* should be explicitly changed to work with *NewClass* instead. We can construct *NewClient* by refinement from *OldClient*. Unfortunately, there are no guarantees that the interface of *NewClient* will conform to that of *OldClient*. Accordingly, we must consider two cases, when *NewClient* is a subclass of *OldClient* and when it is its interface refinement.

When the object type $\tau(OldClass)$ and the types causing the interface change of *OldClass* to *NewClass* are not part of *OldClient* interface, the refinement of *OldClient*, *NewClient*, can be its subclass. In other words, *NewClient* can specify the interface conforming to that of *OldClient*. Naturally, every class using *OldClient* can then use *NewClient* instead and is implicitly refined without respecification.

We feel that there is a strong connection between parameter abstraction relations with respect to which interface refinement is defined and explicit refinement of clients of the refined classes. Investigating how clients can be explicitly refined based on the parameter abstraction relations for the server classes remains the topic of current research.

## 6 Conclusions

Our approach is suited for documenting, constructing, and verifying different kinds of object-oriented systems because of its uniform way of specifying a program at different abstraction levels and the possibility of stepwise development. We have defined the class refinement relation and the interface refinement relation which allow a developer to construct extensible object-oriented programs from specifications and assure reliability of the final program.

Our model of classes, subclassing, and subtyping polymorphism can be used to reason about the meaning of programs constructed using the separate subclassing and interface inheritance hierarchies, like in Java [13], Sather [26], and some other languages. In that approach interface inheritance is the basis for subtyping polymorphism, whereas subclassing is used only for implementation reuse. By associating a specification class with every interface type, we can reason about the behaviour of objects having this interface. All classes claiming to implement a certain interface must refine its specification class. Subclassing, on the other hand, does not, in general, require establishing class refinement between the superclass and the subclass.

For simplicity we consider only single inheritance, but multiple inheritance does not introduce much complication. With a suitable mechanism for resolving clashes in method names, multiple inheritance has the same semantics as we give for single inheritance. Namely, ensuring that a subclass $D$ preserves behaviour of all its declared superclasses $C_1, \ldots, C_n$ requires proving class refinements $C_1 \sqsubseteq D, \ldots, C_n \sqsubseteq D$ for every corresponding superclass-subclass pair.

Using formal specification and verification is especially important for open systems, such as object-oriented frameworks and component-based systems. Frameworks incorporate a reusable design for a specific class of software and dictate a particular architecture of potential applications. When building an application, the user needs to customize framework classes to specific needs of this application. To do so, he must understand the message flow in the framework and the relationship among the framework classes. The intrinsic feature of open component-based systems is a late integration phase, meaning that components are developed by different manufacturers and then integrated together by their users.

A fine-grained specification can accurately describe the fixed behaviour of classes. In this respect, such a specification is a perfect documentation of a framework or a component, because the user does not have to decipher ambiguous, incomplete, and often outdated verbal descriptions. Neither is it necessary to confront the bulk of source code to gain a complete understanding of the system behaviour. The programming notation we use allows the developer to abstract from implementation details and specify classes with abstract state space and non-deterministic behaviour of methods, expressing only the necessary functionality. Moreover, a certain implementation can be a commercial secret, whereas a concise and complete specification distributed instead of source code enables the user to understand the functionality and protects corporate interests.

Formal verification in the form of establishing a class refinement relation between specifications and their implementations guarantees that any behaviour expected from the specifications will be delivered by the implementations.

It has been acknowledged that frameworks are usually developed using a spiral model that takes feedback from actual use of the framework into account. It can be expected that such development iterations may result in an interface change of some classes. In this case, interface refinement can be used to verify behavioural compatibility of the corresponding classes and the rules for interface refinement of clients can be used to refine the whole framework.

## Acknowledgments

We would like to thank Ralph Back for a number of fruitful discussions and Martin Büchi for useful comments.

## References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In *Proceedings of TAPSOFT'97*, LNCS 1214, pages 682–696. Springer, April 1997.
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*, LNCS 489, pages 60–90, New York, N.Y., 1991. Springer-Verlag.
4. R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.
5. R. J. R. Back. Changing data representation in the refinement calculus. In *21st Hawaii International Conference on System Sciences*. IEEE, January 1989.
6. R. J. R. Back and M. Butler. Exploring summation and product operators in the refinement calculus. In B. Möller, editor, *Mathematics of Program Construction, 1995*, volume 947. Springer-Verlag, 1995.
7. R. J. R. Back and J. von Wright. Refinement calculus I: Sequential nondeterministic programs. In W. P. d. J. W. deBakker and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, Lecture Notes in Computer Science, pages 42–66. Springer-Verlag, 1990.
8. M. Broy. (Inter-)Action Refinement: The Easy Way. In M. Broy, editor, *Program Design Calculi*, pages 121–158, Berlin Heidelberg, 1993. Springer-Verlag.
9. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.
10. 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.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
12. P. H. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87(1):143–162, 1991.

13. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Sun Microsystems, Mountain View, 1996.
14. J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *European Symposium on Programming*, LNCS 213. Springer-Verlag, 1986.
15. C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.
16. K. Lano and H. Haughton. Reasoning and refinement in object-oriented specification languages. In O. L. Madsen, editor, *Proceedings of ECOOP'92*, LNCS 615. Springer-Verlag, 1992.
17. K. Lano and H. Haughton. *Object-Oriented Specification Case Studies*. Prentice–Hall, New York, 1994.
18. B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
19. C. C. Morgan. Data refinement by miracles. *Information Processing Letters*, 26:243–246, January 1988.
20. C. C. Morgan. *Programming from Specifications*. Prentice–Hall, 1990.
21. J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.
22. D. A. Naumann. Predicate transformer semantics of an Oberon-like language. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, pages 460–480, San Miniato, Italy, 1994.
23. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, 1991.
24. E. Sekerinski. *Verfeinerung in der Objektorientierten Programmkonstruktion*. Dissertation, Universität Karlsruhe, 1994.
25. E. Sekerinski. A type-theoretic basis for an object-oriented refinement calculus. In S. Goldsack and S. Kent, editors, *Formal Methods and Object Technology*. Springer-Verlag, 1996.
26. C. A. Szyperski, S. Omohundro, and S. Murer. Engineering a programming language – the type and class system of Sather. In *First International Conference on Programming Languages and System Architectures*, LNCS 782, Zurich, Switzerland, March 1994. Springer.

# Paper 2

## Refinement of Generic Classes as Semantics of Correct Polymorphic Reuse

**A. Mikhajlova**

Originally published in *Proceedings of the International Refinement Workshop and Formal Methods Pacific (IRW/FMP'98)*, edited by J. Grundy, M. Schwenke, and T. Vickers, Springer Series in Discrete Mathematics and Theoretical Computer Science, pp. 266–285, July 1998, Springer-Verlag.

# Refinement of Generic Classes as Semantics of Correct Polymorphic Reuse

Anna Mikhajlova

Turku Centre for Computer Science, Åbo Akademi University
Lemminkäisenkatu 14A, Turku 20520, Finland

**Abstract.** Constructing new classes from existing ones by inheritance or subclassing is the main reuse mechanism of the object-oriented programming style. Parametric polymorphism is usually combined with object-orientation to enhance reuse for different types, and increase flexibility of class construction. The combination of parametric polymorphism and subtyping polymorphism permits creation of generic classes where the formal type parameter is constrained (or bounded) to be a subtype of a certain object type. Generic classes parameterized with unbounded and bounded type parameters as well as value parameters can be constructed from existing generic classes by subclassing. This kind of generic subclassing allows polymorphic substitutability of generically derived subclass instances for generically derived superclass instances in clients, and raises the issue of behavioural compatibility of the corresponding class instances. In this paper we consider correctness of the reuse mechanism based on the combination of subclassing and type parameterization. We define the notion of refinement for generic classes by extending the definition of ordinary class refinement, and study special cases when refinement is guaranteed to hold between generically derived classes.

## 1 Introduction

Subclassing (or inheritance) is considered to be the reuse mechanism intrinsic to the object-oriented style of program construction. Parametric polymorphism is usually combined with object-orientation to enhance reuse for different types, and increase flexibility of class construction. Reuse for different types is achieved through parameterizing classes with unconstrained type parameters. Imposing constraints on generic type parameters, known as *bounded parametric polymorphism*, allows more flexibility in class construction when combined with subclassing. In particular, as was noted in [1], this combination can be used to circumvent some typing difficulties due to contravariance of method value parameters. Parameterized classes, which are usually referred to as *generic classes* or *templates*, in addition to bounded and unbounded type parameters can also be parameterized by values. Parameterization by values is used in practice to "fine-tune" reuse, e.g., to allow dynamic specification of list size. Substituting formal type and value parameters with concrete ones in generic classes produces *generically derived* classes.

The combination of parametric polymorphism and subclassing enables construction of new generic classes from existing ones by inheritance. The combination of parametric polymorphism and subtyping polymorphism allows polymorphic substitutability of generically derived subclass instances for generically derived superclass instances in clients, which makes reuse "seamless", but brings up the issue of behavioural compatibility of the corresponding class instances. In this paper we consider correctness of the reuse mechanism based on the combination of subclassing and type parameterization.

We formalize classes, subclassing, subtyping polymorphism, bounded and unbounded type parameterization, and value parameterization of classes in a logical framework known as the *refinement calculus* [5, 19]. Our formalization of object-oriented constructs is based on the simply typed lambda calculus; yet it is powerful enough to model subtyping polymorphism and dynamic binding. Refinement calculus is particularly suited for describing object-oriented programs because it allows us to describe classes at various abstraction levels, using *specification statements* along with ordinary executable statements. The notion of an *abstract class*, specifying behaviour common to its subclasses, can be fully elaborated in this formalization, since the state of class instances can be given using abstract mathematical constructions, like sets and sequences, and class methods can be described as nondeterministic statements, abstractly but precisely specifying the intended behaviour.

We give semantics of subclassing for generic classes, and study the conditions for correct generic subclassing based on the notion of class refinement originally defined in [18]. The results of this study reveal that relying on certain properties of generic classes, which may seem plausible, may lead to problems. Accordingly, we give a definition of generic class refinement, extending the definition of (ordinary) class refinement, to circumvent these problems and ensure that instances of a generically derived subclass will behave as expected from instances of the corresponding generically derived superclass. We also consider special cases in which class refinement is guaranteed to hold between generically derived classes.

## 2 Refinement Calculus Basics

We formalize objects, classes, and relationships between them in the refinement calculus, which is a logic framework for reasoning about correctness and refinement of imperative programs. Let us briefly introduce the main concepts of this formalism.

### 2.1 Predicates, Relations, and Predicate Transformers

A program state with components is modeled by a tuple of values, and a set of states (type) $\Sigma$ is a product space, $\Sigma = \Sigma_1 \times \ldots \times \Sigma_n$. A *predicate* over $\Sigma$ is a boolean function $p : \Sigma \to \mathit{Bool}$ which assigns a truth value to each state. The set of predicates on $\Sigma$ is denoted $\mathcal{P}\Sigma$. The *entailment ordering* on predicates is defined by pointwise extension, so that for $p, q : \mathcal{P}\Sigma$,

$$p \subseteq q \;\widehat{=}\; (\forall \sigma : \Sigma \cdot p\,\sigma \Rightarrow q\,\sigma)$$

A *relation* from $\Sigma$ to $\Gamma$ is a function of type $\Sigma \to \mathcal{P}\Gamma$ that maps each state $\sigma$ to a predicate on $\Gamma$. We write $\Sigma \leftrightarrow \Gamma$ to denote a set of all relations from $\Sigma$ to $\Gamma$. A *predicate transformer* is a function $S : \mathcal{P}\Gamma \to \mathcal{P}\Sigma$ from predicates to predicates. We write

$$\Sigma \mapsto \Gamma \;\widehat{=}\; \mathcal{P}\Gamma \to \mathcal{P}\Sigma$$

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

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

Product operators combine predicates, relations, and predicate transformers by forming cartesian products of their state spaces. For example, 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 $S_2$. For $S : \Sigma \mapsto \Sigma$ we define *lifting* to a product predicate transformer of type $\Sigma \times \Gamma \mapsto \Sigma \times \Gamma$ as $S \times \mathbf{skip}$. Similarly, lifting $S$ to a product predicate transformer of type $\Gamma \times \Sigma \mapsto \Gamma \times \Sigma$ is defined by $\mathbf{skip} \times S$.

For modeling subtyping polymorphism and dynamic binding we employ *sum types*. The sum or disjoint union of two types $\Sigma$ and $\Gamma$ is written $\Sigma + \Gamma$. The types $\Sigma$ and $\Gamma$ are called *base types* of the sum in this case. Associated with the sum types, are the *injection functions* $\iota_1 : \Sigma \to \Sigma + \Gamma$ and $\iota_2 : \Gamma \to \Sigma + \Gamma$, which map elements of the base type to elements of the summation, and the *projection relations* $\pi_1 : \Sigma + \Gamma \leftrightarrow \Sigma$ and $\pi_2 : \Sigma + \Gamma \leftrightarrow \Gamma$, which relate elements of the summation with elements of its base types. The projection is the inverse of the injection, so that $\pi_1^{-1} = |\,\iota_1\,|$, where $|\,\iota_1\,|$ is the injection function lifted to a relation. Since any element of $\Sigma + \Gamma$ comes either from $\Sigma$ or from $\Gamma$, but not both, the ranges of the injections *ran* $\iota_1$ and *ran* $\iota_2$ partition $\Sigma + \Gamma$.

The type $\Sigma$ is defined to be a subtype of $\Sigma'$, written $\Sigma <: \Sigma'$, if $\Sigma = \Sigma'$, or $\Sigma <: \Sigma'_i$, where $\Sigma' = \Sigma'_1 + \ldots + \Sigma'_n$. For example, $\Sigma <: \Sigma + \Sigma'$ and, of course, $\Sigma + \Sigma' <: \Sigma + \Sigma'$. The subtype relation is reflexive, transitive, and antisymmetric. For any $\Sigma$ and $\Sigma'$ such that $\Sigma <: \Sigma'$, we can construct the corresponding injection function $\iota_\Sigma : \Sigma \to \Sigma'$ and projection relation $\pi_\Sigma : \Sigma' \leftrightarrow \Sigma$ in a straightforward way.

### 2.2 Specification Language

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 $\Sigma$ and final state in $\Gamma$ 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 $p$. A statement need not have identical initial and final state spaces, though if it does, we write $S : \Xi(\Sigma)$ instead of $S : \Sigma \mapsto \Sigma$ for the corresponding predicate transformer. Following an established tradition, we identify statements with the monotonic predicate transformers that they determine.

A statement $S$ is said to be correct with respect to specification $(pre, post)$ if $pre \subseteq S\, post$. The refinement ordering on predicate transformers models the notion of total-correctness preserving program refinement. For statements $S$ and $T$, the relation $S \sqsubseteq T$ holds if and only if $T$ satisfies any specification satisfied by $S$.

The predicate transformer **abort** maps each postcondition to the identically false predicate *false*. We know nothing about how **abort** is executed and it is never guaranteed to terminate. Conjunction $\sqcap$ and disjunction $\sqcup$ 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 free and 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.

The specification language includes the *assertion* $\{b\}$ and the *assumption* $[b]$ statements, where $b$ is a predicate stating a condition on program variables. Both the assertion and the assumption behave as **skip** if $b$ is satisfied; otherwise, the assertion aborts, whereas the assumption behaves *miraculously*, i.e., is guaranteed to establish any postcondition. The conditional statement **if** $g$ **then** $S_1$ **else** $S_2$ **fi** is defined by the demonic choice of guarded alternatives $[g]; S_1 \sqcap [\neg g]; S_2$ or the angelic choice of asserted alternatives $\{g\}; S_1 \sqcup \{\neg g\}; S_2$.

Iteration **while** $g$ **do** $S$ **od** is defined as the least fixpoint of a function $(\lambda X \cdot$ **if** $g$ **then** $S; X$ **else skip fi**$)$ on predicate transformers with respect to refinement ordering. A variant of iteration, the *iterative choice* introduced in [9], allows the user to choose repeatedly an alternative that is enabled and have it executed until the user decides to stop:

$$\textbf{do } g_1 :: S_1 \lozenge \ldots \lozenge\ g_n :: S_n \textbf{ od} \ \widehat{=}\ (\mu X \cdot \{g_1\}; S_1; X \sqcup \ldots \sqcup \{g_n\}; S_n; X \sqcup \textbf{skip})$$

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

Following [8], we use the program variable notation as a simple syntactic extension to the typed lambda calculus. Let $(\lambda u \cdot t)$ be a function which replaces the old state $u$ with the new state $t$, changing some components $x_1, \ldots, x_m$ of $u$ to $t_1, \ldots, t_m$, while leaving the others unchanged. The (multiple) assignment statement may be modeled, using the program variable notation, by the *functional update* $\langle f \rangle$, which is a predicate transformer applying the function $f$ to the state $u$ to yield the new state $f\, u$:

$$(\textbf{var } u \cdot x_1, \ldots, x_m := t_1, \ldots, t_m) \ \widehat{=}\ \langle \lambda u \cdot u[x_1, \ldots, x_m := t_1, \ldots, t_m] \rangle$$

A relation $(\lambda u \cdot \lambda u' \cdot b)$ can also be written as $(\lambda u \cdot \{u' \mid b\})$, using the set notation. When such a relation changes a component $x$ of state $u$ to some $x'$ related to $x$ via a boolean expression $b$, this change can be expressed in the variable notation as

$$(\mathbf{var}\ u \cdot x := x' \mid b) \ \widehat{=}\ (\lambda u \cdot \{u[x := x'] \mid b\})$$

The *demonic assignment* $[\mathbf{var}\ u \cdot x := x' \mid b]$ and the *angelic assignment* $\{\mathbf{var}\ u \cdot x := x' \mid b\}$ are the specification statements modeled by the *demonic update* $[P]$ and the *angelic update* $\{P\}$, lifting the relation $P$ to the level of predicate transformers. When started in a state $u$, $[P]$ demonically chooses a new state $u'$ such that $P\,u\,u'$ holds, while $\{P\}$ angelically chooses a new state $u'$ such that $P\,u\,u'$ holds. If no such state exists, then $\{P\}$ aborts, whereas $[P]$ behaves miraculously. Intuitively, the demonic assignment expresses an uncontrollable nondeterministic choice in selecting a new value $x'$ satisfying $b$, whereas the angelic assignment expresses a free choice. The angelic assignment can, e.g., be understood as a request to the user to supply a new value. For example,

$$\{\mathbf{var}\ u \cdot x, e := x', e' \mid x' \geq 0 \wedge e > 0\}; [\mathbf{var}\ u \cdot x := x' \mid -e < x'^2 - x < e]$$

describes how the user gives as input a value $x$ whose square root is to be computed, as well as the precision $e$ with which the system is to compute this square root. The system then computes an approximation to the square root with precision $e$, choosing any new value for $x$ that satisfies this precision.

Finally, the language supports blocks with *local variables*. Entering the local variables and initializing them according to a certain predicate is modeled by a demonic update; removing these local variables is modeled by a functional update:

$$\begin{array}{ll} \mathbf{enter}\ p & \widehat{=}\ [\lambda u \cdot \lambda(x, u') \cdot p\,(x, u') \ \wedge\ u = u'] \\ \mathbf{exit} & \widehat{=}\ \langle \lambda(x, u) \cdot u \rangle \end{array}$$

Here $p$ is the initializing predicate which will normally be written in terms of program variables such as $p = (\mathbf{var}\ x, u \cdot b)$, with boolean expression $b$ relating the new variables $x$ to the old variables $u$. When the old variables $u$ are clear from the context, we will simplify the notation writing $\mathbf{var}\ x \mid b$ for $(\mathbf{var}\ x, u \cdot b)$. The block construct is then modeled as follows:

$$(\mathbf{var}\ u \cdot \mathbf{begin\ var}\ x \mid b \cdot S\ \mathbf{end}) \ \widehat{=}\ (\mathbf{var}\ u \cdot \mathbf{enter\ var}\ x \mid b; S; \mathbf{exit})$$

The program variable declaration can be propagated outside statements and distributed through sequential composition, so that, e.g.,

$$(\mathbf{var}\ u \cdot [x := x' \mid x' \geq 0]; y := x) \ =\ [\mathbf{var}\ u \cdot x := x' \mid x' \geq 0]; (\mathbf{var}\ u \cdot y := x)$$

When the variable declaration is clear from the context, we will omit it.

Data refinement is a general technique by which one can change the state space in a refinement. For statements $S : \Xi(\Sigma)$ and $S' : \Xi(\Sigma')$, let $R : \Sigma' \leftrightarrow \Sigma$ be a relation between the state spaces $\Sigma$ and $\Sigma'$. According to [6], the statement $S$ is said to be data refined by $S'$ via $R$, denoted $S \ \sqsubseteq_R\ S'$, if $\{R\}; S \ \sqsubseteq\ S'; \{R\}$ or, equivalently, $\{R\}; S; [R^{-1}] \ \sqsubseteq\ S'$. Further on we will abbreviate $\{R\}; S; [R^{-1}]$ by $S \downarrow R$.

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 [9, 19].

## 3 Modeling Object-Oriented Constructs

We focus on modeling class-based object-oriented languages, and, accordingly, take a view that *objects* are instantiated by *classes*.

### 3.1 Modeling Classes and Subclasses

A class is a template used to describe objects with identical behaviour through specifying their *interface*. The interface represents the signatures of object methods, i.e., the method name and the types of value and result parameters. We consider the *object type* to be the type of object *attributes*. Without loss of generality, we consider all attributes as private, i.e., hidden, and all methods as public, i.e., visible to clients of the object.

New classes can be constructed from existing ones by *inheriting* some or all of their attributes and methods, possibly *overriding* some attributes and methods, and adding extra methods. This mechanism is known as *subclassing* or *implementation inheritance*. We limit our consideration of class construction to inheritance and overriding; addition of extra methods constitutes the topic of current research.[1]

A class $C$ and a class $C'$ constructed from $C$ by subclassing can be given by the following declarations:

$$
\begin{array}{l}
C \;=\; \textbf{class} \\
\quad \textbf{var}\ attr_1 : \Sigma_1, \ldots, attr_m : \Sigma_m \\
\quad C(\textbf{val}\ x_0 : \Gamma_0) = K, \\
\quad Meth_1(\textbf{val}\ x_1 : \Gamma_1, \textbf{res}\ y_1 : \Delta_1) = M_1, \\
\quad \ldots \\
\quad Meth_n(\textbf{val}\ x_n : \Gamma_n, \textbf{res}\ y_n : \Delta_n) = M_n \\
\textbf{end}
\end{array}
$$

$$
\begin{array}{l}
C' \;=\; \textbf{subclass of}\ C \\
\quad \textbf{var}\ attr_1 : \Sigma_1, \ldots, attr_i : \Sigma_i, \\
\qquad\ attr'_1 : \Sigma'_1, \ldots, attr'_p : \Sigma'_p \\
\quad C'(\textbf{val}\ x'_0 : \Gamma'_0) = K', \\
\quad Meth_1(\textbf{val}\ x_1 : \Gamma_1, \textbf{res}\ y_1 : \Delta_1) = M'_1, \\
\quad \ldots \\
\quad Meth_k(\textbf{val}\ x_k : \Gamma_k, \textbf{res}\ y_k : \Delta_k) = M'_k \\
\textbf{end}
\end{array}
$$

Both classes specify the interface $Meth_1\,(\textbf{val} : \Gamma_1, \textbf{res} : \Delta_1), \ldots, Meth_n\,(\textbf{val} : \Gamma_n, \textbf{res} : \Delta_n)$, where $\Gamma_i$ and $\Delta_i$ are the types of value and result parameters respectively. A method may be parameterless, with both $\Gamma_i$ and $\Delta_i$ being the unit type (), or have only value or only result parameters. We view subclassing as a syntactic relation on classes, since subclasses are distinguished by an appropriate declaration. Syntactic subclassing implies conformance of interfaces, in the sense that a subclass specifies an interface conforming to the one specified by its superclass. In a simple case the interface specified by a subclass is the same as

[1] Addition of extra methods is a non-trivial issue because of inconsistencies possibly introduced by extra methods in presence of subtype aliasing.

that of the superclass. In the next subsection we explain how this requirement can be relaxed.

The class $C$ describes (possibly abstract) attributes, specifies the way the objects are created, and gives a (possibly nondeterministic) specification for each method. Class attributes $(attr_1, \ldots, attr_m)$ have the corresponding types $\Sigma_1$ through $\Sigma_m$. We will use an identifier *self* for the tuple $(attr_1, \ldots, attr_m)$. The type of *self* is then $\Sigma = \Sigma_1 \times \ldots \times \Sigma_m$.[2] A subclass may have attributes different from those of its superclass, inheriting $attr_1, \ldots, attr_i$ and overriding $attr_{i+1}, \ldots, attr_m$ by $attr'_1, \ldots, attr'_p$.

A class *constructor* is used to instantiate objects and has the same name as the class. Due to the fact that the constructor concerns object creation rather than object functionality, it is associated with the class rather than with the specified interface. The statement $K : \Gamma_0 \mapsto \Sigma \times \Gamma_0$, representing the body of the constructor, introduces the attributes into the state space and initializes them using the value parameters $x_0 : \Gamma_0$. Methods $Meth_1$ through $Meth_n$, specified by bodies $M_1, \ldots, M_n$, operate on the attributes and realize the object functionality. Every statement $M_i$ is, in general, of type $\Xi(\Sigma \times \Gamma_i \times \Delta_i)$. The identifier *self* acts in this model as an implicit result parameter of the constructor and an implicit variable parameter of the methods.

The constructor of a subclass is usually redefined without inheriting the superclass constructor. The value parameters $x'_0 : \Gamma'_0$ of the subclass can be different from the value parameters $x_0 : \Gamma_0$ of the superclass. The statements $M'_1, \ldots, M'_k$ override the corresponding definitions of $Meth_1, \ldots, Meth_k$ given in $C$. Methods defined in $C$ and operating only on the attributes inherited by $C'$ can be invoked from $M'_1, \ldots, M'_k$ using a special identifier *super*. Methods of the superclass operating only on the inherited attributes can also be inherited as a whole. In this case their redefinition in the subclass corresponds to super-calling them, passing value and result parameters as arguments. Following the standard convention, we omit such inherited methods from the subclass declaration.

Declared as above, the classes $C$ and $C'$ are the tuples $(K, M_1, \ldots, M_n)$ and $(K', M'_1, \ldots, M'_n)$. The object type specified by a class can always be extracted from the class and we do not need to declare it explicitly. We use $\tau(C)$ to denote the type of objects generated by the class $C$. As such, $\tau(C)$ is just another name for $\Sigma$.

As an example of specifying classes and their subclasses consider specifications of *Bag* and *CountingBag* presented in Fig.1. The subclass *CountingBag* inherits the only attribute of its superclass *Bag*, representing a bag of characters, and adds a counter of bag elements. The constructor of *Bag* initializes the attribute $b$ to the empty bag, and the constructor of *CountingBag* in addition initializes the attribute $n$ to zero. The method *Add* of *CountingBag* overrides the corresponding method of the superclass by incrementing the counter and then super-calling *Add* of *Bag*. The method *AddAll* joins two bags by self-calling

[2] We impose a non-recursiveness restriction on $\Sigma$ so that none of $\Sigma_i$ is equal to $\Sigma$. This restriction allows us to stay within the simple-typed lambda calculus.

```
Bag = class
  var b : bag of Char
  Bag() = enter var b | b = ⟦⟧,
  Add(val c : Char) = b := b ∪ ⟦c⟧,
  AddAll(val nb : bag of Char) =
    while nb ≠ ⟦⟧ do
      begin var c | c ∈ nb ·
        self.Add(c); nb := nb − ⟦c⟧
      end
    od;
end
```

```
CountingBag = subclass of Bag
  var b : bag of Char, n : Nat
  CountingBag() =
    enter var b, n | b = ⟦⟧ ∧ n = 0,
  Add(val c : Char) =
    n := n + 1; super.Add(c)
end
```

**Fig. 1.** Specification of *Bag* and its subclass *CountingBag*

*Add*. *CountingBag* inherits the method *AddAll* from *Bag* which corresponds to super-calling it.

A new variable $c$ of object type $\tau(C)$ is initialized by invoking the corresponding class constructor:

$$\begin{aligned}\textbf{create var } c.C(e) \;\widehat{=}\; & \textbf{enter } (\textbf{var } x_0, u \cdot x_0 = e); K \times \textbf{skip};\\ & \textbf{enter } (\textbf{var } c, (\mathit{self}, x_0), u \cdot c = \mathit{self}); \mathit{Swap}; \textbf{exit}\end{aligned}$$

where $\mathit{Swap} = \langle \lambda x, y, z \cdot y, x, z \rangle$. A variable $x_0 : \Gamma_0$ is first entered into the state space and initialized with the value of $e$. Then the constructor $K$ is "injected" into the global state space, skipping on the global state component $u$. The next statement enters variable $c$ and initializes it to the value of the state component *self*. The state rearranging *Swap* makes the pair $(\mathit{self}, x_0)$ the first state component before exiting it from the block. Naturally, a variable of object type initialized in this way can be local to a block:

$$\textbf{create var } c.C(e) \cdot S \textbf{ end} \;\widehat{=}\; \textbf{create var } c.C(e); S; \textbf{exit}$$

Invocation of a method $Meth_i(\textbf{val } x_i : \Gamma_i, \textbf{res } y_i : \Delta_i)$ on an object $c$ instantiated by class $C$ is modeled as follows:

$$\begin{aligned}(\textbf{var } c, u \cdot c.Meth_i\,(g_i, d_i)) \;\widehat{=}\; (\textbf{var } c, u \cdot & \textbf{ begin var } \mathit{self}, x_i, y_i \mid \mathit{self} = c \wedge x_i = g_i \cdot\\ & M_i \times \textbf{skip}; c, d_i := \mathit{self}, y_i\\ & \textbf{end})\end{aligned}$$

where $u : \Phi$ are global variables, including $d_i : \Delta_i$, and $g_i : \Gamma_i$ is some expression.

### 3.2 Modeling Subtyping Polymorphism and Dynamic Binding

We model subtyping polymorphism as in [18, 7], permitting object types to be sum types. Essentially, an object type of a certain class and object types of all its subclasses are grouped together to form a polymorphic object type. A variable of such a sum type can be instantiated to any base type of the summation, in

other words, to any object instantiated by a class whose object type is the base type of the summation.

A sum of object types, denoted by $\tau(C)^+$ is defined to be such that its base types are $\tau(C)$ and all the object types of subclasses of $C$. For example, if $D$ is the only subclass of $C$ with the object type $\tau(D)$, then $\tau(C)^+ = \tau(C) + \tau(D)$, and we have that $\tau(C) <: \tau(C)^+$ and $\tau(D) <: \tau(C)^+$.

Suppose method $Meth_i\,(\mathbf{val}\ x_i : \Gamma_i, \mathbf{res}\ y_i : \Delta_i)$ is specified in $C$ and $D$ by statements $M_i$ and $M_i'$ respectively. An invocation of this method on an object $p$ of type $\tau(C)^+$ is modeled as a choice between two alternatives each calling $Meth_i$, but one assuming that $p$ is instantiated by class $C$ and the other assuming instantiation by class $D$:

$$p.Meth_i(g_i, d_i) \ \widehat{=}\ \begin{pmatrix} \{p \in ran\ \iota_{\tau(C)}\}; \\ \mathbf{begin\ var}\ c \,|\, \pi_{\tau(C)}\, p\, c \cdot \\ \quad c.Meth_i(g_i, d_i); \\ \quad p := \iota_{\tau(C)}\, c \\ \mathbf{end} \end{pmatrix} \sqcup \begin{pmatrix} \{p \in ran\ \iota_{\tau(D)}\}; \\ \mathbf{begin\ var}\ d \,|\, \pi_{\tau(D)}\, p\, d \cdot \\ \quad d.Meth_i(g_i, d_i); \\ \quad p := \iota_{\tau(D)}\, d \\ \mathbf{end} \end{pmatrix}$$

When $p$ is an instance of $C$, the assertion $\{p \in ran\ \iota_{\tau(C)}\}$ skips, and the method $Meth_i$ is invoked on the object $c$ corresponding to the projection $\pi_{\tau(C)}$ of $p$. Afterwards, the value of $c$ is injected to be of type $\tau(C)^+$ and used to update $p$. The invocation $c.Meth_i(g_i, d_i)$ is modeled as above. The assertion that $p$ is an instance of $D$ is false, aborting the second alternative of the angelic choice, since for all predicates $q$, $\{q\} = \mathbf{if}\ q\ \mathbf{then\ skip\ else\ abort\ fi}$, and for all statements $S$, $\mathbf{abort}; S = \mathbf{abort}$. The angelic choice between the two statements is then equal to the first alternative, since $S \sqcup \mathbf{abort} = S$. Similarly, when $p$ is an instance of $D$, the first alternative aborts, and the choice is equal to the second alternative. As such, the choice between the alternatives is deterministic.

A polymorphic variable $p : \tau(C)^+$ can be instantiated by either class $C$ or its subclass. In practice we occasionally would like to underspecify which particular class instantiates $p$. We can express this by using a demonic choice of possible instantiations. We write $p.C^+(e)$, where $e$ is any expression of type $\Gamma_0$, for this kind of polymorphic instantiation:

$$\mathbf{create\ var}\ p.C^+(e) \ \widehat{=}\ \begin{pmatrix} \mathbf{create\ var}\ c.C(e); \\ \mathbf{enter\ var}\ p \,|\, \pi_{\tau(C)}\, p\, c; \\ \quad Swap; \\ \mathbf{exit} \end{pmatrix} \sqcap \begin{pmatrix} \mathbf{create\ var}\ d.D(e); \\ \mathbf{enter\ var}\ p \,|\, \pi_{\tau(D)}\, p\, d; \\ \quad Swap; \\ \mathbf{exit} \end{pmatrix}$$

The type of constructor value parameter in the subclass must be the same as that in the superclass for the polymorphic instantiation to make sense. Intuitively, the demonic choice can be interpreted as underspecification, which would need to be eliminated in a refinement. Note, that since the demonic choice is refined by either alternative, we have that any concrete instantiation refines the polymorphic instantiation.

Modeling of both the invocation of a method on a polymorphic variable and the instantiation of such a variable generalizes to class hierarchies with several classes in a straightforward way, recursively.

Being equipped with subtyping polymorphism, we can allow overriding methods in a subclass to be *generalized* on the type of value parameters or *specialized* on the type of result parameters. In the first case this type redefinition is *contravariant* and in the second *covariant*.[3] When one interface is the same as the other, except that it can redefine contravariantly value parameter types and covariantly result parameter types, this interface conforms to the original one. For example, $Meth_i\,(\mathbf{val}\ x_i : \Gamma_i, \mathbf{res}\ y_i : \Delta_i)$ specified in class $C$ could be redefined in its subclass $D$ so that the value parameters are of type $\Gamma_i'$, such that $\Gamma_i <: \Gamma_i'$, and the result parameters are of type $\Delta_i'$, such that $\Delta_i' <: \Delta_i$. An invocation of such a method would then need to adjust the input arguments and the result using the corresponding projections and injections. For example, invocation of $Meth_i\,(\mathbf{val}\ x_i' : \Gamma_i', \mathbf{res}\ y_i' : \Delta_i')$ specified by $M_i'$ in class $D$ on an object $d : \tau(D)$ with input argument $g_i : \Gamma_i$ and result argument $d_i : \Delta_i$ is modeled as follows:

$$
\begin{array}{rcl}
d.Meth_i\,(g_i, d_i) & \widehat{=} & \mathbf{begin\ var}\ \mathit{self}, x_i', y_i' \mid \mathit{self} = d \wedge x_i' = \iota_{\Gamma_i}\, g_i \cdot \\
& & \qquad M_i' \times \mathbf{skip}; d, d_i := \mathit{self}, \iota_{\Delta_i'}\, y_i' \\
& & \mathbf{end}
\end{array}
$$

The value parameter $x_i'$ is initialized with the value of the input argument injected into the type $\Gamma_i'$. Similarly, the value of the method result $y_i'$, being of type $\Delta_i'$, cannot be directly assigned to the variable $d_i : \Delta_i$ and is injected into the type $\Delta_i$ using the corresponding injection function.

Dynamic binding of self-referential methods takes place only when a subclass inherits all attributes of its superclass without overriding them. Essentially, a super-call to a method self-calling other methods of the same class resolves the latter with the definitions of the self-called methods in the class which originated the super-call. For example, in the specification of *Bag* and *CountingBag*, the body of the method *AddAll* in *CountingBag*, which implicitly super-calls the corresponding method in *Bag*, is equal to the following statement:

$$
\begin{array}{l}
\mathbf{while}\ nb \neq [\![]\!]\ \mathbf{do} \\
\quad \mathbf{begin\ var}\ c \mid c \in nb \cdot n := n + 1; b := b \cup [\![c]\!]; nb := nb - [\![c]\!]\ \mathbf{end} \\
\mathbf{od}
\end{array}
$$

Formal treatment of dynamic binding in presence of self-referential methods is presented in [7].

## 4 Modeling Parametric Polymorphism

Type parameterization is a general technique for reusing code with different types. As we have mentioned in the introduction, parametric polymorphism is usually combined with object-orientation to enhance reuse for different types and increase flexibility of class construction. In our formalization we model *unbounded type parameterization* and then specialize it to *bounded type parameterization*. Furthermore, we consider parameterization by values which is used in practice to "fine-tune" reuse.

[3] For a more extensive explanation of covariance and contravariance see, e.g., [1].

```
List [Elem] = class
    var elems : seq of Elem

    List [Elem] () = enter var elems | elems = ⟨⟩,

    Empty (res r : Bool) = r := (elems = ⟨⟩),

    Nth (val n : Nat, res e : Elem) = [1 ≤ n ≤ #elems]; e := elems n,

    NumOfElems (res n : Nat) = n := #elems,

    AppendElement (val e : Elem) = elems := elems ⌢ ⟨e⟩,

    Equal (val lst : τ(List[Elem]), res r : Bool) =
        begin var n, i, e | i = 1 · lst.NumOfElems(n); r := (#elems = n);
            while r ∧ i ≤ n do lst.Nth(i, e); r := (elems i = e) od
        end
end
```

**Fig. 2.** Specification of a generic class *List*

Let us first present our model of unbounded type parameterization. Consider class declarations $C_1$ = **class** ... $\Theta_1$ ... **end** and $C_2$ = **class** ... $\Theta_2$ ... **end** which are the same except that the object type named $\Theta_1$ occurs in $C_1$ wherever the object type named $\Theta_2$ occurs in $C_2$. Substituting $\Theta_2$ for $\Theta_1$ in $C_1$ yields $C_2$, i.e., $C_1[\Theta_1 \backslash \Theta_2] = C_2$ and, vice versa, $C_2[\Theta_2 \backslash \Theta_1] = C_1$. We will use a construct $C[T]$ to stand for a collection of all classes $C[\Theta]$ obtained by substituting the type named $\Theta$ for the type parameter $T$. We shall call $C[T]$ a generic class[4] and declare it as follows:

```
C[T] = class
    var self : Σ

    C (val x_0 : Γ_0) = K,
    Meth_1 (val x_1 : Γ_1, res y_1 : Δ_1) = M_1,
    ...
    Meth_n (val x_n : Γ_n, res y_n : Δ_n) = M_n
end
```

Note, that the type parameter $T$ can occur in any of $\Sigma, \Gamma_i$ and $\Delta_i$. However, the type $\tau(C[T])$ cannot occur in $\Sigma$ because of our non-recursiveness requirement. Every class obtained by instantiating some object type $\Theta$ for $T$ specifies objects of type $\tau(C[\Theta])$, and is called a generically derived class, following [17]. Specification of a generic class *List* in Fig.2 is an example of unbounded type parameterization. In this specification the local state of a list is expressed as a sequence of elements, and the behaviour of methods is expressed in terms of operations on sequences. The method returning the $n$'th list element assumes that there are at least $n$ elements in the list, and the client using *Nth* should assert that this condition holds before the invocation. Note how the method *Equal*

[4] In some programming languages, e.g., C++, a generic class is referred to as a *template*.

compares two list instances. Due to the encapsulation assumption, we cannot access the attributes of the value parameter *lst* directly, and, instead, describe the equality comparison in terms of the corresponding method invocations. This example clearly illustrates the advantages of our approach to modeling classes as compared to the approaches based on pre- and post-condition specifications, because this specification describes abstractly (in terms of sequences), but still precisely (in terms of method invocations) the behaviour of polymorphic list objects.

To model bounded type parameterization, we use a similar construct, $C[T <: \Theta]$, which is defined by $\{C[\Theta_i] \,|\, \Theta_i <: \Theta\}$ and corresponds to a collection of classes $C[\Theta_i]$ with the object type $\Theta_i$ being a subtype of $\Theta$. The type $\Theta$ is called the upper bound on the type parameter $T$.

If one of the attributes in a generic class $C[T <: \Theta]$ is of type $T$, the constructor of this class can only initialize this attribute to an existing value of type $T$, because it has no way of knowing which class constructor to invoke for creation of this attribute. Therefore, the initialization of such an attribute can only be done through a constructor value parameter of type $T$.

New generic classes can be constructed from existing ones by subclassing. A declaration $C'[T] = \textbf{subclass of } C[T]$ represents a collection of all class declarations $C'[\Theta] = \textbf{subclass of } C[\Theta]$, obtained by substituting the type named $\Theta$ for the type parameter $T$. Similarly, a declaration $C'[T <: \Theta] = \textbf{subclass of } C[T <: \Theta]$ stands for a collection of class declarations $C'[\Theta_i] = \textbf{subclass of } C[\Theta_i]$, for $i = 1..n$, with the object types $\Theta_1, \ldots, \Theta_n$ being subtypes of $\Theta$. The upper bound in the subclassed generic class can be a subtype of the upper bound in the declaration of the original generic class.

As an example of generic subclassing suppose that *Elem* is a polymorphic object type of elements, and *IndexedElem* is a polymorphic object type of indexed elements, such that *IndexedElem* <: *Elem*. Suppose that we would like to describe indexed lists that can only contain indexed elements. We can subclass a generic class *IndexedList* [*E* <: *IndexedElem*] from a generic class *List* [*E* <: *Elem*] as follows:

| | |
|---|---|
| *List* [*E* <: *Elem*] = **class** | *IndexedList* [*E* <: *IndexedElem*] = |
| | **subclass of** *List* [*E* <: *IndexedElem*] |
| ... | ... |
| *Nth* (**val** $n$ : *Nat*, **res** $e$ : $E$), | *Nth* (**val** $n$ : *Nat*, **res** $e$ : $E$), |
| *AppendElement* (**val** $e$ : $E$), | *AppendElement* (**val** $e$ : $E$), |
| *Equal* (**val** $l$ : $\tau$(*List* [$E$]), **res** $r$ : *Bool*) | *Equal* (**val** $l$ : $\tau$(*List* [$E$])$^+$, **res** $r$ : *Bool*) |
| **end** | **end** |

Here, the possible $\tau$(*IndexedList* [*IndexedElem*]) is a valid object type, whereas the type of indexed lists of possibly non-indexed elements $\tau$(*IndexedList* [*Elem*]) is not well-formed. Naturally, we have that

$$\text{for all } E <: \mathit{IndexedElem},\ \tau(\mathit{IndexedList}\,[E])^+ <:\ \tau(\mathit{List}\,[E])^+$$

This relation is useful for subtype aliasing: it asserts that all indexed lists are the lists with indexed elements.

Finally, let us consider parameterization of classes with values. We will use a construct $C[x : \Phi]$ to stand for a collection of all classes $C[t : \Phi]$ obtained by substituting the value named $t$ for the parameter named $x$, both of type $\Phi$.

Value parameterization as well as bounded and unbounded type parameterization generalize in a straightforward way to parameterization by lists of value parameters and bounded and unbounded type parameters. As an example consider parameterizing the class of lists with the maximal number of elements it can contain. We can subclass a generic class $BoundedList\ [E <: Elem, max : Nat]$ from a class $List\ [E <: Elem]$ as follows:

$BoundedList\,[E <: Elem, max : Nat]\ =\ \textbf{subclass of}\ List\,[E <: Elem]$
  **var** $elems$ : **seq of** $E$

  $BoundedList\,[E, max]\,()\ =\ \textbf{enter var}\ elems \mid elems = \langle\rangle,$

  $AppendElement(\textbf{val}\ e : E)\ =\ [\#elems < max]; super.AppendElement(e)$
**end**

Here, the method *AppendElement* checks that the current number of elements in the list is less than the maximum, and under this assumption adds a new element. The other methods are inherited from the superclass. Using this generic class, we can now create, for example, bounded lists of indexed elements of sizes 5 and 10, specifying the size dynamically:

**create var** $lst1.BoundedList\,[IndexedElem, 5]();$
**create var** $lst2.BoundedList\,[IndexedElem, 10]();$

## 5 Semantics of Correct Polymorphic Reuse

When a subclass overrides some methods of its superclass, there are no guarantees that its instances will deliver the same or refined behaviour as the instances of the superclass. Unrestricted method overriding in a subclass can lead to arbitrary behaviour of its instances. When used in a superclass context, such subclass instances can invalidate their clients. To avoid such problems, we would like to ensure that whenever $C'$ is subclassed from $C$, clients using objects instantiated by $C$ can safely use objects instantiated by $C'$ instead.

Moreover, classes derived from generic classes can specify conforming interfaces, and, therefore, allow syntactic substitutability of the corresponding instances, provided that a suitable substitution mechanism is available in a programming language. Then the semantic relationship between such classes is essential for ensuring correct behaviour of the "subclass" instances with respect to the "superclass" instances. We consider this semantic relationship here as well as class refinement for generic classes.

Before considering generic class refinement, let us briefly explain the notion of ordinary class refinement introduced in [18] and further developed in [7].

### 5.1 Ordinary Class Refinement

Let classes $C$ and $C'$ be modeled by tuples $(K, M_1, \ldots, M_n)$ and $(K', M'_1, \ldots, M'_n)$, where $K : \Gamma_0 \mapsto \Sigma \times \Gamma_0$ and $K' : \Gamma'_0 \mapsto \Sigma' \times \Gamma'_0$ are the class

constructors, and all $M_i : \Xi(\Sigma \times \Gamma_i \times \Delta_i)$ and $M'_i : \Xi(\Sigma' \times \Gamma'_i \times \Delta'_i)$ are the corresponding methods. The value parameter types of the methods are either the same or contravariant, $\Gamma_i <: \Gamma'_i$, and the result parameter types of the methods are either the same or covariant, $\Delta'_i <: \Delta_i$.

The refinement of class constructors $K$ and $K'$ with respect to relations $Q : \Gamma'_0 \leftrightarrow \Gamma_0$ and $R : \Sigma' \leftrightarrow \Sigma$, coercing value parameter types and attribute types of $C'$ to those of $C$, is defined by

$$K \sqsubseteq_{Q,R} K' \quad \widehat{=} \quad \{Q\}; K \quad \sqsubseteq \quad K'; \{R \times Q\} \qquad (\textit{constructor refinement})$$

The refinement of all corresponding methods $M_i$ and $M'_i$ with respect to the relation $R$ is defined by

$$M_i \sqsubseteq_R M'_i \quad \widehat{=} \quad M_i \downarrow (R \times \pi_{\Gamma_i} \times |\iota_{\Delta'_i}|) \quad \sqsubseteq \quad M'_i \qquad (\textit{method refinement})$$

where $\pi_{\Gamma_i} : \Gamma'_i \leftrightarrow \Gamma_i$ projects the corresponding value parameters, and $|\iota_{\Delta'_i}| : \Delta'_i \leftrightarrow \Delta_i$ injects the corresponding result parameters. Obviously, when $\Gamma_i = \Gamma'_i$, the projection relation $\pi_{\Gamma_i}$ is the identity relation *Id*. The same holds when $\Delta_i = \Delta'_i$, namely, $|\iota_{\Delta'_i}| = Id$.

**Definition 1.** *For classes* $C = (K, M_1, \ldots, M_n)$ *and* $C' = (K', M'_1, \ldots, M'_n)$ *refinement* $C \sqsubseteq C'$ *is defined as follows:*

$$C \sqsubseteq C' \quad \widehat{=} \quad \exists Q, R \cdot K \sqsubseteq_{Q,R} K' \wedge (\forall i \mid 1 \leq i \leq n \cdot M_i \sqsubseteq_R M'_i)$$

As was proved in [18], the class refinement relation is reflexive and transitive.

Declaring one class as a subclass of another raises the proof obligation that the class refinement relation holds between these classes. This is, in a way, a semantic constraint that we impose on subclassing to ensure that behaviour of subclasses conforms to the behaviour of their superclasses and that subclass instances can be substituted for superclass instances in all clients.

As it was already mentioned, any concrete instantiation of a polymorphic variable refines the polymorphic instantiation. It is also interesting to examine the relationship between method calls of the same method on a variable of some object type and a variable of its subtype. The following two lemmas, proved in [7], state the complementing results about such method invocations. The first lemma states that invoking a method on a variable of a polymorphic object type refines invocation of the same method on a variable of its subtype with respect to the corresponding projection. This property may at first seem counterintuitive, since a client using instances of some subclass of $C$ in a refinement may use instances of $C$ as well. The intuitive justification behind this property is that a refined client can *do more* than the original client by being able to work with superclass instances along with subclass instances, while the original one worked with subclass instances only.

**Lemma 1.** *Assume that* $Meth_i$ *is specified in class* $C$ *and all its subclasses. Let* $\tau(C)^+$ *be a sum type of* $\tau(C)$ *and* $\tau(C')^+$*, variables* $c$ *and* $c'$ *be of types* $\tau(C)^+$ *and* $\tau(C')^+$ *respectively, a global state* $u : \Phi$ *include a variable* $d_i : \Delta_i$*, and* $g_i : \Gamma_i$ *be some expression. Then for the projection* $\pi_{\tau(C')^+} : \tau(C)^+ \leftrightarrow \tau(C')^+$*,*

$$(\mathbf{var}\ c', u \cdot c'.Meth_i(g_i, d_i))\downarrow(\pi_{\tau(C')^+} \times Id) \quad \sqsubseteq \quad (\mathbf{var}\ c, u \cdot c.Meth_i(g_i, d_i))$$

The second lemma proves a dual result, namely, that invoking a method on a variable of a polymorphic object type is refined by invocation of the same method on a variable of its subtype with respect to the corresponding injection relation.

**Lemma 2.** *Assume that $Meth_i$ is specified in class $C$ and all its subclasses. Let $\tau(C)^+$ be a sum type of $\tau(C)$ and $\tau(C')^+$, variables $c$ and $c'$ be of types $\tau(C)^+$ and $\tau(C')^+$ respectively, a global state $u : \Phi$ include a variable $d_i : \Delta_i$, and $g_i : \Gamma_i$ be some expression. Then for the injection $\iota_{\tau(C')^+} : \tau(C')^+ \to \tau(C)^+$,*

$$(\mathbf{var}\ c, u \cdot c.Meth_i(g_i, d_i))\downarrow(|\iota_{\tau(C')^+}| \times Id) \quad \sqsubseteq \quad (\mathbf{var}\ c', u \cdot c'.Meth_i(g_i, d_i))$$

### 5.2 Class Refinement for Generic Classes

Generic classes describe classes of a very similar kind. Naturally, we would like to study the relationships among these classes and analyze whether we can benefit from this similarity. First we look at classes derived from one generic class, and then consider the relationship among generic classes constructed from other generic classes by subclassing, and the classes generically derived from them.

Consider a generic class $C[T <: \Theta]$ and two classes resulting from instantiation of some subtypes of $\Theta$ for $T$, namely $C[\Theta_1]$ and $C[\Theta_2]$. Syntactically, $C[\Theta_2]$ can be a "subclass" of $C[\Theta_1]$, in the sense that the interface specified by $C[\Theta_2]$ conforms to the interface specified by $C[\Theta_1]$. In this case objects instantiated by $C[\Theta_2]$ can be substituted for objects instantiated by $C[\Theta_1]$ without type errors. The interface of $C[\Theta_2]$ conforms to the interface of $C[\Theta_1]$, making $C[\Theta_2]$ a valid syntactic subclass of $C[\Theta_1]$, if $T$ occurs only in contravariant or only in covariant positions. When $T$ occurs only in contravariant positions, $\Theta_1$ must be a subtype of $\Theta_2$. Correspondingly, when $T$ occurs only in covariant positions, $\Theta_2$ must be a subtype of $\Theta_1$. The intuition behind these requirements is quite straightforward. However, this only gives us a syntactic interface conformance between $C[\Theta_1]$ and $C[\Theta_2]$. The following theorems state the conditions under which the class $C[\Theta_1]$ is refined by the class $C[\Theta_2]$.

**Theorem 1.** *Let $T$ occur only in contravariant positions in $C[T <: \Theta]$, or occur in constructor value parameter type, or be a component of its attribute type. Then for any class $B$ and its subclass $B'$ such that $\tau(B')^+ <: \tau(B)^+ <: \Theta$,*

$$C[\tau(B')^+] \sqsubseteq C[\tau(B)^+]$$

*Proof.* Let $C[\tau(B')^+] = (K, M_1, \ldots, M_n)$ and $C[\tau(B)^+] = (K', M'_1, \ldots, M'_n)$. Without loss of generality we assume that attributes $c$ of $C[\tau(B')^+]$ are of type $\tau(B')^+$ and attributes $c'$ of $C[\tau(B)^+]$ are of type $\tau(B)^+$. We further assume that the types of constructor and method value parameters in classes $C[\tau(B')^+]$ and $C[\tau(B)^+]$ are also $\tau(B')^+$ and $\tau(B)^+$ respectively. Proving class refinement $C[\tau(B')^+] \sqsubseteq C[\tau(B)^+]$ thus amounts to proving that the constructors are in refinement with respect to the projection relation $\pi_{B'} : \tau(B)^+ \leftrightarrow \tau(B')^+$, i.e.,

$K \sqsubseteq_{\pi_{B'},\pi_{B'}} K'$, and all corresponding methods are in refinement with respect to the same relation, i.e., $M_i \sqsubseteq_{\pi_{B'}} M'_i$, for $i = 1..n$.

We prove the desired property by modeling the constructors and the methods as clients [7] of the attributes and the value parameters, invoking methods defined in $B$ and $B'$. Namely, a method working with an attribute $a : \tau(B)^+$ and a value parameter $x : \tau(B)^+$ may be modeled as a program iteratively invoking methods $Bmeth_1, \ldots, Bmeth_k$ specified by classes $B$ and $B'$:

$$\textbf{begin var } l \,|\, p \cdot \textbf{do } \Diamond_{i=1}^{k} q_i :: (a.Bmeth_i(p_i, r_i) \sqcup x.Bmeth_i(p'_i, r'_i)); L_i \textbf{ od end}$$

In this expression, the local variables $l$ are initialized according to the boolean expression $p$; predicates $q_1, \ldots, q_k$ and statements $L_1, \ldots, L_k$ are arbitrary and operate on $l, p_i, r_i, p'_i, r'_i$, and global variables except $a$ and $x$. This client program may choose to invoke a method $Bmeth_i$ on either the attribute or the value parameter, then do $L_i$, and then iterate the choice until it decides to stop. Denoting the iterative choice

$$\textbf{do } \Diamond_{i=1}^{k} q_i :: (a.Bmeth_i(p_i, r_i) \sqcup x.Bmeth_i(p'_i, r'_i)); L_i \textbf{ od}$$

by $\mathcal{M}[a, x]$, we can model the methods $M_i$ and $M'_i$ of $C[\tau(B')^+]$ and $C[\tau(B)^+]$ by

$$\textbf{begin var } l \,|\, p \cdot \mathcal{M}[c, x_i] \textbf{ end} \text{ and } \textbf{begin var } l \,|\, p \cdot \mathcal{M}[c', x'_i] \textbf{ end}$$

respectively, where $x_i : \tau(B')^+$ and $x'_i : \tau(B)^+$ are the corresponding value parameters and $y_i : \Delta_i$ are the result parameters.

The constructors $K$ and $K'$ are modeled as statements which enter the corresponding attributes into the state space, initialize them using the value parameter, and then proceed by invoking methods on either the attribute or the value parameter, and changing some local variables. Thus, $K$ and $K'$ are modeled respectively by

$$\textbf{enter var } c \,|\, c = x_0 \cdot \textbf{begin var } l \,|\, p \cdot \mathcal{M}[c, x_0] \textbf{ end} \text{ and}$$
$$\textbf{enter var } c' \,|\, c' = x'_0 \cdot \textbf{begin var } l \,|\, p \cdot \mathcal{M}[c', x'_0] \textbf{ end}$$

where $x_0 : \tau(B')^+$ and $x'_0 : \tau(B)^+$ are the corresponding constructor value parameters. Now the refinement between the corresponding constructors

$$\{\pi_{B'}\}; \textbf{enter var } c \,|\, c = x_0 \cdot \textbf{begin var } l \,|\, p \cdot \mathcal{M}[c, x_0] \textbf{ end} \sqsubseteq$$
$$\textbf{enter var } c' \,|\, c' = x'_0 \cdot \textbf{begin var } l \,|\, p \cdot \mathcal{M}[c', x'_0] \textbf{ end}; \{\pi_{B'} \times \pi_{B'}\}$$

and the refinement between the corresponding methods

$$\textbf{begin var } l \,|\, p \cdot \mathcal{M}[c, x_i] \textbf{ end} \downarrow (\pi_{B'} \times \pi_{B'} \times Id) \sqsubseteq \textbf{ begin var } l \,|\, p \cdot \mathcal{M}[c', x'_i] \textbf{ end}$$

are proved by rewriting with definitions and using refinement laws as given in [9] and Lemma 1. □

The following theorem states a dual and more interesting from a practical point of view result.

**Theorem 2.** *Let $T$ occur only in covariant positions in $C[T <: \Theta]$, or occur in constructor value parameter type, or be a component of its attribute type. Then for any class $B$ and its subclass $B'$ such that $\tau(B')^+ <: \tau(B)^+ <: \Theta$,*

$$C[\tau(B)^+] \sqsubseteq C[\tau(B')^+]$$

The proof of this theorem is similar to the previous one and we omit it for the lack of space. Intuitively, the theorem states that a class derived from a generic class by substituting a *more concrete* object type for the type parameter is a refinement.

Consider now class refinement for different generic classes. Suppose that $C'[T <: \Theta']$ is constructed from $C[T <: \Theta]$ by subclassing and the object types $\Theta$ and $\Theta'$ are such that $\Theta' <: \Theta$. We know that syntactically for all classes $B$, if $\tau(B) <: \Theta'$ then $\tau(C'[\tau(B)]) <: \tau(C[\tau(B)])^+$. It is interesting, therefore, to analyze a semantic relation between classes generically derived from $C[T <: \Theta]$ and $C'[T <: \Theta']$ by instantiating different object types for $T$. Unfortunately, it turns out that even if we know that $C[\tau(B)] \sqsubseteq C'[\tau(B)]$, we still need to prove class refinement $C[\tau(B')] \sqsubseteq C'[\tau(B')]$ for any subclass $B'$ refining $B$. In other words, it is invalid to assume that if $B \sqsubseteq B'$ and $C[\tau(B)] \sqsubseteq C'[\tau(B)]$ then $C[\tau(B')] \sqsubseteq C'[\tau(B')]$. The example in Fig. 3 illustrates this case. In this example the only method of class $B$ overridden in its subclass is the one calculating the square root of the attribute. Clearly, the class refinement $B \sqsubseteq B'$ holds because the method *Sqrt* of $B'$ is more deterministic, it returns the negative value of the square root, whereas its more abstract counterpart returns a value which can be either positive or negative. Consider now the methods *Set* of classes $C[\tau(B)]$ and $C'[\tau(B)]$. After unfolding definitions of method calls, we can see that in $C[\tau(B)]$ the value of $c$ becomes $\pm v$ and in $C'[\tau(B)]$ simply $v$. The method *Set* defined in $C[\tau(B)]$, being more deterministic, refines its counterpart defined in $C'[\tau(B)]$. So we have that both $B \sqsubseteq B'$ and $C[\tau(B)] \sqsubseteq C'[\tau(B)]$ hold. However, the methods *Set* of classes $C[\tau(B')]$ and $C'[\tau(B')]$ do cause problems, because the one defined in $C'[\tau(B')]$ results in assigning $c$ the absolute value of $v$ as before, whereas *Set* defined in $C[\tau(B')]$ assigns $c$ the negative value. Clearly, refinement does not hold between the classes $C[\tau(B')]$ and $C'[\tau(B')]$ in this case.

The analysis of this problem suggests defining class refinement for generic classes by pointwise extension from refinement on all the classes generically derived from them.

---

$B$ = **class**
  **var** $b$ : *Real*
  *Sqrt*(**res** $r$ : *Real*) =
    $[b \geq 0]; [r := r' \mid r'^2 = b]$,
  *Sqr*() = $b := b^2$,
  *Abs*(**res** $r$ : *Real*) = $r := abs(b)$
**end**

$B'$ = **subclass of** $B$
  **var** $b$ : *Real*
  *Sqrt*(**res** $r$ : *Real*) =
    $[b \geq 0]; r := -\sqrt{b}$,
**end**

$C[T <: \tau(B)^+]$ = **class**
  **var** $c$ : *Real*
  *Set*(**val** $v$ : $T$) = $v.Sqr(); v.Sqrt(c)$
**end**

$C'[T <: \tau(B)^+]$ = **subclass of** $C[T <: \tau(B)^+]$
  **var** $c$ : *Real*
  *Set*(**val** $v$ : $T$) = $v.Abs(c)$
**end**

---

**Fig. 3.** Example of $B \sqsubseteq B' \ \wedge \ C[\tau(B)] \sqsubseteq C'[\tau(B)] \ \not\Rightarrow \ C[\tau(B')] \sqsubseteq C'[\tau(B')]$

**Definition 2.** *A generic class* $C[T <: \tau(B)^+]$ *is refined by a generic class* $C'[T <: \tau(B)^+]$ *whenever for all subclasses* $X$ *of* $B$ *a class* $C[\tau(X)]$ *is refined by a class* $C'[\tau(X)]$*:*

$$C[T <: \tau(B)^+] \sqsubseteq C'[T <: \tau(B)^+] \quad \widehat{=} \quad (\forall X \mid B \sqsubseteq X \cdot C[\tau(X)] \sqsubseteq C'[\tau(X)])$$

From this definition we immediately get the following monotonicity property.

**Theorem 3.** *For any class* $B$ *and any generic classes* $C[T <: \tau(B)^+]$ *and* $C'[T <: \tau(B)^+]$*,*

$$(\forall X \mid B \sqsubseteq X \cdot C[T <: \tau(B)^+] \sqsubseteq C'[T <: \tau(B)^+] \;\Rightarrow\; C[\tau(X)] \sqsubseteq C'[\tau(X)])$$

The definition of class refinement for generic classes is difficult to use in practice, and studying its weaker and more useful form represents an interesting research topic.

Refinement between generic classes involving parameterization by values reduces to one of the described cases of class refinement for generic classes. In particular, in the last example of Sec. 4 refinement holds between *List* [*Element*] and *BoundedList* [*Element*, *maxSize*] for any object type *Element* which is a subtype of *Elem* and any constant value *maxSize* : *Nat*, since skip is always refined by an assumption.

## 6 Conclusions and Related work

We consider correctness of the reuse mechanism based on the combination of parametric polymorphism with subtyping polymorphism and subclassing. We define the notion of refinement for generic classes by extending the definition of ordinary class refinement, and study special cases when refinement is guaranteed to hold between generically derived classes. This work is based on [7], but concentrates on refinement for generic classes.

Related work in formalization of object-oriented concepts includes [10, 20, 21]. Cook and Palsberg in [10] give a denotational semantics of inheritance and prove its correctness with respect to an operational "method lookup" semantics. They model dynamic binding of self-referential methods by representing classes as functions of self-called methods and constructing subclasses using modifying wrappers. There are only functional methods in their model, whereas we consider procedural methods as well. Semantics of a simple Oberon-like programming language with similar specification constructs as here, also based on predicate transformers, is defined by Naumann in [20]. Sekerinski [21] defines a rich object-oriented programming and specification notation by using a type system with subtyping and type parameters, and also using predicate transformers. In both approaches, subtyping is based on extensions of record types. Here we use sum types instead, as suggested by Back and Butler in [4] and further elaborated in [18]. One motivation for moving to sum types is to avoid the complications in the typing and the logic when reasoning about record types: the simply typed lambda calculus as the formal basis is sufficient for our purposes. Also, to allow

objects of a subclass to have different (private) attributes from those of the superclass, hiding by existential types was used in [21]. It turned out that this leads to complications when reasoning about method calls, which are not present when using the model of sum types.

Our model of classes, subclassing, and subtyping polymorphism can be used to reason about the meaning of programs constructed using separate subclassing and interface inheritance hierarchies, like in Java [12]. In that approach interface inheritance is the basis for subtyping polymorphism, whereas subclassing is used only for implementation reuse. By associating a specification class with every interface type, we can reason about the behaviour of objects having this interface. All classes claiming to implement a certain interface must refine its specification class. Subclassing, on the other hand, does not, in general, require establishing class refinement between the superclass and the subclass.

Data refinement of modules, abstract data types, and abstract machines as, e.g., in [14, 19, 2] forms a basis for class refinement. The latter, however, has special features due to subtyping polymorphism and dynamic binding. Behavioural dependencies in presence of subclassing have also been studied in various extensions of Z specification languages, e.g., in [15], but only between class specifications and not implementations. Behavioural subtyping is another area of related work [3, 16, 11] where a distinction is also made between the specification of a class and its implementation. By having specification constructs as part of the (extended) programming language, this distinction becomes unnecessary. It is often claimed, e.g., in [22], that declarative specifications are more abstract and easier to understand than operational ones capturing method invocation order. We, on the contrary, feel that the essence of object-oriented programs is in invoking methods on objects. Also, when reasoning about correctness, it is often necessary to know the method invocation order, which is impossible to specify in terms of pre- and post-conditions. Similar ideas are supported by Helm et al. in [13]. They include method calls in abstract specifications of contracts to express behavioural dependencies between cooperating objects.

Models of parametric polymorphism and, in particular, of bounded parametric polymorphism have been considered, e.g., in [1, 21], but the semantic relationship between generic classes and generically derived classes in presence of subclassing has not, to our knowledge, been studied before.

## Acknowledgments

Discussions with Emil Sekerinski have given me further insight and led to a number of improvements. I also would like to thank Joakim von Wright and Ralph Back for valuable comments on this paper.

## References

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

2. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
3. P. America. A behavioural approach to subtyping in object-oriented programming languages. Technical Report 443, Philips Research Laboratories, Apr. 1989.
4. R. Back and M. Butler. Exploring summation and product operators in the refinement calculus. In B. Möller, editor, *Mathematics of Program Construction, 1995*, volume 947. Springer-Verlag, 1995.
5. R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.
6. R. J. R. Back. Changing data representation in the refinement calculus. In *21st Hawaii International Conference on System Sciences*. IEEE, January 1989.
7. 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.
8. R. J. R. Back and J. von Wright. Programs on product spaces. Technical Report 143, Turku Centre for Computer Science, November 1997.
9. R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, April 1998.
10. 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.
11. 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.
12. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Sun Microsystems, Mountain View, 1996.
13. R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, pages 169–180, Oct. 1990.
14. C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.
15. K. Lano and H. Haughton. Reasoning and refinement in object-oriented specification languages. In O. L. Madsen, editor, *Proceedings of ECOOP'92*, LNCS 615. Springer-Verlag, 1992.
16. B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
17. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.
18. 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.
19. C. C. Morgan. *Programming from Specifications*. Prentice–Hall, 1990.
20. D. A. Naumann. Predicate transformer semantics of an Oberon-like language. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, pages 460–480, San Miniato, Italy, 1994.
21. E. Sekerinski. A type-theoretic basis for an object-oriented refinement calculus. In S. Goldsack and S. Kent, editors, *Formal Methods and Object Technology*. Springer-Verlag, 1996.
22. 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.

# Paper 3

## Reasoning About Interactive Systems

**R.J.R. Back, A. Mikhajlova, and J. von Wright**

To appear in *Proceedings of the World Congress on Formal Methods (FM'99)*, Lecture Notes in Computer Science, September 1999, Springer-Verlag.

# Reasoning About Interactive Systems

Ralph Back, Anna Mikhajlova, Joakim von Wright

Turku Centre for Computer Science, Åbo Akademi University
Lemminkäisenkatu 14A, Turku 20520, Finland
phone: +358-2-215-4032, fax: +358-2-241-0154
e-mail: backrj, amikhajl, jwright @ abo.fi

**Abstract.** The unifying ground for interactive programs and component-based systems is the interaction between a user and the system or between a component and its environment. Modeling and reasoning about interactive systems in a formal framework is critical for ensuring the systems' reliability and correctness. A mathematical foundation based on the idea of contracts permits this kind of reasoning. In this paper we study an iterative choice contract statement which models an event loop allowing the user to repeatedly choose from a number of actions an alternative which is enabled and have it executed. We study mathematical properties of iterative choice and demonstrate its modeling capabilities by specifying a component environment which describes all actions the environment can take on a component, and an interactive dialog box permitting the user to make selections in a dialog with the system. We show how to prove correctness of the dialog box with respect to given requirements, and develop its refinement allowing more complex functionality and providing wider choice for the user.

## 1 Introduction

Most of contemporary software systems are inherently interactive: desk-top applications interact with a user, embedded systems interact with the environment, system integration software interacts with the systems it integrates, etc. In addition, in systems constructed using an object-oriented or a component-based approach objects or components interact with each other.

To be able to verify the behavior of an interactive system in its entirety, it is first necessary to capture this behavior in a precise specification. Formal methods have been traditionally weak in capturing the intricacy of interaction. Probably for this reason, the importance of specifying and verifying program parts describing interaction with the environment (especially in case of interacting with a human user) is considered as secondary to the importance of establishing correctness of some "critical" parts of the program. However, in view of the growing complexity and importance of various interactive systems, the need for verifying correctness of interaction becomes obvious. For instance, embedded systems, which are intrinsically interactive and often used in safety-critical environments, can lead to dramatic consequences if they ignore input from the environment or deliver wrong output.

Component-oriented approach to software design and development is rapidly gaining popularity and stimulates research on methods for analysis and construction of reliable and correct components and their compositions. Component compositions consist of cooperating or interacting components, and for each component all the other components it cooperates with can be collectively considered as the environment. Although various standard methods can be used for reasoning about separate components, component environments present in this respect a challenge. The ability to model and reason about component environments is critical for reasoning about component-based systems. The designer of a component should be aware of the behavior of the environment in which the component is supposed to operate. Knowing the precise behavior of the environment, it is then possible to analyze the effect a change to the component will have on the environment, design an appropriate component interface, etc.

Interaction is often multifaceted in the sense that component-based systems can interact with the user and interactive programs can be component-based. Moreover, for components in a component-based system their environment can be transparent, they will interact with this environment in the same way regardless of whether it is another component or a human user.

To a large extent the weakness of verification techniques for interactive parts of programs can be explained by the lack of modeling methods capable of capturing interaction and the freedom of choice that the environment has. Accordingly, development of a specification and verification method in a formalism expressive enough to model interaction is of critical importance. The mathematical foundation for reasoning about interactive systems, based on the idea of contracts, has been introduced in [4, 6]. In particular, Back and von Wright proposed using an *iterative choice* contract statement which describes an event loop, allowing the user to repeatedly choose from a number of actions an alternative which is enabled and have it executed. In this paper we focus on the iterative choice statement, examine its modeling capabilities, and develop its mathematical properties. In particular, we present rules for proving correctness of iterative choice with respect to given pre- and postconditions, and rules for iterative choice refinement through refining the options it presents and adding new alternatives. We illustrate the expressive power and versatility of iterative choice by specifying a component environment which describes all actions the environment can take on a component, and an interactive dialog box permitting the user to make selections in a dialog with the system. We show how to prove correctness of the dialog box with respect to given requirements, and develop its refinement allowing more complex functionality and providing wider choice for the user.

*Notation:* We use *simply typed higher-order logic* as the logical framework in the paper. The type of functions from a type $\Sigma$ to a type $\Gamma$ is denoted by $\Sigma \to \Gamma$ and functions can have arguments and results of function type. Functions can be described using $\lambda$-abstraction, and we write $f.\ x$ for the application of function $f$ to argument $x$.

## 2 Contracts and Refinement

A computation can generally be seen as involving a number of agents (programs, modules, systems, users, etc.) who carry out actions according to a document (specification, program) that has been laid out in advance. When reasoning about a computation, we can view this document as a contract between the agents involved. In this section we review a notation for *contract statements*. A more detailed description as well as operational and weakest precondition semantics of these statements can be found in [4, 6].

We assume that the world that contracts talk about is described as a *state* $\sigma$. The *state space* $\Sigma$ is the set (type) of all possible states. The state has a number of *program variables* $x_1, \ldots, x_n$, each of which can be observed and changed independently of the others. A program variable $x$ of type $\Gamma$ is really a pair of the *value function* $valx : \Sigma \to \Gamma$ and the *update function* $setx : \Gamma \to \Sigma \to \Sigma$. Given a state $\sigma$, $valx.\ \sigma$ is the value of $x$ in this state, while $\sigma' = setx.\ \gamma.\ \sigma$ is the new state that we get by setting the value of $x$ to $\gamma$. An *assignment* like $x := x + y$ denotes a state changing function that updates the value of $x$ to the value of the expression $x + y$, i.e. $(x := x + y).\ \sigma \ = \ setx.\ (valx.\ \sigma + valy.\ \sigma).\ \sigma$.

A *state predicate* $p : \Sigma \to \mathsf{Bool}$ is a boolean function on the state. Since a predicate corresponds to a set of states, we use set notation ($\cup$, $\subseteq$, etc.) for predicates. Using program variables, state predicates can be written as boolean expressions, for example, $(x + 1 > y)$. Similarly, a *state relation* $R : \Sigma \to \Sigma \to \mathsf{Bool}$ relates a state $\sigma$ to a state $\sigma'$ whenever $R.\ \sigma.\ \sigma'$ holds. We permit a generalized assignment notation for relations. For example, $(x := x' \mid x' > x + y)$ relates state $\sigma$ to state $\sigma'$ if the value of $x$ in $\sigma'$ is greater than the sum of the values of $x$ and $y$ in $\sigma$ and all other variables are unchanged.

### 2.1 Contract notation

Contracts are built from state changing functions, predicates and relations. The *update* $\langle f \rangle$ changes the state according to $f : \Sigma \to \Sigma$. If the initial state is $\sigma_0$ then the agent must produce a final state $f.\ \sigma_0$. An *assignment statement* is a special kind of update where the state changing function is an assignment. For example, the assignment statement $\langle x := x + y \rangle$ (or just $x := x + y$ when it is clear from the context that an assignment statement rather than a state changing function is intended) requires the agent to set the value of program variable $x$ to the sum of the values of $x$ and $y$.

The *assertion* $\{p\}$ of a state predicate $p$ is a requirement that the agent must satisfy in a given state. For instance, $\{x + y = 0\}$ expresses that the sum of (the values of variables) $x$ and $y$ in the state must be zero. If the assertion does not hold, then the agent has *breached* the contract. The *assumption* $[p]$ is dual to an assertion; if the condition $p$ does not hold, then the agent is released from any obligation to carry out his part of the contract.

In the *sequential action* $S_1; S_2$ the action $S_1$ is carried out first, followed by $S_2$. A *choice* $S_1 \sqcup S_2$ allows the agent to choose between carrying out $S_1$ or $S_2$.

In general, there can be a number of agents that are acting together to change the world and whose behavior is bound by contracts. We can indicate explicitly which agent is responsible for each choice. For example, in the contract

$$S \;=\; x := 0; ((y := 1 \sqcup_b y := 2) \sqcup_a x := x + 1); \{y = x\}_a$$

the agents involved are $a$ and $b$. The effect of the update is independent of which agent carries it out, so this information can be lost when writing contract statements.

The *relational update* $\{R\}_a$ is a contract statement that permits an agent to choose between all final states related by the state relation $R$ to the initial state (if no such final state exists, then the agent has breached the contract). For example, the contract statement $\{x := x' \mid x < x'\}_a$ is carried out by agent $a$ by changing the state so that the value of $x$ becomes larger than the current value, without changing the values of any other variables.

A *recursive contract statement* of the form $(\mathsf{rec}_a\ X \bullet S)$ is interpreted as the contract statement $S$, but with each occurrence of statement variable $X$ in $S$ treated as a recursive invocation of the whole contract $(\mathsf{rec}_a\ X \bullet S)$. A more convenient way to define a recursive contract is by an equation of the form $X \;=_a\; S$, where $S$ typically contains some occurrences of $X$. The indicated agent is responsible for termination; if the recursion unfolds infinitely, then the agent has breached the contract.

### 2.2 Using Contracts

Assume that we pick out one or more agents whose side we are taking. These agents are assumed to have a common goal and to coordinate their choices in order to achieve this goal. Hence, we can regard this group of agents as a single agent. The other agents need not share the goals of our agents. To prepare for the worst, we assume that the other agents try to prevent us from reaching our goals, and that they coordinate their choices against us. We will make this a little more dramatic and call our agents collectively the *angel* and the other agents collectively the *demon*. We refer to choices made by our agents as *angelic choices*, and to choices made by the other agents as *demonic choices*.

Having taken the side of certain agents, we can simplify the notation for contract statements. We write $\sqcup$ for the angelic choice $\sqcup_{angel}$ and $\sqcap$ for the demonic choice $\sqcup_{demon}$. Furthermore, we note that if our agents have breached the contract, then the other agents are released from it, i.e. $\{p\}_{angel} = [p]_{demon}$, and vice versa. Hence, we agree to let $\{p\}$ stand for $\{p\}_{angel}$ and $[p]$ stand for $\{p\}_{demon}$. This justifies the following syntax, where the explicit indication of which agent is responsible for the choice, assertion or assumption has been removed:

$$S ::= \langle f \rangle \mid \{p\} \mid [p] \mid S_1; S_2 \mid S_1 \sqcup S_2 \mid S_1 \sqcap S_2$$

This notation generalizes in the obvious way to generalized choices: we write $\sqcup\{S_i \mid i \in I\}$ for the angelic choice of one of the alternatives in the set $\{S_i \mid i \in I\}$

and we write $\sqcap\{S_i \mid i \in I\}$ for the corresponding demonic choice. For relational update, we write $\{R\}$ if the next state is chosen by the angel, and $[R]$ if the next state is chosen by the demon. Furthermore, we write $(\mu X \bullet S)$ for $(\mathsf{rec}_{angel}\ X \bullet S)$ and $(\nu X \bullet S)$ for $(\mathsf{rec}_{demon}\ X \bullet S)$; this notation agrees with the predicate transformer semantics of contracts.

The notation for contracts allows us to express all standard programming language constructs, like sequential composition, assignments, empty statements, conditional statements, loops, and blocks with local variables.

### 2.3 User interaction

Interactive programs can be seen as special cases of contracts, where two agents are involved, the *user* and the *computer system*. The user in this case is the angel, which chooses between alternatives in order to influence the computation in a desired manner, and the computer system is the demon, resolving any internal choices in a manner unknown to the user.

User input during program execution is modeled by an angelic relational assignment. For example, the contract

$$\{x, e := x', e' \mid x' \geq 0 \wedge e > 0\}; [x := x' \mid -e < x'^2 - x < e]$$

describes how the user gives as input a value $x$ whose square root is to be computed, as well as the precision $e$ with which the computer is to compute this square root.

This simple contract *specifies* the interaction between the user and the computing system. The first statement specifies the user's responsibility (to give an input value that satisfies the given conditions) and the second statement specifies the system's responsibility (to compute a new value for $x$ that satisfies the given condition).

### 2.4 Semantics, Correctness, and Refinement of Contracts

Every contract statement has a weakest precondition predicate transformer semantics. A *predicate transformer* $S : (\Gamma \to \mathsf{Bool}) \to (\Sigma \to \mathsf{Bool})$ is a function from predicates on $\Gamma$ to predicates on $\Sigma$. We write

$$\Sigma \mapsto \Gamma \ \widehat{=}\ (\Gamma \to \mathsf{Bool}) \to (\Sigma \to \mathsf{Bool})$$

to denote a set of all predicate transformers from $\Sigma$ to $\Gamma$. A contract statement with initial state in $\Sigma$ and final state in $\Gamma$ determines a monotonic predicate transformer $S : \Sigma \mapsto \Gamma$ that maps any postcondition $q : \Gamma \to \mathsf{Bool}$ to the weakest precondition $p : \Sigma \to \mathsf{Bool}$ such that the statement is guaranteed to terminate in a final state satisfying $q$ whenever the initial state satisfies $p$. Following an established tradition, we identify contract statements with the monotonic predicate transformers that they determine. For details of the predicate transformer semantics, we refer to [4, 6].

The *total correctness assertion* $p \; \{\!|\, S \,|\!\} \; q$ is said to hold if the user can use the contract $S$ to establish the postcondition $q$ when starting in the set of states $p$. The pair of state predicates $(p, q)$ is usually referred to as the pre- and postcondition specification of the contract $S$. The total correctness assertion $p \; \{\!|\, S \,|\!\} \; q$, which is equal to $p \subseteq S.\ q$, means that the user can (by making the right choices) either achieve the postcondition $q$ or be released from the contract, no matter what the other agents do.

A contract $S$ is *refined by* a contract $S'$, written $S \sqsubseteq S'$, if any condition that we can establish with the first contract can also be established with the second contract. Formally, $S \sqsubseteq S'$ is defined to hold if $p \; \{\!|\, S \,|\!\} \; q \Rightarrow p \; \{\!|\, S' \,|\!\} \; q$, for any $p$ and $q$. Refinement is reflexive and transitive. In addition, the contract constructors are monotonic, so a contract can be refined by refining a subcomponent.

The refinement calculus provides rules for transforming more abstract program structures into more concrete ones based on the notion of refinement of contracts presented above. Large collections of refinement rules are given, for instance, in [6, 10].

## 3 Iterative Choice and Its Modeling Capabilities

### 3.1 Modeling Component Environment

To demonstrate how the iterative choice statement can be used to model a component environment, let us first introduce the notion of a component. We view a component as an abstract data type with internal state and methods that can be invoked on the component to carry out certain functionality and (possibly) change the component's state.

$$
\begin{array}{l}
c \;=\; \mathsf{component} \\
\quad x : \Sigma := x_0 \\
\quad m_1 \,(\mathsf{val}\ x_1 : \Gamma_1, \mathsf{res}\ y_1 : \Delta_1) \;=\; M_1, \\
\quad \ldots \\
\quad m_n \,(\mathsf{val}\ x_n : \Gamma_n, \mathsf{res}\ y_n : \Delta_n) \;=\; M_n \\
\mathsf{end}
\end{array}
$$

Here $x : \Sigma$ are the variables which carry the internal component's state. These variables have some initial values $x_0$. Methods named $m_1, \ldots, m_n$ are specified by statements $M_1, \ldots, M_n$ respectively. Invocation of a method on a component has a standard procedure call semantics, with the only difference that the value of the component itself is passed as a value-result argument. We will denote invocation of $m_i$ on $c$ with value and result arguments $v : \Gamma_i$ and $r : \Delta_i$ by $c.m_i(v, r)$.

An environment using a component $c$ does so by invoking its methods. Every time the environment has a choice of which method 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 environment may repeatedly choose a particular option which is enabled and

have it executed. The environment decides on its own when it is willing to stop choosing options. Such an iterative choice of method invocations, followed by arbitrary statements not affecting the component state directly, describes all the actions the environment program might undertake:

$$\mathsf{begin\ var}\ l : \Lambda \bullet p; \mathsf{do}\ q_1 :: c.m_1(g_1, d_1); L_1 \lozenge \ \ldots\ \lozenge\ q_n :: c.m_n(g_n, d_n); L_n\ \mathsf{od\ end}$$

Here the construct inside the keywords do .. od is the iterative choice statement. The alternatives among which the choice is made at each iteration step are separated by $\lozenge$. Variables $l : \Lambda$ are some local variables initialized according to $p$, predicates $q_1 \ldots q_n$ are the asserted conditions on the state, and statements $L_1$ through $L_n$ are arbitrary. The initialization $p$, the assertions $q_1 \ldots q_n$, and the statements $L_1, \ldots, L_n$ do not refer to $c$, which is justified by the assumption that the component state is encapsulated.

The whole program statement is a contract between the component $c$ and *any environment* using $c$. The method enabledness condition $q_i$ corresponds to the assumptions made by the corresponding method $m_i$, as stated in its subcontract (the method body definition). For example, in a component *EntryField* a method *SetLength*(val $l$ : Nat) can begin with an assumption that the length $l$ does not exceed some constant value *lmax*. An environment invoking *SetLength* on *EntryField* will then have to assert that a specific *length* does indeed satisfy this requirement:

$$\mathsf{do}\ \mathit{length} \leq \mathit{lmax} :: \mathit{EntryField}.\mathit{SetLength}(\mathit{length}); \ldots\ \mathsf{od}$$

The assumption of this condition in the body of *SetLength* will pass through, as $\{p\}; [p] = \{p\}$, for all predicates $p$.

### 3.2 Modeling an Interactive Dialog Box

Suppose that we would like to describe a font selection dialog box, where the user is offered the choice of selecting a particular font and its size. The user can select a font by typing the font name in the entry field; the selection is accepted if the entered font name belongs to the set of available fonts. The size of the font can also be chosen by typing the corresponding number in the entry field. The user may change the selections of both the font and the size any number of times before he presses the OK button, which results in closing the dialog box and changing the corresponding text according to the last selection. We can model this kind of a dialog box as shown in Fig. 1. In this specification *fentry* : String and *sentry* : Nat are global variables representing current selections of the font name and its size in the corresponding entry fields of the dialog box. The constants *Fonts* : set of String and *Sizes* : set of Nat represent sets of available font names and font sizes.

When the user opens the dialog box, he assumes that the default entries for the font name and size are among those available in the system, as expressed by the corresponding assumption in *DialogBoxSpec*. If this assumption is met by the system, the user may enter new font name, or new font size, or leave the current

$$
\begin{array}{lll}
\mathit{DialogBoxSpec} & = & [\mathit{fentry} \in \mathit{Fonts} \wedge \mathit{sentry} \in \mathit{Sizes}]; \\
 & & \mathsf{do}\ \ \mathsf{true} :: \{\mathit{fentry} := s \mid s \in \mathit{Fonts}\} \\
 & & \ \ \Diamond\ \ \mathsf{true} :: \{\mathit{sentry} := n \mid n \in \mathit{Sizes}\} \\
 & & \mathsf{od}
\end{array}
$$

**Fig. 1.** Specification of a dialog box

selections intact. The user may select any alternative any number of times until he is satisfied with the choice and decides to stop the iteration. Note that to model dialog closing, we do not need to explicitly maintain a boolean variable *Ok_pressed*, have all the options enabled only when $\neg \mathit{Ok\_pressed}$ holds, and set it explicitly to true to terminate iteration: all this is implicit in the model.

This is a very general specification of *DialogBoxSpec*, but still it is a useful abstraction precisely and succinctly describing the intended behavior. In Sec. 4.3 we will show how one can check correctness of this specification with respect to a given precondition and postcondition. Also, this specification can be refined to a more detailed one, specifying an extended functionality, as we will demonstrate in Sec. 4.5.

## 4 Definition and Properties of Iterative Choice

We begin with studying mathematical properties of an *angelic iteration* operator, which is used to define iterative choice.

### 4.1 Angelic Iteration and Its Properties

Let $S$ be a monotonic predicate transformer (i.e., the denotation of a contract). We define an iteration construct over $S$, *angelic iteration*, as the following fixpoint:

$$S^{\phi} \ \widehat{=}\ (\mu X \bullet S; X \sqcup \mathsf{skip}) \qquad (\textit{Angelic iteration})$$

As such, this construct is a dual of the weak iteration $S^*$ defined in [6] by $(\nu X \bullet S; X \sqcap \mathsf{skip})$.

**Theorem 1.** *Let $S$ be an arbitrary monotonic predicate transformer. Then*

$$S^{\phi} \ = \ ((S^{\circ})^*)^{\circ}$$

Intuitively, the statement $S^{\phi}$ is executed so that $S$ is repeated an angelically chosen (finite) number of times before the iteration is terminated by choosing skip. For example, $(x := x{+}1)^{\phi}$ increments $x$ an angelically chosen finite number of times, and has, therefore, the same effect as the angelic update $\{x := x' \mid x \leq x'\}$.

A collection of basic properties of angelic iteration follows by duality from the corresponding properties of weak iteration proved in [5].

**Theorem 2.** *Let $S$ and $T$ be arbitrary monotonic predicate transformers. Then*

$(a)$ $S^\phi$ is monotonic and terminating

$(b)$ $S^\phi$ preserves termination, strictness, and disjunctivity

$(c)$ $S \sqsubseteq S^\phi$

$(d)$ $(S^\phi)^\phi \;=\; S^\phi$

$(e)$ $S^\phi; S^\phi \;=\; S^\phi$

$(f)$ $S \sqsubseteq T \;\Rightarrow\; S^\phi \sqsubseteq T^\phi$

Here, a predicate transformer $S$ is said to be *terminating* if $S.\,\mathsf{true} = \mathsf{true}$, *strict* if $S.\,\mathsf{false} = \mathsf{false}$, and *disjunctive* if $S.\,(\cup i \in I \bullet q_i) \;=\; (\cup i \in I \bullet S.\,q_i)$, for $I \neq \emptyset$.

To account for tail recursion, angelic iteration can be characterized as follows:

**Lemma 1.** *Let $S$ and $T$ be arbitrary monotonic predicate transformers. Then*

$$S^\phi; T \;=\; (\mu X \bullet S; X \sqcup T)$$

This lemma provides us with general unfolding and induction rules. For arbitrary monotonic predicate transformers $S$ and $T$,

$$S^\phi; T \;=\; S; S^\phi; T \sqcup T \qquad (\textit{unfolding})$$

$$S; X \sqcup T \sqsubseteq X \;\Rightarrow\; S^\phi; T \sqsubseteq X \qquad (\textit{induction})$$

From the unfolding rule with $T$ taken to be $\mathsf{skip}$ we get the useful property that doing nothing is refined by angelic iteration:

$$\mathsf{skip} \sqsubseteq S^\phi$$

Angelic iteration can also be characterized on the level of predicates:

**Lemma 2.** *Let $S : \Sigma \mapsto \Sigma$ be an arbitrary monotonic predicate transformer and $q : \mathcal{P}\Sigma$ an arbitrary predicate. Then*

$$S^\phi.q \;=\; (\mu x \bullet S.\,x \cup q)$$

When applied to monotonic predicate transformers, the angelic iteration operator has two interesting properties known from the theory of regular languages, namely, the *decomposition property* and the *leapfrog property*.

**Lemma 3.** *Let $S$ and $T$ be arbitrary monotonic predicate transformers. Then*

$$(S \sqcup T)^\phi \;=\; S^\phi; (T; S^\phi)^\phi \qquad (\textit{decomposition})$$

$$(S; T)^\phi; S \;\sqsubseteq\; S; (T; S)^\phi \qquad (\textit{leapfrog})$$

(if $S$ is disjunctive, then the leapfrog property is an equality).

Lemma 1, Lemma 2, and Lemma 3 follow by duality from the corresponding properties of weak iteration as given in [6].

Let us now study under what conditions the total correctness assertion $p \; \{\!|\, S^\phi \,|\!\} \; q$ is valid. In lattice theory, the general least fixpoint introduction rule states that

$$\frac{t_w \;\sqsubseteq\; f.\, t_{<w}}{t \;\sqsubseteq\; \mu\; f}$$

where $\{t_w \mid w \in W\}$ is a ranked collection of elements (so that $W$ is a well-founded set and $v < w \Rightarrow t_v \;\sqsubseteq\; t_w$), $t_{<w}$ is an abbreviation for $(\sqcup v \mid v < w \bullet t_v)$, and $t = (\sqcup w \in W \bullet t_w)$. When used for predicates, with $S^\phi.\; q = (\mu x \bullet S.\; x \cup q)$, this rule directly gives us the correctness rule for angelic iteration

$$\frac{p_w \;\subseteq\; (S.\; p_{<w}) \cup q}{p \; \{\!|\, S^\phi \,|\!\} \; q} \qquad \begin{array}{l}(\textit{angelic iteration}\\ \textit{correctness rule})\end{array}$$

where $\{p_w \mid w \in W\}$ is a ranked collection of predicates and $p = (\cup w \in W \cdot p_w)$. If the ranked predicates are written using an invariant $I$ and a termination function $t$, then we have

$$\frac{I \cap t = w \;\subseteq\; S.\; (I \cap t < w) \cup q}{I \; \{\!|\, S^\phi \,|\!\} \; q}$$

where $w$ is a fresh variable. Intuitively, this rule says that at every step either the invariant $I$ is preserved (with $t$ decreasing) or the desired postcondition $q$ is reached directly and the iteration can terminate. This corresponds to temporal logic assertions "$I$ *until* $q$" and "*eventually not* $I$". Since $t$ cannot decrease indefinitely, this guarantees that the program eventually reaches $q$ if it started in $I$.

### 4.2 Iterative Choice and Its Properties

Now we consider a derivative of the angelic iteration $S^\phi$, the *iterative choice* statement. This specification construct was defined in [6] as follows:

$$\begin{array}{l}\mathsf{do}\; \Diamond_{i=1}^{n} g_i :: S_i \;\mathsf{od} \;\;\widehat{=}\qquad\qquad (\textit{Iterative choice})\\ \quad (\mu\, X \bullet \{g_1\}; S_1; X \sqcup \ldots \sqcup \{g_n\}; S_n; X \sqcup \mathsf{skip})\end{array}$$

As such, iterative choice is equivalent to the angelic iteration of the statement $\sqcup_{i=1}^{n} \{g_i\}; S_i$,

$$\mathsf{do}\; \Diamond_{i=1}^{n} g_i :: S_i \;\mathsf{od} \;=\; (\sqcup_{i=1}^{n} \{g_i\}; S_i)^\phi$$

and its properties can be derived from the corresponding properties of the angelic iteration.

An angelic iteration is refined if every alternative in the old system is refined by the angelic choice of all the alternatives in the new system.

**Theorem 3.** *For arbitrary state predicates $g_1, \ldots, g_n$ and $g'_1, \ldots, g'_m$, and arbitrary contract statements $S_1, \ldots, S_n$ and $S'_1, \ldots, S'_m$ we have that*

$$(\forall i \mid 1 \leq i \leq n \bullet \{g_i\}; S_i \sqsubseteq \sqcup_{j=1}^{m} \{g'_j\}; S'_j) \Rightarrow \mathsf{do}\ \Diamond_{i=1}^{n} g_i :: S_i\ \mathsf{od} \sqsubseteq \mathsf{do}\ \Diamond_{j=1}^{m} g'_j :: S'_j\ \mathsf{od}$$

This can be compared with the rule for Dijkstra's traditional do-loop, where every alternative of the new loop must refine the demonic choice of the alternatives of the old loop (and the exit condition must be unchanged).

Two useful corollaries state that whenever every option is refined, the iterative choice of these options is a refinement, and also that adding alternatives in the iterative choice is a refinement.

**Corollary 1.** *For arbitrary state predicates $g_1, \ldots, g_n$ and $g'_1, \ldots, g'_n$, and arbitrary contract statements $S_1, \ldots, S_n$ and $S'_1, \ldots, S'_n$ we have that*

$$g_1 \subseteq g'_1 \wedge \ldots \wedge g_n \subseteq g'_n \wedge \{g_1\}; S_1 \sqsubseteq S'_1 \wedge \ldots \wedge \{g_n\}; S_n \sqsubseteq S'_n \Rightarrow \mathsf{do}\ \Diamond_{i=1}^{n} g_i :: S_i\ \mathsf{od} \sqsubseteq \mathsf{do}\ \Diamond_{i=1}^{n} g'_i :: S'_i\ \mathsf{od}$$

**Corollary 2.** *For arbitrary state predicates $g_1, \ldots, g_{n+1}$ and arbitrary contract statements $S_1, \ldots, S_{n+1}$ we have that*

$$\mathsf{do}\ \Diamond_{i=1}^{n} g_i :: S_i\ \mathsf{od} \sqsubseteq \mathsf{do}\ \Diamond_{i=1}^{n+1} g_i :: S_i\ \mathsf{od}$$

The correctness rule for iterative choice states that for each ranked predicate which is stronger than the precondition there should be a choice decreasing the rank of this predicate or the possibility of establishing the postcondition directly:

$$\frac{p_w \subseteq \cup_{i=1}^{n} (g_i \cap S_i.\ p_{<w}) \cup q}{p\ \{\!|\ \mathsf{do}\ \Diamond_{i=1}^{n} g_i :: S_i\ \mathsf{od}\ |\!\}\ q} \qquad \begin{array}{l}(\textit{iterative choice} \\ \textit{correctness rule})\end{array}$$

When the ranked predicates are written using an invariant $I$ and a termination function $t$, this rule becomes

$$\frac{p \subseteq I \qquad I \cap t = w \subseteq \cup_{i=1}^{n} (g_i \cap S_i.\ (I \cap t < w)) \cup q}{p\ \{\!|\ \mathsf{do}\ \Diamond_{i=1}^{n} g_i :: S_i\ \mathsf{od}\ |\!\}\ q}$$

From the correctness rule we immediately get the iterative choice introduction rule

$$\frac{p_w \subseteq \cup_{i=1}^{n} (g_i \cap S_i.\ p_{<w}) \cup q[x' := x]}{\{p\}; [x := x' \mid q] \sqsubseteq \mathsf{do}\ \Diamond_{i=1}^{n} g_i :: S_i\ \mathsf{od}} \qquad \begin{array}{l}(\textit{iterative choice} \\ \textit{introduction rule})\end{array}$$

where $x$ does not occur free in $q$.

### 4.3 Proving Correctness of the Interactive Dialog Box

Suppose that the font "Times" belongs to the set of available fonts, *Fonts*, and the size 12 is in the set of available sizes, *Sizes*. Can the user, by making the right choices, select this font with this size? The answer to this question can be given by verifying the following total correctness assertion:

$$\begin{array}{c}\text{"Times"} \in \mathit{Fonts} \cap \\ 12 \in \mathit{Sizes}\end{array} \quad \{\!| \begin{array}{l}\mathsf{do}\ \ \mathsf{true} :: \{\mathit{fentry} := s \mid s \ \mathsf{in}\ \mathit{Fonts}\} \\ \ \ \Diamond \\ \mathsf{od}\end{array} \begin{array}{l}\mathsf{true} :: \{\mathit{sentry} := n \mid n \ \mathsf{in}\ \mathit{Sizes}\}\end{array} |\!\} \quad \begin{array}{c}\mathit{fentry} = \text{"Times"} \cap \\ \mathit{sentry} = 12\end{array}$$

Using the rule for the correctness of iterative choice with the invariant $I$ and the termination function $t$ such that

$$\begin{array}{lcl} I & = & \text{"Times"} \in \mathit{Fonts} \cap 12 \in \mathit{Sizes} \\ t & = & \#\left(\{\text{"Times"}, 12\} \setminus \{\mathit{fentry}, \mathit{sentry}\}\right)\end{array}$$

we then need to prove two subgoals:

1. $\text{"Times"} \in \mathit{Fonts} \cap 12 \in \mathit{Sizes} \ \subseteq \ I$
2. $I \cap t = w \ \subseteq \ \begin{array}{l}\mathsf{true} \cap \{\mathit{fentry} := s \mid s \ \mathsf{in}\ \mathit{Fonts}\}.\ (I \cap t < w) \cup \\ \mathsf{true} \cap \{\mathit{sentry} := n \mid n \ \mathsf{in}\ \mathit{Sizes}\}.\ (I \cap t < w) \cup \\ \mathit{fentry} = \text{"Times"} \ \cap \ \mathit{sentry} = 12\end{array}$

The first subgoal states that the precondition is stronger than the invariant and is trivially true. The second subgoal states that, when the invariant holds, at least one of the alternatives will decrease the termination function while preserving the invariant. It can be proved by using the definition of angelic relational update and rules of logic.

Being very simple, this example nethertheless demonstrates the essence of establishing correctness in the presence of iterative choice. By verifying that this specification is correct with respect to the given pre- and postcondition, we can guarantee that any refinement of it will preserve the correctness.

### 4.4 Data Refinement of Iterative Choice

Data refinement is a general technique by which one can change data representation in a refinement. A contract statement $S$ may begin in a state space $\Sigma$ and end in a state space $\Gamma$, written $S : \Sigma \mapsto \Gamma$. Assume that contract statements $S$ and $S'$ operate on state spaces $\Sigma$ and $\Sigma'$ respectively, i.e. $S : \Sigma \mapsto \Sigma$ and $S' : \Sigma' \mapsto \Sigma'$. Let $R : \Sigma' \to \Sigma \to \mathsf{Bool}$ be a relation between the state spaces $\Sigma'$ and $\Sigma$. Following [3], the statement $S$ is said to be *data refined* by the statement $S'$ via the relation $R$, denoted $S \sqsubseteq_{\{R\}} S'$, if $\{R\}; S \sqsubseteq S'; \{R\}$. An alternative and equivalent characterization of data refinement using the inverse relation $R^{-1}$ arises from the fact that $\{R\}$ and $[R^{-1}]$ are each others inverses, in the sense that $\{R\}; [R^{-1}] \sqsubseteq \mathsf{skip}$ and $\mathsf{skip} \sqsubseteq [R^{-1}]; \{R\}$. Abbreviating $\{R\}; S; [R^{-1}]$ by $S \downarrow \{R\}$ we have that

$$S \sqsubseteq_{\{R\}} S' \ \equiv \ S \downarrow \{R\} \sqsubseteq S'$$

We will call $D$ an *abstraction statement* if $D$ is such that $D = \{R\}$, for some $R$. In this case, our notion of data refinement is the standard one, often referred to as forward data refinement or downward simulation.

Data refinement properties of angelic iteration and iterative choice cannot be proved directly by a duality argument from the corresponding results for the traditional iteration operators. However, they can still be proved:

**Theorem 4.** *Assume that $S$ and $D$ are monotonic predicate transformers and that $D$ is an abstraction statement. Then*

$$S^\phi \downarrow D \ \sqsubseteq \ (S \downarrow D)^\phi$$

As a consequence, the angelic iteration operator preserves data refinement:

**Corollary 3.** *Assume that $S, S'$ and $D$ are monotonic predicate transformers and that $D$ is an abstraction statement. Then*

$$S \sqsubseteq_D S' \ \Rightarrow \ S^\phi \sqsubseteq_D S'^\phi$$

Proofs of Theorem 4 and Corollary 3 can be found in [2].

Data refinement rules for iterative choice also arise from the corresponding rules for angelic iteration. First, data refinement can be propagated inside iterative choice:

**Theorem 5.** *Assume that $g_1, \ldots, g_n$ are arbitrary state predicates, $S_1, \ldots, S_n$ are arbitrary contract statements, and $D$ is an abstraction statement. Then*

$$\mathsf{do}\ \Diamond_{i=1}^n g_i :: S_i\ \mathsf{od} \downarrow D \ \sqsubseteq \ \mathsf{do}\ \Diamond_{i=1}^n D.\ g_i :: S_i \downarrow D\ \mathsf{od}$$

A more general rule shows how a proof of data refinement between iterative choices can be reduced to proofs of data refinement between the iterated alternatives.

**Theorem 6.** *Assume that $g_1, \ldots, g_n$ and $g'_1, \ldots, g'_m$ are arbitrary state predicates, $S_1, \ldots, S_n$ and $S'_1, \ldots, S'_m$ are arbitrary contract statements, and $D$ is an abstraction statement. Then*

$$\begin{array}{l}(\forall i \mid 1 \le i \le n \bullet \{g_i\}; S_i \sqsubseteq_D \sqcup_{j=1}^m \{g'_j\}; S'_j) \ \Rightarrow \\ \qquad \mathsf{do}\ \Diamond_{i=1}^n g_i :: S_i\ \mathsf{od} \sqsubseteq_D \ \mathsf{do}\ \Diamond_{j=1}^m g'_j :: S'_j\ \mathsf{od}\end{array}$$

Proofs of Theorems 5 and 6 can be found in [2]. A useful special case of these theorems is when the number of choices is the same and they are refined one by one.

**Corollary 4.** *Assume that $g_1, \ldots, g_n$ and $g'_1, \ldots, g'_n$ are arbitrary state predicates, $S_1, \ldots, S_n$ and $S'_1, \ldots, S'_n$ are arbitrary contract statements, and $D$ is an abstraction statement. Then*

$$\begin{array}{l}D.\ g_1 \subseteq g'_1 \wedge \ldots \wedge D.\ g_n \subseteq g'_n \wedge \{g_1\}; S_1 \sqsubseteq_D S'_1 \wedge \ldots \wedge \{g_n\}; S_n \sqsubseteq_D S'_n \ \Rightarrow \\ \qquad \mathsf{do}\ \Diamond_{i=1}^n g_i :: S_i\ \mathsf{od} \sqsubseteq_D \ \mathsf{do}\ \Diamond_{i=1}^n g'_i :: S'_i\ \mathsf{od}\end{array}$$

### 4.5 Data Refinement of Interactive Dialog Box

Let us now demonstrate how our original specification of a dialog box can be data refined to a more concrete one. Suppose that we would like to describe a dialog box, where the user can select a font by choosing it from the list of available fonts or by typing the font name in the entry field. The size of the font can also be chosen either from the list of sizes or by typing the corresponding number in the entry field. Using the iterative choice statement, we can model this kind of a dialog box as shown in Fig. 2.

In this specification the arrays $fonts$ : array $1..fmax$ of String and $sizes$ : array $1..smax$ of Nat are used to represent lists of the corresponding items. When the user opens the dialog box, the system initializes $fonts$ and $sizes$ to contain elements from the constant sets $Fonts$ and $Sizes$. The function $array\_to\_set$, used for this purpose, is given as follows:

$$array\_to\_set \;=\; (\lambda\,(a,n)\,.\,\{e \mid \exists\, i \bullet 1 \le i \le n \,\wedge\, a[i] = e\})$$

The initialization conditions $\#Fonts = fmax$ and $\#Sizes = smax$ state, in addition, that the arrays contain exactly as many elements as the corresponding constant sets. Indices $fpos$ : Nat and $spos$ : Nat represent the currently chosen selections in the corresponding arrays and are initialized to index some items in $fonts$ and $sizes$; the variables $fentry$ and $sentry$ are initialized with values of these items. The implicit invariant maintained by $DialogBox$ states that $fonts[fpos] = fentry$ and $sizes[spos] = sentry$, i.e. the currently selected font in the list of available fonts is the same as the one currently typed in the font entry field, and similarly for font sizes.

The iterative choice statement is the contract stipulating the interaction between the user making choices and the system reacting to these choices. Consider,

---

$$
\begin{array}{lll}
DialogBox & = & [fentry, sentry, fonts, sizes, fpos, spos := \\
 & & \quad fentry', sentry', fonts', sizes', fpos', spos' \mid \\
 & & \quad\quad array\_to\_set(fonts', fmax) = Fonts \,\wedge\, \#Fonts = fmax \,\wedge \\
 & & \quad\quad array\_to\_set(sizes', smax) = Sizes \,\wedge\, \#Sizes = smax \,\wedge \\
 & & \quad\quad fentry' = fonts'[fpos'] \,\wedge\, sentry' = sizes'[spos'] \,\wedge \\
 & & \quad\quad 1 \le fpos' \le fmax \,\wedge\, 1 \le spos' \le smax]; \\
 & & \mathsf{do}\ \ \mathsf{true} :: \{fentry := fentry' \mid \exists\, i \bullet 1 \le i \le fmax \,\wedge\, fonts[i] = fentry'\}; \\
 & & \quad\quad [fpos := fpos' \mid fonts[fpos'] = fentry] \\
 & & \Diamond\ \ \mathsf{true} :: \{sentry := sentry' \mid \exists\, i \bullet 1 \le i \le smax \,\wedge\, sizes[i] = sentry'\}; \\
 & & \quad\quad [spos := spos' \mid sizes[spos'] = sentry] \\
 & & \Diamond\ \ \mathsf{true} :: \{fpos := fpos' \mid 1 \le fpos' \le fmax\}; fentry := fonts[fpos] \\
 & & \Diamond\ \ \mathsf{true} :: \{spos := spos' \mid 1 \le spos' \le smax\}; sentry := sizes[spos] \\
 & & \mathsf{od}
\end{array}
$$

---

**Fig. 2.** Specification of a dialog box refinement

for example, the case when the user wants to select a font by directly choosing it from the list of available fonts, as modeled by the third alternative. First, the user is offered to pick an index $fpos'$, identifying a certain font in the list of fonts, and then the system updates the variable *fentry* to maintain the invariant $fonts[fpos] = fentry$.

The abstraction relation coercing the state of *DialogBox* to the state of *DialogBoxSpec* is essentially an invariant on the concrete variables:

$$\begin{array}{l} array\_to\_set(fonts, fmax) = Fonts \wedge \#Fonts = fmax \wedge 1 \leq fpos \leq fmax \wedge \\ array\_to\_set(sizes, smax) = Sizes \wedge \#Sizes = smax \wedge 1 \leq spos \leq smax \wedge \\ fentry = fonts[fpos] \wedge sentry = sizes[spos] \end{array}$$

Strictly speaking, we should distinguish between $fentry, sentry$ of *DialogBoxSpec* and $fentry, sentry$ of *DialogBox*; the abstraction relation also includes the conditions $fentry = fentry_0$ and $sentry = sentry_0$, where $fentry_0$ and $sentry_0$ denote *fentry* and *sentry* of *DialogBoxSpec*. It can be shown that $DialogBoxSpec \sqsubseteq_{\{R\}} DialogBox$, where

$$\begin{array}{lll} R.\ concrete.\ abstract & = & array\_to\_set(fonts, fmax) = Fonts \wedge \#Fonts = fmax \wedge \\ & & array\_to\_set(sizes, smax) = Sizes \wedge \#Sizes = smax \wedge \\ & & 1 \leq fpos \leq fmax \wedge 1 \leq spos \leq smax \wedge \\ & & fentry = fonts[fpos] \wedge sentry = sizes[spos] \wedge \\ & & fentry = fentry_0 \wedge sentry = sentry_0 \end{array}$$

with $concrete = fentry, sentry, fonts, sizes, fpos, spos$ and $abstract = fentry_0, sentry_0$.

## 5 Conclusions and Related Work

We have described an interactive computing system in terms of contracts binding participating agents and stipulating their obligations and assumptions. In particular, we have focused on the iterative choice contract and studied its algebraic properties and modeling capabilities. This work extends [4] where Back and von Wright introduced the notions of correctness and refinement for contracts and defined their weakest precondition semantics.

The notion of contracts is based on the fundamental duality between demonic and angelic nondeterminism (choices of different agents), abortion (breaching a contract), and miracles (being released from a contract). The notion of angelic nondeterminism goes back to the theory of nondeterministic automata and the nondeterministic programs of Floyd [8]. Broy in [7] discusses the use of demonic and angelic nondeterminism with respect to concurrency. Some applications of angelic nondeterminism are shown by Ward and Hayes in [12]. Abadi, Lamport, and Wolper in [1] study realizability of specifications, considering them as "determined" games, where the system plays against the environment and wins if it produces a correct behavior. Specifications are identified with the properties that they specify, and no assumptions are made about how they are written.

Moschovakis in [11] studies non-deterministic interaction in concurrent communication also considering it from the game-theoretic perspective.

Another direction of related work concentrates on studying the role of interaction in computing systems. Wegner in [13] proposes to use *interaction machines* as "a formal framework for interactive models". Interaction machines are described as extensions of Turing machines with unbounded input streams, which "precisely capture fuzzy concepts like open systems and empirical computer science". The main thesis of work presented in [13] and further developed in [14] is that "Logic is too weak to model interactive computation" and, instead, empirical models should be used for this purpose. Apparently, first-order logic is meant by the author, which is indeed too weak for modeling interaction. However, our formalization is based on an extension of higher-order logic and, as such, is perfectly suitable for this purpose. Also, it is claimed in [13] that "Interaction machines are incomplete in the sense of Gödel: their nonenumerable number of true statements cannot be enumerated by a set of theorems. [...] The incompleteness of interactive systems implies that proving correctness is not merely hard but impossible." We believe that our work presents a proof to the contrary.

As future work we intend to investigate modeling capabilities of iterative choice further. In particular, its application to modeling client and server proxies in distributed object-oriented systems appears to be of interest. Various architectural solutions, such as implicit invocation [9], can also be described in this framework, and the work on this topic is the subject of current research.

## References


1. M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable specifications of reactive systems. In *Proceedings of 16th ICALP*, volume 372 of *LNCS*, pages 1–17, Stresa, Italy, 11–15 July 1989. Springer-Verlag.
2. R. Back, A. Mikhajlova, and J. von Wright. Modeling component environments and interactive programs using iterative choice. Technical Report 200, Turku Centre for Computer Science, September 1998.
3. R. J. R. Back. Changing data representation in the refinement calculus. In *21st Hawaii International Conference on System Sciences*. IEEE, January 1989.
4. R. J. R. Back and J. von Wright. Contracts, games and refinement. Technical Report 138, Turku Centre for Computer Science, October 1997.
5. R. J. R. Back and J. von Wright. Reasoning algebraically about loops. Technical Report 144, Turku Centre for Computer Science, November 1997.
6. R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, April 1998.
7. M. Broy. A theory for nondeterminism, parallelism, communication, and concurrency. *Theoretical Computer Science*, 45:1–61, 1986.
8. R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical aspects of computer science*, volume 19, pages 19–31. American Mathematical Society, 1967.
9. D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM 91, Volume 1: Conference Contributions*, LNCS 551, pages 31–44. Springer-Verlag, Oct. 1991.

10. C. C. Morgan. *Programming from Specifications*. Prentice–Hall, 1990.
11. Y. N. Moschovakis. A model of concurrency with fair merge and full recursion. *Information and Computation*, 93(1):114–171, July 1991.
12. N. Ward and I. Hayes. Applications of angelic nondeterminism. In P.A.C.Bailes, editor, *6th Australian Software Engineering Conference*, pages 391–404, Sydney, Australia, 1991.
13. P. Wegner. Interactive software technology. In J. Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, in cooperation with ACM, 1997.
14. P. Wegner. Interactive foundations of computing. *Theoretical Computer Science*, 192(2):315–351, Feb. 1998.

# Paper 4

## Class Refinement as Semantics of Correct Object Substitutability

**R.J.R. Back, A. Mikhajlova, and J. von Wright**

Extends Paper 1, previous version published as a technical report of Turku Centre for Computer Science, TUCS-TR-147, December 1997. Submitted to *Formal Aspects of Computing*.

# Class Refinement as Semantics of Correct Object Substitutability

Ralph Back, Anna Mikhajlova, Joakim von Wright

Turku Centre for Computer Science, Åbo Akademi University
Lemminkäisenkatu 14A, Turku 20520, Finland

**Abstract.** Subtype polymorphism, based on syntactic conformance of objects' methods and used for substituting subtype objects for supertype objects, is a characteristic feature of the object-oriented programming style. While certainly very useful, typechecking of syntactic conformance of subtype objects to supertype objects is insufficient to guarantee correctness of object substitutability. In addition, the behaviour of subtype objects must be constrained to achieve correctness. In class-based systems classes specify the behaviour of the objects they instantiate. In this paper we define the class refinement relation which captures the semantic constraints that must be imposed on classes to guarantee correctness of substitutability in all clients of the objects these classes instantiate. Clients of class instances are modelled as programs making an iterative choice over invocation of class methods, and we formally prove that when a class $C'$ refines a class $C$, substituting instances of $C'$ for instances of $C$ is refinement for the clients.

## 1 Introduction

The issue of correctness of object-oriented programs deserves close consideration in view of the popularity of this programming style and the necessity to enhance reliability of programs. Not only correctness is a crucial requirement for safety-critical systems, but also it is becoming increasingly important for distributed object-oriented systems and frameworks, which are composed by independent users and characterized by a late integration phase. We consider here a methodology for ensuring correctness of class-based statically-typed object-oriented systems.

Subtype polymorphism, which is generally recognized as central to object-orientation, is based on syntactic conformance of objects' methods and used for substituting subtype objects for supertype objects. In most object-oriented languages, such as Simula, Eiffel, and C++, *subclassing* or *implementation inheritance* forms a basis for subtype polymorphism, i.e. signatures of subclass methods automatically conform to those of superclass methods, and, syntactically, subclass instances can be substituted for superclass instances. As the mechanism of polymorphic substitutability is, to a great extent, independent of the mechanism of implementation reuse, languages like Java and Sather separate the subtyping and subclassing hierarchies.

With both approaches, typechecking can be used to verify syntactic conformance of subtype objects to their supertype objects. However, it has been recognized that, while certainly very useful, typechecking is insufficient to guarantee correctness of object substitutability. An attempt to establish behavioral conformance along with syntactic one has created a research direction known as *behavioral subtyping* [4, 26, 15, 16]. The essence of behavioral subtyping is to associate behavior with type signatures and to identify subtypes that conform to their supertypes not only syntactically but also semantically.

In our view subtyping is a mechanism for substituting objects with certain method signatures for other objects with conforming method signatures and, as such, is a purely syntactic concept. Behavior of objects, on the other hand, has little to do with their syntactic interfaces and is expressed in the specification of the objects' methods manipulating the objects' attributes. Most importantly, syntactic subtyping is decidable and can be checked by a computer, while behavior-preserving subtyping is undecidable. Hence, in our approach we separate syntactic subtyping from behavioral conformance of subtype objects to supertype objects. We consider classes to be the carriers of behavior and compare them for behavioral compatibility. Instances of one class are guaranteed to behave as expected from instances of another, more abstract, class if the more concrete class is a *refinement* of the more abstract. We give a definition of *class refinement*, which we regard as semantics of correct substitutability of subclass instances for superclass instances in clients. We formally prove that when a class $C'$ refines a class $C$, substituting instances of $C'$ for instances of $C$ is refinement for the clients.

Class refinement is orthogonal to subclassing. A class and its subclass may not be in refinement, and two classes can be in refinement even if one of them is not declared to be a subclass of the other. With separate interface inheritance and implementation inheritance hierarchies, a subclass may not even be intended for behavioral conformance with its superclass, as the substitution mechanism is completely independent of the reuse mechanism. Syntactic conformance of method signatures, however, is a prerequisite for class refinement, as it is meaningless to compare behavior of classes whose instances are not intended for substitution. For simplicity, we consider subclassing to be the basis for subtyping and, consequently, require that class refinement be established between a class and its declared subclasses. However, the same principles also apply to systems with separate subclassing and interface inheritance hierarchies, as we will explain in the concluding section.

We build a logical framework for reasoning about object-oriented programs as a conservative extension of the *refinement calculus* [30, 10], which is used for reasoning about correctness and refinement of imperative programs in a rigorous, mathematically precise manner. The refinement calculus is particularly suited for describing object-oriented programs because it allows us to describe classes at various abstraction levels, using *specification statements* along with ordinary executable statements. The notion of an *abstract class*, specifying behavior common to its subclasses, can be fully elaborated in this formalization, since the

state of class instances can be given using abstract mathematical constructions, like sets and sequences, and class methods can be described as nondeterministic statements, abstractly but precisely specifying the intended behavior. Versatility of the specification language that we use permits treating specifications and implementations in a uniform manner considering implementations to be just deterministic specifications.

Expressiveness of higher-order logic, which is the formal basis of the refinement calculus, allows us to define relations between classes, such as class refinement, entirely within logic. Reasoning about these relations can, therefore, be carried out completely formally, whereas formalizations based on first-order logic can only allow informal reasoning. The detailed elaboration of our formalization permits mechanized reasoning and mechanical verification, because, being so precisely defined, every concept can be formalized within a theorem proving environment such as HOL [18] or PVS [32].

## 2 Refinement Calculus Basics

We formalize objects, classes, and relationships between them in the refinement calculus, which is used for reasoning about correctness and refinement of imperative programs. Let us briefly introduce the main concepts of this formalism.

### 2.1 Predicates, Relations, and Predicate Transformers

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

A *predicate* over $\Sigma$ is a boolean function $p : \Sigma \to \mathit{Bool}$ which assigns a truth value to each state. The set of predicates on $\Sigma$ is denoted $\mathcal{P}\Sigma$. The *entailment ordering* on predicates is defined by pointwise extension, so that for $p, q : \mathcal{P}\Sigma$,

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

Conjunction $\cap$ and disjunction $\cup$ of (similarly-typed) predicates are also defined pointwise.

A *relation* from $\Sigma$ to $\Gamma$ is a function of type $\Sigma \to \mathcal{P}\Gamma$ that maps each state $\sigma$ to a predicate on $\Gamma$. We write $\Sigma \leftrightarrow \Gamma$ to denote a set of all relations from $\Sigma$ to $\Gamma$. This view of relations is isomorphic to viewing them as predicates on the cartesian space $\Sigma \times \Gamma$. A function $f : \Sigma \to \Gamma$ can always be lifted to a (deterministic) relation $|\, f \,| : \Sigma \leftrightarrow \Gamma$. Functional and relational compositions are defined in a standard way.

A *predicate transformer* is a function $S : \mathcal{P}\Gamma \to \mathcal{P}\Sigma$ from predicates to predicates. We write

$$\Sigma \mapsto \Gamma \;\widehat{=}\; \mathcal{P}\Gamma \to \mathcal{P}\Sigma$$

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

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

Product operators combine predicates, functions, relations, and predicate transformers by forming cartesian products of their state spaces. For example, 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 $S_2$. In addition to many other useful properties, presented, e.g., in [8, 9], the product operator preserves refinement:

$$S_1 \sqsubseteq S_1' \ \wedge\ S_2 \sqsubseteq S_2' \ \Rightarrow\ (S_1 \times S_2) \sqsubseteq (S_1' \times S_2')$$

For modelling subtype polymorphism and dynamic binding we employ *sum types*. The sum or disjoint union of two types $\Sigma$ and $\Gamma$ is written $\Sigma + \Gamma$. The types $\Sigma$ and $\Gamma$ are called *base types* of the sum in this case. Associated with the sum types, are the *injection functions* which map elements of the base type to elements of the summation

$$\iota_1 : \Sigma \to \Sigma + \Gamma \qquad \iota_2 : \Gamma \to \Sigma + \Gamma$$

and *projection relations* which relate elements of the summation with elements of its base types

$$\pi_1 : \Sigma + \Gamma \leftrightarrow \Sigma \qquad \pi_2 : \Sigma + \Gamma \leftrightarrow \Gamma$$

The projection is the inverse of the injection, so that $\pi_1^{-1} = |\, \iota_1 \,|$, where $|\, \iota_1 \,|$ is the injection function lifted to a relation. Since any element of $\Sigma + \Gamma$ comes either from $\Sigma$ or from $\Gamma$, but not both, the ranges of the injections *ran* $\iota_1$ and *ran* $\iota_2$ partition $\Sigma + \Gamma$. For $\sigma : \Sigma + \Gamma$, the projection $\pi_1$ will relate it to a unique $\sigma' : \Sigma$ only if $\sigma \in$ *ran* $\iota_1$, and similarly for $\pi_2$. Sum types, as well as product types, associate to the right, so that $\Sigma_1 + \Sigma_2 + \Sigma_3 = \Sigma_1 + (\Sigma_2 + \Sigma_3)$.

We define the type $\Sigma$ to be a subtype of $\Sigma'$, written $\Sigma <: \Sigma'$, if $\Sigma = \Sigma'$, or $\Sigma <: \Sigma_i'$, where $\Sigma' = \Sigma_1' + \ldots + \Sigma_n'$. For example, $\Sigma <: \Sigma + \Sigma'$ and, of course, $\Sigma + \Sigma' <: \Sigma + \Sigma'$. The subtype relation is reflexive, transitive, and antisymmetric. For any $\Sigma$ and $\Sigma'$ such that $\Sigma <: \Sigma'$, we can construct the corresponding injection function $\iota_\Sigma : \Sigma \to \Sigma'$ and projection relation $\pi_\Sigma : \Sigma' \leftrightarrow \Sigma$ in a straightforward way.

### 2.2 Specification Language

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 $\Sigma$ and final state in $\Gamma$ 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 $p$. A statement need not have identical initial and final state spaces, though if it does, we write $S : \Xi(\Sigma)$ instead of $S : \Sigma \mapsto \Sigma$ for the corresponding predicate transformer. Following an established tradition, we will from now on identify statements with the monotonic predicate transformers that they determine in this manner.

The total correctness assertion $p \ \{\!|\, S \,|\!\} \ q$ is said to hold if execution of the statement $S$ establishes the postcondition $q$ when started in the set of states $p$. The pair of state predicates $(p, q)$ is usually referred to as the pre- and postcondition specification of the statement $S$. Formally, the total correctness assertion $p \ \{\!|\, S \,|\!\} \ q$ is defined to be $p \subseteq S \ q$. The refinement ordering on predicate transformers models the notion of total-correctness preserving program refinement. For statements $S$ and $T$, the relation $S \sqsubseteq T$ holds if and only if $T$ satisfies any specification satisfied by $S$.

Predicate transformers form a complete lattice under the refinement ordering. The bottom element is the predicate transformer **abort** that maps each postcondition to the identically false predicate *false*, and the top element is the predicate transformer **magic** that maps each postcondition to the identically true predicate *true*. We know nothing about how **abort** is executed and it is never guaranteed to terminate. The **magic** statement 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. Intuitively, **magic** can be understood as an infinite *wait* statement, and, although not directly implementable, it serves as a convenient abstraction in manipulating program statements.

Conjunction $\sqcap$ and disjunction $\sqcup$ of (similarly-typed) predicate transformers are defined pointwise, e.g.,

$$(\sqcap \ i \in I \ \cdot \ S_i) \ q \ \widehat{=} \ (\cap \ i \in I \ \cdot \ S_i \ q)$$

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 free and aimed at establishing the postcondition.

Sequential composition of program statements is modelled by functional composition of predicate transformers and the program statement **skip** is modelled by the identity predicate transformer:

$$\begin{array}{lcl} (S;T) \ q & \widehat{=} & S \ (T \ q) \\ \textbf{skip} \ q & \widehat{=} & q \end{array}$$

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

$$\begin{array}{lcl} \langle f \rangle \ q \ \sigma & \widehat{=} & q \ (f \ \sigma) \\ \{P\} \ q \ \sigma & \widehat{=} & (\exists \ \gamma : \Gamma \ \cdot \ P \ \sigma \ \gamma \ \wedge \ q \ \gamma) \\ [\mathrm{P}] \ q \ \sigma & \widehat{=} & (\forall \gamma : \Gamma \ \cdot \ P \ \sigma \ \gamma \ \Rightarrow \ q \ \gamma) \end{array}$$

The functional update applies the function $f$ to the state $\sigma$ to yield the new state $f\ \sigma$. When started in a state $\sigma$, $\{P\}$ angelically chooses a new state $\gamma$ such that $P\ \sigma\ \gamma$ holds, while $[P]$ demonically chooses a new state $\gamma$ 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**.

Following [9], 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 $(\lambda u \cdot t)$ 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

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

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

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

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. Unfortunately, lambda terms do not maintain consistent naming of state components, due to the possibility of $\alpha$-conversion of bound variables. To enforce the naming consistency, we use the program variable notation, writing, e.g., $(\mathbf{var}\ x, y \cdot (x := x+y); (y := 0))$ to express that each function term is to be understood as a lambda abstraction over the bound variables $x, y$:

$$(\mathbf{var}\ x, y \cdot (x := x + y); (y := 0)) \ =\ (\lambda x, y \cdot x := x + y); (\lambda x, y \cdot y := 0)$$

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

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

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

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

Intuitively, the demonic assignment expresses an uncontrollable nondeterministic choice in selecting a new value $x'$ satisfying $b$, whereas the angelic assignment expresses a free choice. The angelic assignment can, e.g., be understood as a request to the user to supply a new value.

Our specification language also includes the *assertion* and the *assumption* statements, written $\{b\}$ and $[b]$ respectively, where $b$ is a predicate stating a condition on program variables. Both the assertion and the assumption behave as **skip** if $b$ is satisfied; otherwise, the assertion aborts, whereas the assumption behaves as **magic**.

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

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

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

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

A variant of iteration, the *iterative choice* [10, 6], allows the user to choose repeatedly an alternative that is enabled and have it executed until the user decides to stop:

$$\textbf{do } g_1 :: S_1 \lozenge \ldots \lozenge\ g_n :: S_n \textbf{ od } \mathrel{\widehat{=}} (\mu X \cdot \{g_1\}; S_1; X \sqcup \ldots \sqcup \{g_n\}; S_n; X \sqcup \textbf{skip})$$

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

Finally, the language supports blocks with *local variables*. Block beginning and end are modelled by demonic and functional updates respectively:

$$\begin{array}{ll} \textbf{enter } p & \mathrel{\widehat{=}} [\lambda u \cdot \lambda(x, u') \cdot p\,(x, u') \wedge u = u'] \\ \textbf{exit} & \mathrel{\widehat{=}} \langle \lambda(x, u) \cdot u \rangle \end{array}$$

Here $p$ is the predicate initializing the local variables. We can define a block introducing a program variable $x$ initialized according to a boolean expression $b$ as follows:

$$(\textbf{var } u \cdot \textbf{begin } (\textbf{var } x, u \cdot b); S; \textbf{end}) \mathrel{\widehat{=}} (\textbf{var } u \cdot \textbf{enter } (\textbf{var } x, u \cdot b); S; \textbf{exit})$$

When the variable declaration is clear from the context, we will for simplicity write just **begin var** $x \cdot b; S;$ **end**.

The program variable declaration can be propagated outside statements and distributed through sequential composition, so that, e.g.,

$$(\textbf{var } u \cdot [x := x' \,|\, x' \geq 0]; y := x) = [\textbf{var } u \cdot x := x' \,|\, x' \geq 0]; (\textbf{var } u \cdot y := x)$$

When the variable declaration is clear from the context, we will omit it.

As suggested in [10], we can define an interactive executable language, where a statement is built by the following syntax:

| | | | |
|---|---|---|---|
| $S$ | ::= | **abort** | *(abortion)* |
| | \| | **skip** | *(skip)* |
| | \| | $\{b\}$ | *(assertion)* |
| | \| | $x_1, \ldots, x_m := t_1, \ldots, t_m$ | *(assignment)* |
| | \| | $\{x := x' \mid b\}$ | *(angelic assignment)* |
| | \| | $S_1; S_2$ | *(sequential composition)* |
| | \| | **if** $g$ **then** $S_1$ **else** $S_2$ **fi** | *(conditional statement)* |
| | \| | **while** $g$ **do** $S$ **od** | *(iteration)* |
| | \| | **do** $g_1 :: S_1 \lozenge \ldots \lozenge g_n :: S_n$ **od** | *(iterative choice)* |
| | \| | **begin var** $x \cdot b; S;$ **end** | *(block with local variables)* |

When extended with miraculous statements, this executable language becomes a general specification language:

| | | | |
|---|---|---|---|
| $S$ | ::= | $\ldots$ | |
| | \| | **magic** | *(magic)* |
| | \| | $[b]$ | *(assumption)* |
| | \| | $[x := x' \mid b]$ | *(demonic assignment)* |

In the next section we explain how to extend this language with object-oriented constructs.

### 2.3 Data Refinement

Data refinement is a general technique by which one can change the state space in a refinement. For statements $S : \Xi(\Sigma)$ and $S' : \Xi(\Sigma')$, let $R : \Sigma' \leftrightarrow \Sigma$ be a relation between the state spaces $\Sigma$ and $\Sigma'$. According to [7], 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 notion of data refinement is the standard one, often referred to as forward data refinement or downward simulation. 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 $[R^{-1}]$ are each others inverses, in the sense that $\{R\};[R^{-1}] \sqsubseteq$ **skip** and **skip** $\sqsubseteq [R^{-1}];\{R\}$. Further on we will abbreviate $\{R\};S;[R^{-1}]$ by $S \downarrow R$ and $[R^{-1}];S';\{R\}$ by $S' \uparrow R$. 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 [10, 30].

## 3 Modelling Object-Oriented Constructs

We focus on modelling class-based statically-typed object-oriented languages, which form the mainstream of object-oriented programming. Accordingly, we take a view that *objects* are instances of *classes*. A class describes objects with similar behavior through specifying their *interface*. The interface represents signatures of object methods, i.e. the method name and the types of value and result parameters. For simplicity, we consider all object attributes as private or hidden, and all methods as public or visible to clients of the object. We consider an *object type* to be the type of object attributes having an additional unique global identifier distinguishing this object type from the others.

A class can be given by the following declaration:

$$
\begin{array}{l}
C \;=\; \textbf{class} \\
\quad \textbf{var}\ attr_1 : \Sigma_1, \ldots, attr_m : \Sigma_m \\
\quad C\,(\textbf{val}\ x_0 : \Gamma_0) \;=\; K, \\
\quad Meth_1\,(\textbf{val}\ x_1 : \Gamma_1, \textbf{res}\ y_1 : \Delta_1) \;=\; M_1, \\
\quad \ldots \\
\quad Meth_n\,(\textbf{val}\ x_n : \Gamma_n, \textbf{res}\ y_n : \Delta_n) \;=\; M_n \\
\textbf{end}
\end{array}
$$

This class specifies the interface $Meth_1\,(\textbf{val} : \Gamma_1, \textbf{res} : \Delta_1), \ldots, Meth_n\,(\textbf{val} : \Gamma_n, \textbf{res} : \Delta_n)$, where $\Gamma_i$ and $\Delta_i$ are the types of value and result parameters respectively. A method may be parameterless, with both $\Gamma_i$ and $\Delta_i$ being the unit type (), or may have only value or only result parameters.

The class $C$ describes (possibly abstract) attributes, specifies the way the objects are created, and gives a (possibly nondeterministic) specification for each method. Class attributes $(attr_1, \ldots, attr_m)$ have the corresponding types $\Sigma_1$ through $\Sigma_m$. Apart from the declared attributes, every class has an implicit constant attribute $type : String$ which contains the name of the object type specified by this class. This constant identifier is unique in every class. We will use an identifier *self* for the tuple $(attr_1, \ldots, attr_m, type)$. The type of *self* is then $\Sigma \;=\; \Sigma_1 \times \ldots \times \Sigma_m \times String$. We impose a non-recursiveness restriction on $\Sigma$ so that none of $\Sigma_i$ is equal to $\Sigma$. This restriction allows us to stay within the simple-typed lambda calculus.

A class *constructor* is used to instantiate objects and has the same name as the class. Due to the fact that the constructor concerns object creation rather than object functionality, it is associated with the class rather than with the specified interface. Semantically, the constructor is equivalent to a stand-alone global procedure which is associated with the class for encapsulation reasons. The statement $K : \Gamma_0 \mapsto \Sigma \times \Gamma_0$, representing the body of the constructor, introduces the attributes into the state space and initializes them using the value parameter(s) $x_0 : \Gamma_0$. Methods $Meth_1$ through $Meth_n$, specified by bodies $M_1, \ldots, M_n$, operate on the attributes and realize the object functionality. Every statement $M_i$ is, in general, of type $\Xi(\Sigma \times \Gamma_i \times \Delta_i)$. The identifier *self* acts in

this model as an implicit result parameter of the constructor and an implicit variable parameter of the methods.

Being declared as such, the class $C$ is a tuple $(K, M_1, \ldots, M_n)$. Further on we will refer to $K$ as the constructor and to $M_1, \ldots, M_n$ as the methods, unless stated otherwise. The object type specified by a class can always be extracted from the class and we do not need to declare it explicitly. We use $\tau(C)$ to denote the type of objects generated by the class $C$; as such, $\tau(C)$ is just another name for $\Sigma$.

### 3.1 Object Instantiation and Method Invocations

Initialization of a new variable $c$ of object type $\tau(C)$ takes invoking the corresponding class constructor:

$$\begin{array}{rcl} \textbf{create var } c.C(e) & \widehat{=} & \textbf{enter } (\textbf{var } x_0, u \cdot x_0 = e); K \times \textbf{skip}; \\ & & \textbf{enter } (\textbf{var } c, (\mathit{self}, x_0), u \cdot c = \mathit{self}); \mathit{Swap}; \textbf{exit} \end{array}$$

where $\mathit{Swap} = \langle \lambda x, y, z \cdot y, x, z \rangle$. A variable $x_0 : \Gamma_0$ is first entered into the state space and initialized with the value of $e$. Then the constructor $K$ is "injected" into the global state space, skipping on the global state component $u$. The next statement enters a variable $c$ and initializes it to the value of the state component $\mathit{self}$. The state rearranging $\mathit{Swap}$ makes the pair $(\mathit{self}, x_0)$ the first state component before exiting it from the block. Naturally, a variable of an object type initialized in this way can be local to a block:

$$\textbf{create var } c.C(e); S; \textbf{end} \;\widehat{=}\; \textbf{create var } c.C(e); S; \textbf{exit}$$

Invocation of a method $\mathit{Meth}_i(\textbf{val } x_i : \Gamma_i, \textbf{res } y_i : \Delta_i)$ on an object $c$ instantiated by class $C$ is modelled by

$$\begin{array}{rcl} (\textbf{var } c, u \cdot c.\mathit{Meth}_i\,(g_i, d_i)) & \widehat{=} & \textbf{begin } (\textbf{var } (\mathit{self}, x_i, y_i), c, u \cdot \\ & & \quad \mathit{self} = c \wedge x_i = g_i); \\ & & \quad M_i \times \textbf{skip}; c, d_i := \mathit{self}, y_i; \\ & & \textbf{end} \end{array}$$

where $u : \Phi$ are global variables including $d_i : \Delta_i$, and $g_i : \Gamma_i$ is some expression.

### 3.2 Modelling Object Clients

A client program using an object $c : \tau(C)$ does so by invoking its methods. Every time a client has a choice of which method 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 client may repeatedly choose a particular option which is enabled and have it executed. The client decides on its own when it is willing to stop choosing options. Such an iterative choice of method invocations, followed by arbitrary

statements not affecting the object directly, describes all the actions the client program might undertake:

$$(\mathbf{var}\ c, u\ \cdot\ \mathbf{begin\ var}\ l\ \cdot\ b; \mathbf{do}\ \Diamond_{i=1}^{n} q_i :: c.Meth_i(g_i, d_i); L_i\ \mathbf{od}; \mathbf{end})$$

Here $u : \Phi$ are global variables including $d_i$, $l : \Lambda$ are some local variables initialized according to $b$, predicates $q_1 \ldots q_n$ are the asserted conditions on the state, and statements $L_1$ through $L_n$ are arbitrary. The initialization $b$, the assertions $q_1 \ldots q_n$, and the statements $L_1, \ldots, L_n$ do not refer to $c$, which is justified by the assumption that the object state is encapsulated. Therefore, $c$ is not free in $b$, every $q_i$ is of the form $q_i' \times \mathit{true} \times q_i''$, with $q_i' : \mathcal{P}\Lambda$ and $q_i'' : \mathcal{P}\Phi$, and every $L_i$ is of the form $L_i' \times \mathbf{skip} \times L_i''$, with $L_i' : \Xi(\Lambda)$ and $L_i'' : \Xi(\Phi)$.

Objects can, of course, be their own clients, and any method of class $C$ can invoke other methods of $C$, including itself. The most general behavior of a method $Meth_j(\mathbf{val}\ x_j : \Gamma_j, \mathbf{res}\ y_j : \Delta_j)$ can therefore be described by

$$(\mathbf{var}\ \mathit{self}, x_j, y_j\ \cdot\ \mathbf{begin\ var}\ l\ \cdot\ b; \mathbf{do}\ \Diamond_{i=1}^{n} q_i :: \mathit{self}.Meth_i(g_i, d_i); L_i\ \mathbf{od}; \mathbf{end})$$

where *self* is free in $b$, all $q_i$ can directly access *self*, and all $L_i$ can directly access and modify it. For example, a method self-calling itself is an instance of this general definition. The meaning of such a recursive method is given by the least fixpoint of the corresponding function with respect to the refinement ordering.

### 3.3 Modelling Dynamic Objects

Following [10], we model pointers to class instances as indices of an array of these class instances. Natural numbers can be used as the index set of such an array, and we can declare a program variable *heap* to contain the whole dynamic data structure:

$$\mathbf{var}\ \mathit{heap} : \mathbf{array}\ \mathit{Nat}\ \mathbf{of}\ \tau(C)$$

The type of pointers to instances of class $C$ can then simply be defined as the type of natural numbers. The index value 0 can be used as a *nil* pointer. New pointer (index) values can be generated dynamically, on demand, by keeping a separate counter *new* for the next unused index:

$$\begin{array}{l}\mathbf{type\ pointer\ to}\ \tau(C)\ \mathrel{\widehat{=}}\ \mathit{Nat};\\ \mathbf{var}\ \mathit{nil}, \mathit{new} : \mathbf{pointer\ to}\ \tau(C) := 0, 1;\end{array}$$

Dynamic creation of an object of type $\tau(C)$ and association of a pointer $p : \mathbf{pointer\ to}\ \tau(C)$ with this object are modelled as follows:

$$p := \mathbf{new}\ C(e)\ \mathrel{\widehat{=}}\ p, \mathit{new} := \mathit{new}, \mathit{new}+1; \mathbf{create\ var}\ c.C(e); \mathit{heap}\,[p] := c; \mathbf{end}$$

To keep the array of class instances implicit, the notation $p\uparrow$ is used for the access operation $\mathit{heap}\,[p]$, so that $p\uparrow := e$ stands for the update operation $\mathit{heap}\,[p] := e$, and $p\uparrow.Meth_i(g_i, d_i)$ stands for the method invocation $\mathit{heap}\,[p].Meth_i(g_i, d_i)$.

### 3.4 Example

As an example of class specifications consider a text-editing application in which a text document may be viewed and possibly changed in several different windows. Whenever the text is changed in any of the windows in response to, e.g., user actions, all the other windows displaying the same text are notified of this change and updated to achieve consistency in presenting the data. We specify these interactions in Fig. 1. Views are responsible for presenting the textual data in various windows and providing operations for changing it. A client of a text

```
TextDoc = class
  var text : String,
      views : set of pointer to τ(View)

  TextDoc (val t : String) =
    enter var text, views ·
      text = t ∧ views = {},

  AddView (val v : pointer to τ(View)) =
    [v ≠ nil]; views := views ∪ {v},

  AddText (val t : String) =
    text := text⌢t; self.Notify(),

  GetText (res t : String) = t := text,

  Notify () =
    begin var (vs, v) · vs = views;
      while vs ≠ {} do
        [v := v' | v' ∈ vs];
        vs := vs \ v; v↑.Update()
      od;
    end
end
```

```
View = class
  var txt : String,
      doc : pointer to τ(TextDoc)

  View (val d : pointer to
    τ(TextDoc)) =
    [d ≠ nil];
    enter var txt, doc · doc = d;
      doc↑.GetText(txt),

  AddText (val t : String) =
    doc↑.AddText(t),

  Update () = doc↑.GetText(txt)
end
```

```
TextEditor = class
  ...
  OpenTextDoc () =
    begin var d : pointer to τ(TextDoc); d := new TextDoc("");
      begin var (v1, v2 : pointer to τ(View));
        v1 := new View(d); d↑.AddView(v1);
        v2 := new View(d); d↑.AddView(v2);
      end;
    end,
  ...
end
```

**Fig. 1.** Example of class specification

editor can open a new text document, displayed in two different windows, by invoking the method *OpenTextDoc*. When one of these views is asked to add some text to the existing one, the method *AddText* is invoked. This method forwards the request to the method *AddText* of the current view's *doc* attribute. After the new text is concatenated to the old one, all views on the document are notified of the change and asked to update their state.

A point to notice here is that such a specification, although being rather abstract, precisely documents the behavior of the involved parties without resorting to verbal descriptions. The necessity for a precise documentation was pointed out in [17] when discussing the Observer pattern which our example follows. In particular, it was advised to document which *Subject* (in our case *TextDoc*) methods trigger modifications. Also, the place of the *Notify* method invocation can be fixed in the specification. We chose to call it from the state-modifying *AddText* method of *TextDoc* after the change. Alternatively, this method could be called from *AddText* of *View* after invoking the corresponding method on the *doc* attribute. The advantages and disadvantages of both approaches are discussed in [17], we only would like to note that fixing the invocation of this method in the specification helps avoiding the problem of calling this method at inappropriate times or, even worse, not calling it at all from the overridden methods in subclasses of *TextDoc* and *View*.

### 3.5 Subclassing

New classes can be constructed from existing ones by *inheriting* some or all of their attributes and methods, possibly *overriding* some attributes and methods, and adding extra methods. This mechanism is known as *subclassing*.[1]

A class constructed from $C$ by subclassing is declared as follows:

$$
\begin{aligned}
C' \;=\; & \textbf{subclass of } C \\
& \textbf{var } attr_1 : \Sigma_1, \ldots, attr_i : \Sigma_i, attr'_1 : \Sigma'_1, \ldots, attr'_p : \Sigma'_p \\
& C' \,(\textbf{val } x'_0 : \Gamma'_0) \;=\; K', \\
& \mathit{Meth}_1 \,(\textbf{val } x_1 : \Gamma_1, \textbf{res } y_1 : \Delta_1) \;=\; M'_1, \\
& \ldots \\
& \mathit{Meth}_k \,(\textbf{val } x_k : \Gamma_k, \textbf{res } y_k : \Delta_k) \;=\; M'_k, \\
& \mathit{NMeth}_1 \,(\textbf{val } u_1 : \Phi_1, \textbf{res } v_1 : \Psi_1) \;=\; N_1, \\
& \ldots \\
& \mathit{NMeth}_p \,(\textbf{val } u_p : \Phi_p, \textbf{res } v_p : \Psi_p) \;=\; N_p
\end{aligned}
$$

A subclass may have attributes different from those of its superclass, inheriting $attr_1, \ldots, attr_i$ and overriding $attr_{i+1}, \ldots, attr_m$ by $attr'_1, \ldots, attr'_p$. The class constructor is not inherited from the superclass, but rather redefined in every subclass. The statements $M'_1, \ldots, M'_k$ override the corresponding definitions of

[1] We prefer the term *subclassing* to *implementation inheritance* because the latter literally means reuse of existing methods and does not, as such, suggest the possibility of method overriding.

$Meth_1, \ldots, Meth_k$ given in $C$. The methods $NMeth_1, \ldots, NMeth_p$ with bodies given by $N_1, \ldots, N_p$ are new.

When a subclass $C'$ inherits all attributes of its superclass $C$ without overriding them, methods defined in the superclass can be invoked from methods $M'_1, \ldots, M'_k, N_1, \ldots, N_p$ using a special identifier *super*. For example, a method $Meth_i\,(\mathbf{val}\ x_i : \Gamma_i, \mathbf{res}\ y_i : \Delta_i)$ defined in $C$ by $M_i$ can be super-called inside any of $M'_1, \ldots, M'_k, N_1, \ldots, N_p$ by writing $super.Meth_i(g_i, d_i)$, where $g_i$ and $d_i$ are some value and result arguments respectively. Such a super-call corresponds to executing statement $M_i \times \mathbf{skip}$, with **skip** operating on the additional attributes of $C'$. Methods of the superclass can also be inherited as a whole. In this case their redefinition in the subclass corresponds to super-calling them, passing value and result parameters as arguments. Following the standard convention, we omit such inherited methods from the subclass declaration.

We view subclassing as a syntactic relation on classes, since subclasses are distinguished by an appropriate declaration. Subclassing implies conformance of interfaces, meaning that the interface specified by a subclass is an extension of the interface specified by the superclass, having at least all the method signatures of the latter and possibly introducing new ones. In an extended interface the inherited method signatures can be modified to allow more flexibility in polymorphic object substitutability. In the next section we explain this can be achieved.

### 3.6 Modelling Subtype Polymorphism and Dynamic Binding

To model subtype polymorphism, we allow object types to be sum types. The idea is to group together an object type of a certain class and object types of all its subclasses, to form a polymorphic object type. A variable of such a sum type can be instantiated to any base type of the summation, in other words, to any object instantiated by a class whose object type is the base type of the summation.

A sum of object types, denoted by $\tau(C)^+$ is defined to be such that its base types are $\tau(C)$ and all the object types of subclasses of $C$. For example, if $D$ is the only subclass of $C$ with the object type $\tau(D)$, then $\tau(C)^+ = \tau(C) + \tau(D)$, and we have that

$$\tau(C) <: \tau(C)^+ \text{ and } \tau(D) <: \tau(C)^+$$

The diagram in Fig. 2 illustrates the relationship between subclassing and subtyping hierarchies. The subclassing hierarchy on the left-hand side corresponds to the subtyping hierarchy on the right-hand side, with the arrows meaning "is the type of instances of".

Suppose a method $Meth_i\,(\mathbf{val}\ x_i : \Gamma_i, \mathbf{res}\ y_i : \Delta_i)$ is specified in both $C$ and $D$. An invocation of this method on an object $p$ of type $\tau(C)^+$ is modelled as a choice between two alternatives each calling $Meth_i$, but one assuming that $p$ is instantiated by class $C$ and the other assuming instantiation by class $D$:

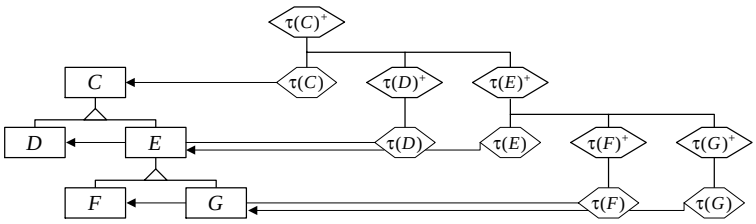



**Fig. 2.** The relationship between subclassing and subtyping hierarchies

$$p.Meth_i(g_i, d_i) \;\widehat{=}\; \begin{pmatrix} \{p \in ran\ \iota_{\tau(C)}\}; \\ \textbf{begin var}\ c \cdot \pi_{\tau(C)}\ p\ c; \\ \quad c.Meth_i(g_i, d_i); \\ \quad p := \iota_{\tau(C)}\ c; \\ \textbf{end} \end{pmatrix} \sqcup \begin{pmatrix} \{p \in ran\ \iota_{\tau(D)}\}; \\ \textbf{begin var}\ d \cdot \pi_{\tau(D)}\ p\ d; \\ \quad d.Meth_i(g_i, d_i); \\ \quad p := \iota_{\tau(D)}\ d; \\ \textbf{end} \end{pmatrix}$$

When $p$ is an instance of $C$, the assertion $\{p \in ran\ \iota_{\tau(C)}\}$ skips, and the method $Meth_i$ is invoked on the object $c$ corresponding to the projection $\pi_{\tau(C)}$ of $p$. Afterwards, the value of $c$ is injected to be of type $\tau(C)^+$ and used to update $p$. The invocation $c.Meth_i(g_i, d_i)$ is modelled as in Sec. 3.1. The assertion that $p$ is an instance of $D$ is false, aborting the second alternative of the angelic choice, since for all predicates $q$, $\{q\} = \textbf{if}\ q\ \textbf{then skip else abort fi}$, and for all statements $S$, $\textbf{abort}; S = \textbf{abort}$. The angelic choice between the two statements is then equal to the first alternative, since $S \sqcup \textbf{abort} = S$. Similarly, when $p$ is an instance of $D$, the first alternative aborts, and the choice is equal to the second alternative. As such, the choice between the alternatives is deterministic.

A polymorphic variable $p : \tau(C)^+$ can be instantiated by either class $C$ or its subclass. In practice we occasionally would like to underspecify which particular class instantiates $p$. We can express this by using a demonic choice of possible instantiations. We will write $p.C^+(e)$, where $e$ is any expression of type $\Gamma_0$, for this kind of polymorphic instantiation:

$$\textbf{create var}\ p.C^+(e) \;\widehat{=}\; \begin{pmatrix} \textbf{create var}\ c.C(e); \\ \textbf{begin var}\ p \cdot \pi_{\tau(C)}\ p\ c; \\ \quad Swap; \\ \textbf{end} \end{pmatrix} \sqcap \begin{pmatrix} \textbf{create var}\ d.D(e); \\ \textbf{begin var}\ p \cdot \pi_{\tau(D)}\ p\ d; \\ \quad Swap; \\ \textbf{end} \end{pmatrix}$$

Intuitively, the demonic choice can be interpreted as underspecification, which would eventually be eliminated in a refinement. Note that since the demonic choice is refined by either alternative, we have that any concrete instantiation refines the polymorphic instantiation. Modelling of both the invocation of a method on a polymorphic variable and the instantiation of such a variable generalizes to class hierarchies with several classes in a straightforward way, recursively.

Being equipped with subtype polymorphism, we can allow overriding methods in a subclass to be *generalized* on the type of value parameters or *specialized* on the type of result parameters. In the first case this type redefinition is *contravariant* and in the second *covariant*.[2] When one interface is the same as the other, except that it can redefine contravariantly value parameter types and covariantly result parameter types, this interface *conforms* to the original one. For example, $Meth_i\,(\mathbf{val}\ x_i : \Gamma_i, \mathbf{res}\ y_i : \Delta_i)$ specified in class $C$ could be redefined in its subclass $D$ so that the value parameters are of type $\Gamma_i'$, such that $\Gamma_i <: \Gamma_i'$, and the result parameters are of type $\Delta_i'$, such that $\Delta_i' <: \Delta_i$. An invocation of such a method would then need to adjust the input arguments and the result using the corresponding projections and injections. For example, invocation of $Meth_i\,(\mathbf{val}\ x_i' : \Gamma_i', \mathbf{res}\ y_i' : \Delta_i')$ specified by $M_i'$ in class $D$ on an object $d : \tau(D)$ with input argument $g_i : \Gamma_i$ and result argument $d_i : \Delta_i$ is modelled by

$$\begin{array}{rcl} d.Meth_i\,(g_i, d_i) & \widehat{=} & \mathbf{begin\ var}\ self, x_i', y_i' \cdot self = d \wedge x_i' = \iota_{\Gamma_i}\ g_i; \\ & & \qquad M_i' \times \mathbf{skip}; d, d_i := self, \iota_{\Delta_i'}\ y_i'; \\ & & \mathbf{end} \end{array}$$

Here the value parameter $x_i'$ is initialized with the value of the input argument injected into the type $\Gamma_i'$. Similarly, the value of the method result $y_i'$, being of type $\Delta_i'$, cannot be directly assigned to the variable $d_i : \Delta_i$ and is injected into the type $\Delta_i'$ using the corresponding injection function.

Subtype polymorphism extends in a natural way to pointer types. A sum of pointer types $\mathbf{pointer\ to}\ \tau(C)^+$ is defined to be such that its base types are $\mathbf{pointer\ to}\ \tau(C)$ and all the pointer types to subclasses of $C$. A variable of such a polymorphic pointer type cannot be instantiated using **new**, because the latter is defined to generate a new index in some array of class instances associated with a base pointer type. A polymorphic pointer variable can, however, be assigned a value of an existing index to one of the arrays $heap_C, heap_{C_1}, \ldots, heap_{C_n}$, which keep instances of $C$ and instances of its subclasses $C_1, \ldots, C_n$. Before assignment, this pointer value should be injected into the corresponding sum type.

Dynamic binding of self-referential methods can occur only when a subclass inherits all attributes of its superclass without overriding them. Essentially, a super-call to a method self-calling other methods of the same class resolves the latter with the definitions of the self-called methods in the class which originated the super-call.

Suppose that class $C'$ inherits all attributes of its superclass $C$ and has some new attributes, so that the first projection of $self : \Sigma \times \Sigma'$ in $C'$ is equal to $self : \Sigma$ in $C$. The general behavior of a self-referential method $Meth_j(\mathbf{val}\ x_j : \Gamma_j, \mathbf{res}\ y_j : \Delta_j)$ in $C$ can be described by

$$(\mathbf{var}\ self, x_j, y_j \cdot \mathbf{begin\ var}\ l \cdot b; \mathbf{do}\ \Diamond_{i=1}^{n} q_i :: self.Meth_i(g_i, d_i); L_i\ \mathbf{od}; \mathbf{end})$$

[2] For a more extensive explanation of covariance and contravariance see, e.g., [1].

```
Bag = class
  var b : bag of Char
  Bag() = enter var b · b = ⟦⟧,
  Add(val c : Char) = b := b ∪ ⟦c⟧,
  AddAll(val nb : bag of Char) =
    while nb ≠ ⟦⟧ do
      begin var c · c ∈ nb;
        self.Add(c); nb := nb − ⟦c⟧;
      end
    od
end
```

```
CountingBag = subclass of Bag
  var b : bag of Char, n : Nat
  CountingBag() =
    enter var b, n · b = ⟦⟧ ∧ n = 0,
  Add(val c : Char) =
    n := n + 1; super.Add(c)
end
```

**Fig. 3.** Example of subclassing with dynamic binding of self-referential methods

Let the behavior of this method be given in $C'$ by $super.Meth_j(x_j, y_j)$. Then the super-call is defined to invoke the self-called methods on the current *self* object:

$$(\textbf{var } self, x_j, y_j \cdot super.Meth_j(x_j, y_j)) \;\widehat{=}\; (\textbf{var } self, x_j, y_j \cdot \textbf{ begin } \langle\rho\rangle\ (\textbf{var } l, self, x_j, y_j \cdot b); \textbf{ do } \Diamond_{i=1}^{n}\langle\rho\rangle\ q_i :: self.Meth_i(g_i, d_i); L_i\downarrow \mid \rho \mid \textbf{ od}; \textbf{ end})$$

where $\rho = (\lambda\, x, (y, y'), z \cdot x, y, z)$ is the projection function removing the extra attribute of *self*. Applying the functional update $\langle\rho\rangle$ to the predicates $(\textbf{var } l, self, x_j, y_j \cdot b)$ and $q_1, \ldots, q_n$, and wrapping the statements $L_1, \ldots, L_n$ in the relation $\mid \rho \mid$, coerces them to operate on the extended state space $\Lambda \times (\Sigma \times \Sigma') \times \Gamma_j \times \Delta_j$. As such, this is a technicality not changing the meaning of the corresponding statements. Self-calls to $Meth_1, \ldots, Meth_n$ are resolved with the definitions of these methods given in $C'$.

As an example consider specifications of *Bag* and *CountingBag* presented in Fig. 3. The subclass *CountingBag* inherits the only attribute of its superclass *Bag*, representing a bag of characters, and adds a counter of bag elements. The method *Add* overrides the corresponding method of the superclass by incrementing the counter and then super-calling *Add* of *Bag*.

The method *AddAll* joins two bags by self-calling *Add*. The self-call in the definition of *AddAll* in *Bag* is resolved by substituting the body of *Add* as defined in *Bag*:

```
Bag :: AddAll(val nb : bag of Char) = while nb ≠ ⟦⟧ do
                                        begin var c · c ∈ nb;
                                          b := b ∪ ⟦c⟧; nb := nb − ⟦c⟧;
                                        end
                                      od
```

The definition of the method *AddAll* in *CountingBag* exemplifies dynamic binding of self-referential methods. First of all, inheriting this method from *Bag* corresponds to super-calling it:

$$\mathit{CountingBag} :: \mathit{AddAll}(\mathbf{val}\ nb : \mathbf{bag\ of}\ \mathit{Char}) \ = \ \mathit{super}.\mathit{AddAll}(nb)$$

According to the definition of a super-called method involving self-calls, we then have that for $\mathit{self} = (b, n)$, $\mathit{super}.\mathit{AddAll}(nb)$ is equal to

$$\begin{array}{l}(\mathbf{var}\ (b,n), nb \cdot\ \mathbf{while}\ \langle\rho\rangle\ (\mathbf{var}\ b, nb \cdot\ nb \neq [\![]\!])\ \mathbf{do} \\ \qquad\qquad \mathbf{begin}\ \langle\rho\rangle\ (\mathbf{var}\ c, b, nb \cdot\ c \in nb); \\ \qquad\qquad\quad \mathit{self}.\mathit{Add}(c); (\mathbf{var}\ c, b, nb \cdot\ nb := nb - [\![c]\!]) \downarrow \mid \rho \mid; \\ \qquad\qquad \mathbf{end} \\ \qquad\quad \mathbf{od})\end{array}$$

which, using the definitions of $\rho, \downarrow$, and functional update, is equal to

$$\begin{array}{l}(\mathbf{var}\ (b,n), nb \cdot\ \mathbf{while}\ (\mathbf{var}\ (b,n), nb \cdot\ nb \neq [\![]\!])\ \mathbf{do} \\ \qquad\qquad \mathbf{begin}\ (\mathbf{var}\ c, (b,n), nb \cdot\ c \in nb); \\ \qquad\qquad\quad \mathit{self}.\mathit{Add}(c); (\mathbf{var}\ c, (b,n), nb \cdot\ nb := nb - [\![c]\!]); \\ \qquad\qquad \mathbf{end} \\ \qquad\quad \mathbf{od})\end{array}$$

The self-call, being on $\mathit{self} = (b, n)$, is resolved with the definition of *Add* in *CountingBag*.

## 4 Class Refinement

When a subclass overrides some methods of its superclass, there are no guarantees that its instances will deliver the same or refined behavior as the instances of the superclass. Unrestricted method overriding in a subclass can lead to arbitrary behavior of its instances. When used in a superclass context, such subclass instances can invalidate their clients. To avoid such problems, we would like to ensure that whenever $C'$ is subclassed from $C$, clients using objects instantiated by $C$ can safely use objects instantiated by $C'$ instead. First we consider class refinement between two classes having the same number of methods and then extend the definition to account for additional methods defined in a subclass.

### 4.1 Class Refinement Without New Methods

Suppose classes $C$ and $C'$ specify interfaces

$$\begin{array}{l}\mathit{Meth}_1\,(\mathbf{val}\ : \Gamma_1, \mathbf{res}\ : \Delta_1), \ldots, \mathit{Meth}_n\,(\mathbf{val}\ : \Gamma_n, \mathbf{res}\ : \Delta_n) \text{ and} \\ \mathit{Meth}_1\,(\mathbf{val}\ : \Gamma'_1, \mathbf{res}\ : \Delta'_1), \ldots, \mathit{Meth}_n\,(\mathbf{val}\ : \Gamma'_n, \mathbf{res}\ : \Delta'_n)\end{array}$$

respectively. Let $C$ and $C'$ be modelled by tuples $(K, M_1, \ldots, M_n)$ and $(K', M'_1, \ldots, M'_n)$, where $K : \Gamma_0 \mapsto \Sigma \times \Gamma_0$ and $K' : \Gamma'_0 \mapsto \Sigma' \times \Gamma'_0$ are the class

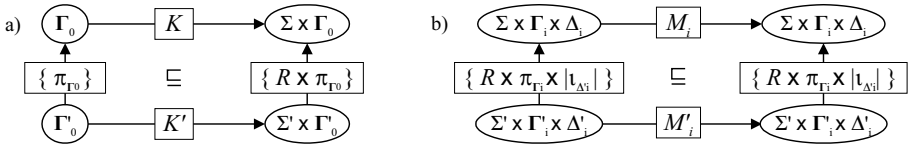



**Fig. 4.** Constructor refinement a) and method refinement b)

constructors, and all $M_i : \Xi(\Sigma \times \Gamma_i \times \Delta_i)$ and $M'_i : \Xi(\Sigma' \times \Gamma'_i \times \Delta'_i)$ are the corresponding methods. The value parameter types of the constructors and the methods in $C'$ are either the same or contravariant, so that $\Gamma_0 <: \Gamma'_0$ and $\Gamma_i <: \Gamma'_i$, and the result parameter types of its methods are either the same or covariant, $\Delta'_i <: \Delta_i$.

Let $R : \Sigma' \leftrightarrow \Sigma$ be a relation coercing attribute types of $C'$ to those of $C$, so that $R$ is of the form $(\lambda c \cdot \{a \mid R\ c\ a\})$. The refinement of class constructors $K$ and $K'$ with respect to $R$ is defined as follows:

$$K \sqsubseteq_R K' \quad \widehat{=} \quad \{\pi_{\Gamma_0}\}; K \quad \sqsubseteq \quad K'; \{R \times \pi_{\Gamma_0}\} \qquad (\textit{constructor refinement})$$

where $\pi_{\Gamma_0}$ is the projection relation coercing $\Gamma'_0$ to $\Gamma_0$. The commuting diagram in Fig. 4 (a) illustrates constructor refinement.

The refinement of all corresponding methods $M_i$ and $M'_i$ with respect to the relation $R$ is defined by

$$M_i \sqsubseteq_R M'_i \quad \widehat{=} \quad M_i \downarrow (R \times \pi_{\Gamma_i} \times |\iota_{\Delta'_i}|) \quad \sqsubseteq \quad M'_i \qquad (\textit{method refinement})$$

where $\pi_{\Gamma_i} : \Gamma'_i \leftrightarrow \Gamma_i$ projects the corresponding value parameters, and $|\iota_{\Delta'_i}| : \Delta'_i \leftrightarrow \Delta_i$ injects the corresponding result parameters. Obviously, when $\Gamma_i = \Gamma'_i$, the projection relation $\pi_{\Gamma_i}$ is the identity relation *Id*. The same holds when $\Delta_i = \Delta'_i$, namely, $|\iota_{\Delta'_i}| = Id$. The commuting diagram in Fig. 4 (b) illustrates method refinement.

**Definition 1.** *For classes $C = (K, M_1, \ldots, M_n)$ and $C' = (K', M'_1, \ldots, M'_n)$, class refinement $C \sqsubseteq C'$ is defined as follows:*

$$C \sqsubseteq C' \quad \widehat{=} \quad (\exists R \cdot K \sqsubseteq_R K' \wedge (\forall i \mid 1 \leq i \leq n \cdot M_i \sqsubseteq_R M'_i))$$

The class refinement relation is reflexive and transitive. This definition of class refinement is constructive in the sense that given two classes we can always check whether these classes are in refinement. However, from this definition alone we cannot make any conclusions about the behavior of clients using instances of classes that are in refinement. Before presenting a theorem which relates class refinement to object substitutability in clients, let us introduce two useful lemmas.

**Lemma 1.** *Let classes $C$ and $C'$ have constructors $K : \Gamma_0 \mapsto \Sigma \times \Gamma_0$ and $K' : \Gamma'_0 \mapsto \Sigma' \times \Gamma'_0$ with $\Gamma_0 <: \Gamma'_0$. In a global state $u : \Phi$, for any relation $R : \Sigma' \leftrightarrow \Sigma$, any statement $S : \Xi(\Sigma \times \Phi)$, and any constructor input argument $e : \Gamma_0$,*

$K \sqsubseteq_R K' \;\Rightarrow$
$\quad$ **create var** $c.C(e); S;$ **end** $\sqsubseteq$ **create var** $c'.C'(e); S\downarrow(R \times Id);$ **end**

**Lemma 2.** *Let classes $C$ and $C'$ have methods $M_i : \Xi(\Sigma \times \Gamma_i \times \Delta_i)$ and $M'_i : \Xi(\Sigma' \times \Gamma'_i \times \Delta'_i)$ with $\Gamma_i <: \Gamma'_i$ and $\Delta'_i <: \Delta_i$. In a global state $u : \Phi$ including a variable $d_i : \Delta_i$, for any relation $R : \Sigma' \leftrightarrow \Sigma$ and any input argument $g_i : \Gamma_i$,*

$M_i \sqsubseteq_R M'_i \;\Rightarrow$
$\quad (\mathbf{var}\ c, u \cdot c.Meth_i(g_i, d_i))\downarrow(R \times Id) \;\sqsubseteq\; (\mathbf{var}\ c', u \cdot c'.Meth_i(g_i, d_i))$

The following theorem proves that clients using objects instantiated by some class are refined when using objects instantiated by its refinement.

**Theorem 1.** *For any classes $C$ and $C'$, any program $\mathcal{K}$ expressible as an iterative choice of invocations of $C$ methods, and any constructor input argument $e : \Gamma_0$,*

$C \sqsubseteq C' \;\Rightarrow$ **create var** $c.C(e); \mathcal{K}\ [c];$ **end** $\sqsubseteq$ **create var** $c'.C'(e); \mathcal{K}\ [c'];$ **end**

Proofs of Lemma 1, Lemma 2, and Theorem 1 are presented in the appendix. Declaring one class as a subclass of another raises the proof obligation that the class refinement relation holds between these classes. This is, in a way, a semantic constraint that we impose on subclassing to ensure that behavior of subclasses conforms to the behavior of their superclasses and that subclass instances can be substituted for superclass instances in all clients.

### 4.2 The Problem Introduced by New Methods

As was pointed out in [26] the effects of new methods become visible in the presence of subsumption (subtype aliasing) as well as in the general computational environment that allows sharing of objects by multiple users. For example, when a client is working with an object $c'$ of a subclass $C'$ of $C$, it may freely call new methods defined in $C'$. Other clients of $c'$ considering it as an instance of the polymorphic type $\tau(C)^+$ can only anticipate changes to $c'$ specified by the methods of the class $C$. New methods specifying some "unexpected behavior" could take $c'$ to (what for $C$ is) an unreachable state, and clients of this object considering it from the superclass perspective would be damaged.

Let us consider an example illustrating this problem. Suppose that a class *Counter* introduces methods *Val* and *Inc2* which, respectively, return the value of the counter and increment the counter by two. A subclass *Counter′* inherits these methods and, in addition, defines a method *Inc1* incrementing the counter by one:

*Counter* = **class**
$\quad$ **var** $n$ : *Nat*
$\quad$ *Counter* () = **enter var** $n \cdot n = 0$,
$\quad$ *Inc2* () = $n := n + 2$,
$\quad$ *Val* (**res** $r$ : *Nat*) = $r := n$
**end**

*Counter′* = **subclass of** *Counter*
$\quad$ **var** $n$ : *Nat*
$\quad$ *Counter′* () = **enter var** $n \cdot n = 0$,
$\quad$ *Inc1* () = $n := n + 1$
**end**

The implicit (or the strongest) invariant established by the class constructor and preserved by the methods of *Counter* states that the predicate *Even* holds of all states reachable by objects instantiated by *Counter*. The new method *Inc1* defined in the subclass breaks this invariant. A client $\mathcal{K}$ of a polymorphic object $c : \tau(Counter)^+$ might assume that this invariant holds of all states reachable by $c$ and execute some statement $S$ relying on the invariant:

$$\mathcal{K}[c] \;=\; \textbf{if } (Even\ c.Val()) \textbf{ then } S \textbf{ else abort fi}$$

When $c$ is instantiated by *Counter*, other clients in the environment of $\mathcal{K}$ will only be able to call the methods defined in the class *Counter* which preserve the strongest invariant. When operating in such an environment, $\mathcal{K}$ will always execute $S$. However, when $c$ is instantiated by $Counter'$, $\mathcal{K}$ may also work in the environment where other clients of $c$ know about its origin and may call the method *Inc1* in addition to the methods *Inc2* and *Val*. Suppose, for example, that there is a client $\mathcal{K}'$ which sees the class $Counter'$, with the new method $Inc1()$, and tries to increase the counter by as little as possible:

$$\mathcal{K}'[c] \;=\; \textbf{if } (c \textbf{ is } \tau(Counter')) \textbf{ then } c.Inc1() \textbf{ else } c.Inc2() \textbf{ fi}$$

If objects $c$ and $c'$ are now instantiated by *Counter* and $Counter'$ respectively and initialized to zero, executing $\mathcal{K}'[c];\mathcal{K}[c]$ equals executing $S$, whereas $\mathcal{K}'[c'];\mathcal{K}[c']$ aborts because the strongest invariant is broken by $\mathcal{K}'$.

To avoid this and similar problems, we want to ensure that invocation of a new method does not result in any unexpected behavior or, in other words, that the new method *preserves the strongest invariant of its superclass*. Let us formally analyze this consistency property and the requirements that can be imposed on new methods to enforce this property.

### 4.3 Ensuring New Method Consistency

Let us first define the notion of the strongest class invariant. As suggested by its name, the strongest class invariant is the least state predicate established by the class constructor and preserved by all its methods.

**Definition 2.** *For a class $C = (K, M_1, \ldots, M_n)$, a state predicate $I$ is the strongest class invariant if it is the invariant of $C$ and of all invariants of $C$ it is the least one:*

$$\begin{aligned} Inv\,(C, I) \;\widehat{=}\;\; & true\ \{\!|\, K \,|\!\}\ I \wedge (\forall i \mid 1 \le i \le n \cdot I\ \{\!|\, M_i \,|\!\}\ I) \wedge \\ & (\forall J \cdot true\ \{\!|\, K \,|\!\}\ J \wedge (\forall i \mid 1 \le i \le n \cdot J\ \{\!|\, M_i \,|\!\}\ J) \Rightarrow I \subseteq J) \end{aligned}$$

Suppose now that a class $C = (K, M_1, \ldots, M_n)$ specifies the interface

$$Meth_1\,(\textbf{val } : \Gamma_1, \textbf{res } : \Delta_1), \ldots, Meth_n\,(\textbf{val } : \Gamma_n, \textbf{res } : \Delta_n)$$

and a class $C' = (K', M'_1, \ldots, M'_n, N_1, \ldots, N_p)$ specifies the interface

$$\begin{aligned} & Meth_1\,(\textbf{val } : \Gamma_1, \textbf{res } : \Delta_1), \ldots, Meth_n\,(\textbf{val } : \Gamma_n, \textbf{res } : \Delta_n), \\ & NMeth_1\,(\textbf{val } : \Phi_1, \textbf{res } : \Psi_1), \ldots, NMeth_p\,(\textbf{val } : \Phi_p, \textbf{res } : \Psi_p) \end{aligned}$$

For simplicity, we assume that methods $Meth_1, \ldots, Meth_n$ in $C'$ have the same types of value and result parameters as the corresponding methods in $C$. The case when the value parameter types are contravariant and the result parameter types are covariant is treated similarly. We can express the meaning of an invariant $I$ of $C$ on the attributes of $C'$ as $\{R\}\ I$, where $R$ is a relation coercing the attributes of $C'$ to those of $C$. To guarantee that a new method $N_j$ of $C'$ preserves the strongest class invariant of $C$, we then need to prove the correctness assertion $\{R\}\ I\ \{\!|\, N_j \,|\!\}\ \{R\}\ I$ for $I$ such that $Inv\,(C, I)$. By satisfying this correctness assertion, the new method of $C'$ preserves the set of reachable states of $C$. In general, preserving a coerced invariant $\{R\}\ I$ by a statement $S' : \Xi(\Sigma')$ is the same as preserving the invariant $I$ by the statement $S'$ coerced to operate on the state space $\Sigma$, as expressed in the following lemma.

**Lemma 3.** *For a statement $S' : \Xi(\Sigma')$, a relation $R : \Sigma' \leftrightarrow \Sigma$, and a state predicate $I : \mathcal{P}\Sigma$, we have*

$$\{R\}\ I\ \{\!|\, S' \,|\!\}\ \{R\}\ I\ =\ I\ \{\!|\, S' \uparrow R \,|\!\}\ I$$

A proof of this lemma is presented in the appendix.

Class refinement between a class $C$ and a class $C'$ introducing new methods is given as an extension of Def. 1 requiring that every new method of $C'$ preserves the strongest class invariant of $C$.

**Definition 3.** *For a class $C = (K, M_1, \ldots, M_n)$ and a class $C' = (K', M'_1, \ldots, M'_n, N_1, \ldots, N_p)$, class refinement $C \sqsubseteq C'$ is defined as follows:*

$$\begin{aligned} C \sqsubseteq C' \ \widehat{=}\ & (\exists R \cdot\ K \sqsubseteq_R K' \wedge (\forall i \mid 1 \le i \le n \cdot\ M_i \sqsubseteq_R M'_i) \wedge \\ & (\forall I \cdot\ Inv\,(C, I) \Rightarrow (\forall j \mid 1 \le j \le p \cdot\ \{R\}\ I\ \{\!|\, N_j \,|\!\}\ \{R\}\ I))) \end{aligned}$$

As one can expect, Theorem 1, relating class refinement to object substitutability in clients, holds for the extended definition of class refinement as well. Unfortunately, verifying correctness assertions for new methods can be difficult in practice, because the strongest invariant of a superclass cannot always be easily calculated from its specification, e.g., in the case of recursive method invocations. When such verification is infeasible, we could instead verify that new methods satisfy certain restrictions such that the correctness assertions hold automatically. Intuitively, a new method preserves the strongest invariant of the superclass if it does not modify attributes at all, or if it modifies them as the old methods could have done. More precisely, the strongest invariant of the superclass is preserved in the following cases:

- the new method is an observer, i.e. a non-modifying method
- the subclass adds new attributes without overriding the original attributes of the superclass and the new method modifies only these new attributes
- the new method is composed of calls to old methods
- the new method is a refinement of (a combination of) old methods

Note that in the last case the new method can either data refine the old method definitions as given in the superclass, or refine the old method definitions as given in the subclass, or be a refinement of any combination of these.

Formally, weak iteration of a demonic choice of statements $S_1, \ldots, S_n$, namely $(\sqcap_{i=1}^{n}\ S_i)^*$, describes all possible combinations of these statements. Any combination of statements refines this statement, e.g., $(\sqcap_{i=1}^{3}\ S_i)^* \sqsubseteq S_1; S_3; S_2; S_1$. To be consistent, a new method should data refine an arbitrary combination of old methods prefixed by enabledness guards and intermixed with arbitrary statements. We require that the arbitrary statements do not update the attributes and necessarily terminate. A statement $S$ is guaranteed to terminate if it can establish any postcondition from any initial state, i.e. $true = S\ true$.

As old methods and new methods operate on different state spaces, we first have to adjust them to operate on the common state space. Recall that methods $M_i$ of $C$ operate on $\Sigma \times \Gamma_i \times \Delta_i$, while methods $M_i'$ of $C'$ operate on $\Sigma' \times \Gamma_i \times \Delta_i$ and new methods $N_j$ on $\Sigma' \times \Phi_j \times \Psi_j$. We can construct a common state space $\Pi$ including all value and result parameter types of all methods in $C'$ so that

$$\Pi = \Gamma_1 \times \Delta_1 \times \ldots \times \Gamma_n \times \Delta_n \times \Phi_1 \times \Psi_1 \times \ldots \times \Phi_p \times \Psi_p$$

Then a projection function $\xi_i : \Pi \to \Gamma_i \times \Delta_i$, for $i = 1..n$, will give us the types of value and result parameters of method $M_i'$. Similarly, a projection function $\xi_{n+j} : \Pi \to \Phi_j \times \Psi_j$, for $j = 1..p$, will give us the types of value and result parameters of method $N_j$. We can always coerce $M_i'$ to operate on the state space $\Sigma' \times \Pi$ using the corresponding projection function.

As methods $M_i$ of $C$ have to operate on the attributes of $C'$ rather than $C$, they have to be appropriately coerced using the abstraction relation $R : \Sigma' \leftrightarrow \Sigma$. The resulting statement $M_i \downarrow R$, being of type $\Xi(\Sigma' \times \Gamma_i \times \Delta_i)$, still has to be coerced to operate on the common state space $\Sigma' \times \Pi$, using the corresponding projection function.

Putting everything together, we can now define consistency of new methods as follows.

**Definition 4.** *For classes* $C = (K, M_1, \ldots, M_n)$ *and* $C' = (K', M_1', \ldots, M_n', N_1, \ldots, N_p)$*, some guards* $q_i$*, and some terminating statements* $K_i$ *skipping on the attributes of* $C'$*, consistency of a new method* $N_j$*, for* $j = 1..p$*, with respect to* $C$ *and an abstraction relation* $R : \Sigma' \leftrightarrow \Sigma$ *is defined as follows:*

$$\begin{array}{l} \mathit{Consistent}\ (N_j, C, R) \ \widehat{=} \\ \quad \mathbf{begin\ var}\ l \cdot b; (\sqcap_{i=1}^{n}\ [q_i]; (\mathbf{skip} \times M_i \downarrow R) \downarrow |\rho_i|; K_i)^*; \mathbf{end} \sqsubseteq N_j \end{array}$$

Here the local block variables $l$ introduce the value and result parameters of all methods $M_1', \ldots, M_n'$ and all new methods except $N_j$, whose value and result parameters are already present in the state. Effectively, the state space inside the block is $\Pi' \times \Sigma' \times \Phi_j \times \Psi_j$, where $\Pi'$ is the same as $\Pi$ with $\Phi_j \times \Psi_j$ projected away. The statement $\mathbf{skip} \times M_i \downarrow R$ operates on the state space $\Pi'' \times \Sigma' \times \Gamma_i \times \Delta_i$, where $\Pi''$ is the same as $\Pi$ with $\Gamma_i \times \Delta_i$ projected away. To coerce this statement to operate on the state space of the block, which has the same state components but

in a slightly different order (unless $M_i$ and $N_j$ happen to have value and result parameters of the same types), we wrap it in the function $\rho_i : \Pi' \times \Sigma' \times \Phi_j \times \Psi_j \to \Pi'' \times \Sigma' \times \Gamma_i \times \Delta_i$. Note that wrappings in the state-reassociating functions $\rho_i$ are just technicalities not changing the meaning of the corresponding statements.

Definition 4 allows a new method to be an arbitrary non-modifying method refining **skip**, since $(\sqcap_{i=1}^{n}\ [q_i]; (\mathbf{skip} \times M_i \downarrow R) \downarrow |\rho_i|; K_i)^* \sqsubseteq \mathbf{skip}$. No less important, it follows from Def. 4 that a new method $N_j$ is also consistent if it is composed of calls to overriding methods intermixed with arbitrary statements or refines an arbitrary composition of such calls:

**Corollary 1.** *For classes* $C = (K, M_1, \ldots, M_n)$ *and* $C' = (K', M'_1, \ldots, M'_n, N_1, \ldots, N_p)$*, some guards* $q_i$*, and some terminating statements* $K_i$ *skipping on the attributes of* $C'$*,*

$$
\begin{array}{l}
(\forall i \mid 1 \leq i \leq n \cdot M_i \sqsubseteq_R M'_i) \ \wedge \\
\mathbf{begin\ var}\ l \cdot b; (\sqcap_{i=1}^{n}\ [q_i]; (\mathbf{skip} \times M'_i) \downarrow |\rho_i|; K_i)^*; \mathbf{end} \ \sqsubseteq \ N_j \ \Rightarrow \\
\quad \mathit{Consistent}\ (N_j, C, R)
\end{array}
$$

If all new methods in a class $C'$ are consistent, the constructor of $C$ is refined by the constructor of $C'$ and all old methods of $C$ are refined by the corresponding old methods of $C'$, then class refinement between $C$ and $C'$ is guaranteed to hold, as proved by the following theorem.

**Theorem 2.** *For classes* $C = (K, M_1, \ldots, M_n)$ *and* $C' = (K', M'_1, \ldots, M'_n, N_1, \ldots, N_p)$*,*

$$
\begin{array}{l}
(\exists R \cdot \ K \sqsubseteq_R K' \wedge (\forall i \mid 1 \leq i \leq n \cdot M_i \sqsubseteq_R M'_i) \wedge \\
\qquad (\forall j \mid 1 \leq j \leq p \cdot \mathit{Consistent}\ (N_j, C, R))) \ \Rightarrow \ C \sqsubseteq C'
\end{array}
$$

A proof of this theorem is given in the appendix.

## 5 Conclusions and Related Work

This work is based on [29], but concentrates on class refinement and its relation to object substitutability. One of the main contributions of the present paper is in modelling clients of class instances by an iterative choice of method invocations. In our opinion, polymorphic substitutability of objects in clients is central to the object-oriented programming style, and, in this respect, the ability to reason about the behavior of object clients, and not only objects, is very important. Our model allows us to reason formally about the relationship between refinement on classes and substitutability of class instances in clients. We prove that substituting instances of a refined class for instances of the original class is refinement for the clients.

### 5.1 Related Work in Formalization of Object-Oriented Concepts

Related work in formalization of object-oriented concepts includes [13, 31, 33, 2]. William Cook and Jens Palsberg in [13] give a denotational semantics of inheritance and prove its correctness with respect to an operational "method lookup" semantics. They model dynamic binding of self-referential methods by representing classes as functions of self-called methods and constructing subclasses using modifying wrappers. There are only functional methods in their model, whereas we consider the methods modifying object state as well.

Martín Abadi and Rustan Leino in [2] develop a logic of object-oriented programs in the style of Hoare [20], prove its soundness and discuss completeness issues. Rather than building a new logic, we extend a logic for reasoning about imperative programs (the refinement calculus) with definitions of classes, subclassing, subtyping, and class refinement. Our extension is conservative in the sense that it does not extend the set of theorems over the original constants in the underlying logic. Accordingly, our logic of object-oriented programs inherits all meta-logical properties of the refinement calculus, including soundness. Being itself a conservative extension of higher-order logic, the refinement calculus has the syntax of higher-order logic, with some syntactic sugaring, and the simple set-theoretic semantics of higher-order logic. As the refinement calculus identifies program statements with the monotonic predicate transformers that they determine, it makes no distinction between syntax, semantics, and proof theory that is traditional in programming logics [10].

Semantics of a simple imperative Oberon-like programming language with similar specification constructs as here, also based on predicate transformers, is defined by David Naumann in [31]. Emil Sekerinski [33] defines a rich object-oriented programming and specification notation by using a type system with subtyping and type parameters, and also using predicate transformers. In both approaches, subtyping is based on extensions of record types. Here we use sum types instead, as suggested by Ralph Back and Michael Butler in [8] and further elaborated in [29]. One motivation for moving to sum types is to avoid complications in the typing and the logic when reasoning about record types: the simply typed lambda calculus as the formal basis is sufficient for our purposes. Also, to allow objects of a subclass to have different (private) attributes from those of the superclass, hiding by existential types was used in [33]. It turned out that this leads to complications when reasoning about method calls, which are not present when using the model of sum types. Leonid Mikhajlov and Emil Sekerinski in [27] give semantics to object-oriented constructs in the refinement calculus, modelling dynamic binding of self-referential methods following [13] but permitting state-modifying methods as we do here. As their formalization is tailored for studying a particular problem, namely the fragile base class problem, they consider a limited set of object-oriented constructs and mechanisms.

The detailed elaboration of our formalization, especially the fact that we define all object-oriented constructs and mechanisms on the semantic level, within the logic, rather than by syntactic definitions, opens the possibility of mechanized reasoning and mechanical verification. An interesting recent work by Bart

Jacobs et al. in [22] reports a work in progress on building a front-end tool for translating Java classes to higher-order logic in PVS [32]. The authors state that "current work involves incorporation of Hoare logic [20], via appropriate definitions and rules in PVS", and present in [22] a description of the tool "directly based on definitions". We develop a theoretic foundation for reasoning about object-oriented programs based on the logical framework for reasoning about imperative programs. A tool supporting verification of correctness and refinement of imperative programs and known as the Refinement Calculator [23] already exists and extending it to handling object-oriented programs based on the formalization presented here appears to be only natural.

### 5.2 Related Work on Behavioral Compatibility of Objects

The general idea behind our approach and the research direction known as behavioral subtyping is essentially the same – to develop a specification and verification methodology for reasoning about correctness of object-oriented programs. Our work has been to a great extent inspired by works of Pierre America, Barbara Liskov, Jeannette Wing, Gary Leavens, and others [4, 5, 26, 25, 16]. However, our approach differs in a number of ways. First of all, as was already mentioned in the introduction, we consider it essential to separate decidable syntactic properties of interface conformance or subtyping from undecidable but provable properties of behavioral conformance or refinement. We use classes to express (at different abstraction levels) the behavior of objects and class refinement to express behavioral conformance. Here we for simplicity consider systems where subclassing forms a basis for subtype polymorphism. However, our model of classes, subclassing, and subtype polymorphism as well as the definition of class refinement can be used to reason about the meaning of programs using separate subclassing and interface inheritance hierarchies. By associating a specification class with every interface type, we can reason about the behavior of objects having this interface. All classes claiming to implement a certain interface must refine its specification class. Subclassing in this layout does not, in general, require establishing class refinement between the superclass and the subclass.

When used in the context of separate subclassing and subtyping hierarchies, class refinement is very similar to behavioral subtyping. Consider a graphical representation of the corresponding settings in Fig. 5. In both cases $I$ and $I'$ are certain interfaces (types) such that $I'$ is a syntactic subtype of $I$. In the case of behavioral subtyping in Fig. 5 (a) the behavior of methods is specified in terms of pre- and postconditions. To verify that $I'$ is a behavioral subtype of $I$, written $I' < I$, America, Liskov, and Wing require proving that every precondition $pre_i$ is stronger than the corresponding $pre'_i$ and every postcondition $post_i$ is weaker than the corresponding $post'_i$, while Dhara and Leavens in [16] weaken the requirement for postconditions. In addition to proving behavioral subtyping, one must also verify that the classes $C$ and $D$ claiming to implement the types $I$ and $I'$ respectively really do so. America in [5] proposes a rigorous verification method that can be used for this purpose. For verifying, e.g., that $C$ implements $I$, he uses a representation function mapping concrete states of

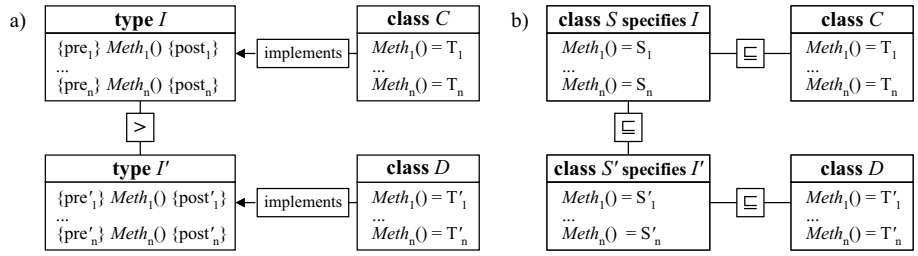



**Fig. 5.** behavioral subtyping (a) and class refinement (b) in the case of separate interface and implementation inheritance hierarchies

$C$ to the set of abstract states associated with $I$ as well as a representation invariant constraining the values of attributes in $C$, and requires proving that every method $T_i$ of $C$ preserves the representation invariant and establishes $post_i$ coerced to the state space of $C$ when $pre_i$ also coerced to the state space of $C$ holds. Since in [5] and other works on behavioral subtyping no formal semantics is given to implementation constructs and mechanisms, such as, e.g., super-calls or dynamic binding, this verification can only be done semi-formally.

Consider now the diagram (b) of Fig. 5 illustrating class refinement. First of all, we can reason about specification classes $S$ and $S'$ and implementation classes $C$ and $D$ in a uniform manner, and the behavioral conformance between the participating classes is the class refinement. Since class refinement is transitive, we get directly that $D$, implementing $I'$ by refining its specification $S'$, also refines the specification $S$ of $I$.

Class refinement can be used to verify correctness even if $D$ happens to be a subclass of $C$. Dynamic binding of self-referential methods, which becomes possible in this case, can be resolved as described in Sec. 3.6, and then we can prove that, e.g., $S' \sqsubseteq D$ using the definition of class refinement. With behavioral subtyping, however, it is not clear how one can prove that a method satisfies certain pre- and postconditions in the presence of dynamic binding of self-referential methods.

When used for reasoning about systems with unified subclassing and subtyping, our methodology eliminates a significant amount of proof obligations as compared to behavioral subtyping. We do not need to prove separately that a class and its subclass implement the corresponding type and its behavioral subtype, all that needs to be proved is class refinement between the subclass and the superclass.

Finally, the formalisms presented in [4, 5, 26, 25, 16] permit verification of only *partial correctness*. Essentially, the correctness of a program with respect to its specification can only be verified under the assumption that the program terminates. Class refinement defined here guarantees *total correctness*, i.e. no termination assumption has to be made. In general, we shift the focus from correctness reasoning to establishing refinements between methods.

Researchers working in the area of behavioral subtyping, e.g., America in [5], maintain that specifications in terms of pre- and postconditions are more ab-

stract and easier to understand than those in a more operational style, capturing method invocation order. We feel that the essence of object-oriented programs is in invoking methods on objects, and, as our *TextDoc - View* example shows, it might be necessary to specify explicitly that a certain method calls other methods. When reasoning about correctness, it is often necessary to know the method invocation order, which is more difficult to specify in terms of pre- and postconditions. Therefore, we consider it essential for a specification language to support both declarative and operational specification styles, permitting abstract specifications when it is desirable to abstract away from implementation details and also permitting capturing method invocation order when it is essential. Similar ideas are supported by Richard Helm et al. in [19]. They include method calls in abstract specifications of contracts to express behavioral dependencies between co-operating objects. Martin Büchi and Wolfgang Weck in [12] also advocate a specification language combining specification statements with method calls.

Mark Utting in his PhD thesis [34] extends the refinement calculus to support a variety of object-oriented programming styles. One of the main contributions of [34] is a formal definition in the refinement calculus of modular reasoning advocated by Leavens in [25]. It is assumed that all objects are ordered by a substitution relation $\leq$ which must be a preorder but otherwise is unrestricted. An object-oriented system is defined to support modular reasoning if methods of an object $a$, such that $a \leq b$, are refined by the corresponding methods of $b$. Clearly our methodology of object-oriented system development supports modular reasoning, because, if the substitution ordering is chosen so that $a \leq b$ whenever the class of $a$ is refined by the class of $b$, then the corresponding methods are in refinement. Our definition of class refinement is constructive, meaning that it can be used to formally verify behavioral conformance between given classes. Proving refinement between classes guarantees correctness of substitutability in all clients of the objects these classes instantiate. Utting's definition of modular reasoning, on the other hand, is non-constructive; to cite Liskov and Wing's description in [26], "it tells you what to look for, but not how to prove that you got it".

As it follows the style of behavioral subtyping, the approach reported in [34] separates implementations and specifications (types) and checks behavioral conformance of types to their supertypes. Data refinement is only allowed between the implementation and a specification of an object, although a way of generalizing data refinement for the (behavioral) subtyping is discussed in the future work section. Utting's approach to formalization of object-oriented programs differs from ours in several aspects, motivated primarily by the fact that the refinement calculus used as the basis for his object-oriented extensions was formalized within infinitary rather than higher-order logic. In particular, with the state space modelled by a product space as we have here, encapsulation is built-in rather naturally in the model: methods operate only on the instances of the corresponding class and cannot access or modify instances of other classes. In [34] the state is not considered to be a tuple of state components, but rather a function from all variables (including object variables) to all values (including

object values) in the program. Methods of all objects operate on the global state and encapsulation is only assumed.

Behavioral dependencies in the presence of subclassing have also been studied in various extensions of Z specification languages, e.g., [24, 14], but only between class specifications and not implementations. By having specification constructs as part of the (extended) programming language, we do not have to treat specifications and implementations separately.

Data refinement of modules, abstract data types, and abstract machines as, e.g., in [21, 30, 3] forms a basis for class refinement. The latter, however, has special features due to subtype polymorphism and dynamic binding.

Our treatment of new methods follows that of Liskov and Wing as presented in [26]. They describe two approaches to dealing with new method consistency. The first approach requires that new methods satisfy the explicit class invariant and the history constraint, whereas the second approach forces new methods to preserve the strongest superclass invariant. Here we do not consider explicit class invariants and refer to [28] for a detailed analysis of consistency requirements that must be imposed in the presence of explicit invariants. In this paper we present a formal analysis of the requirements that, when satisfied by new methods, are guaranteed to preserve the strongest superclass invariant. Our definition of new method consistency is more permissive than that of Liskov and Wing. They informally require that "for each extra method an explanation be given of how its behavior could be effected by just those methods already defined for the supertype". Our definition of consistency permits new methods not only to be composed of calls to existing methods, but also refine an arbitrary combination of the old methods as defined in the subclass or data refine an arbitrary combination of the old methods as defined in the superclass.

Applying our methodology in practice represents the subject of current research. In particular, we are trying to specify the Java Collections Framework which is a part of the standard JDK2.0.

## Acknowledgments

The authors would like to thank Emil Sekerinski, Leonid Mikhajlov, Michael Butler, and Martin Büchi for valuable comments on this paper.

## References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In *Proceedings of TAPSOFT'97*, LNCS 1214, pages 682–696. Springer, April 1997.
3. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
4. P. America. Inheritance and subtyping in a parallel object-oriented language. In *Proceedings of ECOOP'87*, LNCS 276, pages 234–242, Paris, France, 1987. Springer-Verlag.

5. 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*, LNCS 489, pages 60–90, New York, N.Y., 1991. Springer-Verlag.
6. R. Back, A. Mikhajlova, and J. von Wright. Reasoning about interactive systems. To appear in *Proceedings of the World Congress on Formal Methods (FM'99)*, LNCS, Springer-Verlag, September 1999. Previous version appeared as Technical Report No. 200, Turku Centre for Computer Science. http://www.tucs.abo.fi/publications/techreports/TR200.
7. R. J. R. Back. Changing data representation in the refinement calculus. In *21st Hawaii International Conference on System Sciences*. IEEE, January 1989.
8. R. J. R. Back and M. Butler. Exploring summation and product operators in the refinement calculus. In B. Möller, editor, *Mathematics of Program Construction, 1995*, volume 947. Springer-Verlag, 1995.
9. R. J. R. Back and J. von Wright. Programs on product spaces. Technical Report 143, Turku Centre for Computer Science, November 1997.
10. R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, April 1998.
11. R. J. R. Back and J. von Wright. Encoding, decoding and data refinement. Technical Report 236, Turku Centre for Computer Science, March 1999.
12. M. Büchi and W. Weck. A plea for grey-box components. Technical Report 122, Turku Center for Computer Science, Presented at the Workshop on Foundations of Component-Based Systems, Zurich, September 1997.
13. 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.
14. E. Cusack. Inheritance in object-oriented Z. In P. America, editor, *Proceedings of ECOOP'91*, LNCS 512, pages 167–179, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.
15. K. K. Dhara and G. T. Leavens. Weak behavioral subtyping for types with mutable objects. In *Mathematical Foundations of Programming Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.
16. 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.
17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
18. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
19. R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Proceedings of OOPSLA/ECOOP'90*, ACM SIGPLAN Notices, pages 169–180, Oct. 1990.
20. C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–583, 1969.
21. C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.
22. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes (preliminary report). In *Proceedings of OOPSLA'98*, pages 329–340, Vancouver, Canada, Oct. 1998. Association for Computing Machinery.

23. T. Långbacka, R. Ruksenas, and J. von Wright. TkWinHOL: A tool for window inference in HOL. *Higher Order Logic Theorem Proving and its Applications: 8th International Workshop*, 971:245–260, September 1995.
24. K. Lano and H. Haughton. Reasoning and refinement in object-oriented specification languages. In O. L. Madsen, editor, *Proceedings of ECOOP'92*, LNCS 615. Springer-Verlag, 1992.
25. G. T. Leavens and W. E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). In *Proceedings of OOPSLA/ECOOP'90*, volume 25(10) of *ACM SIGPLAN Notices*, pages 212–223, 1990.
26. B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
27. L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In E. Jul, editor, *Proceedings of ECOOP'98*, pages 355–382. Springer, July 1998.
28. A. Mikhajlova. Consistent extension of components in the presence of explicit invariants. In *Technology of Object-Oriented Languages and Systems (TOOLS 29)*, pages 76–85. IEEE Computer Society Press, June 1999.
29. A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In *Proceedings of the 4th International Formal Methods Europe Symposium, FME'97*, LNCS 1313, pages 82–101. Springer, 1997.
30. C. C. Morgan. *Programming from Specifications*. Prentice–Hall, 1990.
31. D. A. Naumann. Predicate transformer semantics of an Oberon-like language. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, pages 460–480, San Miniato, Italy, 1994.
32. S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.
33. E. Sekerinski. A type-theoretic basis for an object-oriented refinement calculus. In S. Goldsack and S. Kent, editors, *Formal Methods and Object Technology*. Springer-Verlag, 1996.
34. M. Utting. *An Object-Oriented Refinement Calculus with Modular Reasoning*. PhD thesis, University of New South Wales, Kensington, Australia, 1992.

## Appendix

### Algorithmic Refinement Rules

Let us first present algorithmic refinement rules that will be used in proofs of Lemmas 1-3, and Theorems 1 and 2. Proofs of these rules can be found in [10, 9].

Skip is unit of sequential composition :
$S; \mathbf{skip} \;=\; S \;=\; \mathbf{skip}; S$

Relational product distribution through composition :
$(P_1 \times Q_1); (P_2 \times Q_2) \;=\; (P_1; P_2) \times (Q_1; Q_2)$

Distribution of sequential composition through updates :

(a) $\langle f \rangle; \langle g \rangle \;=\; \langle f; g \rangle$
(b) $[P]; [Q] \;=\; [P; Q]$
(c) $\{P\}; \{Q\} \;=\; \{P; Q\}$

Product distribution through updates :

(a) $\langle f \rangle \times \langle g \rangle \;=\; \langle f \times g \rangle$
(b) $[P] \times [Q] \;=\; [P \times Q]$
(c) $\{P\} \times \{Q\} \;=\; \{P \times Q\}$

Product distribution through sequential composition :

(a) $(S_1; T_1) \times (S_2; T_2) \;\sqsubseteq\; (S_1 \times S_2); (T_1 \times T_2)$
(b) $(S_1 \times \mathbf{skip}); (S_2 \times \mathbf{skip}) \;=\; (S_1; S_2) \times \mathbf{skip}$
(c) $\mathbf{skip} \times \{R\}; (S \times \mathbf{skip}) \;=\; S \times \{R\}$
(d) $[P \times Q] \;=\; [P \times Id]; [Id \times Q] \;=\; [Id \times Q]; [P \times Id]$
(e) $\{P \times Q\} \;=\; \{P \times Id\}; \{Id \times Q\} \;=\; \{Id \times Q\}; \{P \times Id\}$

### Data Refinement Rules

Proofs of most rules presented here can be found in [11], and the other rules can easily be derived using definitions of the involved constructs.

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

$$(S_1; S_2) \downarrow R \;\sqsubseteq\; (S_1 \downarrow R); (S_2 \downarrow R)$$

Data refinement also distributes through demonic and angelic choice:

$$(S \sqcap T) \downarrow R \;\sqsubseteq\; S \downarrow R \sqcap T \downarrow R$$
$$(S \sqcup T) \downarrow R \;\sqsubseteq\; S \downarrow R \sqcup T \downarrow R$$

The *indifferent block rule* reduces data refinement of a block with local variables to a data refinement of a statement inside that block, retaining the local

variables:

$$\textbf{begin}\ (p \times \mathit{true}); S; \textbf{end} \downarrow R \ \sqsubseteq\ \textbf{begin}\ (p \times \mathit{true}); S \downarrow (\mathit{Id} \times R); \textbf{end}$$

When the initializing predicate is effected by the data refinement, the *block rule* requires that this predicate is coerced accordingly:

$$\begin{array}{c}\textbf{begin}\ p; S; \textbf{end} \downarrow R \ \sqsubseteq\ \textbf{begin}\ p'; S \downarrow (\mathit{Id} \times R); \textbf{end}, \\ \text{where } p' \subseteq (\lambda(x, y') \cdot \exists y \cdot R\, y'\, y \ \wedge\ p(x, y))\end{array}$$

There are also two auxiliary block begin rules:

$$\begin{array}{ll}\text{(a)} & \textbf{begin}\ p; [(R \times \mathit{Id})^{-1}] \ \sqsubseteq\ \textbf{begin}\ p', \\ & \quad \text{where } p' \subseteq (\lambda(x', y) \cdot \exists x \cdot R\, x'\, x \ \wedge\ p(x, y)) \\ \text{(b)} & \{R\}; \textbf{begin}\ p \ \sqsubseteq\ \textbf{begin}\ p'; \{\mathit{Id} \times R\}, \\ & \quad \text{where } p' \subseteq (\lambda(x, y') \cdot \exists y \cdot R\, y'\, y \ \wedge\ p(x, y))\end{array}$$

Another block-related *local variable rule* allows us to change local variables in a refinement. For any $S : \Xi(\Lambda \times \Sigma)$ and $R : \Lambda' \leftrightarrow \Lambda$,

$$\begin{array}{c}\textbf{begin}\ p; S; \textbf{end} \ \sqsubseteq\ \textbf{begin}\ p'; S \downarrow (R \times \mathit{Id}); \textbf{end}, \\ \text{where } p' \subseteq (\lambda(x', y) \cdot \exists x \cdot R\, x'\, x \ \wedge\ p(x, y))\end{array}$$

Using the program variable notation, this rule can be expressed as follows:

$$\textbf{begin}\ (\textbf{var}\ l, u \cdot b); S; \textbf{end} \ \sqsubseteq\ \begin{array}{l}\textbf{begin}\ (\textbf{var}\ l', u \cdot (\exists l \cdot R\, l'\, l \wedge b)); \\ \quad S \downarrow (R \times \mathit{Id}); \\ \textbf{end}\end{array}$$

The *indifferent statement rule* describes the cases when a statement is not affected by data refinement:

$$(\textbf{skip} \times S) \downarrow (R \times \mathit{Id}) \ \sqsubseteq\ \textbf{skip} \times S \quad \text{and} \quad (S \times \textbf{skip}) \downarrow (\mathit{Id} \times R) \ \sqsubseteq\ S \times \textbf{skip}$$

The *iterative choice rule* states that for any $S_i : \Sigma \mapsto \Sigma, R : \Sigma' \leftrightarrow \Sigma$, and any $q_i : \mathcal{P}\Sigma$ indifferent to $R$,

$$\textbf{do}\ \Diamond_{i=1}^{n} q_i :: S_i\ \textbf{od} \downarrow R \ \sqsubseteq\ \textbf{do}\ \Diamond_{i=1}^{n} q_i :: S_i \downarrow R\ \textbf{od}$$

where indifference means that $q_i = q_i' \times \mathit{true}$ when $R$ is of the form $\mathit{Id} \times R'$, and $q_i = \mathit{true} \times q_i'$ when $R = R' \times \mathit{Id}$.

Finally, the *identity of inverse coercion rule* states that wrapping the statement $S \downarrow R$ in $\uparrow R$ undoes the effect of wrapping $S$ in $\downarrow R$:

$$S \ \sqsubseteq\ (S \downarrow R) \uparrow R$$

**Correctness Rules**

Now let us present the rules that can be used for proving correctness assertions.

$$
\begin{array}{l}
\text{(a) } p \ \{\!|\, S_1; S_2 \,|\!\}\ q \ = \ (\exists r \cdot p \ \{\!|\, S_1 \,|\!\}\ r \wedge r \ \{\!|\, S_2 \,|\!\}\ q) \\
\text{(b) } p \ \{\!|\, (\sqcap i \in I \cdot S_i) \,|\!\}\ q \ = \ (\forall i \in I \cdot p \ \{\!|\, S_i \,|\!\}\ q) \\
\text{(c) } p \ \{\!|\, [r] \,|\!\}\ q \ = \ p \cap r \subseteq q \\
\text{(d) } r \ \{\!|\, S \,|\!\}\ r \ \Rightarrow \ r \ \{\!|\, S^* \,|\!\}\ r \\
\text{(e) } r \ \{\!|\, S \,|\!\}\ r \ \Rightarrow \ (\mathit{true} \times r) \ \{\!|\, \mathbf{skip} \times S \,|\!\}\ (\mathit{true} \times r) \\
\text{(f) } \mathit{true} = S\ \mathit{true} \ \Rightarrow \ (\mathit{true} \times r) \ \{\!|\, S \times \mathbf{skip} \,|\!\}\ (\mathit{true} \times r) \\
\text{(g) } (\mathit{true} \times p) \cap (\mathbf{var}\ x, u \cdot b) \ \{\!|\, S \,|\!\}\ (\mathit{true} \times q) \ \Rightarrow \\
\qquad\qquad p \ \{\!|\, \mathbf{begin}\ (\mathbf{var}\ x, u \cdot b); S; \mathbf{end} \,|\!\}\ q
\end{array}
$$

Proofs of the rules (a)-(d) can be found in [10], and the rules (e)-(g) can easily be derived using definitions of the involved constructs. Finally, the following rule presented in [10] shows that refining a statement is the same as preserving correctness of the statement:

$$S \sqsubseteq S' \ = \ (\forall p\ q \cdot p \ \{\!|\, S \,|\!\}\ q \ \Rightarrow \ p \ \{\!|\, S' \,|\!\}\ q)$$

**Lemma 1.** *Let classes $C$ and $C'$ have constructors $K : \Gamma_0 \mapsto \Sigma \times \Gamma_0$ and $K' : \Gamma'_0 \mapsto \Sigma' \times \Gamma'_0$ with $\Gamma_0 <: \Gamma'_0$. In a global state $u : \Phi$, for any relation $R : \Sigma' \leftrightarrow \Sigma$, any statement $S : \Xi(\Sigma \times \Phi)$, and any constructor input argument $e : \Gamma_0$,*

$$
\begin{array}{l}
K \sqsubseteq_R K' \ \Rightarrow \\
\quad \mathbf{create\ var}\ c.C(e); S; \mathbf{end} \ \sqsubseteq \ \mathbf{create\ var}\ c'.C'(e); S \downarrow (R \times \mathit{Id}); \mathbf{end}
\end{array}
$$

*Proof.* We assume the antecedent and prove the consequent by refining the left-hand side to the right-hand side:

$$
\begin{array}{l}
\quad \mathbf{create\ var}\ c.C(e); S; \mathbf{end} \\
\equiv \ \{\, \text{definition of constructor invocation} \,\} \\
\quad \mathbf{enter}\ (\mathbf{var}\ x_0, u \cdot x_0 = e); K \times \mathbf{skip}; \\
\quad \mathbf{enter}\ (\mathbf{var}\ c, (\mathit{self}, x_0), u \cdot c = \mathit{self}); \mathit{Swap}; \mathbf{exit}; S; \mathbf{exit} \\
\equiv \ \{\, \text{definitions} \,\} \\
\quad [\lambda u \cdot \lambda(x_0, u') \cdot x_0 = e \wedge u = u']; K \times \mathbf{skip}; \\
\quad [\lambda((\mathit{self}, x_0), u) \cdot \lambda(c, (\mathit{self}', x'_0), u') \cdot c = \mathit{self} \wedge ((\mathit{self}, x_0), u) = ((\mathit{self}', x'_0), u')]; \\
\quad \langle \lambda(c, (\mathit{self}, x_0), u) \cdot ((\mathit{self}, x_0), c, u) \rangle; \langle \lambda((\mathit{self}, x_0), c, u) \cdot (c, u) \rangle; S; \langle \lambda(c, u) \cdot u \rangle
\end{array}
$$

$$\equiv \left\{ \begin{array}{l} \text{demonic update of a functional relation } |f| \text{ is equal to} \\ \text{a functional update of a function } f\text{: } [|f|] = \langle f \rangle, \\ \text{distribution of sequential composition through functional updates,} \\ \text{definition of functional composition, logic} \end{array} \right\}$$

$$\begin{array}{l} [\lambda u \cdot \lambda(x_0, u') \cdot x_0 = e \wedge u = u']; K \times \mathbf{skip}; \\ \langle \lambda((self, x_0), u) \cdot (self, (self, x_0), u) \rangle; \\ \langle \lambda(c, (self, x_0), u) \cdot (c, u) \rangle; S; \langle \lambda(c, u) \cdot u \rangle \end{array}$$

$\sqsubseteq \{$ focus on a subexpression $\}$

$$[\lambda u \cdot \lambda(x_0, u') \cdot x_0 = e \wedge u = u']; K \times \mathbf{skip};$$

$\sqsubseteq \{$ general rule $\mathbf{skip} \sqsubseteq [R^{-1}]; \{R\}$ $\}$

$$\begin{array}{l} [\lambda u \cdot \lambda(x_0, u') \cdot x_0 = e \wedge u = u']; \\ [(\pi_{\Gamma_0} \times Id)^{-1}]; \{\pi_{\Gamma_0} \times Id\}; K \times \mathbf{skip} \end{array}$$

$$\equiv \left\{ \begin{array}{l} \text{distribution of sequential composition through demonic updates,} \\ \text{definition of relational composition, logic} \end{array} \right\}$$

$$[\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\ e \wedge u = u']; \{\pi_{\Gamma_0} \times Id\}; K \times \mathbf{skip}$$

$$\equiv \left\{ \begin{array}{l} \text{product distribution through angelic updates, then} \\ \text{product distribution through sequential composition (b)} \end{array} \right\}$$

$$[\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\ e \wedge u = u']; (\{\pi_{\Gamma_0}\}; K) \times \mathbf{skip}$$

$\sqsubseteq \{$ assumption $\}$

$$[\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\ e \wedge u = u']; (K'; \{R \times \pi_{\Gamma_0}\}) \times \mathbf{skip}$$

$$\equiv \left\{ \begin{array}{l} \text{product distribution through sequential composition (b), then} \\ \text{product distribution through angelic updates} \end{array} \right\}$$

$$[\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\ e \wedge u = u']; K' \times \mathbf{skip}; \{(R \times \pi_{\Gamma_0}) \times Id\}$$

$\equiv$

$$\begin{array}{l} [\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\ e \wedge u = u']; K' \times \mathbf{skip}; \\ \{(R \times \pi_{\Gamma_0}) \times Id\}; \\ \langle \lambda((self, x_0), u) \cdot (self, (self, x_0), u) \rangle; \\ \langle \lambda(c, (self, x_0), u) \cdot (c, u) \rangle; S; \langle \lambda(c, u) \cdot u \rangle \end{array}$$

$$\sqsubseteq \left\{ \begin{array}{l} \text{definition of sequential composition,} \\ \text{definitions of angelic and functional updates, logic} \end{array} \right\}$$

$$\begin{array}{l} [\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\ e \wedge u = u']; K' \times \mathbf{skip}; \\ \langle \lambda((self', x_0'), u) \cdot (self', (self', x_0'), u) \rangle; \\ \{R \times (R \times \pi_{\Gamma_0}) \times Id\}; \\ \langle \lambda(c, (self, x_0), u) \cdot (c, u) \rangle; S; \langle \lambda(c, u) \cdot u \rangle \end{array}$$

$\sqsubseteq \{$ definitions, logic $\}$

$$
\begin{array}{l}
\quad [\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\ e \wedge u = u'];\ K' \times \mathbf{skip};\\
\quad \langle \lambda((self', x_0'), u) \cdot (self', (self', x_0'), u)\rangle;\\
\quad \langle \lambda(c', (self', x_0'), u) \cdot (c', u)\rangle;\\
\quad \{R \times Id\};\\
\quad S; \langle \lambda(c, u) \cdot u\rangle \\
\sqsubseteq \{ \text{general rule } \mathbf{skip} \sqsubseteq [R^{-1}]; \{R\}, \text{definition of } \downarrow \} \\
\quad [\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\ e \wedge u = u'];\ K' \times \mathbf{skip};\\
\quad \langle \lambda((self', x_0'), u) \cdot (self', (self', x_0'), u)\rangle;\\
\quad \langle \lambda(c', (self', x_0'), u) \cdot (c', u)\rangle;\\
\quad S \downarrow (R \times Id);\\
\quad \{R \times Id\}; \langle \lambda(c, u) \cdot u\rangle \\
\sqsubseteq \{ \text{definitions, logic} \} \\
\quad [\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\ e \wedge u = u'];\ K' \times \mathbf{skip};\\
\quad \langle \lambda((self', x_0'), u) \cdot (self', (self', x_0'), u)\rangle;\\
\quad \langle \lambda(c', (self', x_0'), u) \cdot (c', u)\rangle; S \downarrow (R \times Id); \langle \lambda(c', u) \cdot u\rangle \\
\equiv \{ \text{definitions} \} \\
\quad \mathbf{create\ var}\ c'.C'(e); S \downarrow (R \times Id); \mathbf{end} \\
\square
\end{array}
$$

**Lemma 2.** *Let classes $C$ and $C'$ have methods $M_i : \Xi(\Sigma \times \Gamma_i \times \Delta_i)$ and $M_i' : \Xi(\Sigma' \times \Gamma_i' \times \Delta_i')$ with $\Gamma_i <: \Gamma_i'$ and $\Delta_i' <: \Delta_i$. In a global state $d_i : \Delta_i, u : \Phi$, for any relation $R : \Sigma' \leftrightarrow \Sigma$ and any input argument $g_i : \Gamma_i$,*

$$
\begin{array}{l}
M_i \sqsubseteq_R M_i' \ \Rightarrow \\
\quad (\mathbf{var}\ c, d_i, u \cdot c.Meth_i(g_i, d_i)) \downarrow (R \times Id) \ \sqsubseteq\ (\mathbf{var}\ c', d_i, u \cdot c'.Meth_i(g_i, d_i))
\end{array}
$$

*Proof.* We prove the goal by assuming the antecedent and deriving the right-hand side from the left-hand side as follows:

$$
\begin{array}{l}
\quad (\mathbf{var}\ c, d_i, u \cdot c.Meth_i(g_i, d_i)) \downarrow (R \times Id) \\
\equiv \{ \text{definition of method invocation} \} \\
\quad (\mathbf{var}\ c, d_i, u \cdot \mathbf{begin}\ (\mathbf{var}\ (self, x_i, y_i), c, d_i, u \cdot self = c \wedge x_i = g_i);\\
\quad M_i \times \mathbf{skip}; c, d_i := self, y_i; \mathbf{end}) \downarrow (R \times Id) \\
\equiv \{ \text{definitions} \} \\
\quad \{R \times Id\}; [\lambda(c, d_i, u) \cdot \lambda((self, x_i, y_i), c', d_i', u') \cdot \\
\qquad self = c \wedge x_i = g_i \wedge (c, d_i, u) = (c', d_i', u')];\\
\quad M_i \times \mathbf{skip};\\
\quad \langle \lambda((self, x_i, y_i), c, d_i, u) \cdot ((self, x_i, y_i), self, y_i, u)\rangle;\\
\quad \langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u)\rangle; [(R \times Id)^{-1}] \\
\equiv \left\{ \begin{array}{l} \text{demonic and angelic updates of a functional relation are equal,} \\ \text{distribution of sequential composition through angelic updates} \end{array} \right\}
\end{array}
$$

$\{R \times Id; \lambda(c, d_i, u) \cdot \lambda((self, x_i, y_i), c', d_i', u') \cdot$
$\quad self = c \wedge x_i = g_i \wedge (c, d_i, u) = (c', d_i', u')\};$
$M_i \times \mathbf{skip};$
$\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot ((self, x_i, y_i), self, y_i, u)\rangle;$
$\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u)\rangle; [(R \times Id)^{-1}]$

$\sqsubseteq \{$ definition of relational composition, logic $\}$

$\{\lambda(c', d_i, u) \cdot \lambda((self', x_i', y_i'), c'', d_i', u') \cdot$
$\quad self' = c' \wedge x_i' = \iota_{\Gamma_i}\ g_i \wedge (c', d_i, u) = (c'', d_i', u');$
$(R \times \pi_{\Gamma_i} \times |\iota_{\Delta_i'}|) \times R \times Id\};$
$M_i \times \mathbf{skip};$
$\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot ((self, x_i, y_i), self, y_i, u)\rangle;$
$\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u)\rangle; [(R \times Id)^{-1}]$

$\equiv \left\{ \begin{array}{l} \text{distribution of sequential composition through angelic updates,} \\ \text{demonic and angelic updates of a functional relation are equal} \end{array} \right\}$

$[\lambda(c', d_i, u) \cdot \lambda((self', x_i', y_i'), c'', d_i', u') \cdot$
$\quad self' = c' \wedge x_i' = \iota_{\Gamma_i}\ g_i \wedge (c', d_i, u) = (c'', d_i', u')];$
$\{(R \times \pi_{\Gamma_i} \times |\iota_{\Delta_i'}|) \times R \times Id\};$
$M_i \times \mathbf{skip};$
$\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot ((self, x_i, y_i), self, y_i, u)\rangle;$
$\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u)\rangle; [(R \times Id)^{-1}]$

$\equiv \left\{ \begin{array}{l} \text{product distribution through sequential composition} \\ \text{rules (e), (c) then } \mathbf{skip} \text{ is unit of sequential composition} \end{array} \right\}$

$[\lambda(c', d_i, u) \cdot \lambda((self', x_i', y_i'), c'', d_i', u') \cdot$
$\quad self' = c' \wedge x_i' = \iota_{\Gamma_i}\ g_i \wedge (c', d_i, u) = (c'', d_i', u')];$
$\{(R \times \pi_{\Gamma_i} \times |\iota_{\Delta_i'}|)\} \times \mathbf{skip};$
$(M_i; \mathbf{skip}) \times (\mathbf{skip}; \{R \times Id\});$
$\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot ((self, x_i, y_i), self, y_i, u)\rangle;$
$\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u)\rangle; [(R \times Id)^{-1}]$

$\sqsubseteq \left\{ \begin{array}{l} \text{product distribution through sequential composition} \\ \text{rules (a) then (b)} \end{array} \right\}$

$[\lambda(c', d_i, u) \cdot \lambda((self', x_i', y_i'), c'', d_i', u') \cdot$
$\quad self' = c' \wedge x_i' = \iota_{\Gamma_i}\ g_i \wedge (c', d_i, u) = (c'', d_i', u')];$
$(\{R \times \pi_{\Gamma_i} \times |\iota_{\Delta_i'}|\}; M_i) \times \mathbf{skip}; \{Id \times R \times Id\});$
$\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot ((self, x_i, y_i), self, y_i, u)\rangle;$
$\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u)\rangle; [(R \times Id)^{-1}]$

$\sqsubseteq \{$ assumption, using general rule $S \downarrow R \sqsubseteq S' \ = \ \{R\}; S \sqsubseteq S'; \{R\}\ \}$

$[\lambda(c', d_i, u) \cdot \lambda((self', x_i', y_i'), c'', d_i', u') \cdot$
$\quad self' = c' \wedge x_i' = \iota_{\Gamma_i}\ g_i \wedge (c', d_i, u) = (c'', d_i', u')];$
$(M_i'; \{R \times \pi_{\Gamma_i} \times |\iota_{\Delta_i'}|\}) \times \mathbf{skip}; \{Id \times R \times Id\});$
$\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot ((self, x_i, y_i), self, y_i, u)\rangle;$
$\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u)\rangle; [(R \times Id)^{-1}]$

$\equiv$ { product distribution through sequential composition rule (b), then (e) }

$[\lambda(c', d_i, u) \cdot \lambda((self', x_i', y_i'), c'', d_i', u') \cdot$
$\quad self' = c' \wedge x_i' = \iota_{\Gamma_i}\ g_i \wedge (c', d_i, u) = (c'', d_i', u')];$
$M_i' \times \mathbf{skip}; \{(R \times \pi_{\Gamma_i} \times |\iota_{\Delta_i'}|) \times R \times Id\};$
$\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot ((self, x_i, y_i), self, y_i, u) \rangle;$
$\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u) \rangle; [(R \times Id)^{-1}]$

$\sqsubseteq$ { definition of sequential composition, definitions of angelic and functional updates, logic }

$[\lambda(c', d_i, u) \cdot \lambda((self', x_i', y_i'), c'', d_i', u') \cdot$
$\quad self' = c' \wedge x_i' = \iota_{\Gamma_i}\ g_i \wedge (c', d_i, u) = (c'', d_i', u')];$
$M_i' \times \mathbf{skip};$
$\langle \lambda((self', x_i', y_i'), c', d_i, u) \cdot ((self', x_i', y_i'), self', \iota_{\Delta_i'}\ y_i, u) \rangle;$
$\{(R \times \pi_{\Gamma_i} \times |\iota_{\Delta_i'}|) \times R \times Id\};$
$\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u) \rangle; [(R \times Id)^{-1}]$

$\sqsubseteq$ { definition of sequential composition, definitions of angelic and functional updates, logic }

$[\lambda(c', d_i, u) \cdot \lambda((self', x_i', y_i'), c'', d_i', u') \cdot$
$\quad self' = c' \wedge x_i' = \iota_{\Gamma_i}\ g_i \wedge (c', d_i, u) = (c'', d_i', u')];$
$M_i' \times \mathbf{skip};$
$\langle \lambda((self', x_i', y_i'), c', d_i, u) \cdot ((self', x_i', y_i'), self', \iota_{\Delta_i'}\ y_i, u) \rangle;$
$\langle \lambda((self', x_i', y_i'), c', d_i, u) \cdot (c', d_i, u) \rangle; \{R \times Id\}; [(R \times Id)^{-1}]$

$\sqsubseteq$ { general rule $\{R\}; [R^{-1}] \sqsubseteq \mathbf{skip}$ }

$[\lambda(c', d_i, u) \cdot \lambda((self', x_i', y_i'), c'', d_i', u') \cdot$
$\quad self' = c' \wedge x_i' = \iota_{\Gamma_i}\ g_i \wedge (c', d_i, u) = (c'', d_i', u')];$
$M_i' \times \mathbf{skip};$
$\langle \lambda((self', x_i', y_i'), c', d_i, u) \cdot ((self', x_i', y_i'), self', \iota_{\Delta_i'}\ y_i, u) \rangle;$
$\langle \lambda((self', x_i', y_i'), c', d_i, u) \cdot (c', d_i, u) \rangle$

$\equiv$ { definition of method invocation }

$(\mathbf{var}\ c', d_i, u \cdot c'.Meth_i(g_i, d_i))$

$\Box$

**Theorem 1.** *For any classes $C$ and $C'$, any program $\mathcal{K}$ expressible as an iterative choice of invocations of $C$ methods, and any constructor input argument $e : \Gamma_0$,*

$$C \sqsubseteq C' \ \Rightarrow\ \mathbf{create\ var}\ c.C(e); \mathcal{K}\ [c]; \mathbf{end} \sqsubseteq \mathbf{create\ var}\ c'.C'(e); \mathcal{K}\ [c']; \mathbf{end}$$

*Proof.* Rewriting this implication with the definition of class refinement, we get

$(\exists R \cdot K \sqsubseteq_R K' \wedge (\forall i \mid 1 \leq i \leq n \cdot M_i \sqsubseteq_R M_i')) \ \Rightarrow$
$\quad \mathbf{create\ var}\ c.C(e); \mathcal{K}\ [c]; \mathbf{end} \sqsubseteq \mathbf{create\ var}\ c'.C'(e); \mathcal{K}\ [c']; \mathbf{end}$

Assume that $R$ is a relation that satisfies the antecedent of this implication. Using the first conjunct of the antecedent and Lemma 1, the implication is then reduced to

$$(\forall i \mid 1 \leq i \leq n \cdot M_i \sqsubseteq_R M_i') \;\Rightarrow\; (\mathbf{var}\ c, u \cdot \mathcal{K}\ [c])\downarrow(R \times Id) \;\sqsubseteq\; (\mathbf{var}\ c', u \cdot \mathcal{K}\ [c'])$$

Assuming the antecedent and expressing $\mathcal{K}$ as an iterative choice of method invocations, we get

$$\mathbf{begin}\ (\mathbf{var}\ l, c, u \cdot p); \mathbf{do}\ \Diamond_{i=1}^{n} q_i :: c.Meth_i(g_i, d_i); L_i\ \mathbf{od}; \mathbf{end}\downarrow(R \times Id) \sqsubseteq \mathbf{begin}\ (\mathbf{var}\ l, c', u \cdot p); \mathbf{do}\ \Diamond_{i=1}^{n} q_i :: c'.Meth_i(g_i, d_i); L_i\ \mathbf{od}; \mathbf{end}$$

The indifferent block rule allows us to reduce this goal to

$$(\mathbf{var}\ l, c, u \cdot \mathbf{do}\ \Diamond_{i=1}^{n} q_i :: c.Meth_i(g_i, d_i); L_i\ \mathbf{od})\downarrow(Id \times R \times Id) \;\sqsubseteq\; (\mathbf{var}\ l, c', u \cdot \mathbf{do}\ \Diamond_{i=1}^{n} q_i :: c'.Meth_i(g_i, d_i); L_i\ \mathbf{od})$$

which we prove by beginning with the left-hand side and refining it to the right-hand side as follows:

$$\begin{array}{ll} & (\mathbf{var}\ l, c, u \cdot \mathbf{do}\ \Diamond_{i=1}^{n} q_i :: c.Meth_i(g_i, d_i); L_i\ \mathbf{od})\downarrow(Id \times R \times Id) \\ \sqsubseteq & \{\ \text{iterative choice rule, since every } q_i \text{ is of the form } q_i' \times \mathit{true} \times q_i''\ \} \\ & (\mathbf{var}\ l, c, u \cdot \mathbf{do}\ \Diamond_{i=1}^{n} q_i :: (c.Meth_i(g_i, d_i); L_i)\downarrow(Id \times R \times Id)\ \mathbf{od}) \\ \sqsubseteq & \left\{\begin{array}{l}\text{sequential composition and indifferent statement rules,}\\ \text{since all statements } L_i \text{ are indifferent to } Id \times R \times Id\end{array}\right\} \\ & (\mathbf{var}\ l, c, u \cdot \mathbf{do}\ \Diamond_{i=1}^{n} q_i :: c.Meth_i(g_i, d_i)\downarrow(Id \times R \times Id); L_i\ \mathbf{od}) \\ \sqsubseteq & \{\ \text{Lemma 2, using the assumption}\ \} \\ & (\mathbf{var}\ l, c', u \cdot \mathbf{do}\ \Diamond_{i=1}^{n} q_i :: c'.Meth_i(g_i, d_i); L_i\ \mathbf{od}) \end{array}$$

This completes the proof. □

**Lemma 3.** *For a statement $S' : \Xi(\Sigma')$, a relation $R : \Sigma' \leftrightarrow \Sigma$, and a state predicate $I : \mathcal{P}\Sigma$, we have*

$$\{R\}\ I\ \{\!|\ S'\ |\!\}\ \{R\}\ I \;=\; I\ \{\!|\ S' \uparrow R\ |\!\}\ I$$

*Proof.* We prove the goal by mutual implication. The first part

$$\{R\}\ I\ \{\!|\ S'\ |\!\}\ \{R\}\ I \;\Rightarrow\; I\ \{\!|\ S' \uparrow R\ |\!\}\ I$$

is proved as follows:

$\{R\}\ I\ \{\!|\ S'\ |\!\}\ \{R\}\ I$

$\equiv \{$ definition of correctness assertion $\}$

$\{R\}\ I\ \subseteq\ S'\ \{R\}\ I$

$\Rightarrow \{$ monotonicity of demonic update $\forall P\ p\ q \cdot p \subseteq q\ \Rightarrow\ [P]\ p \subseteq [P]\ q \}$

$[R^{-1}]\ \{R\}\ I\ \subseteq\ [R^{-1}]\ S'\ \{R\}\ I$

$\equiv \{$ definition of sequential composition $\}$

$([R^{-1}]; \{R\})\ I\ \subseteq\ ([R^{-1}]; S'; \{R\})\ I$

$\Rightarrow \{$ general rule $\mathbf{skip}\ \sqsubseteq\ [R^{-1}]; \{R\}\ \}$

$\mathbf{skip}\ I\ \subseteq\ ([R^{-1}]; S'; \{R\})\ I$

$\equiv \{ \uparrow R$ abbreviates $[R^{-1}]; S'; \{R\}\ \}$

$\mathbf{skip}\ I\ \subseteq\ (S' \uparrow R)\ I$

$\equiv \{$ definition of $\mathbf{skip}$, definition of correctness assertion $\}$

$I\ \{\!|\ S' \uparrow R\ |\!\}\ I$

The second implication, $I\ \{\!|\ S' \uparrow R\ |\!\}\ I\ \Rightarrow\ \{R\}\ I\ \{\!|\ S'\ |\!\}\ \{R\}\ I$, is proved similarly:

$I\ \{\!|\ S' \uparrow R\ |\!\}\ I$

$\equiv \{ \uparrow R$ abbreviates $[R^{-1}]; S'; \{R\}\ \}$

$I\ \{\!|\ [R^{-1}]; S'; \{R\}\ |\!\}\ I$

$\equiv \{$ definition of correctness assertion $\}$

$I\ \subseteq\ ([R^{-1}]; S'; \{R\})\ I$

$\Rightarrow \{$ monotonicity of angelic update $\forall P\ p\ q \cdot p \subseteq q\ \Rightarrow\ \{P\}\ p \subseteq \{P\}\ q \}$

$\{R\}\ I\ \subseteq\ \{R\}\ ([R^{-1}]; S'; \{R\})\ I$

$\equiv \{$ definition of sequential composition $\}$

$\{R\}\ I\ \subseteq\ (\{R\}; [R^{-1}])\ S'\ \{R\}\ I$

$\Rightarrow \{$ general rule $\{R\}; [R^{-1}]\ \sqsubseteq\ \mathbf{skip}\ \}$

$\{R\}\ I\ \subseteq\ S'\ \{R\}\ I$

$\equiv \{$ definition of correctness assertion $\}$

$\{R\}\ I\ \{\!|\ S'\ |\!\}\ \{R\}\ I$

$\Box$

**Theorem 2.** *For classes* $C = (K, M_1, \ldots, M_n)$ *and* $C' = (K', M'_1, \ldots, M'_n, N_1, \ldots, N_p)$,

$$(\exists R \cdot\ K \sqsubseteq_R K' \wedge (\forall i \mid 1 \leq i \leq n \cdot M_i \sqsubseteq_R M'_i) \wedge (\forall j \mid 1 \leq j \leq p \cdot \mathit{Consistent}\ (N_j, C, R))) \ \Rightarrow\ C \sqsubseteq C'$$

*Proof.* We begin with weakening the antecedent by eliminating the existential quantification and rewriting the consequent with the definition of class refinement:

$$\begin{array}{l} K \sqsubseteq_R K' \wedge (\forall i \mid 1 \leq i \leq n \cdot M_i \sqsubseteq_R M'_i) \wedge \\ (\forall j \mid 1 \leq j \leq p \cdot \mathit{Consistent}\ (N_j, C, R)) \ \Rightarrow \\ \quad (\exists R \cdot K \sqsubseteq_R K' \wedge (\forall i \mid 1 \leq i \leq n \cdot M_i \sqsubseteq_R M'_i) \wedge \\ \quad\quad (\forall I \cdot \mathit{Inv}\,(C, I) \Rightarrow (\forall j \mid 1 \leq j \leq p \cdot \{R\}\ I\ \{\!|\, N_j \,|\!\}\ \{R\}\ I))) \end{array}$$

Next, we instantiate the existentially quantified relation variable to $R$, getting

$$\begin{array}{l} K \sqsubseteq_R K' \wedge (\forall i \mid 1 \leq i \leq n \cdot M_i \sqsubseteq_R M'_i) \wedge \\ (\forall j \mid 1 \leq j \leq p \cdot \mathit{Consistent}\ (N_j, C, R)) \ \Rightarrow \\ \quad K \sqsubseteq_R K' \wedge (\forall i \mid 1 \leq i \leq n \cdot M_i \sqsubseteq_R M'_i) \wedge \\ \quad (\forall I \cdot \mathit{Inv}\,(C, I) \Rightarrow (\forall j \mid 1 \leq j \leq p \cdot \{R\}\ I\ \{\!|\, N_j \,|\!\}\ \{R\}\ I)) \end{array}$$

Rewriting with the assumptions and simplifying, we get

$$\begin{array}{l} (\forall j \mid 1 \leq j \leq p \cdot \mathit{Consistent}\ (N_j, C, R)) \ \Rightarrow \\ \quad (\forall I \cdot \mathit{Inv}\,(C, I) \Rightarrow (\forall j \mid 1 \leq j \leq p \cdot \{R\}\ I\ \{\!|\, N_j \,|\!\}\ \{R\}\ I)) \end{array}$$

Next, we strip off the universal quantification in the consequent and move $\mathit{Inv}\,(C, I)$ to the antecedent:

$$\begin{array}{l} (\forall j \mid 1 \leq j \leq p \cdot \mathit{Consistent}\ (N_j, C, R)) \wedge \mathit{Inv}\,(C, I) \ \Rightarrow \\ \quad (\forall j \mid 1 \leq j \leq p \cdot \{R\}\ I\ \{\!|\, N_j \,|\!\}\ \{R\}\ I) \end{array}$$

Using standard logical rules for quantifier manipulation, we further reduce this goal to

$$\mathit{Consistent}\ (N_j, C, R) \wedge \mathit{Inv}\,(C, I) \ \Rightarrow\ \{R\}\ I\ \{\!|\, N_j \,|\!\}\ \{R\}\ I$$

Rewriting with the definition of consistency for new methods and using the rule $S \sqsubseteq S' \ =\ (\forall p\ q \cdot p\ \{\!|\, S \,|\!\}\ q \ \Rightarrow\ p\ \{\!|\, S' \,|\!\}\ q)$, we reduce the goal as follows:

$$\begin{array}{l} \mathit{Inv}\,(C, I) \ \Rightarrow \\ \quad \{R\}\ I\ \{\!|\, \mathbf{begin\ var}\ l \cdot b; (\sqcap_{i=1}^{n}\ [q_i]; (\mathbf{skip} \times M_i \downarrow R) \downarrow |\rho_i|; K_i)^*; \mathbf{end} \,|\!\}\ \{R\}\ I \end{array}$$

Using the rule for proving correctness assertions for blocks, we reduce this goal to

$$\begin{array}{l} \mathit{Inv}\,(C, I) \ \Rightarrow \\ \quad (\mathit{true} \times \{R\}\ I) \cap \\ \quad (\mathbf{var}\ l, \mathit{self}, u_j, v_j \cdot b)\ \{\!|\, (\sqcap_{i=1}^{n}\ [q_i]; (\mathbf{skip} \times M_i \downarrow R) \downarrow |\rho_i|; K_i)^* \,|\!\}\ (\mathit{true} \times \{R\}\ I) \end{array}$$

Next, we apply the rules for proving correctness assertions for weak iteration and demonic choice, getting

$$\begin{array}{l} \mathit{Inv}\,(C, I) \;\Rightarrow \\ \quad (\forall i \mid 1 \leq i \leq n \;\cdot \\ \qquad (\mathit{true} \times \{R\}\; I) \cap \\ \qquad (\mathbf{var}\; l, \mathit{self}, u_j, v_j \;\cdot\; b)\; \{\!|\, [q_i]; (\mathbf{skip} \times M_i \downarrow R) \downarrow |\rho_i|; K_i \,|\!\}\; (\mathit{true} \times \{R\}\; I)) \end{array}$$

Stripping off the universal quantification in the consequent and discharging the unnecessary condition $1 \leq i \leq n$, we get

$$\begin{array}{l} \mathit{Inv}\,(C, I) \;\Rightarrow \\ \qquad (\mathit{true} \times \{R\}\; I) \cap \\ \qquad (\mathbf{var}\; l, \mathit{self}, u_j, v_j \;\cdot\; b)\; \{\!|\, [q_i]; (\mathbf{skip} \times M_i \downarrow R) \downarrow |\rho_i|; K_i \,|\!\}\; (\mathit{true} \times \{R\}\; I) \end{array}$$

Applying now the rules for proving correctness assertions for sequential composition, we get

$$\begin{array}{l} \mathit{Inv}\,(C, I) \;\Rightarrow \\ \qquad (\exists r \;\cdot\; (\mathit{true} \times \{R\}\; I) \cap (\mathbf{var}\; l, \mathit{self}, u_j, v_j \;\cdot\; b)\; \{\!|\, [q_i] \,|\!\}\; r \;\wedge \\ \qquad\quad r\; \{\!|\, (\mathbf{skip} \times M_i \downarrow R) \downarrow |\rho_i|; K_i \,|\!\}\; (\mathit{true} \times \{R\}\; I)) \end{array}$$

Next, instantiating $r$ to $(\mathit{true} \times \{R\}\; I)$, we get

$$\begin{array}{l} \mathit{Inv}\,(C, I) \;\Rightarrow \\ \qquad (\mathit{true} \times \{R\}\; I) \cap (\mathbf{var}\; l, \mathit{self}, u_j, v_j \;\cdot\; b)\; \{\!|\, [q_i] \,|\!\}\; (\mathit{true} \times \{R\}\; I) \;\wedge \\ \qquad (\mathit{true} \times \{R\}\; I)\; \{\!|\, (\mathbf{skip} \times M_i \downarrow R) \downarrow |\rho_i|; K_i \,|\!\}\; (\mathit{true} \times \{R\}\; I) \end{array}$$

Applying now the rules for proving correctness assertions for guards and simplifying, we get

$$\begin{array}{l} \mathit{Inv}\,(C, I) \;\Rightarrow \\ \qquad (\mathit{true} \times \{R\}\; I)\; \{\!|\, (\mathbf{skip} \times M_i \downarrow R) \downarrow |\rho_i|; K_i \,|\!\}\; (\mathit{true} \times \{R\}\; I) \end{array}$$

Applying again the rule for proving correctness assertions for sequential composition and instantiating the existentially quantified predicate variable to $(\mathit{true} \times \{R\}\; I)$, we get

$$\begin{array}{l} \mathit{Inv}\,(C, I) \;\Rightarrow \\ \qquad (\mathit{true} \times \{R\}\; I)\; \{\!|\, (\mathbf{skip} \times M_i \downarrow R) \downarrow |\rho_i| \,|\!\}\; (\mathit{true} \times \{R\}\; I) \;\wedge \\ \qquad (\mathit{true} \times \{R\}\; I)\; \{\!|\, K_i \,|\!\}\; (\mathit{true} \times \{R\}\; I) \end{array}$$

Using the assumption that every $K_i$ is terminating and skipping on the attributes of $C'$, and applying the rule for proving correctness assertions (f), we reduce this goal to

$$\mathit{Inv}\,(C, I) \;\Rightarrow\; (\mathit{true} \times \{R\}\; I)\; \{\!|\, (\mathbf{skip} \times M_i \downarrow R) \downarrow |\rho_i| \,|\!\}\; (\mathit{true} \times \{R\}\; I)$$

Using the definition of $\downarrow$ and then the rule for proving correctness assertions for sequential composition, we get

$$\begin{aligned} \mathit{Inv}\,(C, I) \;\Rightarrow\; & (\exists r_1 \cdot (\mathit{true} \times \{R\}\ I)\ \{\!|\ \{|\rho_i|\}\ |\!\}\ r_1 \;\wedge \\ & (\exists r_2 \cdot r_1\ \{\!|\ \mathbf{skip} \times M_i \downarrow R\ |\!\}\ r_2 \;\wedge\; r_2\ \{\!|\ [|\rho_i|^{-1}]\ |\!\}\ (\mathit{true} \times \{R\}\ I))) \end{aligned}$$

Instantiating both $r_1$ and $r_2$ to $(\mathit{true} \times \{R\}\ I)$ and using the fact that $\rho_i$ is a state-reassociating function not updating any of the state components, we reduce the goal to

$$\mathit{Inv}\,(C, I) \;\Rightarrow\; (\mathit{true} \times \{R\}\ I)\ \{\!|\ \mathbf{skip} \times M_i \downarrow R\ |\!\}\ (\mathit{true} \times \{R\}\ I)$$

Applying the rule for proving correctness assertions (e) and then Lemma 3, we reduce this goal to

$$\mathit{Inv}\,(C, I) \;\Rightarrow\; I\ \{\!|\ (M_i \downarrow R) \uparrow R\ |\!\}\ I$$

and then, using the identity of inverse coercion rule, to

$$\mathit{Inv}\,(C, I) \;\Rightarrow\; I\ \{\!|\ M_i\ |\!\}\ I$$

which holds according to the definition of $\mathit{Inv}\,(C, I)$. □

# Paper 5

## Consistent Extension of Components in the Presence of Explicit Invariants

**A. Mikhajlova**

Originally published in *Proceedings of the 29th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 29)*, IEEE Computer Society Press, pp. 76–85, Nancy, France, June 1999.

# Consistent Extension of Components in the Presence of Explicit Invariants *

Anna Mikhajlova

Turku Centre for Computer Science, Åbo Akademi University
Lemminkäisenkatu 14A, Turku 20520, Finland
e-mail: Anna.Mikhajlova@abo.fi

**Abstract.** Component extensions must semantically conform to the components they extend to guarantee consistency of the extended system. Semantic conformance usually means preservation of observable properties while decreasing nondeterminism; in the presence of explicit invariants it also involves preservation of invariants by the extended and the extending components. Depending on the reuse technique employed for constructing extensions, the requirements that must be imposed on components to guarantee consistency vary. We concentrate on the issue of ensuring consistency of extensions with forwarding as the reuse technique, formulating requirements that allow consistent extension of components in the presence of explicit invariants. Also, we discuss additional problems arising with the use of inheritance and propose solutions to these problems.

## 1 Introduction

In an open component-based system, the ultimate goal of creating an extension is to improve and enhance functionality of an existing component by tuning it for specific needs, making it more concrete, implementing a faster algorithm, and so on. Effectively, the client of a component benefits from using its extension, only if the extension does not invalidate the client. Imposing semantic constraints on extensions ensures their consistency from the client perspective.

Explicit invariants binding values of component attributes state the properties the client of a component may safely assume about the component behaviour, and play an important rôle in maintaining consistency of component extensions. We formulate requirements for components and their extensions that guarantee consistency of the extended system in the presence of explicit invariants. We concentrate on the issue of extension

consistency for component-based systems employing forwarding as the reuse technique, in the style of Microsoft COM [21]. Our analysis indicates that ensuring consistency of component extensions in the presence of explicit invariants is easier with forwarding than with inheritance, and we discuss the additional consistency problems that arise with the use of inheritance. Based on our analysis we present solutions to the described problems.

## 2 Components, contracts, and invariants

The notion of a component does not have a standardised meaning and various researchers and practitioners understand under components fairly different things. For our purposes it is irrelevant whether a component conceptually is a distributed object whose methods are subject to remote procedure calls or an object having a graphical user interface through which the user can compose it with other components in an application. We view a component as an abstract data type having an encapsulated local state, carried in component attributes, and a set of globally visible methods, which are used to access the attributes and modify them. In addition, every component usually has a constructor, initialising the attributes. Each component implements a certain interface, which is a set of method signatures, including the name and the types of value and result parameters. An extending component implements an interface which includes all method signatures of the original component and, in addition, may have new method signatures. This conformance of interfaces forms a basis for subtype polymorphism and subsumption of components.

We consider a component composition scenario when a component is delivered to a client, who might also be an extension developer, as a formal specification with the implementation hidden behind this specification. For motivation of such an organisation of a component market see, e.g., [5, 20]. In general, several components can implement the same specification, and one component can implement several different specifications. We assume that the specification language combines standard executable statements, such as assignments, conditionals, and loops, with specification statements including assertions, assumptions, and nondeterministic specification statements, which abstractly yet precisely describe the intended behaviour. A proposal and motivation for such a language are given in [5], and formal semantics of the involved constructs may be found in [18].

Essentially, the formal specification of a component is a contract binding the developer of the implementation and the clients, including extension developers. Assumptions $[p]$ and assertions $\{p\}$ of a state predicate $p$ are the main constituents of such a contract. The assumptions state expectations of one party that must be met by the other party, whereas the assertions state promises of one party, that the other party may rely on. Naturally, the assumptions of one party are the assertions of the other and vice versa. When a party fails to keep its promise (the asserted predicate does not hold in a state), this party aborts. When the assumptions of a party are not met (the assumed predicate does not hold in a state), it is released from the contract and the other party aborts. For a detailed discussion of contracts and their formal semantics see, e.g., [2].

Invariants binding values of component attributes play an important rôle in maintaining consistency of component extensions. An implicit, or the strongest, invariant characterises exactly all reachable states of the component, whereas an explicit invariant restricts the values the component might have. The implicit invariant is established by the component constructor, preserved by all its methods, and can be calculated from the component specification. As suggested by its name, the explicit invariant, on the other hand, is stated explicitly in the component specification, being part of the contract the component promises to satisfy. The component developer is supposed to meet the contract by verifying that the constructor establishes the explicit invariant and all methods preserve it.

The advantages of stating invariants explicitly in abstract data types and especially components have been stressed by several researchers, e.g., [11, 5, 14]. Indeed, explicit invariants are useful for facilitation of implementation, consistency checking, and for guiding revisions and extensions. Namely, component designers might want to guarantee that certain invariant holds in all states, and by explicitly stating this fact in the component specification, they would allow clients to assume it. In most existing component frameworks the implicit invariant is not safe to assume, and clients relying on it may get invalidated. This is especially the case when one component implements several specifications with different interfaces. One client, using this component as the implementation of a certain specification, may take it to a state which is perceived as unreachable from the perspective of another client having a different specification of this component's behaviour. Moreover, the implicit invariant is, in general, stronger than necessary, and preserving it in client extensions might be too restrictive. When one component implements several specifications,

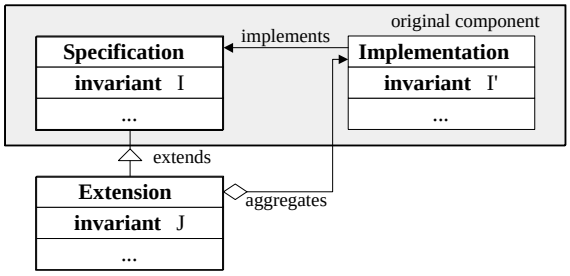



**Fig. 1.** Illustration of component extension layout

ensuring that it preserves the strongest invariants of all these specifications can be unimplementable.

Explicit invariants are supported by at least one programming language, Eiffel [15], and although, to our knowledge, none of the existing component-based systems supports explicit invariants at present, research in this direction is underway.

## 3 Consistent extension of components in the presence of explicit invariants

When the implementation of a component is hidden behind a formal specification, with the invariant stated explicitly in the latter, both the extension and the implementation must satisfy certain requirements with respect to this invariant to obtain a consistent component composition. In what follows, we will refer to the component being extended as the original component, distinguishing when necessary between its specification and implementation. The corresponding layout is illustrated in Fig. 1.

### 3.1 Requirements imposed on extensions

When extension is achieved through forwarding, the extending component plays a dual rôle. On one hand, it offers services to the clients of the original component, when substituted for that component, and, on the other hand, it is a client of the original component. This duality requires that the extension matches in a certain sense the contract of the original component specification and simultaneously satisfies this contract.

Let us analyse the restrictions that must be imposed on components and their extensions to guarantee consistency. Consider a component *Bag*, which represents a bag of characters, and its extension *CountBag*, which aggregates *Bag* and has an attribute of its own $n : \mathsf{Nat}$:

$Bag$ = **component**
  $b$ : **bag of** Char
  **invariant** $I$ = $\#b \leq max$
  $Bag()$ = $b := [\![]\!]$
  $Size($**res** $r$ : Nat$)$ = $r := \#b$
  ...
**end**

$CountBag$ = **component extends** $Bag$
  $Bag$; $n$ : Nat
  **invariant** $J$ = $\#Bag.b \leq max \wedge$
    $\#Bag.b = n$
  $CountBag()$ = $Bag(); n := 0$
  $Size($**res** $r$ : Nat$)$ = $r := n$
  ...
**end**

As such, *Bag* is the specification of a component whose implementation is hidden from the developer of *CountBag*. By aggregating *Bag*, the extension *CountBag*, in fact, aggregates its implementation which will be substituted at run-time. The invariant of *Bag* states that the size of the bag does not exceed some constant value *max*, and the invariant of *CountBag* in addition stipulates that $n$ is the counter of elements in the bag. Maintaining the explicit invariant means that the body of the constructor *Bag* is equal to $b := [\![]\!]; \{I\}$, and the body of the method *Size* in the component *Bag* is equal to $[I]; r := \#b; \{I\}$. Similarly, bodies of *CountBag* and *Size* in the component *CountBag* are equal respectively to $Bag(); n := 0; \{J\}$ and $[J]; r := n; \{J\}$. That is, the constructors unconditionally assert the corresponding invariant, whereas the methods, assuming that the invariant holds in the beginning, promise to establish it in the end.

It is well-known from the theory of abstract data types that to guarantee safe substitutability, the invariant in the more concrete ADT must be stronger than that in the more abstract ADT. Similarly, a client of the component *Bag*, assuming the invariant $I$ maintained by *Bag*, can safely use the extension *CountBag* only if the invariant $J$ is at least as strong as $I$, written $J \subseteq I$. Note that by establishing $J$, the constructor and the methods of *CountBag* also establish $I$, which is a weaker property. Also, the assumption of the weaker invariant $I$ passes through, whenever the stronger assumption $[J]$ holds. For example, consider specifications of method *Contains* in *Bag*

$$Contains(\textbf{val}\ c : \mathsf{Char}, \textbf{res}\ r : \mathsf{Bool}) = [I]; r := c \in b; \{I\}$$

and in *CountBag*

$$Contains(\textbf{val}\ c : \mathsf{Char}, \textbf{res}\ r : \mathsf{Bool}) = [J]; Bag.Contains(c, r); \{J\}$$

To facilitate reasoning, we have written out assumptions and assertions of the corresponding invariants. Substituting the definition of *Bag* :: *Contains* for the method invocation, we have that the body of *Contains* in *CountBag* is equal to

$$[J]; [I]; r := c \in Bag.b; \{I\}; \{J\}$$

Since $[J]; [I] = [J]$ and $\{I\}; \{J\} = \{J\}$, for all $I, J$ such that $J \subseteq I$, this is further equal to

$$[J]; r := c \in Bag.b; \{J\}$$

Essentially, this means that, if the assumption $[J]$ holds, then the assumption $[I]$ skips, and also means that establishing $J$ establishes $I$ as well.

The extension has no way of breaking the invariant of the original component because it changes the attributes of the latter only by invoking its methods, which are guaranteed to preserve the invariant. However, if a component breaks its own invariant before invoking its own methods, then the assumption of this invariant in the self-called methods will lead to a crash. Consider, for example, specifications of methods *Add* and *AddSet*:

```
Bag = component
  ...
  Add(val c : Char) =
    [I]; if #b < max then
           b := b ∪ ⟦c⟧
         else skip fi;
    {I}
  AddSet(val nb : set of Char) =
    [I];
    [b := b' | #b' ≤ max ∧
       b ⊆ b' ∧ b' ⊆ b ∪ nb];
    {I}
```

```
CountBag = component extends Bag
  ...
  Add(val c : Char) =
    [J]; if n < max then
           n := n + 1; Bag.Add(c)
         else skip fi;
    {J}
  AddSet(val nb : set of Char) =
    [J]; if n + #nb > max then
           n := max
         else n := n + #nb fi;
         while nb ≠ ⟦⟧ do
           begin var c · c ∈ nb;
             self.Add(c); nb := nb − ⟦c⟧
           end
         od;
    {J}
```

Here, the addition of elements from the set $nb$ to the original bag is specified in *Bag* using the nondeterministic specification statement [3] which expresses that $b$ is assigned a nondeterministically chosen value $b'$, satisfying the invariant, and also satisfying the conditions that the previous value of $b$ is included in $b'$ and that every element in $b'$ comes

either from $b$ or from $nb$. The extension redefines the method *AddSet* by updating the counter and iteratively adding elements from $nb$ to the bag until $nb$ becomes empty. The method *Add*, iteratively invoked in *CountBag* :: *AddSet*, adds elements to the bag provided that its size is smaller than the maximum, and otherwise does nothing. Therefore, it may seem that everything should work fine, if there are more elements in $nb$ than can be added to the bag, then the extra elements are simply discarded. However, only when $nb$ is empty will this method work correctly, as the method *Add* promises to carry out its contract under the assumption that the counter $n$ is equal to the size of the bag, and this assumption is broken by the conditional statement updating $n$. The broken assumption will result in aborting the party which initiated the self-call, namely, the method *AddSet*. To fix the problem, it is sufficient to remove the conditional statement altogether, since then the invariant $J$ is preserved before all self-calls to *Add*; adding $(n < max)$ to the termination condition of the while-loop would also eliminate unnecessary iteration steps.

Based on this example, we formulate a requirement that the invariant must be re-established in a component before every self-call to a method which makes use of the assumed invariant. Naturally, this requirement must be satisfied in all components under consideration, including specification components and implementation components.

Unfortunately, the three requirements we have stated so far, namely, preserving the corresponding invariants in the original component specification and its extension, strengthening the specification invariant in the extension, and re-establishing the corresponding invariants before self-calls, do not alone guarantee consistency of extensions as seen from the client perspective. For example, the method *AddSet* of *CountBag* could preserve the invariant, yet add completely arbitrary elements to the *Bag*. The client of the original component, when using the extension, could become invalidated if, after passing a certain set of characters to *AddSet*, it would suddenly discover that the resulting component contains characters different from those that it expects.

To avoid this kind of situation, the extension developer should ensure that the extension $E$ is a *refinement* of the original component specification $S$, written $S \sqsubseteq E$. Refinement means preservation of observable behaviour, while decreasing nondeterminism. To verify that $E$ is a refinement of $S$, one must show that the constructor of $E$ refines the constructor of $S$ and that the methods of $E$ refine the corresponding methods of $S$.

Usually, this verification will be trivial in case a method is forwarded to the corresponding method of the aggregated component.

Since the new methods added in the extension may not be invoked by the client of the original component, there is no danger of breaking client expectations about their behaviour. The only requirements the new methods must satisfy are preserving the invariant of the extension and re-establishing this invariant before self-calls.

### 3.2 Requirements imposed on implementations

The implementation of a component has the freedom to change the attributes of the specification completely, being hidden behind this specification. However, in the presence of an explicit invariant, it must ensure that the new attributes are such that it is possible to formulate an invariant which is stronger than the specification invariant with respect to an *abstraction relation*. Let us illustrate this idea with an example of a bag implementation using an array to represent a bag.

$$
\begin{array}{l}
\mathit{BagImp} \;=\; \textbf{component implements } \mathit{Bag} \\
\quad \mathit{bag} : \textbf{array } [1..\mathit{max}] \textbf{ of } \mathsf{Char}, \mathit{size} : \mathsf{Nat} \\
\quad \textbf{invariant } I' \;=\; \mathit{size} \leq \mathit{max} \\
\quad \mathit{BagImp}() \;=\; \mathit{size} := 0 \\
\quad \mathit{Add}(\textbf{val } c : \mathsf{Char}) \;= \\
\qquad \textbf{if } \mathit{size} < \mathit{max} \textbf{ then} \\
\qquad\quad \mathit{size} := \mathit{size} + 1; \mathit{bag}[\mathit{size}] := c \\
\qquad \textbf{else skip fi}; \\
\quad \ldots \\
\textbf{end}
\end{array}
$$

The attribute *size* keeps the number of elements in the array *bag* which have, so far, been added to this array. Note that the number of all elements in the array is always *max* with the elements from $\mathit{size}+1$ to *max* being arbitrary characters. The invariant $I'$ does not, as such, give us all this information and it has no way of doing so. The implementation invariant, in general, has to be related to the specification invariant via an abstraction relation which stipulates how the abstract attributes can be coerced to the concrete attributes. In the case of our example, the abstraction relation $R$ is such that

$$
\begin{array}{rcl}
R\,(\mathit{bag}, \mathit{size})\,(b) & = & (\mathit{toBag}(\mathit{bag}, \mathit{size}) = b), \text{ where} \\
\mathit{toBag}(a, 0) & = & [\![]\!] \\
\mathit{toBag}(a, n) & = & \mathit{toBag}(a, n-1) \cup [\![a[n]]\!]
\end{array}
$$

As we have already mentioned, the invariant $I'$ of the implementation must be stronger than the invariant $I$ of the specification with respect to an abstraction relation $R$. Namely, for implementation attributes $c$ and specification attributes $a$ we have,

$$I' \subseteq_R I \quad = \quad I'\, c \Rightarrow (\exists a \cdot R\, c\, a \wedge I\, a)$$

Intuitively this means that whenever the attributes $c$ satisfy the invariant $I'$ there exist images $a$ of these attributes with respect to $R$ satisfying the invariant $I$. The rationale behind this requirement is similar to the one for the requirement that the extension invariant must be stronger than the specification invariant. Strictly speaking, the former must be stronger than the latter also with respect to an abstraction relation which is a projection. In both the extension and the implementation if the attributes of the specification are not changed, the abstraction relation is the identity. The relation between the invariants in our example is expressed by

$$I' \Rightarrow (\exists b' \cdot R\,(bag, size)\,(b') \wedge I[b \leftarrow b'])$$

where $I[b \leftarrow b']$ stands for $I$ with all occurrences of $b$ substituted with $b'$. It is easy to see that this relation is satisfied since instantiating $b'$ with $toBag(bag, size)$ reduces it to

$$size \leq max \Rightarrow$$
$$\quad toBag(bag, size) = toBag(bag, size) \wedge \#toBag(bag, size) \leq max$$

which obviously holds since $\#toBag(a, n) = n$.

Just as was the case with the specification and the extension, semantic conformance in the form of refinement must be established between the specification of the component and its implementation. Namely, it must be verified that the concrete constructor refines the abstract one with respect to an abstraction relation, and that every method of the implementation refines the corresponding method of the specification with respect to the same relation. For details we refer to [18, 1].

As was mentioned in Sec. 2, one specification can be implemented by several components, and each component can implement several specifications with different interfaces. In the latter case the implementation must semantically conform to every specification it implements. For example, apart from *BagImp*, the specification of *Bag* can also be implemented by, e.g, *EntryField* which is a standard widget used in dialog boxes. As such,

*EntryField* must also implement a component specification *Widget* with interface including such methods as *Move*, *Resize*, *HandleKey* and so on.

In general, if a component $C$ implements several component specifications $S_1, \ldots, S_n$, maintaining the invariants $I_1, \ldots, I_n$, then the invariant $J$ of $C$ must be stronger than all of $I_i, i = 1..n$ with respect to the corresponding abstraction relations: $J \subseteq_{R_1} I_1 \wedge \ldots \wedge J \subseteq_{R_n} I_n$. The constructor of $C$ must establish $J$, all methods must preserve it and re-establish before self-calls. Moreover, every method of $C$ must refine the corresponding method of $S_i$ with respect to the abstraction relation $R_i$.

## 4 Explicit invariants and inheritance

Inheriting attributes of an original component opens a possibility for method inheritance through super-calling from an extension methods of the original component. Moreover, self-referential method invocations, also known as call-backs, become possible in this case. As such, a component and its extension become mutual clients, and are required to satisfy each other's contracts when invoking methods via self and super. Unfortunately, re-establishing invariants before all self and super-calls does not solve all problems and does not guarantee consistency. Consider components $Bag'$ and $CountBag'$ inheriting from $Bag'$:

```
Bag' = component
  b : bag of Char
  invariant I = #b ≤ max
  ...
  Add(val c : Char) =
    [I]; if #b < max then
           b := b ∪ ⟦c⟧
         else skip fi;
    {I}
  AddSet(val nb : set of Char) =
    [I]; while nb ≠ ⟦⟧ do
           begin var c · c ∈ nb;
             self.Add(c);
             nb := nb − ⟦c⟧
           end
         od;
    {I}
```

```
CountBag' = component inherits Bag'
  n : Nat
  invariant J = I ∧ #b = n
  ...
  Add(val c : Char) =
    [J]; if n < max then
           n := n + 1; super.Add(c)
         else skip fi;
    {J}
  AddSet(val nb : set of Char) =
    [J]; if n + #nb > max then
           n := max
         else n := n + #nb fi;
         super.AddSet(nb);
    {J}
```

In the specification of method *Add* in $CountBag'$ incrementing $n$ before the super-call breaks the extending part $\#b = n$ of the invariant $J$.

Since $Bag'$ :: $Add$ only relies on $I$ which holds before super-calling $Add$, everything works fine and the incoming element is added to the bag. However, breaking the extending part of the invariant in $AddSet$ before the super-call leads to a crash, because $J$ does not hold before a self-call to $Add$ in $Bag'$ :: $AddSet$ and this call gets redirected to $CountBag'$.

The analysis of this example indicates two solutions to the problem. Since an original component is, in general, unaware of extensions and their invariants, there is no possibility of re-establishing such invariants in the original component before self-calls. The question is then whether the original component can preserve the extension invariant or not. If it can preserve the extension invariant then for avoiding the problem it is sufficient to establish this invariant before super-calls to the original component; otherwise, another solution is required. The extension invariant can be preserved in the original component if it can be split into the inherited part and the extending part, with the inherited part being the invariant of the original component and the extending part placing constraints only on new attributes, without relating them to the inherited attributes.

Most often, however, we would want to relate new attributes to the inherited ones, like in our example when the invariant $J$ inherits $I$ and restricts $n$ to be the counter of elements in bag $b$. Re-establishing the extension invariant before super-calls would not help in this situation, because modification of attributes in the original component can break it and any following call-back would crash. Only if the original and the extension invariants are equal with respect to an abstraction relation, re-establishing them before self- and super-calls solves the problem. Namely, if the original and the extension invariants $I$ and $J$ are such that $I =_R J$, then re-establishing $I$ before all self-calls in the original component and before all self- and super-calls in the extension eliminates the possibility of crashing.

It may seem that explicit invariants are to be blamed for all these problems and complications, but even without them inheritance across component boundaries is known to create a lot of consistency problems. Also, disciplining inheritance [17] helps to avoid the problems, but reduces flexibility, which is always advocated as the major advantage of inheritance over forwarding. For instance, our interest in explicit invariants was initiated when analysing the problems the addition of new methods creates in the presence of subtype aliasing [13]. Our analysis reveals that when invariants are not stated explicitly, new methods do not introduce inconsistencies only if they preserve the strongest invariant maintained

by the original component. In practice this means that the new methods may change the additional attributes of the extension and invoke the old methods. Obviously, this is no much more flexible than what can be done with forwarding.

## 5 Conclusion

Explicit statement of invariant properties benefits in a number of ways independent component composition and extensibility. However, to achieve consistency in component composition, including composition of original components with their extensions, certain requirements must be imposed on all components. We have analysed the issue of consistent component composition, illustrating possible consistency problems, and formulated the requirements that must be imposed on components to avoid such problems.

Our analysis has focused on the scenario when a client or an extension developer works with a component seeing only its formal specification, explicitly stating invariant properties, whereas the actual implementation is hidden from the client. This layout, which we consider to be the most promising foundation for a component market, requires that all implementations of a component semantically conform to the specification of that component, and so do all extensions. Semantic conformance means consistency with respect to explicit invariants and preservation of observable properties while decreasing nondeterminism. We also considered the additional problems which arise with the use of inheritance and proposed solutions to these problems.

Although the rôle of explicit invariants has been considered before by various researchers, e.g., [14, 9], we discuss this issue in proper relief, accenting on details specific to component- and class-based systems with a particular composition scenario. Meyer's "Design by Contract" [14, 16] advocates the use of explicit invariants in classes, and Eiffel [15] even supports the proposed construct. However, the problems with explicit invariants in the presence of dynamic binding of self-referential methods are not discussed in [14, 16]. Neither they are discussed in works on various extensions of the specification language Z [7, 8, 6, 12], where inheritance is considered between class specifications only. As we treat specifications and implementations in a unified logical framework, considering specifications to be abstract programs and implementations – executable specifications, we can abstract away from implementation details through using specification statements and yet express complex behavioural de-

pendencies by including method calls in the specifications. This approach allows us to reason about specifications and implementations in a uniform manner, identifying the existing problems such as, e.g., the effect of self-reference in the presence of explicit invariants. Contracts of Helm et al. [9], as in our approach, are intended to capture behavioural dependencies and also support invariants. However, they are separated from classes, which have to be mapped to the contracts via a conformance declaration. Alternatively, we capture the behavioural dependencies by using specification statements and explicitly including method calls in specifications. As such, class or component methods, on both the abstract and the concrete levels, are the contracts themselves. Besides, as contracts in [9] have no formal semantics, reasoning about conformance and refinement can only be done informally, whereas in our framework every construct has a precise mathematical meaning as described in [18, 2].

The related work on abstract data types [10, 11] usually concentrates on algebraic specifications of methods and procedures, where self-calls are not present altogether, concealing the problems with re-establishment of invariants. Treatment of explicit invariants in the presence of dynamic binding of self-referential methods also requires special consideration, whereas the classical theory of ADTs does not deal with this issue.

A recently completed work on specification and verification of Java Collections Framework [4] reported in [19] makes extensive use of explicit (class) invariants and demonstrates verification of refinement and consistency with respect to explicit invariants. Dealing only with specifications of Java interfaces and not concrete classes related through inheritance, the work in [19] does not address the issue of ensuring consistency of explicit invariants in the presence of self-referential method calls. Verifying practical applicability of our recommendations in this case as well appears to be of interest and represents the topic of current research. Perhaps the restrictions we imposed on components and their extensions can be relaxed, and investigating the possibilities for doing so is a topic for future work.

## References


1. 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. http://www.tucs.abo.fi/publications/techreports/TR147.html.
2. R. J. R. Back and J. von Wright. Contracts, games and refinement. In *4th Workshop on Expressiveness in Concurrency, EXPRESS'97*, volume 7 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 1997.

3. R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.
4. J. Bloch. Java Collections Framework: Collections 1.2. http://java.sun.com/docs/books/tutorial/collections/index.html.
5. M. Büchi and E. Sekerinski. Formal methods for component software: The refinement calculus perspective. In *Second Workshop on Component-Oriented Programming (WCOP'97) held in conjunction with ECOOP'97*, pages 23–32. TUCS General Publication, No. 5, June 1997.
6. D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In *Formal Description Techniques (FORTE '89), Vancouver*, pages 281–296. North-Holland Publishing Co., Dec. 1989.
7. E. Cusack. Inheritance in object-oriented Z. In P. America, editor, *Proceedings of ECOOP'91*, LNCS 512, pages 167–179, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.
8. E. Cusack and M. Lai. Object-oriented specification in LOTOS and Z, or my cat really is object-oriented! In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *LNCS*, pages 179–202. Springer-Verlag, New York, N.Y., 1991.
9. R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Proceedings of OOPSLA/ECOOP'90*, ACM SIGPLAN Notices, pages 169–180, Oct. 1990.
10. C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.
11. C. Jones. *Systematic Software Development Using VDM*. Prentice–Hall International, 1986.
12. K. Lano and H. Haughton. Reasoning and refinement in object-oriented specification languages. In O. L. Madsen, editor, *Proceedings of ECOOP'92*, LNCS 615. Springer-Verlag, 1992.
13. B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
14. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct. 1992.
15. B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.
16. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.
17. L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In E. Jul, editor, *Proceedings of ECOOP'98*, pages 355–382. Springer, July 1998.
18. A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In *Proceedings of 4th International Formal Methods Europe Symposium, FME'97*, LNCS 1313, pages 82–101. Springer, 1997.
19. A. Mikhajlova and E. Sekerinski. Ensuring correctness of Java frameworks: A formal look at JCF. Technical Report 250, Turku Centre for Computer Science, March 1999.
20. C. Szyperski. *Component Software – Beyond Object-Oriented Software*. Addison-Wesley, 1997.
21. S. Williams and C. Kinde. The component object model: Technical overview. *Dr. Dobbs Journal*, December 1994.

# Paper 6

## Ensuring Correctness of Java Frameworks: A Formal Look at JCF

**A. Mikhajlova and E. Sekerinski**

Published as a technical report of Turku Centre for Computer Science, TUCS-TR-250, March 1999.
A shorter version of this paper by Anna Mikhajlova appeared in *Proceedings of the 30th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 30)*, IEEE Computer Society Press, pp. 136–145, Santa Barbara, USA, August 1999.

# Ensuring Correctness of Java Frameworks: A Formal Look at JCF


## Anna Mikhajlova

Turku Centre for Computer Science,
Åbo Akademi University
Lemminkäisenkatu 14A, Turku 20520, Finland

## Emil Sekerinski

McMaster University,
1280 Main Street West, Hamilton,
Ontario, Canada, L8S4K1




**Turku Centre for Computer Science**
**TUCS Technical Report No 250**
**March 1999**

**ISBN 952-12-0402-8**
**ISSN 1239-1891**

**Abstract**

In this paper we propose a novel approach to specification, development, and verification of object-oriented frameworks employing separate interface inheritance and implementation inheritance hierarchies. In particular, we illustrate how our method of framework specification and verification can be used to specify the Java Collections Framework, which is a part of the standard Java Development Kit 2.0, and ensure its correctness. We propose to associate with Java interfaces formal descriptions of the behavior that classes implementing these interfaces and their subinterfaces must deliver. Verifying behavioral conformance of classes implementing given interfaces to the specifications integrated with these interfaces allows us to ensure correctness of the system.

The characteristic feature of our specification method is that the specification language used combines standard executable statements of the Java language with possibly nondeterministic specification statements. A specification of the intended behavior of a particular interface given in this language can serve as a precise documentation guiding implementation development. Since subtype polymorphism in Java is based on interface inheritance, behavioral conformance of subinterfaces to their superinterfaces is essential for correctness of object substitutability in clients. As we view interfaces augmented with formal specifications as abstract classes, verifying behavioral conformance amounts to proving *class refinement* between specifications of superinterfaces and subinterfaces. Moreover, the logic framework that we use also allows verification of behavioral conformance between specifications of interfaces and classes implementing these interfaces. The uniform treatment of specifications and implementations and the relationships between them permits verifying correctness of the whole framework and its extensions.

**Keywords:** formal specification, reasoning, object-oriented frameworks, separate subtyping and subclassing, nondeterminism, correctness, verification, class refinement, Java

**TUCS Research Group**
Programming Methodology Research Group

# 1 Introduction

One of the defining features of object-oriented frameworks is their extensibility, i.e., the ability of frameworks to call user extensions. In view of this important characteristic it is critical to build user extensions which *behaviorally conform* to the part of the framework that they extend. Moreover, frameworks themselves are usually build in a hierarchical manner, starting with some basic functionality and specializing this functionality in various directions to meet different demands. Naturally, behavioral conformance also underlies this hierarchy, with the most general behavior at the top level and specialized or refined behaviors at the lower levels. Verification of behavioral conformance both within a framework and between the framework and its extensions is critical for ensuring correctness and reliability of the resulting system.

In this paper we propose a specification and verification method supporting development of provably correct object-oriented frameworks. The method has been originally described in [19] in application to systems with unified interface and implementation inheritance hierarchies. Here we focus on object-oriented frameworks employing separate interface inheritance and implementation inheritance hierarchies and illustrate how our method of framework development can be used to specify the Java Collections Framework (JCF) and ensure its correctness. Essentially, we propose to associate with Java interfaces formal descriptions of the behavior that classes implementing these interfaces and their subinterfaces must deliver. Interfaces always have an informal semantics as expressed in their names and in the names and parameter types of their methods, we just make this semantics explicit and express it mathematically. Such formal specifications can be distributed as part of the framework documentation, contributing to the detailed understanding of its functionality and guiding extension development. The characteristic feature of our specification method is that the specification language used combines standard executable statements of the Java language with possibly nondeterministic specification statements. Every statement in this language has a precise mathematical meaning in the refinement calculus as described in [19, 3]. In this paper we present only informal explanations of the specification constructs used in specifications of JCF interfaces.

Since subtype polymorphism in Java is based on interface inheritance, behavioral conformance of subinterfaces to their superinterfaces is essential for correctness of object substitutability in clients. Our verification of behavioral conformance is based on the notion of *class refinement* first described in [19] and developed in [3]. One class (usually more abstract or nondeterministic) is refined by another class (usually more concrete or deterministic) if the externally observable behavior of the first class is preserved in the second class while decreasing nondeterminism. For a detailed description of

refinement in the refinement calculus we refer to [21, 6]. Class refinement per se is based on data refinement [11, 14, 20, 4] which takes place when a state space is changed in a refinement step. An extensive collection of "high level" refinement laws that has been developed within the refinement calculus permits verification of class refinement in practice, and mechanical verification tools that are currently being developed [9] open the possibility of mechanized verification.

As we view interfaces augmented with formal specifications as abstract classes, verifying behavioral conformance amounts to proving class refinement between specifications of superinterfaces and subinterfaces. Moreover, the logical framework that we use also enables verification of behavioral conformance between specifications of interfaces and classes implementing these interfaces: class refinement must be established between the specification and the implementation classes. The uniform treatment of specifications and implementations and the relationships between them permits us to verify correctness of the original framework and then prove that user extensions preserve this correctness, ensuring in this way the correctness of the whole system.

The paper is organized as follows. In Sec. 2 we describe the Java Collections Framework which we use to illustrate our approach, and specify the interface *Collection* with its *Iterator* and the interface *List* with its *ListIterator*. Our specifications are entirely based on informal descriptions of the interface semantics as described in [7], and we reflect on the clarity and preciseness of these descriptions. In Sec. 3 we explain the notion of class refinement, present a number of refinement laws, and demonstrate verification of class refinement between the specifications of *Iterator* and *ListIterator*. Finally, in Sec. 4 we draw some conclusions, and describe future work.

Notation. We use *simply typed higher-order logic* as the logical framework in the paper. The type of functions from a type $\Sigma$ to a type $\Gamma$ is denoted by $\Sigma \to \Gamma$ and functions can have arguments and results of function type. Functions can be described using $\lambda$-abstraction and we write $f\ x$ for the application of function $f$ to argument $x$. Whenever necessary to clarify the argument in the application of a function, especially in the case when the argument is a tuple of elements, we also use brackets around the argument, writing $f(x)$.

The use of equality and assignment symbols deserves special attention. The Java language uses = to denote assignment and == to denote equality of two values, and we will follow this convention in specifications. However, being reluctant to redefine the symbol = traditionally used to denote mathematical equality, we will also use it in logical formulas and definitions, clarifying the intended meaning when necessary. In particular, we will use = rather than == to represent logical equality on right-hand sides of definitions and between the two parts of equational rules.

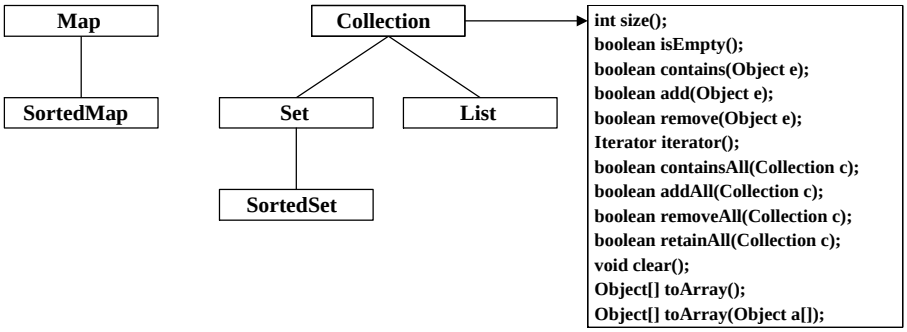



Figure 1: Collection hierarchy

# 2 Specifying the Java Collections Framework

As was stated in the documentation of JCF [7], "A collections framework is a unified architecture for representing and manipulating collections." This particular framework contains three parts: interfaces, implementations, and algorithms. In this paper we focus on the interfaces, formalizing their informal descriptions as given in [7] and studying behavioral conformance between formal specifications of interfaces and formal specifications of their subinterfaces as we define them. The following description of JCF is based on [7].

The interfaces at the core of JCF form a hierarchy as shown in Fig. 1. The root of the hierarchy, the *Collection* interface, represents a group of objects, known as its elements. *Collection* is used to pass collections around and manipulate them when maximum generality is desired. Some *Collection* specializations allow duplicate elements and others do not. Some are ordered and others are not. For example, *Set* is an unordered collection that cannot contain duplicate elements, and *List* is an ordered collection that can contain duplicates.

## 2.1 Specifying the Collection Interface

In the *Collection* interface the method names suggest the intended functionality, for example, the method *size* returns the size of the underlying collection. The interface type *Iterator* returned by the method *iterator* is used to access collection elements and structurally modify the collection. In Fig. 2 we illustrate the hierarchy formed by *Iterator* and its subinterface *ListIterator*. The methods *hasNext*, *next*, and *remove* check whether there are more elements in the collection, return the next element, and remove the current element respectively. The description of JCF states that the behavior of an iterator is unspecified if the underlying collection is structurally modified while the iteration is in progress in any way other than by calling

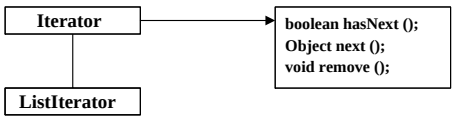



Figure 2: Iterator hierarchy

the method *remove*.

To specify the behavior of *Collection* methods we must model the underlying data structure the methods operate on. It appears to be rather natural to model this data structure by a bag (multiset) of *Object*[1] elements, as we want the collection to contain polymorphic elements, possibly duplicated or unordered. Furthermore, to specify the history of structural modifications, we will use an integer attribute *modified* which will be increased whenever elements are added to the original collection or removed from it. We begin with specifying the data attributes, the constructor, and the basic operations of *Collection* as follows:

```
public interface Collection {
  bag of Object elems;
  int modified;
  Collection() {
    elems, modified = ⟦⟧, 0;
  }
  int size() {
    return min(#elems, Integer.MAX_VALUE);
  }
  boolean isEmpty() {
    return (#elems == 0);
  }
  boolean contains(Object o) {
    return (o ∈ elems);
  }
  boolean add(Object o) {
    boolean r | r == false;
    if (o ∈ elems){
      choose {skip;}
       or {elems, modified, r = elems + ⟦o⟧, modified + 1, true;};
    }
    else {elems, modified, r = elems + ⟦o⟧, modified + 1, true;};
```

[1]In Java the standard class *Object* is a superclass of all other classes, and a variable of type *Object* can hold a reference to an object of any other type.

```
    return r;
}
boolean remove(Object o) {
    boolean r | r == false;
    if (o ∈ elems){
        elems, modified = elems \ o, modified + 1;
        r = true;
    }
    return r;
}
Iterator iterator() {
    Iterator i = new Iterator(this);
    return i;
}
```

In this specification highlighted in bold is the original *Collection* interface and the rest is the precise description of the intended behavior. The behavior of the constructor and the methods is specified in terms of operations on bags and integers, with # returning the number of elements in a bag, $\in, +$, and $\setminus$ representing containment of an element in a bag, bag summation, and element removal respectively:

$$\begin{array}{rcl} \#b & \widehat{=} & \sum e \bullet b\ e \\ e \in b & \widehat{=} & b\ e > 0 \\ b_1 + b_2 & \widehat{=} & (\lambda e \bullet b_1\ e + b_2\ e) \\ b \setminus a & \widehat{=} & (\lambda e \bullet (e = a)\,?\,\mathsf{max}\,((b\ e - 1),\, 0) : b\ e) \end{array}$$

As bags are functions from elements to the number of their occurrences, function application $b\ e$ returns the number of elements $e$ in the bag $b$. In the last definition the equality on the right-hand side of definition sign $\widehat{=}$ is the logical equality. The conditional expression $b\,?\,e_1 : e_2$ is equal to the expression $e_1$ if the boolean condition $b$ holds and to $e_2$ otherwise.

Finally, the statement $\mathsf{choose}\ S_1\ \ \mathsf{or}\ \ldots\ \ \mathsf{or}\ S_n$, used in the specification of the method *add*, represents a nondeterministic choice between the alternatives $S_1$ through $S_n$.[2]

Although the specifications of the constructor and the methods intuitively are quite straightforward, a few points are of interest here. First of all, assignment of a bag to a variable of type *bag of Object*, as in the constructor, results in the corresponding variable containing the value which

[2]Dijkstra's nondeterministic choice statement is usually written as $S_1 \,[\!]\, \ldots \,[\!]\, S_n$ or $S_1 \sqcap \ldots \sqcap S_n$, e.g., in [13, 6]. Here we use the syntax $\mathsf{choose}\ S_1\ \ \mathsf{or}\ \ldots\ \ \mathsf{or}\ S_n$ instead because we believe that it improves readability.

is equal to the value being assigned, in this case $[\![]\!]$, with equality on bags defined as follows:

$$b1 == b2 \quad \widehat{=} \quad (\forall e \bullet b1\ e\ =\ b2\ e)$$

The description of method *contains* in [7] states that this method "returns true if and only if this Collection contains at least one element $e$ such that $(o == null\ ?\ e == null\ :\ o.equals\,(e))$". Looking up the description of method *Object.equals*, we see that "for any reference values $x$ and $y$, this method returns true if and only if $x$ and $y$ refer to the same object ($x == y$ has the value true)". Our specification states that the object reference $o$, be it a null or a non-null value, is one of the elements in the bag *elems*, which directly corresponds to the above description, still being more succinct and concise.

The description of method *add* states that it ensures that the current *Collection* instance contains the specified element, returning true if *Collection* changed as a result of the call and false if it does not permit duplicates and already contains the specified element. The nondeterministic choice operator choose used in our specification allows us to express these variations in the behavior succinctly and precisely: if the element to be added is already present in the current instance of *Collection*, this element can either be added to *Collection* or the addition of the element can be skipped, with the choice between the options made nondeterministically. When the element is not present, it is necessarily added to *Collection*. The declaration and initialization of a local variable $r$ is equivalent to the declaration followed by assigning $r$ the boolean value *false*. Further on in specifications we will use this kind of initialization along with nondeterministic initialization where a local variable is initialized according to some predicate.

The method *remove* is described as an operation removing an element $e$ such that $(o == null\ ?\ e == null\ :\ o.equals\,(e))$, if *Collection* contains one or more such elements. Further it is stated in [7] that this method "returns true if the Collection contained the specified element (or equivalently, if the Collection changed as a result of the call)". In our specification we stipulate that if $o$ is present in *Collection* at least once, its number of occurrences is decreased by one and the method returns true.

The iterator returned in the identically named method of the interface *Collection* is constructed by calling the constructor *Iterator* and passing it the reference to the current instance of *Collection*. Although *Iterator* is just an interface which cannot be used to produce instances, we provide its formal specification in the same way as for *Collection*, and by giving the specification of *Iterator*'s constructor, we define the precise meaning of its invocation in the method *Iterator* of *Collection*. An implementation of *Iterator* will have to define its own constructor, and an implementation of *Collection* will then return an instance created by this constructor in the method *Iterator*.


Figure 3: Simultaneous modification of a collection by different iterators

Before presenting a formal specification of the interface *Iterator*, let us consider a collaboration between a collection and iterators attached to it. As described in [7], "*Iterator.remove* is the only safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress." Obviously, this description is rather ambiguous, because it is unclear the behavior of which methods is unspecified, and how modifications are being monitored, and what it means for an iteration to be in progress. To get an intuitive understanding of object interaction in this case, let us consider Fig. 3. Suppose that two iterators $i_1$ and $i_2$ are used to iterate over a collection implemented as a list, as shown in Fig. 3(a). Now, if we execute $i_2.next(); i_1.next(); i_1.remove()$, the iterator $i_2$ will be indexing a non-existing list element, as shown in Fig. 3(b). Further invocations of methods on $i_2$ will produce erroneous results or simply abort. However, the iterator $i_1$, which has carried out the structural modification of the underlying collection, will continue to work correctly. Accordingly, we have to specify the conditions under which iterators can be sure that the underlying data structure hasn't been structurally modified. The data attribute *modified* of *Collection* can be used for this purpose. Maintaining in *Iterator* an invariant that its own *modified* data attribute is equal to the one of the underlying *Collection*, helps solve the problem. Furthermore, the description of method *remove* states that this method can be called only once per call to *next*. To reflect this requirement in the specification, we maintain a data attribute *canRemove* and set it to true after resetting the next element and to false after removing the current element. The interface *Iterator* can, therefore, be specified as follows:

**public interface Iterator {**

*Collection col*;
*bag of Object current*;
*boolean canRemove*;
*int modified*;
*Object next*;

| | | |
|---|---|---|
| invariant $I$ | $==$ | $col \mathrel{!}= null \wedge$ |
| | | $(canRemove \Rightarrow next \in current)$ |
| interclass invariant $intI$ | $==$ | $current \subseteq col.elems \wedge$ |
| | | $modified == col.modified$ |

```
  Iterator(Collection c) {
    assert c != null;
    col, current, canRemove, modified, next = c, ⟦⟧, false, c.modified, null;
  }
  boolean hasNext() {
    return current ⊂ col.elems;
  }
  Object next() {
    assert current ⊂ col.elems;
    [next = e | e ∈ (col.elems \ current)];
    current, canRemove = current + ⟦next⟧, true;
    return next;
  }
  void remove() {
    assert canRemove;
    col.elems, col.modified = col.elems \ next, col.modified + 1;
    current, canRemove, modified = current \ next, false, modified + 1;
    next = null;
  }
}
```

As elements in a bag cannot be indexed, we use the data attribute *current* to store the elements of the underlying collection that have been returned by the method *next* in the current iteration. The attribute *next* stores the element returned by the last call to the method *next*. The class invariant $I$ states that the iterator is always attached to an existing collection ($col \mathrel{!}= null$) and that the next element to be removed is one of the elements currently "indexed" ($canRemove \Rightarrow next \in current$). This class invariant holds of all *Iterator* instances during their whole life cycle, being established by the constructor and preserved by all the methods. Apart from the class invariant, *Iterator* maintains another invariant *intI* which captures the invariance in the relation between the attributes of *Iterator* and the attributes of *Collection* that it aggregates, stating that the elements returned by the method *next* are always in the underlying collection ($current \subseteq col.elems$) and that structural modifications made so far have been made by the current instance of *Iterator* ($modified == col.modified$). This invariant is different from the class invariant proper in that it is maintained mutually by *Iterator* and *Collection*. We choose to call this invariant "interclass invariant" to reflect that, on the one hand, it is an invariant established by the constructor and preserved by all the methods of *Iterator*, and, on the other hand, it is the predicate which cannot be assumed to hold of all *Iterator* instances at all times because *Iterator* alone cannot guarantee its preservation between method calls to its methods. In other words, creating an instance of *Iterator* through calling the constructor establishes *intI*, and, although there are no

guarantees that *intI* holds at all moments in a life cycle of this instance, if it does then a call to any method of *Iterator* will preserve it. Note that the methods *add* and *remove* of *Collection* break *intI* which suggests potential behavioral problems with structural modification of the underlying collection by different iterators. The interclass invariant of a particular *Iterator* instance will be preserved only if this instance is used by the underlying collection to structurally modify itself through calls to *Iterator* methods. In this respect, the fact that *Collection* has the method *add*, while *Iterator* does not, might indicate the possibility of inadequate framework design.

Bertrand Meyer in [17] discusses the problem of interclass invariants, although in a slightly different setting with two classes maintaining mutual references to each other, and proposes to do run-time monitoring of these invariants, effectively adding them to pre- and postconditions of methods in the classes whose attributes are related through such invariants. We define the semantics of the interclass invariant construct similarly, by adding it as the implicit assert condition in the end of the class constructor and the implicit assume\assert conditions in, respectively, the beginning and the end of every class method. Proving consistency of a class with respect to its class invariant and interclass invariant amounts to verifying that both kinds of invariants are established by the class constructor and preserved by all its methods.

The additional operations on bags used in the specification of *Iterator* are defined as follows:

$$
\begin{array}{rcl}
b_1 \subseteq b_2 & \widehat{=} & (\forall e \bullet b_1\ e \leq b_2\ e) \\
b_1 \subset b_2 & \widehat{=} & b_1 \subseteq b_2 \wedge \#b_1 < \#b_2 \\
b_1 \setminus\ b_2 & \widehat{=} & (\lambda e \bullet \mathsf{max}\,((b_1\ e - b_2\ e),\, 0))
\end{array}
$$

Apart from standard Java language constructs we use the multiple assignment statement $x_1, \ldots, x_n = e_1, \ldots, e_n$ which stands for a simultaneous assignment of expressions $e_1, \ldots, e_n$ to variables $x_1, \ldots, x_n$ respectively. Assuming that $x_1, \ldots, x_n$ do not occur free in $e_1, \ldots, e_n$, multiple assignment can always be rewritten as a sequential composition of the corresponding individual assignments in arbitrary order. Moreover, we use two specification statements, assertion and nondeterministic update. The assertion statement assert $p$, where $p$ is a boolean-valued expression, skips if $p$ holds in a current state and aborts otherwise.[3] The nondeterministic update $[x = x' \mid b]$ assigns $x$ a value $x'$ satisfying a boolean condition $b$; if such a value cannot be found, the execution stops.

The constructor creates a new *Iterator* instance only under the condition that the collection referred by $c$ is some existing object, as expressed in the

[3]The syntax of the assertion statement is different in [6] where the semantics of this statement is defined; it is written as $\{p\}$ instead of assert $p$ that we use here. Using the syntax $\{p\}$ to denote assertions in Java specifications would be confusing, as the curly brackets are used to delineate blocks.

assertion assert $c$ != *null*; otherwise, the constructor aborts. The method *next* returns a next object in the underlying data structure only under the condition that the end of the structure hasn't been reached, as expressed in the assertion assert $current \subset col.elems$. Note that the element to be returned by this method is chosen nondeterministically from the elements in the underlying collection that haven't been returned by *next* in the current iteration run. This element is added to the bag of currently iterated elements *current* and the boolean flag *canRemove* is set to true, permitting removal of the next element. In turn, the method *remove* agrees to remove the next element only if *canRemove* holds in a state, encoding the requirement that *remove* can be called only once per call to *next*.

Note that in the specification of *Iterator* we directly modify data attributes of the aggregated collection *col*. Normally, in object-oriented programming such practice is rightfully criticized for breaking encapsulation. In specifications, however, we will permit such direct access and modification because this significantly simplifies specifications, as there is no need to specify the behavior solely in terms of method calls on the aggregated objects. There is no danger of breaking encapsulation because implementations can (and usually will) use completely different attributes for achieving what is required in the specification, and in the implementations direct access to data attributes of another class will be completely eliminated and substituted with method calls preserving encapsulation.

Now we can continue with specifying the bulk operations of *Collection* as follows:

```
public interface Collection {
  ...
  boolean containsAll(Collection c) {
    assert c != null;
    return c.elems ⊆ elems;
  }
  boolean addAll(Collection c) {
    assert c != null ∧ (c == this ⇒ c.elems == ⦇⦈);
    bag of Object old, int cm | old == elems ∧ cm == c.modified;
    [elems, modified = e, m | e == elems + c.elems ∧
      m ≥ modified ∧ c.modified == cm];
    return old != elems;
  }
  boolean removeAll(Collection c) {
    assert c != null ∧ (c == this ⇒ c.elems == ⦇⦈);
    boolean r, int cm | r == false ∧ cm == c.modified;
    if (∃e • e ∈ c.elems ∧ e ∈ elems) {
      [elems, modified = e, m | e == elems \ toSet(c.elems) ∧
```

```
        m ≥ modified ∧ c.modified == cm];
      r = true;
    };
    return r;
  }
  boolean retainAll(Collection c) {
    assert c != null ∧ (c == this ⇒ c.elems == ⟦⟧);
    boolean r, int cm | r == false ∧ cm == c.modified;
    if (∃e • e ∈ c.elems ∧ e ∈ elems) {
      [elems, modified = e, m | e == elems \ toSet(elems \ toSet(c.elems)) ∧
        m ≥ modified ∧ c.modified == cm];
      r = true;
    };
    return r;
  }
  void clear() {
    elems = ⟦⟧;
    [modified = m | m ≥ modified];
  }
}
```

The specification of method *containsAll* is quite straightforward: under the condition that the reference to the incoming *Collection* is non-null this method returns true if all elements in the incoming *Collection* are present in the current instance of *Collection*. Note that in the specification it is assumed that the incoming *Collection* is an instance of the specification class *Collection* whose attribute *elems* is a bag of *Object* elements. The behavior of this method in the case when an instance of some other class implementing the interface *Collection* is passed as input is underspecified. The implementation of *containsAll* will have to be polymorphic and deliver the behavior as specified in *Collection.containsAll* regardless of the dynamic type of the input argument.

The informal description of method *addAll* states that the behavior of this operation is undefined if the incoming *Collection* is modified while the operation is in progress. To express this restriction in the specification, we use the local variable *cm* to keep the number of modifications made to *c* up to the moment it was passed as input to *addAll*. Elements of *c* are guaranteed to be added to the current *Collection* only if *cm* remains equal to *c.modified* during the whole operation. Also it is mentioned in the documentation [7] that the behavior of *addAll* is undefined if the incoming *Collection* is the current instance of *Collection* and is nonempty. We address this restriction by stating the corresponding assertion in the beginning of the method specification.

Note how this specification of *addAll* uses specification constructs to express the required complex functionality. On the one hand, we avoid unnecessary details, such as checking whether the current *Collection* gets modified as a result of each call to *add* and simply return the result of comparing the original bag with the resulting one. This specification is inefficient but it succinctly and clearly captures the intended behavior. On the other hand, we do not oversimplify the specification sacrificing preciseness: writing just $elems = elems + c.elems$ would certainly make the specification of this method shorter, but wouldn't express the necessary requirement that the collection $c$ cannot be modified during the addition. An implementation of *addAll* will add elements iteratively, and in order to meet the requirement about non-modification of $c$ it will have to check that this requirement is satisfied before adding each element of c.

Surprisingly, the description of method *removeAll* in the original documentation does not stipulate the requirement that the incoming collection $c$ cannot be modified during its execution. However, based on the statement that "After this call returns, this *Collection* will contain no elements in common with the specified *Collection*" we can justify the need for such a requirement. Suppose that, while iterating over $c$, an element which already has been removed from the current *Collection* is added again to $c$ before the iterator used in *removeAll*. When the execution of *removeAll* completes, the current *Collection* will still have some elements in common with $c$. Similarly the result of *removeAll* can be undefined if some elements of $c$ are externally removed during the iteration. It is also easy to see that the behavior of this method becomes undefined if the current *Collection* is passed to it as an input argument. In the specification we address all these requirements using the corresponding assertions and the nondeterministic assignment statement. The latter states that the new value assigned to *elems* is equal to the difference between the current *Collection* and the set obtained from converting the bag of elements in $c$; in addition, it is stipulated that $c$ does not undergo any structural modifications: its *modified* attribute remains unchanged. The function *toSet* used in this specification is defined for a bag $b$ as follows:

$$toSet(b) \quad \widehat{=} \quad \{e \mid b\ e > 0\}$$

The function returning the difference between a bag $b$ and a set $s$ is given as follows:

$$b \setminus s \quad \widehat{=} \quad (\lambda e \bullet (e \in s)\,?\,0 : b\ e)$$

In the specification of method *retainAll* we state that the new value assigned to *elems* contains only the elements that are common to the original bag *elems* and the incoming $c.elems$. The same non-modification requirements as in *removeAll* are imposed on $c$ for similar reasons. Finally, the method *clear* results in assigning to *elems* the empty bag $[\![]\!]$.

The next two methods to be specified deal with converting *Collection* to an array. The first method *toArray* is described in [7] as one returning an array containing all of the elements in the current *Collection*. It is stated that "the returned array will be 'safe' in that no references to it are maintained by *Collection*". As such this is a rather vague description of the behavior, because it is unclear whether elements of the original collection are copied to the returned array by reference or by value. When writing a formal specification we must address this issue, and we choose to copy the collection elements by value rather than by reference, which appears to be safer than copying by reference.

The behavior of the second *toArray* method is described as follows: "Returns an array containing all of the elements in this *Collection*, whose runtime type is that of the specified array. If *Collection* fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this *Collection*. If *Collection* fits in the specified array with room to spare (i.e., the array has more elements than *Collection*), the element in the array immediately following the end of the collection is set to null." We specify the two array conversion methods as follows:

**public interface Collection {**

...

**Object**[] **toArray**() {

$Object[]a = \mathsf{new}\ Object[\#elems];$

$bag\ of\ Object\ be \mid be == elems;$

$\mathsf{for}\ (i = 0; i < \#elems; i = i + 1)\ \{$

$[a[i], be = a, b \mid a \notin be \wedge (\exists a' \bullet a' \in be \wedge a\uparrow == a'\uparrow \wedge b == be \setminus a']$

};

$\mathsf{return}\ a;$

}

**Object**[] **toArray**(**Object a**[]) {

$Class\ typeOfArray = a.getClass().getComponentType();$

$bag\ of\ Object\ be \mid be == elems;$

$\mathsf{if}\ (a.length() < \#elems)\ \{$

$Object[]c = \mathsf{new}\ typeOfArray[\#elems];$

$\mathsf{for}\ (i = 0; i < \#elems; i = i + 1)\ \{$

$[c[i], be = c, b \mid c \notin be \wedge (\exists c' \bullet c' \in be \wedge c\uparrow == c'\uparrow \wedge$

$b == be \setminus c' \wedge c.getClass() == typeOfArray)]$

};

$\mathsf{return}\ c;$

}

$\mathsf{else}\ \{$

$\mathsf{for}\ (i = 0; i < \#elems; i = i + 1)\ \{$

$$[a[i], be = a, b \mid a \notin be \wedge (\exists a' \bullet a' \in be \wedge a\uparrow == a'\uparrow \wedge$$
$$b == be \setminus a' \wedge a.getClass() == typeOfArray)]$$
};
$a[\#elems] = null$;
return $a$;
};
}
}

One interesting point to note here is the use of method invocations *getClass* and *getComponentType*. Although the precise definitions of these methods, supported by the array interface, are not available, we include these method invocations in the specification of method *toArray* to indicate that these methods should be called in implementations of *Collection*. Being partial, such a specification of the behavior of *toArray* is nevertheless very useful, as it succinctly describes the intended actions and guides implementation development.

This concludes our specification of *Collection* and now we can specify the interface *List* which extends *Collection*.

## 2.2 Specifying List and ListIterator

A *List* is an ordered *Collection* sometimes called a sequence. In addition to the operations inherited from *Collection*, the interface *List* includes operations for positional access, search for a specified object in the list, list iteration, and range operations on the list. In addition to the ordinary *Iterator*, *List* provides a richer *ListIterator* that allows one to traverse the list in either direction, modify the list during iteration, and obtain the current position of the iterator. The interfaces of *List* and *ListIterator* are shown in Fig. 4.

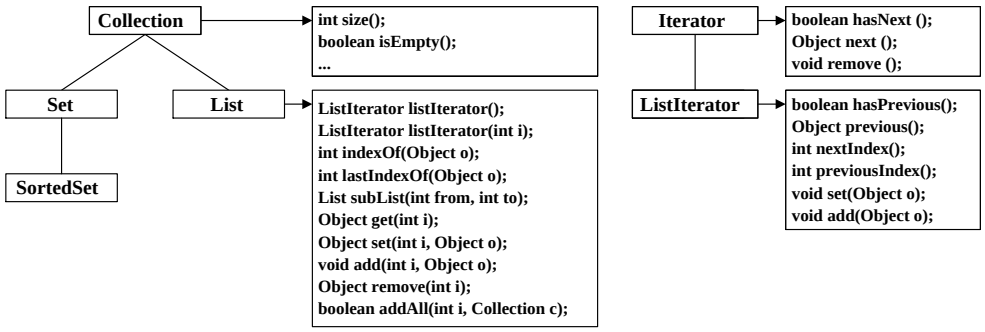



Figure 4: List and ListIterator interfaces

It appears to be natural to model the underlying data structure of *List* by a sequence of *Object* elements. As before, to specify the history of structural modifications, we will use an integer attribute *modified* which will be increased whenever elements are added to the original list or removed from it. We begin with specifying the data attributes, the constructor, and the operations inherited from *Collection*:

```
public interface List extends Collection {
  seq of Object elems;
  int modified;
  List() {
    elems, modified = ⟨⟩, 0;
  }
  int size() {
    return min(#elems, Integer.MAX_VALUE);
  }
  boolean isEmpty() {
    return (#elems == 0);
  }
  boolean contains(Object o) {
    return (o in elems);
  }
  boolean add(Object o) {
    elems, modified = elems ⌢ ⟨o⟩, modified + 1;
    return true;
  }
  boolean remove(Object o) {
    boolean r | r == false;
    if (o in elems){
      [elems = e | (∃l1, l2 •
        l1 ⌢ ⟨o⟩ ⌢ l2 = elems ∧ ¬(o in l1) ∧ l1 ⌢ l2 = e)];
      modified = modified + 1;
      r = true;
    }
    return r;
  }
  Iterator iterator() {
    Iterator i = new ListIterator(this);
    return i;
  }
```

Specifications of the constructor and the methods *size*, *isEmpty*, and *contains* are quite straightforward with the membership operation in on sequences defined as follows:

$$e \text{ in } l \quad \widehat{=} \quad (\exists i \,|\, 0 \leq i < \#l \bullet l[i] = e)$$

To improve readability we use the notation $l[i]$ rather than the function application $l\ i$ to represent the $i$'th element of the sequence $l$.

The method *add* appends the specified element to the end of *List*, whereas the method *remove* removes the first occurrence of the specified element from *List*. The iterator returned by the identically named method is an instance of *ListIterator* which is an extension of *Iterator*. In addition to the methods of *Iterator*, *ListIterator* provides methods allowing positional access through an index.

The bulk operations and the array conversion operations of *List* are specified similarly to those of *Collection*:

```
public interface List extends Collection {
  ...
  boolean containsAll(Collection c) {
    assert c != null;
    return c.elems ⊆ toBag(elems);
  }
  boolean addAll(Collection c) {
    assert c != null ∧ (c == this ⇒ c.elems == ⟨⟩);
    seq of Object old, int cm | old == elems ∧ cm == c.modified;
    [elems, modified = e, m | ∃e' •
      toBag(e') == c.elems ∧ e == elems⌢e' ∧
      m ≥ modified ∧ c.modified == cm];
    return old != elems;
  }
  boolean removeAll(Collection c) {
    assert c != null ∧ (c == this ⇒ c.elems == ⟨⟩);
    boolean r, int cm | r == false ∧ cm == c.modified;
    if (∃e • e ∈ c.elems ∧ e in elems) {
      [elems, modified = e, m | e == remAll(elems, toSet(c.elems)) ∧
        m ≥ modified ∧ c.modified == cm];
      r = true;
    };
    return r;
  }
  boolean retainAll(Collection c) {
    assert c != null ∧ (c == this ⇒ c.elems == ⟨⟩);
```

```
    boolean r, int cm | r == false ∧ cm == c.modified;
    if (∃e • e ∈ c.elems ∧ e ∈ elems) {
      [elems, modified = e, m |
        e == remAll(elems, toSet(elems) \ toSet(c.elems)) ∧
        m ≥ modified ∧ c.modified == cm];
      r = true;
    };
    return r;
  }
  void clear() {
    elems = ⟨⟩;
    [modified = m | m ≥ modified];
  }
}
```

The specifications of array conversion methods are similar to those of *Collection* and we omit them for the lack of space. The function *toBag* used in the specifications of methods *containsAll* and *addAll* is given as follows:

$$toBag(l) \;\widehat{=}\; \begin{cases} [\![]\!] & \text{if } l = \langle\rangle \\ toBag(front) + [\![e]\!] & \text{if } l = front \frown \langle e \rangle \end{cases}$$

The function *remAll* used in the specifications of methods *removeAll* and *retainAll* is given as follows:

$$remAll(l, s) \;\widehat{=}\; \begin{cases} \langle\rangle & \text{if } l = \langle\rangle \\ remAll(front, s) & \text{if } l = front \frown \langle e \rangle \;\wedge\; e \in s \\ remAll(front, s) \frown \langle e \rangle & \text{if } l = front \frown \langle e \rangle \;\wedge\; e \notin s \end{cases}$$

Before proceeding with the specification of the new methods of *List*, let us present the specification of *ListIterator*. In this specification we use $l[f..t]$ to denote a subsequence of the given sequence $l$ between indices $f$ and $t$ inclusive. We define this function to be total, returning a subsequence starting at index $f$ and ending at index $t$, if these indices are such that $0 \le f < t < \#l$, returning part of the sequence within range $f..\#l-1$ if the lower index $f$ satisfies $0 \le f < t$ but the upper index $t$ is greater than $\#l-1$, and returning an empty sequence if the indices $f$ and $t$ are misplaced in some way, being in the wrong order or reaching outside the bounds $0..\#l-1$.

$$l[f..t] \;\widehat{=}\; \begin{cases} \langle l[t] \rangle & \text{if } 0 \le f = t < \#l \\ l[f..t-1] \frown \langle l[t] \rangle & \text{if } 0 \le f < t < \#l \\ l[f..\#l-1] & \text{if } t \ge \#l - 1 \\ \langle\rangle & \text{otherwise} \end{cases}$$

The specification of *ListIterator* can now be given as follows:

```
public interface ListIterator extends Iterator {
  List lst;
  int ind;
  boolean canModify;
  int modified;
  invariant J              ==  lst != null
  interclass invariant intJ ==  −1 ≤ ind ≤ #lst.elems ∧
                                (canModify ⇒ 0 ≤ ind < #lst.elems) ∧
                                modified = lst.modified
  ListIterator(List l) {
    assert l != null;
    lst, ind, modified, canModify = l, −1, l.modified, false;
  }
  boolean hasNext() {
    return ind < #lst.elems − 1;
  }
  boolean hasPrevious() {
    return ind > 0;
  }
  Object next() {
    assert ind < #lst.elems − 1;
    ind, canModify = ind + 1, true;
    return lst.elems[ind];
  }
  Object previous() {
    assert ind > 0;
    ind, canModify = ind − 1, true;
    return lst.elems[ind];
  }
  int nextIndex() {
    return min (ind + 1, #lst.elems);
  }
  int previousIndex() {
    return max (ind − 1, −1);
  }
  void remove() {
    assert canModify;
    lst.elems = lst.elems[0..ind − 1]⌢
      lst.elems[ind + 1..#lst.elems − 1];
    ind, modified, canModify = ind − 1, modified + 1, false;
    lst.modified = modified;
  }
  void set(Object o) {
    assert canModify;
    lst.elems[ind] = o;
  }
  void add(Object o) {
    ind = ind + 1;
    [lst.elems = s | ∃s1, s2 • s == s1⌢⟨o⟩⌢s2 ∧
      lst.elems == s1⌢s2 ∧ s[ind] == o];
    canModify, modified = false, modified + 1;
    lst.modified = modified;
  }
}
```

Just as *Iterator* is used to iterate over its aggregated *Collection*, *ListIterator* is used to iterate over *List*. The class invariant $J$ of *ListIterator* states that at all times it aggregates a non-null reference to a *List* instance. In addition, the interclass invariant *intJ* states that the integer-valued index *ind* is used to iterate over $lst.elems$ and can range in the interval $[-1..\#lst.elems]$, with valid index values being in the interval $[0..\#lst.elems - 1]$. The data field *canModify* is similar to *canRemove* of *Iterator* and is used to regulate the order of calls to *next* and *previous* before calls to *remove*, *add*, and *set*. Finally, the data field *modified* is used to regulate structural modifications made to the underlying list.

In the description of *List* interface in [7] the index is said to always be between two elements, the one that would be returned by a call to *previous* and the one that would be returned by a call to *next*. With this layout the index has $n + 1$ valid positions for the list of size $n$, starting with 0 and ending with $n$. In our opinion this intuitive picture is somewhat confusing, especially in the two boundary cases when the index is before the first element or past the last one. In fact, this layout is so confusing that we have found contradicting descriptions of method behavior. For example, in the section describing the interface *List* the method *nextIndex* is said to return $list.size() + 1$ when the cursor is after the final element, whereas in the documentation describing *ListIterator* proper it is stated that this method "returns list size if the list iterator is at the end of the list". Apparently, the confusion arises because of the ambiguity of the valid values of the index pointing between elements rather than at elements. With our specification the index positions $-1$ and $\#lst.elems$ are boundary, whereas if the index *ind* is in the interval $[0..\#lst.elems - 1]$ inclusive, it points to the elements $lst.elems[0]$ through $lst.elems[\#lst.elems - 1]$. Having decided on the relationship between the index and the list elements, we can specify the behavior of *ListIterator* constructor and methods unambiguously. Namely, in the constructor the index is set to the boundary position $-1$. The methods *next* and *previous* first check that moving the index to the next (previous) position would not take it outside the bounds, then increment (decrement) it and return the currently indexed list element. The method *nextIndex* returns the minimum between $ind + 1$ and the size of the list, whereas the method *previousIndex* returns the maximum between $ind - 1$ and the boundary value $-1$. Obviously, the specifications of these methods are not only unambiguous but also very concise.

The specifications of methods *remove* and *set* are quite straightforward but the specification of *add* is worthy of a few comments. Let us first consider it description in [7]: "The element is inserted immediately before the next element that would be returned by next, if any, and after the next element that would be returned by previous, if any. (If the list contains no elements, the new element becomes the sole element on the list.) The new element is inserted before the implicit index: a subsequent call to next

would be unaffected, and a subsequent call to previous would return the new element. (This call increases by one the value that would be returned by a call to nextIndex or previousIndex.)" The first sentence in this description is somewhat equivocal because it is unclear whether the element is inserted before the next element that would be returned by *next* if the inserted element wouldn't have been inserted, or it is inserted before the next element that would be returned by *next* if we call *next* after a call to *add*. In the first case, the result of *add* should be insertion of the new element into the position after the implicit index, whereas in the second case, the element returned by the method *next* depends not only on the position where the new element is inserted but also on the position where the implicit index is placed as the effect of *add*. Only the following sentences clarify that the intention is to place the new element into the position "before the implicit index".

Now that we know the exact behavior of *ListIterator*, we can proceed with specification of *List* operations for positional access, search, and range extraction.

```
public interface List extends Collection {
  ...
  ListIterator listIterator() {
    ListIterator itr = new ListIterator(this);
    return itr;
  }
  ListIterator listIterator(int i) {
    assert 0 ≤ i ≤ #elems;
    ListIterator itr = new ListIterator(this);
    itr.ind = i;
    return itr;
  }
  int indexOf(Object o) {
    return min ({i | elems[i] == o} ∪ {−1});
  }
  int lastIndexOf(Object o) {
    return max ({i | elems[i] == o} ∪ {−1});
  }
  List subList(int from, int to) {
    assert 0 ≤ from ≤ to ≤ #elems;
    seq of Object s | (∀i | from ≤ i < to • elems[i] == s[i − from]);
    List sub = new List();
    sub.elems, sub.modified = s, 0;
    return sub;
  }
```

```
  Object get(int i) {
    assert 0 ≤ i < #elems;
    return elems[i];
  }
  Object set(int i, Object o) {
    assert 0 ≤ i < #elems;
    Object s | s == elems[i];
    [elems = e | ∃s1, s2 • elems == s1⌢⟨s⟩⌢s2 ∧ #s1 == i ∧ e == s1⌢⟨o⟩⌢s2];
    return s;
  }
  void add(int i, Object o) {
    assert 0 ≤ i ≤ #elems;
    [elems = e | ∃s1, s2 • elems == s1⌢s2 ∧ #s1 == i ∧ e == s1⌢⟨o⟩⌢s2];
    modified = modified + 1;
  }
  Object remove(int i) {
    assert 0 ≤ i < #elems;
    Object o | o == elems[i];
    [elems = e | ∃s1, s2 • elems == s1⌢⟨o⟩⌢s2 ∧ #s1 == i ∧ e == s1⌢s2];
    modified = modified + 1;
    return o;
  }
  boolean addAll(int i, Collection c) {
    assert 0 ≤ i ≤ #elems ∧ (c == this ⇒ c.elems == ⦇⦈);
    Iterator itr = c.iterator();
    int cm, seq of Object s, old | s == ⟨⟩ ∧ old == elems ∧ cm == c.modified;
    while (itr.hasNext()) {s = s⌢itr.next();};
    [elems, modified = e, m | (∃s1, s2 • elems == s1⌢s2 ∧ #s1 == i ∧
      e == s1⌢s⌢s2) ∧ m ≥ modified ∧ c.modified == cm];
    return old != elems;
  }
}
```

The first two methods construct new *ListIterator* instances, setting their indices to $-1$ and the specified index $i$ respectively. The next two methods *indexOf* and *lastIndexOf* return the indices of the first and the last occurrence of the specified element in the current *List*, or $-1$ if it does not contain this element. We specify these methods by saying that the returned index is, respectively, the minimal and the maximal element of the set containing all indices at which the list element is equal to the specified element or $-1$, if this set is empty. The description of method *subList*, whose specification is given next, states that the returned list is a portion of the current *List*

between the specified *from* index, inclusive, and *to* index, exclusive. We specify this behavior by constructing a subsequence *s* of *elems* such that the elements of *s* are equal to the elements of *elems* starting at index *from* and finishing at index $to - 1$. A new *List* instance initialized with this subsequence is then returned as the result of method *subList*. The specifications of methods *set*, *add*, and *remove* are rather straightforward and hardly require further explanation. The behavior of method *addAll* adding the specified *Collection* at a specified position is very similar to the ordinary *addAll*. The only interesting point here is that the informal description of this method stipulates that the new elements will appear in the current *List* in the order that they are returned by the specified *Collection*'s iterator. We address this requirement by iteratively constructing from *Collection* elements a sequence and adding this sequence at the specified position in the current list.

# 3 Ensuring Correctness of JCF

As was already mentioned in the introduction, correctness of a framework can be ensured by verifying behavioral conformance between classes whose instances are intended for polymorphic substitution in clients. In systems with separate interface inheritance and implementation inheritance hierarchies, such as JCF, subtype polymorphism is based on interface inheritance. Therefore, there are two ways of achieving polymorphic reuse, through passing instances of classes implementing an interface where objects with this interface are expected and through substituting objects of subinterface type for objects of superinterface type. In the first case, the concrete class must be shown to refine the specification of the interface it implements. In the second case, verifying behavioral conformance between the superinterface objects and the subinterface objects amounts to proving class refinement between the specification of the original interface and the specification of its subinterface. These two cases are illustrated in Fig. 5. The classes *AbstractCollection*, *ConcreteCollection* and *SpecialCollection* are different implementations of *Collection* interface and the classes *AbstractList* and *LinkedList* are different implementations of *List* interface. Both *Collection*

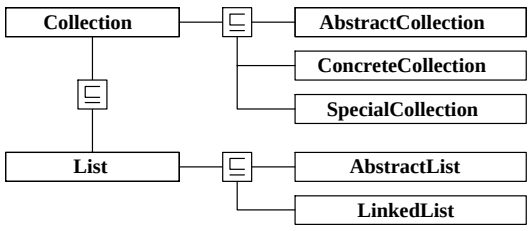



Figure 5: Behavioral conformance in systems with separate interface inheritance and implementation inheritance hierarchies

and *List* interfaces are augmented with formal specifications of the intended behavior. If we verify that the specification of *Collection* is refined by its implementations, i.e. if we prove class refinements *Collection* $\sqsubseteq$ *AbstractCollection*, *Collection* $\sqsubseteq$ *ConcreteCollection* and *Collection* $\sqsubseteq$ *SpecialCollection*, then clients specified to work with a variable $c$ of type *Collection* will continue to work correctly if $c$ is assigned an instance of any of the classes *AbstractCollection*, *ConcreteCollection* and *SpecialCollection*. Similarly, using an instance of *AbstractList* or *LinkedList* in the context where it is viewed as an object of type *List* will be correct if *List* $\sqsubseteq$ *AbstractList* and *List* $\sqsubseteq$ *LinkedList*.

Moreover, since *List* is a subinterface of *Collection*, instances of *AbstractList* and *LinkedList* can be used in the context where an object of type *Collection* is expected. If we verify that *Collection* $\sqsubseteq$ *List* then by transitivity *AbstractList* and *LinkedList*, as well as all other correct implementations of *List*, will be refinements of *Collection*.

We will illustrate the verification of class refinement by proving that *Iterator* is refined by *ListIterator*. But first we would like to explain the notion of refinement is more detail. For a formal treatment of class refinement we refer to [19, 3, 2].

## 3.1 Formal Background: Class Refinement

### 3.1.1 Semantics, Correctness and Refinement of Program Statements

Every program statement has a weakest precondition predicate transformer semantics. A *predicate transformer* $S : (\Gamma \to \mathit{Bool}) \to (\Sigma \to \mathit{Bool})$ is a function from predicates on $\Gamma$ to predicates on $\Sigma$. We write

$$\Sigma \mapsto \Gamma \;\widehat{=}\; (\Gamma \to \mathit{Bool}) \to (\Sigma \to \mathit{Bool})$$

to denote the type of all predicate transformers from $\Sigma$ to $\Gamma$. A statement with initial state in $\Sigma$ and final state in $\Gamma$ determines a monotonic predicate transformer $S : \Sigma \mapsto \Gamma$ that maps any postcondition state predicate $q : \Gamma \to \mathit{Bool}$ to the weakest precondition state predicate $p : \Sigma \to \mathit{Bool}$ such that the statement is guaranteed to terminate in a final state satisfying $q$ whenever the initial state satisfies $p$. In the refinement calculus program statements are identified with the monotonic predicate transformers that they determine. For details of the predicate transformer semantics, we refer to [6].

The *total correctness assertion* $p \,\{\!|\, S \,|\!\}\, q$ is said to hold if the statement $S$ can be used to establish the postcondition $q$ when starting in the set of states $p$. Formally, the total correctness assertion $p \,\{\!|\, S \,|\!\}\, q$ is defined to be equal to $p \subseteq S\ q$, which means that $p$ is stronger than the weakest precondition of $S$ with respect to $q$.

A statement $S$ is *refined by* a statement $S'$, written $S \sqsubseteq S'$, if any condition that we can establish with the first statement can also be established with the second statement. Formally, $S \sqsubseteq S'$ is defined to hold if $p \; \{\!|\, S \,|\!\} \; q \Rightarrow p \; \{\!|\, S' \,|\!\} \; q$, for any $p$ and $q$. Refinement is reflexive and transitive.

The refinement calculus provides rules for transforming more abstract program structures into more concrete ones based on the notion of refinement of statements presented above. For example, we have the following law for assignment introduction:

$$\mathsf{assert}\ b_1; [x = x' \,|\, b_2] \quad \sqsubseteq \quad x = e, \ \text{if}\ b_1 \ \Rightarrow \ b_2[x' \leftarrow e] \tag{1}$$

This law states that a nondeterministic assignment to $x$ of a new value satisfying the boolean expression $b_2$ under the condition that $b_1$ holds initially is refined by a deterministic assignment of an expression $e$ to $x$ if $b_1$ is stronger than $b_2$ with all variables $x'$ substituted with $e$. For example, $[n = n' \,|\, n'^2 == n]$ is refined by $n = -\sqrt{n}$ because assertion of a universally true predicate *true* always skips, so that $[n = n' \,|\, n'^2 == n]$ is the same as $\mathsf{assert}\ \mathit{true}; [n = n' \,|\, n'^2 == n]$, and also because $\mathit{true} \ \Rightarrow \ ((-\sqrt{n})^2 == n)$. Effectively, this law expresses the fact that decreasing nondeterminism is a refinement.

### 3.1.2 Data Refinement of Program Statements

Data refinement is a general technique by which one can change data representation in a refinement. Assume that statements $S$ and $S'$ operate on state spaces $\Sigma$ and $\Sigma'$ respectively, i.e. $S : \Sigma \mapsto \Sigma$ and $S' : \Sigma' \mapsto \Sigma'$. Let $R : \Sigma' \to \Sigma \to \mathit{Bool}$ be a relation between the state spaces $\Sigma'$ and $\Sigma$. Following [4], the statement $S$ is said to be *data refined* by the statement $S'$ via the relation $R$, denoted $S \sqsubseteq_R S'$, if coercing the concrete state $\Sigma'$ to the abstract state $\Sigma$ followed by executing $S$ is refined by executing $S'$ followed by coercing the concrete state to the abstract:

$$S \sqsubseteq_R S' \quad \widehat{=} \quad \{R\}; S \sqsubseteq S'; \{R\}$$

The angelic nondeterministic assignment $\{R\}$ used here coerces the concrete state to the abstract. Usually, if the concrete state is represented by the variable $c : \Sigma'$ and the abstract one by the variable $a : \Sigma$, the relation $R$ applied to $c$ and $a$ is equal to some boolean expression $t$ which may refer to $a, c$ and other program variables over the global state. The *abstraction statement* $\{R\}$ written in terms of program variables will then have the form $\{a = a' \,|\, t[a \leftarrow a']\}$, where $t[a \leftarrow a']$ is $t$ with all occurrences of $a$ substituted with $a'$.

To illustrate data refinement laws, let us present the rule for data refinement of demonic nondeterministic assignment:

$$\mathsf{assert}\ p; [a, u = a', u' \,|\, b_1] \ \sqsubseteq_R \ \mathsf{assert}\ p'; [c, u = c', u' \,|\, b_2], \tag{2}$$

$$\text{if } p \wedge t \Rightarrow p' \text{ and } p \wedge t \wedge b_2 \Rightarrow (\exists a' \bullet b_1 \wedge t')$$

Here $t$ stands for $R$ applied to $c$ and $a$, and $t'$ is equal to $t$ with all occurrences of $a, c$ and $u$ substituted with $a', c'$ and $u'$, i.e. $t' = t[a, c, u \leftarrow a', c', u']$. According to this rule, for example, the nondeterministic assignment to a variable $e$ of some element of a nonempty set $s$ is refined by the nondeterministic assignment to $e$ of some element of a nonempty sequence $l$:

$$\begin{array}{l} \mathsf{assert}\ s\ != \emptyset; [s, e = s', e' \mid e' \in s] \sqsubseteq_R \\ \mathsf{assert}\ l\ != \langle\rangle; [l, e = l', e' \mid \exists i \bullet 0 \leq i < \#l \wedge e' == l[i]] \end{array}$$

Here $R\ l\ s\ =\ (\forall e \bullet e \in s == e\ \mathsf{in}\ l)$ and verification of the necessary preconditions can be done using the basic properties of sets and sequences.

Presenting other rules of data refinement is outside the scope of this paper and we refer the interested reader to [6, 21] which contain large collections of refinement rules.

### 3.1.3 Class Refinement

Class refinement is defined to hold between classes $C$ and $D$ if there exists a relation $R$ such that the constructor of $C$ is data refined by the constructor of $D$ with respect to $R$ and every method of $C$ is data refined by the corresponding method of $D$ with respect to $R$. Suppose that the constructors of $C$ and $D$ with input parameters $g_0$ and $g_0'$ of types $\Gamma_0$ and $\Gamma_0'$ are specified by statements $K$ and $K'$ of types $\Gamma_0 \mapsto \Sigma \times \Gamma_0$ and $\Gamma_0' \mapsto \Sigma' \times \Gamma_0'$ respectively. Further suppose that a relation $R : \Sigma \leftrightarrow \Sigma'$ coerces the attributes $d$ of $D$ to the attributes $c$ of $C$; if $D$ has the same attributes as $C$, this relation is the identity relation $Id$, if $D$ inherits all the attributes of $C$ and adds some new, this relation is the projection. Similarly, a relation $Q : \Gamma_0' \leftrightarrow \Gamma_0$ coerces the input parameter $g_0'$ to the input parameter $g_0$. In case the input parameters are of the same type, $Q$ is equal to the identity relation. Constructor refinement with respect to the relations $Q$ and $R$ is defined as follows:

$$\{Q\}; K \sqsubseteq K'; \{R \times \mathit{True}\}$$

Here the relational product $R \times \mathit{True}$ relates pairs of states $(d, g_0')$ and $(c, g_0)$ so that $R$ holds of $d$ and $c$ and $\mathit{True}$ holds of $g_0'$ and $g_0$. As the values of the input parameters in the end of the constructors are irrelevant, coercing them using the relation $\mathit{True}$ will always succeed. In terms of program variables for the attributes $c$ of $C$ and $d$ of $D$ the rule for constructor refinement can be expressed as follows:

$$\{g_0 = g' \mid Q\ g_0'\ g'\}; K \sqsubseteq K'; \{c, g_0 = c', g' \mid R\ d\ c'\}$$

Consider now refinement of statements $M_i$ and $M_i'$ that specify the behavior of some method called $\mathit{Meth}_i$ in $C$ and $D$ respectively. Formally, a method with input parameters $g_i : \Gamma_i$ and an output parameter $d_i$ of

type $\Delta_i$, operating on the attributes of type $\Sigma$ is a statement of type $\Sigma \times \Gamma_i \times \Delta_i \mapsto \Sigma \times \Gamma_i \times \Delta_i$. As the types of input and return parameters of $Meth_i$ in $C$ and $D$ are necessarily identical in Java, we can coerce the parameters using the identity relation. Refinement of methods with the respect to the relation $R$ coercing the corresponding attributes is then given as follows:

$$\{R \times Id\}; M_i \sqsubseteq M_i'; \{R \times Id\}$$

In terms of program variables for the attributes $c$ of $C$ and $d$ of $D$ the rule for method refinement can be expressed as follows:

$$\{c = c' \mid R\ d\ c'\}; M_i \sqsubseteq M_i'; \{c = c' \mid R\ d\ c'\}$$

If $D$ has new methods there is an additional proof obligation that every new method of $D$ preserves the set of reachable states of $C$. This requirement is necessary because if new methods take an instance of $D$ into a state which is perceived as unreachable in the context of $C$, clients of $D$ may get invalidated. In practice, the set of reachable states is preserved by all non-modifying methods and by modifying methods that refine an arbitrary composition of the original methods. For a formal treatment of class refinement and consistency in the presence of new methods see [2].

When classes have explicit invariants, apart from proving class refinement it is necessary to verify that the classes are *consistent* with respect to the corresponding class invariants, and that these class invariants are related via an abstraction relation. Namely, if $I$ is the invariant of $C$ (in the case when the interclass invariant of $C$ is different from *true*, the invariant $I$ is the conjunction of the class invariant and the interclass invariant, otherwise it is just the class invariant), we have to prove that the constructor of $C$ establishes $I$ and all methods of $C$ preserve $I$. Let the constructor of $C$ be specified by a statement $K$, then verification of establishing $I$ by $K$ amounts to proving that the total correctness assertion $true\ \{\!|\, K \,|\!\}\ I$ holds. If $K$ has some precondition $p$, e.g., places some restrictions on input parameters, then we have to conjoin $p$ to the precondition of the correctness assertion, getting to prove $p\ \{\!|\, K \,|\!\}\ I$, which means that if $p$ holds in the beginning then $K$ guarantees to establish $I$ in the end. Similarly, verification of preserving the invariant $I$ by a method $M_i$ of $C$ requires verifying the correctness assertion $I\ \{\!|\, M_i \,|\!\}\ I$. If $M$ itself has a precondition $p_i$, this correctness assertion becomes $I \wedge p_i\ \{\!|\, M_i \,|\!\}\ I$.

Coercing an abstract invariant using an abstraction relation produces an invariant on a concrete state that restricts the possible values of the concrete state as the abstract invariant restricts the possible values of the abstract state. More formally, if $a$ and $c$ are the program variables representing an abstract and a concrete states respectively, $I$ and $J$ are boolean expressions on $a$ and $c$ representing the corresponding invariants, and $R$ is an abstraction

relation, then $I$ can be expressed on the concrete state $c$ as $(\exists a \bullet R\ c\ a \wedge I)$. In verifying class refinement between $C$ and $D$, we have to prove that the invariant $J$ of $D$ is stronger than or equal to the invariant $I$ of $C$ with respect to $R$, i.e. $J \Rightarrow (\exists a \bullet R\ c\ a \wedge I)$. Verifying that this relation between the invariants holds, allows us to make sure that instances of $D$ preserve the invariant of $C$ with respect to the abstraction relation, which is important if they are to be dynamically substituted for instances of $C$. Moreover, if $D$ is a subclass of $C$, self-referential method invocations in $C$ can get redirected to $D$. To prevent such a *down-call* of a subclass method from a superclass method from aborting, the subclass invariant must be equal to the superclass invariant with respect to the abstraction relation, i.e. $J = (\exists a \bullet R\ c\ a \wedge I)$, with $=$ standing for logical equality. This condition guarantees that both $C$ and $D$ preserve mutual invariants, which is a critical requirement in the presence of subtype polymorphism and possible self-referential calls between $C$ and $D$. For a detailed discussion of these issues we refer to [18]. In the next subsection we will illustrate all these concepts and requirements with an example of proving class refinement between *Iterator* and *ListIterator*, both having non-trivial class invariants.

### 3.2 Proving Class Refinement in Practice

In proving the class refinement $Iterator \sqsubseteq ListIterator$ we have to select an abstraction relation coercing the attributes $lst, ind, canModify, modified$ of *ListIterator* to the attributes $col, current, canRemove, modified, next$ of *Iterator*. To distinguish between the attributes *modified* in the two classes, we will call $cm$ the one in *Iterator* and $lm$ the one in *ListIterator*. Also, for convenience we will abbreviate $(col, current, canRemove, cm, next)$ by $attr$ and $(lst, ind, canModify, lm)$ by $attr'$.

The abstraction relation $R$ can now be given as follows:

$$\begin{aligned} R\ attr'\ attr \quad = \quad & I' \wedge J' \wedge Q\ lst\ col \wedge canModify == canRemove \wedge \\ & (canModify \Rightarrow next == lst.elems[ind]) \wedge \\ & toBag(lst.elems[0..ind]) == current \end{aligned}$$

Here $I'$ and $J'$ are the combined class and interclass invariants of *Iterator* and *ListIterator* with the *modified* parameter called $cm$ in *Iterator* and $lm$ in *ListIterator*, and $Q$ is an abstraction relation coercing *List* to *Collection*:

$$\begin{aligned} I' \quad = \quad & col\ != null \wedge current \subseteq col.elems \wedge cm == col.modified \wedge \\ & (canRemove \Rightarrow next \in current) \end{aligned}$$

$$\begin{aligned} J' \quad = \quad & lst\ != null \wedge -1 \leq ind \leq \#lst.elems \wedge lm == lst.modified \wedge \\ & (canModify \Rightarrow 0 \leq ind < \#lst.elems) \end{aligned}$$

$$\begin{aligned} Q\ l\ c \quad = \quad & (l == null \wedge c == null) \vee (l\ != null \wedge c\ != null \wedge \\ & toBag(l.elems) == c.elems \wedge l.modified == c.modified) \end{aligned}$$

We distinguish the relation $Q$ because it will be used not only as a part of $R$, but also as an abstraction relation coercing constructor input parameters.

### 3.2.1 Proving Constructor and Method Refinement

We begin with proving data refinement between the constructors of *Iterator* and *ListIterator* with respect to the relations $Q$ and $R$. The goal we have to prove is as follows:

$$
\begin{array}{l}
\{c = c' \mid Q\ l\ c'\}; \\
\mathsf{assert}\ c\ != null; \\
col, current, canRemove, cm, next = c, [\!], false, c.modified, null \\
\sqsubseteq \\
\mathsf{assert}\ l\ != null; lst, ind, canModify, lm = l, -1, false, l.modified; \\
\{col, current, canRemove, cm, next, c = col', cur', r', cm', n', c' \mid \\
\quad R\ (lst, ind, canModify, lm)\ (col', cur', r', cm', n')\}
\end{array}
$$

Deterministic assignment can always be rewritten as angelic nondeterministic assignment, according to the rule

$$x = e \quad = \quad \{x = x' \mid x' == e\} \tag{3}$$

Also, assertion can be propagated inside an adjacent angelic assignment,

$$\mathsf{assert}\ p; \{x = x' \mid b\} \quad = \quad \{x = x' \mid p \wedge b\} \tag{4}$$

$$\{x = x' \mid b\}; \mathsf{assert}\ p \quad = \quad \{x = x' \mid p[x \leftarrow x'] \wedge b\} \tag{5}$$

Applying these rules we get

$$
\begin{array}{l}
\{c = c' \mid Q\ l\ c' \wedge c'\ != null\}; \\
\{col, current, canRemove, cm, next = col', cur', r', cm', n' \mid \\
\quad col' == c \wedge cur' == [\!] \wedge r' == false \wedge cm' == c.modified \wedge n' == null\} \\
\sqsubseteq \\
\{lst, ind, canModify, lm = lst', ind', m', lm' \mid \\
\quad l\ != null \wedge lst' == l \wedge ind' == -1 \wedge m' == false \wedge lm' == l.modified\}; \\
\{col, current, canRemove, cm, next, c = col', cur', r', cm', n', c' \mid \\
\quad R\ (lst, ind, canModify, lm)\ (col', cur', r', cm', n')\}
\end{array}
$$

Two angelic assignment statements can be merged together according to the following rule:

$$\{x = x' \mid b\}; \{y = y' \mid c\} \quad = \quad \{x, y = x', y' \mid b \wedge c[x \leftarrow x']\} \tag{6}$$

As the abstraction statement removes concrete attributes, replacing them with abstract ones, application of the above rule gives us the following:

$$
\begin{array}{l}
\{col, current, canRemove, cm, next, c = col', cur', r', cm', n', c' \mid \\
\quad Q\ l\ c' \wedge c'\ != null \wedge col' == c' \wedge cur' == [\!] \wedge \\
\quad r' == false \wedge cm' == c'.modified \wedge n' == null\} \\
\sqsubseteq \\
\{col, current, canRemove, cm, next, c = col', cur', r', cm', n', c' \mid \\
\quad l\ != null \wedge R\ (l, -1, false, l.modified)\ (col', cur', r', cm', n')\}
\end{array}
$$

Using the rule

$$(b \Rightarrow c) \quad \Rightarrow \quad \{x = x' \mid b\} \sqsubseteq \{x = x' \mid c\} \tag{7}$$

we can now reduce the proof to

$$\begin{array}{l} Q\ l\ c' \wedge c' \mathrel{!}= null \wedge col' == c' \wedge cur' == [\![]\!] \wedge \\ r' == false \wedge cm' == c'.modified \wedge n' == null \\ \Rightarrow \\ l \mathrel{!}= null \wedge col' \mathrel{!}= null \wedge cur' \subseteq col'.elems \wedge cm' == col'.modified \wedge \\ (r' \Rightarrow n' \in cur') \wedge l \mathrel{!}= null \wedge -1 \leq -1 \leq \#l.elems \wedge \\ l.modified == l.modified \wedge (false \Rightarrow 0 \leq -1 \leq \#l.elems - 1) \\ Q\ l\ col' \wedge false == r' \wedge (false \Rightarrow n' == l.elems[-1]) \wedge \\ toBag(l.elems[0..-1]) == cur' \end{array}$$

Applying simple logic transformations, we reduce this goal to true, completing our proof of constructor refinement.

For the proof of method refinement between the methods *hasNext* as defined in *Iterator* and *ListIterator*, we would need to show that the values returned in these methods are equal under the abstraction relation $R$:

$$R\ attr'\ attr \Rightarrow (current \subset col.elems = ind < \#lst.elems - 1)$$

We prove the boolean equality by proving mutual implications:

1. $R\ attr'\ attr \Rightarrow (current \subset col.elems \Rightarrow ind < \#lst.elems - 1)$
2. $R\ attr'\ attr \Rightarrow (ind < \#lst.elems - 1 \Rightarrow current \subset col.elems)$

For the proof of the first subgoal we use a lemma $c \subseteq b \Rightarrow \#c < \#b$, which can easily be proved for arbitrary bags $c$ and $b$, to get

$$R\ attr'\ attr \Rightarrow (\#current < \#col.elems \Rightarrow ind < \#lst.elems - 1)$$

Using the clause $toBag(lst.elems[0..ind]) == current$, which is a part of $R\ attr'\ attr$, and then a lemma $\#toBag(l) == \#l$, we get

$$\begin{array}{l} R\ attr'\ attr \Rightarrow \\ \quad (\#(lst.elems[0..ind]) < \#col.elems \Rightarrow ind < \#lst.elems - 1) \end{array}$$

Assuming that $ind \geq \#lst.elems - 1$ and using the definition of subsequence we get

$$\begin{array}{l} R\ attr'\ attr \wedge ind \geq \#lst.elems - 1 \Rightarrow \\ \quad (\#lst.elems < \#col.elems \Rightarrow ind < \#lst.elems - 1) \end{array}$$

Now from $R\ attr'\ attr$ we get that $toBag(lst.elems) == col.elems$ and, therefore, $\#lst.elems == \#col.elems$, using the above mentioned lemmas.

We reach a contradiction in the assumptions, thus proving the goal:

$$\begin{array}{ll} & R\ attr'\ attr\ \wedge\ ind \geq \#lst.elems - 1\ \wedge\ \#lst.elems == \#col.elems\ \Rightarrow \\ & \quad (\#lst.elems < \#col.elems\ \Rightarrow\ ind < \#lst.elems - 1) \\ = & R\ attr'\ attr\ \wedge\ ind \geq \#lst.elems - 1\ \wedge\ \#lst.elems == \#col.elems\ \wedge \\ & \quad \#lst.elems < \#col.elems\ \Rightarrow\ ind < \#lst.elems - 1) \\ = & false\ \Rightarrow\ ind < \#lst.elems - 1 \\ = & true \end{array}$$

The second subgoal

$$R\ attr'\ attr\ \Rightarrow\ (ind < \#lst.elems - 1\ \Rightarrow\ current \subset col.elems)$$

is proved similarly to the first subgoal, using lemmas

$$i < \#l - 1\ \Rightarrow\ \#l[0..i] < \#l \text{ and}$$
$$c \subseteq b\ \wedge\ \#c < \#b\ \Rightarrow\ c \subset b$$

The next method refinement we must prove is between the methods *next* in *Iterator* and *ListIterator* respectively. Namely, we have to prove the following data refinement:

$$\begin{array}{l} \mathsf{assert}\ current \subset col.elems; [next = e \mid e \in (col.elems \setminus current)]; \\ current, canRemove = current + [\![next]\!], true; \mathsf{return}\ next \\ \sqsubseteq_R \\ \mathsf{assert}\ ind < \#lst.elems - 1; ind, canModify = ind + 1, true; \\ \mathsf{return}\ lst.elems[ind] \end{array}$$

First of all, returning a value from a method can be modeled by assigning the returned value to a variable *res* representing the result parameter. Therefore, we can rewrite the above data refinement as follows:

$$\begin{array}{l} \mathsf{assert}\ current \subset col.elems; [next = e \mid e \in (col.elems \setminus current)]; \\ current, canRemove = current + [\![next]\!], true; res = next \\ \sqsubseteq_R \\ \mathsf{assert}\ ind < \#lst.elems - 1; ind, canModify = ind + 1, true; \\ res = lst.elems[ind] \end{array}$$

Two demonic assignment statements can be merged together according to the following rule:

$$[x = x' \mid b]; [y = y' \mid c] \quad = \quad [x, y = x', y' \mid b\ \wedge\ c[x \leftarrow x']] \tag{8}$$

Transforming deterministic assignments into demonic assignments and ap-

plying this rule, we get

$$
\begin{array}{l}
\mathsf{assert}\ current \subset col.elems; \\
[next, current, canRemove, res = n', cur', r', res' \mid n' \in (col.elems \setminus current) \wedge \\
\quad cur' == current + \llbracket n' \rrbracket \wedge r' == true \wedge res' == n'] \\
\sqsubseteq_R \\
\mathsf{assert}\ ind < \#lst.elems - 1; \\
[ind, canModify, res = ind', m', res' \mid \\
\quad ind' == ind + 1 \wedge m' == true \wedge res' == lst.elems[ind']]
\end{array}
$$

Applying the rule for data refinement of nondeterministic assignment statements, we can reduce the proof of this goal to two subgoals:

1. $current \subset col.elems \wedge R\ attr'\ attr \Rightarrow ind < \#lst.elems - 1$
2. $current \subset col.elems \wedge R\ attr'\ attr \wedge$
   $ind' == ind + 1 \wedge m' == true \wedge res' == lst.elems[ind']$
   $\Rightarrow$
   $(\exists col', cur', r', cm', n' \bullet$
   $n' \in (col.elems \setminus current) \wedge cur' == current + \llbracket n' \rrbracket \wedge r' == true \wedge$
   $res' == n' \wedge R\ (lst, ind', m', lm)\ (col', cur', r', cm', n'))$

The first subgoal, using the logical shunting rule

$$p \wedge q \Rightarrow r \ = \ p \Rightarrow (q \Rightarrow r)$$

is reduced to the first subgoal in the proof of method refinement between the methods *hasNext* and is already proved.

In the proof of the second subgoal we instantiate the existentially quantified variables by $col$, $current + \llbracket lst.elems[ind'] \rrbracket$, $true$, $cm$ and $lst.elems[ind']$ respectively, getting

$$
\begin{array}{l}
current \subset col.elems \wedge R\ attr'\ attr \wedge \\
ind' == ind + 1 \wedge m' == true \wedge res' == lst.elems[ind'] \\
\Rightarrow \\
lst.elems[ind'] \in (col.elems \setminus current) \wedge \\
current + \llbracket lst.elems[ind'] \rrbracket == current + \llbracket lst.elems[ind'] \rrbracket \wedge \\
true == true \wedge res' == lst.elems[ind'] \wedge col \mathrel{!}= null \wedge \\
current + \llbracket lst.elems[ind'] \rrbracket \subseteq col.elems \wedge cm == col.modified \wedge \\
(true \Rightarrow lst.elems[ind'] \in current + \llbracket lst.elems[ind'] \rrbracket) \wedge \\
lst \mathrel{!}= null \wedge -1 \leq ind' \leq \#lst.elems \wedge lm == lst.modified \wedge \\
(m' \Rightarrow 0 \leq ind' < \#lst.elems) \wedge Q\ lst\ col \wedge m' == true \wedge \\
(m' \Rightarrow lst.elems[ind'] == lst.elems[ind']) \wedge \\
toBag(lst.elems[0..ind']) == current + \llbracket lst.elems[ind'] \rrbracket
\end{array}
$$

Simplifying and rewriting with the definition of $R\ attr'\ attr$, we get

$$
\begin{array}{l}
current \subset col.elems \wedge R\ attr'\ attr \\
\Rightarrow \\
lst.elems[ind+1] \in (col.elems \setminus current) \wedge \\
current + \llbracket lst.elems[ind+1] \rrbracket \subseteq col.elems \wedge \\
lst.elems[ind+1] \in current + \llbracket lst.elems[ind+1] \rrbracket \wedge \\
-1 \leq ind \leq \#lst.elems - 2 \wedge \\
toBag(lst.elems[0..ind+1]) == current + \llbracket lst.elems[ind+1] \rrbracket
\end{array}
$$

To prove this goal, we use the following lemmas:

$$toBag(\langle e \rangle) \ = \ \llbracket e \rrbracket \tag{9}$$

$$toBag(l_1) + toBag(l_2) \ = \ toBag(l_1 \frown l_2) \tag{10}$$

$$l[0..i+1] \ = \ l[0..i] \frown \langle l[i+1] \rangle \tag{11}$$

$$b_1 \subset b_2 \wedge e \in b_2 \ \Rightarrow \ b_1 + \llbracket e \rrbracket \ \subseteq \ b_2 \tag{12}$$

$$(\exists i \mid 0 \leq i \leq \#l - 1 \bullet l[i] = e) \ \Rightarrow \ e \in toBag(l) \tag{13}$$

$$e \in b_1 \ \wedge \ b_2\ e < b_1\ e \ \Rightarrow \ e \in (b_1 \setminus b_2) \tag{14}$$

Proofs of these lemmas are straightforward from the definitions of the corresponding bag and sequence operators. Rewriting the goal with these lemmas and simplifying, we get

$$
\begin{array}{l}
current \subset col.elems \wedge R\ attr'\ attr \\
\Rightarrow \\
lst.elems[ind+1] \in col.elems \ \wedge \ -1 \leq ind \leq \#lst.elems - 2 \wedge \\
current\ (lst.elems[ind+1]) < (col.elems)\ (lst.elems[ind+1]) \wedge \\
toBag(lst.elems[0..ind]) + \llbracket lst.elems[ind+1] \rrbracket == \\
\quad current + \llbracket lst.elems[ind+1] \rrbracket
\end{array}
$$

Finally, using the earlier proved property

$$current \subset col.elems \wedge R\ attr'\ attr \ \Rightarrow \ ind < \#lst.elems - 1$$

and rewriting with clauses

$$
\begin{array}{l}
-1 \leq ind \leq \#lst.elems \\
toBag(lst.elems[0..ind]) == current \\
toBag(lst.elems) == col.elems
\end{array}
$$

from $R\ attr'\ attr$, we prove this goal.

We omit the proof of $Iterator.remove \sqsubseteq_R ListIterator.remove$ which is carried out in the same manner as the proof of $Iterator.next \sqsubseteq_R ListIterator.next$, using the same lemmas.

### 3.2.2 Proving Preservation of Invariants

While verifying correctness of a class having explicit invariants, we should prove that constructors of this class establish the (combined class and inter-class) invariant and all methods preserve it. Here we will only demonstrate how one can prove that methods preserve the invariant. We show that the method *add* of *ListIterator* preserves the invariant $J'$, expressed as the following correctness assertion:

$$J' \;\{\!|\; \begin{array}{l} ind = ind + 1; [lst.elems = s \mid \exists s_1, s_2 \bullet \\ \quad s == s_1 \frown \langle o \rangle \frown s_2 \wedge lst.elems == s_1 \frown s_2 \wedge s[ind] == o]; \\ canModify, lm = false, lm + 1; lst.modified = lm \end{array} \;|\!\}\; J'$$

In the proof of this correctness assertion we will use the following rules, presented and proved in [6]:

$$p \;\{\!|\, S_1; S_2 \,|\!\}\; q \;=\; (\exists r \bullet p \;\{\!|\, S_1 \,|\!\}\; r \wedge r \;\{\!|\, S_2 \,|\!\}\; q) \tag{15}$$

$$p \;\{\!|\, x = e \,|\!\}\; q \;=\; p \subseteq q[x \leftarrow e] \tag{16}$$

$$p \;\{\!|\, x = x' \mid b \,|\!\}\; q \;=\; p \subseteq (\forall x' \bullet b \Rightarrow q[x \leftarrow x'] \tag{17}$$

Applying rules (15) and (16) and instantiating the existentially quantified predicate to $J_1$ such that

$$J_1 \;=\; \begin{array}{l} lst \mathrel{!}= null \wedge -1 \leq ind \leq \#lst.elems + 1 \wedge lm == lst.modified \wedge \\ (canModify \Rightarrow 0 \leq ind \leq \#lst.elems) \end{array}$$

we get to prove two subgoals

1. $J' \Rightarrow J_1[ind \leftarrow ind + 1]$
   $[lst.elems = s \mid \exists s_1, s_2 \bullet$
2. $J_1 \;\{\!|\; \begin{array}{l} s == s_1 \frown \langle o \rangle \frown s_2 \wedge lst.elems == s_1 \frown s_2 \wedge s[ind] == o]; \\ canModify, lm = false, lm + 1; lst.modified = lm \end{array} \;|\!\}\; J'$

The first subgoal obviously holds, since $-1 \leq ind \leq \#lst.elems \Rightarrow -1 \leq ind + 1 \leq \#lst.elems + 1$ and $0 \leq ind < \#lst.elems \Rightarrow 0 \leq ind + 1 \leq \#lst.elems$. For proving the second subgoal we apply rules (15) and (17), instantiating the existentially quantified predicate to $J'$:

1. $\begin{array}{l} lst \mathrel{!}= null \wedge -1 \leq ind \leq \#lst.elems + 1 \wedge lm == lst.modified \wedge \\ (canModify \Rightarrow 0 \leq ind \leq \#lst.elems) \\ \Rightarrow \\ (\forall s \bullet (\exists s_1, s_2 \bullet s == s_1 \frown \langle o \rangle \frown s_2 \wedge lst.elems == s_1 \frown s_2 \wedge s[ind] == o) \\ \quad \Rightarrow lst \mathrel{!}= null \wedge -1 \leq ind \leq \#s \wedge lm == lst.modified \wedge \\ \quad (canModify \Rightarrow 0 \leq ind \leq \#s - 1)) \end{array}$
2. $J' \;\{\!|\, canModify, lm = false, lm + 1; lst.modified = lm \,|\!\}\; J'$

Using simple logic transformations, the first of these goals can be reduced to

$$
\begin{array}{l}
-1 \leq ind \leq \#lst.elems + 1 \wedge (canModify \Rightarrow 0 \leq ind \leq \#lst.elems) \wedge \\
s == s_1 \frown \langle o \rangle \frown s_2 \wedge lst.elems == s_1 \frown s_2 \\
\Rightarrow \\
-1 \leq ind \leq \#s \wedge (canModify \Rightarrow 0 \leq ind \leq \#s - 1)
\end{array}
$$

and proved using the lemma $\#(l_1 \frown l_2) = \#l_1 + \#l_2$.

For the proof of the second subgoal, we apply rules (15) and (16) instantiating the existentially quantified predicate to $J_2$ such that

$$
\begin{array}{rcl}
J_2 & = & lst \mathrel{!}= null \wedge -1 \leq ind \leq \#lst.elems \wedge lm == lst.modified + 1 \wedge \\
& & (canModify \Rightarrow 0 \leq ind < \#lst.elems)
\end{array}
$$

The resulting subgoals

1. $lst \mathrel{!}= null \wedge -1 \leq ind \leq \#lst.elems \wedge lm == lst.modified \wedge$
   $(canModify \Rightarrow 0 \leq ind < \#lst.elems)$
   $\Rightarrow$
   $lst \mathrel{!}= null \wedge -1 \leq ind \leq \#lst.elems \wedge lm + 1 == lst.modified + 1 \wedge$
   $(false \Rightarrow 0 \leq ind < \#lst.elems)$
2. $lst \mathrel{!}= null \wedge -1 \leq ind \leq \#lst.elems \wedge lm == lst.modified + 1 \wedge$
   $(canModify \Rightarrow 0 \leq ind < \#lst.elems)$
   $\Rightarrow$
   $lst \mathrel{!}= null \wedge -1 \leq ind \leq \#lst.elems \wedge lm == lm \wedge$
   $(canModify \Rightarrow 0 \leq ind < \#lst.elems)$

hold trivially.

This completes our proof of method *ListIterator.add* preserving the combined class and interclass invariant of *ListIterator*. Proofs of invariant preserving for other methods of *ListIterator* can be carried out in a similar manner. Naturally, the same principles apply to proving consistency of methods of *Iterator* with respect to its combined invariant. Moreover, non-modifying methods, such as *hasNext*, *hasPrevious*, *nextIndex*, and *previousIndex*, preserve the corresponding invariants automatically.

When proving refinement between two classes having explicit invariants, we should also verify that the concrete invariant is stronger than or equal to the abstract invariant with respect to an abstraction relation. As in our case *Iterator* and *ListIterator* are the specifications of the corresponding interfaces and *ListIterator* does not inherit from *Iterator*, a stronger requirement that the invariants $I'$ and $J'$ must be equal does not apply. Therefore, we have only to prove that $J'$ is stronger than or equal to $I'$ with respect to $R$, i.e.

$$J' \Rightarrow (\exists attr \bullet R\ attr'\ attr \wedge I')$$

where *attr* abbreviates $(col, current, canRemove, cm, next)$ and $attr'$ abbreviates $(lst, ind, canModify, lm)$, and also $I'$ and $J'$ are the combined invariants of *Iterator* and *ListIterator* with the *modified* parameter called *cm* in *Iterator* and *lm* in *ListIterator*. Rewriting with the definitions of $I'$ and $J'$, we get to prove the following goal:

$$
\begin{array}{l}
lst \mathrel{!}= null \wedge -1 \leq ind \leq \#lst.elems \wedge lm == lst.modified \wedge \\
(canModify \Rightarrow 0 \leq ind < \#lst.elems) \\
\Rightarrow \\
(\exists attr \bullet R\ attr'\ attr \wedge col \mathrel{!}= null \wedge \\
current \subseteq col.elems \wedge cm == col.modified \wedge \\
(canRemove \Rightarrow next \in current))
\end{array}
$$

Instantiating the existentially quantified variables *attr* so that *col* is instantiated with a reference $c$ to an object with $elems == toBag(lst.elems)$ and $modified == lst.modified$, *canRemove* is instantiated with *canModify*, *current* with $toBag(lst.elems[0..ind])$, *cm* with $c.modified$, and *next* with $lst.elems[ind]$, we can prove this goal as well.

# 4 Conclusions and Related Work

In this paper we present a novel approach to specification and verification of object-oriented frameworks. The novelty of our approach is in blurring the difference between specifications and implementations which permits abstracting away from implementation details in a specification, yet being precise about important behavioral issues, such as, e.g., a fixed method invocation order or an iterative execution of a particular statement. The benefits of combining executable statements with specification statements when reasoning about object-oriented and component-based systems are described by Martin Büchi and Emil Sekerinski in [8]. In particular, they note that a popular form of specification in terms of pre- and postconditions does not scale well to reasoning about object-oriented and component-based systems, because pre- and postconditions, being predicates, cannot contain calls to other methods, except when the latter are pure functions. Therefore, one has to reinvent the wheel every time when specifying the behavior of a method implementing some functionality by calling other methods. Specifications in terms of abstract statements, as pointed out in [8], are not affected by this scalability problem. Also, Büchi and Sekerinski note that pre- and postconditions, which are only checked at runtime, help to locate errors but do not prevent them as does static analysis.

Our specification method is supported by a solid formal foundation: every executable statement of the Java language as well as every specification statement that we use has a precise mathematical meaning as described in [19, 3]. Moreover, treating specifications and implementations in a uniform logical framework permits formal reasoning about their relationship

and properties. Of particular interest is the verification of behavioral conformance between specifications of interfaces and implementations of these interfaces. Verifying behavioral conformance of implementations to their specifications as well as behavioral conformance of subinterface specifications to the corresponding superinterface specifications permits ensuring correctness of the whole system.

We illustrate our specification method by specifying a part of the Java Collections Framework. We have developed formal specifications of other subinterfaces of *Collection* as well, but omit them for the reasons of limited space. In the process of specifying JCF interfaces we had to make several important design decisions resolving the ambiguities and inconsistencies in the original documentation. The expressiveness of our specification language and the preciseness that it requires, forced us to deal with the issues that were overlooked, underspecified, left undefined, or even had contradicting descriptions. In particular, from analyzing the description of the relationship between structure-modifying methods of *Collection* and *Iterator* one concludes that the methods *add* and *remove* of *Collection* should perform structural modifications through calls to the identically-named methods of *Iterator*. However, the interface *Iterator* does not even provide a method *add*, which either indicates the problems with the informal description of the behavior or the possibility of inadequate framework design.

Another interesting point to note is that, according to our specification, iterating several times over a collection can every time return elements in a different order, which is the most liberal specification describing iteration over an unordered collection, and which corresponds directly to the description of *Collection.iterator* in [7] stating: "There are no guarantees concerning the order in which the elements are returned [...]". This specification permits both an implementation that imposes no order on the returned elements, and a more deterministic implementation that guarantees the same order of elements every time when iterating over the same collection. To mention just one more place where we have identified inconsistencies in the original documentation, the method *Collection.removeAll* does not stipulate the requirement that the incoming collection cannot be modified during its execution; however, based on the description of this method's behavior, one can conclude that this requirement is absolutely necessary.

The difference in size between the implementation of *Collection*'s *contains* method as given in the class *AbstractCollection*, which is a part of the standard JCF implementation, and between our specification of this method is quite illustrative of the general picture. In *AbstractCollection* the method *contains* is defined by

```
public boolean contains(Object o) {
    Iterator e = iterator();
        if (o==null) {
            while (e.hasNext())
                if (e.next()==null) return true;
        } else {
            while (e.hasNext())
                if (o.equals(e.next())) return true;
        }
    return false;
}
```

while in our specification it is defined by return $(o \in elems)$.

Related work in formal specification of object-oriented systems includes William Cook's specification in [12] of Smalltalk-80 collection class library. Although the library is organized by inheritance, Cook argues that interface inheritance or subtyping is a logical basis for the library organization, supporting this claim by specifying the interfaces and revealing several problems with the current organization of the library. With the Java Collections Framework that we specify here, interface inheritance is separated from the implementation inheritance and, since the former forms the basis for polymorphic object substitutability in client programs, we associate behavioral specifications with interfaces, as does Cook. One of the main differences of our work from that of Cook is that his specifications are given in terms of pre- and postconditions, following Pierre America's approach in [1], while we use a specification language combining specification statements with executable ones.

The detailed elaboration of our formalization of object-oriented constructs and mechanisms, as described in [19, 3], opens the possibility of mechanized reasoning and mechanical verification. An interesting recent work by Bart Jacobs et al. in [15] reports a work in progress on building a front-end tool for translating Java classes to higher-order logic in PVS [22]. The authors state that "current work involves incorporation of Hoare logic [10], via appropriate definitions and rules in PVS", and present in [15] a description of the tool "directly based on definitions". In this work we test the applicability of the theoretic foundation for reasoning about object-oriented programs developed in [19, 3]. This theoretic foundation is based on the logical framework for reasoning about imperative programs. A tool supporting verification of correctness and refinement of imperative programs and known as the Refinement Calculator [16] already exists and extending it to handling object-oriented programs, including Java programs, appears to be only natural.

There are a few issues that we haven't addressed in this project, in particular, the role of exceptions, their relation to assertion statements and their formal semantics are left as a topic for future work. Method early returns are treated somewhat informally: we assume that every method

returning the result inside the conditional statement or inside the loop can be rewritten to an equivalent one returning the result as the last operation. Formal treatment of early returns represents an interesting research topic.

Another important aspect that we haven't addressed in this work is that of concurrent execution of method invocations. In the underlying formalism [19, 3] method calls are modelled as being *atomic*, meaning that once a method has been invoked on an object, no other method will be invoked on the same object before the first method has completed execution. However, our specification of method *addAll*, for example, clearly indicates the need for considering non-atomic (concurrent) method invocations as well, i.e. such invocations that can be interrupted by other method invocations and resumed only upon their completion. Extending the specification and verification method described here to handling concurrency represents an interesting and important research direction. The refinement calculus supports reasoning about concurrent systems as reported in [5] and, therefore, extending it to handling object-oriented concurrency appears to be a challenging yet feasible task.

## Acknowledgements

We would like to thank Joakim von Wright, Ralph Back, Gary Leavens, and Michael Butler for useful comments on this paper.

## References

[1] 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*, LNCS 489, pages 60–90, New York, N.Y., 1991. Springer-Verlag.

[2] R. Back, A. Mikhajlova, and J. von Wright. Class refinement as semantics of correct object substitutability. Unpublished monograph extending [3], submitted for publication.
http://www.abo.fi/~amikhajl/Papers/ClassRefSem.ps.

[3] 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.

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

[5] R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

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

[7] J. Bloch. Java Collections Framework: Collections 1.2. http://java.sun.com/docs/books/tutorial/collections/index.html.

[8] M. Büchi and E. Sekerinski. Formal methods for component software: The refinement calculus perspective. In *Second Workshop on Component-Oriented Programming (WCOP'97) held in conjunction with ECOOP'97*, June 1997.

[9] M. Butler, J. Grundy, T. Långbacka, R. Rukšėnas, and J. von Wright. The refinement calculator: Proof support for program refinement. In L. Groves and S. Reeves, editors, *Formal Methods Pacific'97: Proceedings of FMP'97*, Discrete Mathematics & Theoretical Computer Science, pages 40–61, Wellington, New Zealand, July 1997. Springer-Verlag.

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

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

[12] W. R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *Proceedings of OOPSLA'92*, pages 1–15. ACM SIGPLAN Notices, 27(10), October 1992.

[13] E. W. Dijkstra. *A Discipline of Programming*. Prentice–Hall International, 1976.

[14] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *European Symposium on Programming*, LNCS 213. Springer-Verlag, 1986.

[15] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes (preliminary report). In *Proceedings of OOPSLA'98*, pages 329–340, Vancouver, Canada, Oct. 1998. Association for Computing Machinery.

[16] T. Långbacka, R. Ruksenas, and J. von Wright. TkWinHOL: A tool for window inference in HOL. *Higher Order Logic Theorem Proving and its Applications: 8th International Workshop*, 971:245–260, September 1995.

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

[18] A. Mikhajlova. Consistent extension of components in the presence of explicit invariants. In *Technology of Object-Oriented Languages and Systems (TOOLS 29)*, pages 76–85. IEEE Computer Society Press, June 1999.

[19] A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In *Proceedings of the 4th International Formal Methods Europe Symposium, FME'97*, LNCS 1313, pages 82–101. Springer, 1997.

[20] C. C. Morgan. Data refinement by miracles. *Information Processing Letters*, 26:243–246, January 1988.

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

[22] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

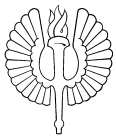
Turku Centre for Computer Science

TUCS Dissertations

1. **Marjo Lipponen**, On Primitive Solutions of the Post Correspondence Problem
2. **Timo Käkölä**, Dual Information Systems in Hyperknowledge Organizations
3. **Ville Leppänen**, Studies on the Realization of PRAM
4. **Cunsheng Ding**, Cryptographic Counter Generators
5. **Sami Viitanen**, Some New Global Optimization Algorithms
6. **Tapio Salakoski**, Representative Classification of Protein Structures
7. **Thomas Långbacka**, An Interactive Environment Supporting the Development of Formally Correct Programs
8. **Thomas Finne**, A Decision Support System for Improving Information Security
9. **Valeria Mihalache**, Cooperation, Communication, Control. Investigations on Grammar Systems
10. **Marina Waldén**, Formal Reasoning About Distributed Algorithms
11. **Tero Laihonen**, Estimates on the Covering Radius When the Dual Distance is Known
12. **Lucian Ilie**, Decision Problems on Orders of Words
13. **Jukkapekka Hekanaho**, An Evolutionary Approach to Concept Learning
14. **Jouni Järvinen**, Knowledge Representation and Rough Sets
15. **Tomi Pasanen**, In-Place Algorithms for Sorting Problems
16. **Mika Johnsson**, Operational and Tactical Level Optimization in Printed Circuit Board Assembly
17. **Mats Aspnäs**, Multiprocessor Architecture and Programming: The Hathi-2 System
18. **Anna Mikhajlova**, Ensuring Correctness of Object and Component Systems

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

http://www.tucs.abo.fi



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

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

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