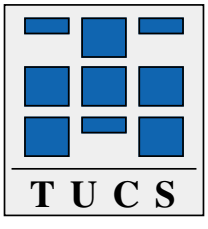
Safe Language Mechanisms for Modularization and Concurrency

Martin Büchi



Turku Centre for Computer Science

TUCS Dissertations

No 28, May 2000

Safe Language Mechanisms for Modularization and Concurrency

Martin Büchi

To be presented —with the permission of the Faculty of Mathematics and Natural Sciences at Åbo Akademi University— for public criticism in Alabama's Auditorium at DataCity in Turku, Finland, on June 9, 2000, at 12 noon.

Department of Computer Science
Åbo Akademi University

Supervised by

Professor Ralph Back
Department of Computer Science
Åbo Akademi University
Lemminkäisenkatu 14A
20520 Turku
Finland

Reviewed by

Professor Cliff B. Jones
Department of Computing Science
University of Newcastle upon Tyne, NE1 7RU
The United Kingdom

Dr. Clemens A. Szyperski
Microsoft Research
One Microsoft Way 31/1288
Redmond, WA 98052
U.S.A.

ISBN 951-29-1727-0
ISSN 1239-1883

Painosalama Oy, 2000
Turku, Finland

Para 恭子

Acknowledgments

I would like to thank my supervisor, Professor Ralph Back, for his encouragement, support, and guidance. As a professor, he provided me with the best possible apprenticeship to scientific research. As a friend, he showed great understanding for my needs and gave me continuous support. Together with Professors Johan Lilius, Kaisa Sere, and Joakim von Wright, he created a very inspiring and human environment for performing research.

Professor Cliff Jones of the University of Newcastle, United Kingdom, and Dr. Clemens Szyperski of Microsoft Research, USA, kindly agreed to review this dissertation. I would like to thank them for their insightful comments and suggestions, which helped me improve the introduction, the appendix, and some of the papers.

I had the good fortune of receiving guidance and support from four former postdocs and researchers at the programming methodology group. Dr. Emil Sekerinski (currently at McMaster University, Canada) introduced me to concurrency theory, the B method, and formal reasoning about object-oriented systems. As we co-authored three papers together, he taught me how to publish scientific articles. Dr. Wolfgang Weck (currently at Oberon microsystems Inc., Switzerland) filled me with enthusiasm for component software and type systems. His never-ending pursuit of an intelligible structure and a clear focus greatly improved the five papers we wrote together. Dr. Jim Grundy (currently at the Australian National University, Australia) and Dr. John Harrison (currently at Intel Corporation, USA) helped me with higher order logic and theorem proving.

Dr. Marina Waldén patiently answered my questions on the B method and on action systems. Furthermore, she bravely endured my first sentences in Finnish and Swedish and is greatly responsible for my present fluency in these two beautiful languages. Dr. Mats Aspnäs was the universal source for help and provided superb lunch company. I had many inspiring discussions with my fellow Ph.D. students Dr. Linas Laibinis, Dr. Leonid Mikhajlov, Dr. Anna Mikhajlova, Iván Porres Paltor, Rimvydas Rukšėnas, Mauno Rönkkö, and Elena Troubitsyna as well as with other members of the programming methodology group.

I would like to thank the staff of Åbo Akademi University and the Turku Centre for Computer Science for creating a pleasant and efficient working environment. The financial support from the Turku Centre for Computer Science and ABB Switzerland is gratefully acknowledged.

Sari Isotalo and Albert Morgades, Kaisa Yli-Jokipii and Jussi Linderborg, and Ulrika and Lasse Nielsen have become fantastic friends during the past 4 years. They have greatly enriched my life. Thanks, too, to the Ikkala extended family for introducing me to Finnish family life.

Finally, I would like to thank my parents, my brother, and Yasko for their continuous support.

Turku, May 10, 2000
Martin Büchi

Abstract

We study safe language mechanisms for modularization and concurrency. Our contributions are a case study and several new language mechanisms with associated theories. Our motivation is twofold. First, the construction of software used in safety-critical systems requires expressive specification and programming languages that are themselves safe. Second, we want to gain insight into models for creating correct programs.

Large programs must be decomposed into smaller parts —such as modules, components, or classes— that can be considered one at a time without too much regard for the remaining parts. Concise formal specifications facilitate the use of these parts. Refinement can be used to prove implementations correct with respect to their specifications. Furthermore, specification and refinement improve the development process by separating the *what* from the *how*.

We have performed a medium-sized case study in the B formal method to explore the practical application of modularization, specification, and refinement. The sharing restrictions among modules in B caught our attention. To overcome these restrictions, we propose a new compositional symmetric sharing mechanism based on roles expressing rely/guarantee conditions.

B and other standard specification and data refinement methods work well for layered systems. However, they cannot in a modular way handle call-backs, which are common in object- and component-based systems. We propose a new specification and refinement method that addresses this need. Our specifications explicitly prescribe making external calls during the execution of a method, and the corresponding notion of refinement considers the sequence of external calls as part of the observable behavior that is preserved.

Compiler-checkable explicitly named and declared types with associated semantic contracts can augment the semantic specification and refinement approach. To be efficiently decidable and safe, type systems must forbid some semantically correct compositions. We isolate such a case, and the novel solution we propose takes a step toward the reconciliation of safety and flexibility. Our solution is especially useful for component software because component assemblers often lack the ability to change and recompile components in order to make them compatible.

Existing composition mechanisms either fail to fully address the safety, modular reasoning, and late composition requirements of component software, or allow only limited reuse due to typing restrictions. We introduce a new object composition mechanism that addresses the above needs of component software.

We also explore the benefits of specification, refinement, and object orientation in concurrent systems. We build on action systems, which give us a simple, yet powerful model for concurrency. We add objects to action systems and investigate both practical and theoretical aspects of our new model. Our compiler and run-time environment provide for animation. On the theoretical side, we define trace refinement of active objects. Our notion of refinement supports algorithmic, data, and atomicity refinement in the implementation of specifications and in behavioral subtyping.

To assure the safety of both our systems and language mechanisms, we have applied tool-based formal methods to achieve the most rigorous proofs. We have used Atelier B to mechanically check our B case study, and have proved type soundness of our extended type systems in Isabelle/HOL.

Contents

List of Original Publications

I. Martin Büchi. The B Bank. In Emil Sekerinski and Kaisa Sere, editors, *Program Development by Refinement: Case Studies Using the B Method*, FACIT series, chapter 4, pages 115–180. Springer Verlag, 1998.

II. Martin Büchi and Ralph Back. Compositional Symmetric Sharing in B. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *Proceedings of FM'99: World Congress on Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 431–451. Springer Verlag, September 1999.

III. Martin Büchi and Wolfgang Weck. The Greybox Approach: When Blackbox Specifications Hide too Much. Submitted for publication.

IV. Martin Büchi and Wolfgang Weck. Compound Types for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) '98*, pages 362–373. ACM Press, 1998.

V. Martin Büchi and Wolfgang Weck. Generic Wrapping. Technical Report 317, Turku Centre for Computer Science, April, 2000. Shortened version: Generic Wrappers. In Proceedings of ECOOP 2000, *Lecture Notes in Computer Science*. Springer Verlag, June 2000.

VI. Ralph Back, Martin Büchi, and Emil Sekerinski. Action-Based Concurrency and Synchronization for Objects. In M. Bertran and T. Reus, editors, *Proceedings of the Fourth AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software (ARTS)*, volume 1231 of *Lecture Notes in Computer Science*, pages 248–262. Springer Verlag, May 1997.

VII. Martin Büchi and Emil Sekerinski. Refining Concurrent Objects. Conditionally accepted to Fundamenta Informaticae with request for changes.

1 Introduction

Computer-based systems are ubiquitous in safety-critical applications. The dependability of such systems —say in airplanes, medical equipment, and banks— is therefore of critical importance. The quality of the instruments used to construct computer-based systems strongly influences the dependability of these systems. The main instruments for building computer-based systems include specification and programming languages. In this thesis, we identify problems with existing languages in the key areas of modularization and concurrency, and propose new mechanisms to address these problems. Because we are concerned with the safety of systems, we use formal methods to show that our mechanisms themselves are safe.

Structure and abstraction are the main techniques leading to understandable and hence trustworthy programs. Programs must be decomposed into partitions that can be considered one at a time without too much regard for the remaining parts.

If a team is to develop a large program, the original task must be split into smaller subproblems that can be handled by individual programmers. Solutions to individual problems may rely on the solutions of other subproblems. For example, the implementation of accounts in a banking application may rely on a database. In turn, the implementation of accounts may be used from various front-ends, such as ATMs and cashiers' terminals. Programmers should be able to build on the work of others, without having to study the inner workings of the others' solutions. Hence, program building blocks need to be equipped with concise abstract descriptions for facilitating their use. The decomposition of programs into smaller parts with concise specifications is called *modularization*.

Refinement allows developers to show that program parts satisfy their specifications. Furthermore, program development by refinement yields a separation of concerns: When we write specifications, we can focus on the *what* without being distracted by the *how*. When we implement a part, we can focus on efficiency and correctness without having to worry about determining the requirements of clients.

The formalization of specifications and of refinement enables tool-based mechanical reasoning about properties of our programs. Formalization can provide insight, deliver assurance, and help with debugging.

Types can enhance modular development. Explicitly declared and named types can stand for external semantic specifications. Unlike semantic specifications and refinement, types can be automatically checked by compilers and run-time environments.

In this thesis we introduce several novel language mechanisms for the concise specification of modules and for the composition of modules and instances of items defined therein. Two of these mechanisms use types for automatic checking.

Language mechanisms used for constructing safety-critical systems must themselves be safe. A language mechanism is safe if it is free from defects and produces the expected results. Whereas everyone is likely to be interested in safety, there are different ways of asserting it. In this thesis, we have chosen formal methods to give the most rigorous safety proofs for our systems and language mechanisms. To further increase the confidence in proofs, we have partly used mechanized theorem provers.

The second area of concern in this thesis is *concurrency*. Many computer systems are concurrent in the sense that multiple things can happen at the same time. For exam-

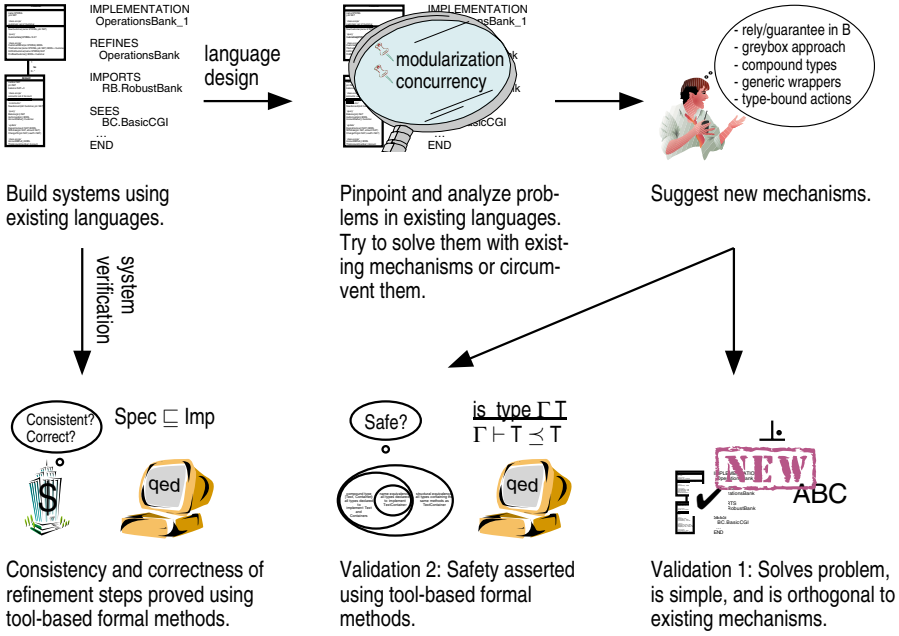


Figure 1: Overview of the system construction and language design processes

ple, several travel agents in different countries can make reservations and cancellations for the same flight at the same time. Concurrent systems are intrinsically more complicated to develop than their sequential counterparts. This is due to synchronization, communication, and non-interference requirements. The choice of a conceptual model for concurrency determines how well we can focus on the actual requirements without having to worry about unwanted effects, such as the deadlock of two processes waiting for each other. In this thesis we extend an existing model for concurrency, i.e. action systems, by adding objects to achieve a more expressive, yet still simple formalism.

Figure 1 summarizes the general approach of this thesis: Existing languages are used to build systems. The dependability of these systems is asserted using tool-based formal methods. Problems that are encountered with existing languages are analyzed and new mechanisms are proposed for solving them. The new mechanisms are validated in two steps. First they must solve the problems at hand, be as simple as possible, and be orthogonal to existing mechanisms that they do not replace. Second, their safety is proved using formal methods.

Publications and practical work The body of this thesis comprises seven articles. In addition to the papers, I have for this thesis produced several software artifacts and mechanized proofs constituting a total effort of approximately one person year. For Paper I, I have produced a B case study with 2,324 lines of B in 15 constructs and a total of 1,397 proof obligations, all of which I have discharged with the mechanized theorem

prover of Atelier B. For Papers IV and V, I have extended a formalization of Java's type system in Isabelle/HOL and proved type soundness for the extended systems. Additionally, I have written an Action-Oberon compiler and a run-time environment to experiment with object-oriented action systems, as described in Papers VI and VII.

Overview The remainder of this introduction is organized as follows. The first three sections introduce some basic concepts used throughout this thesis. Section 2 characterizes modules, components, and classes. Special emphasis is put on modular information hiding. In Sect. 3 we discuss formal specifications. We illustrate the advantage of a formal foundation and argue for the use of tools. Refinement to assert the correctness of implementations with respect to their specifications is discussed in Sect. 4. Modularization, specification, and refinement are also key ingredients of the case study in Paper I.

In Sect. 5 we describe an interference problem of shared modules in B and sketch how we solve it in Paper II. In Sect. 6 we discuss the specification and refinement of external calls. This topic is studied in detail in Paper III

Types and their use as automatically checkable aids for the construction of semantically consistent and correct programs are described in Sect. 7. In Sect. 8 we introduce compound types to solve a typing problem in component software. In Sect. 9 we propose a new language construct for late composition. Targeted at strongly-type languages, it utilizes the type system to provide maximal static and dynamic safety.

Section 10 is devoted to concurrency. We introduce object-oriented action systems as a simple, yet powerful model for concurrency. Furthermore, we reiterate the ideas of specification and refinement in the concurrent context.

Section 11 contains the conclusions. Throughout the introduction, we point out which aspects we have studied in Papers I–VII and what contributions we have made.

The seven papers listed above form the body of this thesis. The appendix gives a short introduction to the B method for the purpose of Papers I and II.

2 Modules, Components, and Classes

In this section we present the basic concepts of modular, component-oriented, and object-oriented programming.

2.1 Modules

A *module*, as in Modula-2 [59], Oberon-2 [47], or B [1], is a closed static unit that encapsulates embedded abstractions, such as types, variables, constants, functions, procedures, and classes. A module is closed in the sense that others cannot add, remove, or change any items. In contrast, Java [27] packages, to which anyone can add new classes, are open. Because modules are closed, their encapsulated domains are fixed and the modules can be fully analyzed. Modules can *import* other modules and thereby gain access to exported items of the imported modules. Modules can be understood, refined, implemented, compiled, and reasoned about separately, that is knowing only the

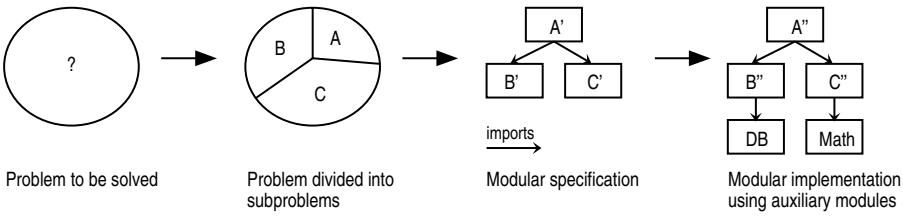


Figure 2: Modular solution of a problem

specifications of imported modules. Modularization leads to a separation of concerns and thus reduces the detailed reasoning needed to a doable amount.

Figure 2 illustrates a modular solution of a problem. The problem to be solved is divided into subproblems *A*, *B*, and *C*. The subproblems and their dependencies are specified. Different programmers then implement the three specifications. Modules B'' and C'' use the auxiliary modules *DB* and *Math* to complete their tasks.

A module can hide details from clients that import it. Thus, modules establish a new level of abstraction. This is called *information hiding*. Figure 3 gives part of a simple banking application in Oberon-2 as an example of information hiding: Clients of module *Bank1* can call the exported (indicated by the '*' after the name) procedure *Withdraw*, which appears in the interface. On the other hand, clients cannot directly manipulate the balances because they are not exported.

Information hiding has three benefits:

1. Implementers of client modules are not bothered by implementation details of the imported module. They must only understand the specification to utilize a module. In the example, they do not have to know that accounts are implemented with two arrays. This is an advantage even if the same person develops both the provider and the client module. She or he doesn't have to remember the details of the provider implementation when coding the client.

2. The implementer is free to choose and change the actual data structure and algorithms. For example, accounts could be implemented with linked lists or using an SQL database instead.

3. Clients cannot tamper with and corrupt the data structure, read secret information, or make unauthorized modifications.[1] In the example, clients cannot overdraw accounts, read pins, or withdraw money without the correct pin.

In Paper I we make extensive use of modular information hiding. Every module is divided into a semantic specification and an implementation. The concise specifications helped us to reduce the amount of detailed reasoning when writing client modules. Correctness proofs of clients were greatly simplified by using the abstract descriptions of imported modules, rather than the lengthy implementations.

[1] Most modularization mechanisms are suitable as software engineering aids, but not as security mechanisms because they do not enforce information hiding on the binary level.

```
DEFINITION Bank1; (* Interface visible to clients *)
  PROCEDURE Withdraw(no, pin, amount: INTEGER; VAR ok: BOOLEAN);
  ...
END Bank1.
```

```
MODULE Bank1; (* Implementation *)
  VAR nofAccounts: INTEGER;
    pins, balances: ARRAY 10 OF INTEGER;

  PROCEDURE Withdraw*(no, pin, amount: INTEGER; VAR ok: BOOLEAN);
  BEGIN
    IF (0<=no) & (no<nofAccounts) & (pins[no]=pin) & (amount>0) &
        (balances[no]>=amount) THEN
      balances[no]:=balances[no]-amount; ok:=TRUE
    ELSE ok:=FALSE
    END
  END Withdraw;
  ...
END Bank1.
```

Figure 3: Information hiding: Visible interface and implementation of *Bank1*

Additionally, modular information hiding allowed us to specify provider modules and check whether their specifications would provide suitable services to clients without being concerned about the implementations. When writing the implementations, we only needed to think of the specification, but not of all the different client usages.

Finally, information hiding guaranteed that client modules could not invalidate the invariants of service modules. Thus, information hiding enabled us to establish invariants on a modular base.

On the negative side, information hiding also prevents clients from performing sensible operations directly on the data, which would sometimes be simpler and more efficient. Instead of hiding the data and providing procedures to access it, we could also make the data visible to clients and give them rules as to how they may manipulate it. However, such an approach has two disadvantages. First, such rules are semantic and can therefore, unlike hiding, not be enforced by a compiler. Second, changes of internal requirements may invalidate clients. For example, we might decide to introduce a log of all withdrawals. With information hiding, we simply change the procedure *Withdraw*; there is no need to reanalyze or change the clients. On the other hand, if clients can directly modify the balances instead of calling *Withdraw* we must change the rules for these accesses and, therefore, change all clients so that they also update the log. Thus, modularization is also possible without language support. However, in this case the compiler cannot warn us if we inadvertently break our conventions.

There are three typical kinds of modules:

1. A *subroutine library* module contains no data of its own, but exports a collection of procedures. A typical example is a module that exports mathematical functions.

2. An *abstract data structure* module contains hidden data and procedures to manipulate that data. Module *Bank1* (Fig. 3) is an example of an abstract data structure.

3. An *abstract data type* module exports a data type together with associated operations. In contrast to 2, clients can create arbitrary numbers of instances of the abstract data type at run time.

Modules in most languages have two drawbacks. First, traditional modules can only be combined into a system if they are all written in the same language. Second, modules, such as Fortran numerical subroutine libraries, are statically linked together by the developer. For some purposes, this is too restrictive. For example, we might want to use a different spell checker in our word processor, or embed an editable spreadsheet into a text document. This may not be possible if the word processor is a statically linked monolithic program. Component software addresses these problems.

2.2 Components

Software *components* are binary units of independent production, acquisition, and deployment that interact to form a functioning system [56]. Components should have contractually specified interfaces and explicit context dependencies only. Whereas modules state which other modules they import, components state which services, such as a database or mathematical functions, they require. The decision which other component will provide these services is delayed until the components are assembled. Some systems may even contain several components providing the same service. Which component is used to provide which service to which other component may then be statically configurable, or this may be decided when a request is made on the basis of such factors as the current load.

Component software makes it possible for separate teams to develop different parts of large software systems and replace individual software parts that evolve at different speeds. Replacement can be done without having to change or reanalyze other parts. Furthermore, marketing of independently developed building blocks becomes possible.

As in other engineering disciplines, component markets allow customers to combine off-the-shelf components for standard tasks with custom-made components for business-specific requirements. Customers can thus save time and get large parts of their systems at predictable costs. Component manufacturers profit from the market because they can focus on their core competence. For example, a company with special knowledge in checking the spelling of the Finnish language can build a spell checker component for word processors [40]. They do not have to create themselves a full word processor that can compete with Word. Because a given component can be used in different applications, manufactures are likely to sell large quantities.

Figure 4[2] illustrates a component market, as it exists for JavaBeans and ActiveX components (e.g. [18]) and in more restricted forms for plug-ins for standard software, such as Adobe Photoshop, Quark XPress, and Netscape Navigator. A component market has three kinds of players. Vendors produce and sell components. Assemblers

[2]Figures 4 and 17 originally created by Wolfgang Weck. Modified and used with permission.

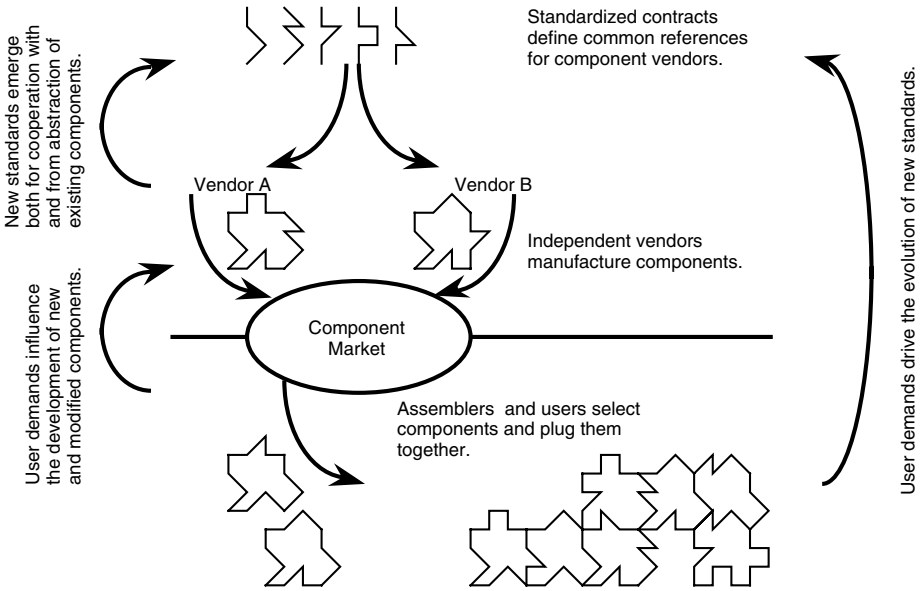


Figure 4: Component market

combine components and possibly some custom 'glue' into applications. Users acquire such applications as well as single components that they themselves insert into their applications. As in other component markets, standards emerge from user demands and successful components.

Component-based development also provides many benefits if the different players depicted in Fig. 4 are within the same company. The engineering of ever growing and changing applications appears to be possible only using loosely coupled components with contractually specified dependencies. Therefore, even large companies that would seem to profit from monolithic software switch to component-based designs [50].

The idea of software components dates back to the NATO conference on software engineering in 1968 [42]. In 1990 Brad Cox even advocated an industrial revolution in the software realm [20], observing that software components are not just a technological issue but a cultural one as well.

To meet the above demands, component-oriented programming requires [56]:

1. *Encapsulation.* Component-oriented programming requires encapsulation and information hiding like modular programming.

2. *Polymorphism.* Polymorphism is the ability to appear in multiple forms. It allows a component to work with any other component implementing a required interface. The most common form is subtype (inclusion) polymorphism, which allows variables to reference at run time instances of subtypes of their declared types. For example, a variable of type *IPrinter* in an ATM could reference a driver object for the specific printer installed.

3. *Late binding and loading.* Late binding and loading allow us to defer deciding which component implementation to use for a certain service to assembly, load, or even run time. Late binding and loading are crucial for independent deployment and for data-driven use of additional components as in compound documents and Web applets.

4. *Safety.* Safety is the property of a component to prevent certain kinds of problems by (preferably) statically excluding certain kinds of errors or by dynamically detecting them as early as possible to avoid failure. Safety is crucial because components are assembled by third parties, rendering integration testing of complete systems by component developers impossible.

5. *Specifications.* Components need to be equipped with concise semantic specifications or with a label stating to which standard specification they adhere. If this is not done, the use of third-party components is difficult and may not be economically profitable.

Components being binary units, communication among components requires binary wiring standards, such as Microsoft's COM [51], Sun's JavaBeans [54], and the OMG's CORBA Components [46]. Components can be created in any language for which a mapping to the desired binary standard exists. However, binary standards are most easily programmed to in languages that directly support the same interface specification and composition mechanisms. We call such languages component-oriented. Furthermore, only language-level support can provide the desired machine checkable safety using types.

2.3 Classes and objects

Components and modules have many points of contact with classes and objects: Components are often programmed with object-oriented languages and have 'object-oriented interfaces'. Furthermore, component-oriented programming is sometimes presented as an evolution of object-oriented programming, with which it is occasionally also confused. Finally, some languages unify modules, classes, and components. It is our aim here to separate the different concepts and show where combining them may be fruitful.

Object-oriented programming is the construction of software systems as structured collections of abstract data type implementations [43]. Simula-67 [21] is generally accredited to have been the first object-oriented programming language. By now, the object-oriented paradigm spans all aspects of software engineering. Paper I shows how object-oriented analysis and design can successfully be combined with formal specifications and implementations in a procedural style.

Figure 5 gives a variant of our banking example in Java. It defines two classes, *Account* and *SavingsAccount*. Classes are templates for the instantiation of objects at run time.

Object-oriented programming languages rest on four pillars:

1. *Encapsulation.* Data and the methods to modify the data are encapsulated in objects. For example, each account contains a number, a pin, and a balance as

well as a withdrawal method. The encapsulated data cannot be accessed directly outside the defining class.

2. *Inheritance.* Inheritance allows classes or objects to be defined as extensions of others. In the example, *SavingsAccount* inherits from *Account*. Inheritance means that *SavingsAccount* has all attributes and methods of *Account*. Additionally, inheritance implies subtyping in most languages. (A subtype is a subcollection of a type.)

3. *Subtype polymorphism.* Subtype polymorphism allows a polymorphic variable to reference objects of any subtype of the variable's static type. For example, a variable *acc* of declared type *Account* could at run time reference an instance of *SavingsAccount*. Subsumption allows us to consider an element of a subtype as an element of a supertype.

4. *Dynamic binding.* Dynamic binding implies that the actual type of a referenced object determines which version of an operation to apply. For example, if we would override *withdraw* in *SavingsAccount* to allow at most 10,000 mk to be withdrawn per month, calling the method on a variable *acc* of declared type *Account* results in the specialized method being executed if *acc* references an instance of *SavingsAccount*.

The relationship between classes and modules varies from language to language. In Oberon-2, modules are receptacles for multiple classes with closer communication [55]. In Java and Eiffel, on the other hand, classes are unified with modules. Most Java and Eiffel classes are abstract data type modules. The main differences to abstract data type implementations in procedural languages like Modula-2 [59] are the support for inheritance, subtype polymorphism, and dynamic binding.

Since 'component-oriented' has replaced 'object-oriented' as the high-tech synonym of good, object-oriented languages have been rebranded as component-oriented.

```
class Account {
  private int number, pin, balance;

  public boolean withdraw(int pin, int amount) {
    if(this.pin==pin && this.balance>=amount) {
      this.balance=this.balance-amount; return true;
    } else {
      return false;
    }
  }
  ...
}

class SavingsAccount extends Account {
  public void addInterest() {...}
  ...
}
```

Figure 5: Classes *Account* and *SavingsAccount*

Whereas these languages can be used to program to binary component standards, they may not provide optimal support and some of their coding practices may contradict the principles of components.

In object-oriented programming we specify classes and directly access objects via variables of class type. In contrast, we specify interfaces and perform all accesses indirectly through them in component-oriented programming.

The component-oriented approach has two advantages. Because clients only talk about interfaces, they are not bound to a specific implementation from one vendor. Second, avoiding code inheritance across component boundaries leads to more robust systems. Code inheritance across component boundaries, as practiced in object-oriented programming, results in an overly tight binding leading to the semantic fragile base class problem [44]. Seemingly valid maintenance changes to a base class break subclasses. Consider the stylized classes *C* and *D*, which both contain two methods for incrementing *x*:

```
class C {
  int x;
  void inc1() {x=x+1;}
  void inc2() {x=x+1;}
}
```

```
class D extends C {

  void inc1() {inc2();}

}
```

If the developer of *C* changes the implementation of *inc2* in a new release to

```
void inc2() {inc1();}
```

then calling either *inc1* or *inc2* of an instance of *D* will result in an infinite mutual recursion. As long as both *C* and *D* are written by the same developer, he or she may spot this bug. However, if the two classes are in components from different developers, then the combination of *D* with the new version of *C* may occur at the customer's site.

Additionally, object-oriented programming languages do not support safe late composition well, as required by component software (Sect. 9).

3 Formal Specifications

Information hiding frees developers of clients from studying the source code of a part (module, class, or component) they want to use. However, developers can only utilize a part if they know what it can do for them. Hence, they need a *specification* of the functionality of the part.

A specification is a contract between the provider and the clients of a part. From the clients' perspective, the contract states how they may use a part and what results they will get. Reciprocally, from the provider's perspective the contract states how the part must perform under which conditions.

The specification of a part must describe the essential aspects of the part's observable behavior. Otherwise it cannot be used as a contract. In this thesis, the *observable behavior* is taken to be limited to functional aspects, that is, the return values of operations and the state of accessible variables. We are not concerned about time or memory requirements. In addition to the sine qua non of describing the essential aspects of

the observable behavior, specifications should not prescribe unnecessary details, they should be simple to read and write, and they should lend themselves to formal reasoning.

3.1 Formal vs informal specifications

Specifications can have different degrees of formality. Plain English specifications are at the informal end of the spectrum. Except for simple cases, they are usually not clear enough. They are subject to interpretation, which in turn depends upon the particular context of the reader. Different interpretations of specifications can then lead to incompatible parts.

The problem is intensified by the fact that specifications should deliberately leave certain aspects undefined. Otherwise, the implementer is too limited in his or her choices. Furthermore, leaving certain aspects undefined increases the chances that future versions will be able to conform to the same specification. This, in turn, implies that clients do not need to be revised to work with new versions of the provided part.

With informal specifications it is often not clear what is intentionally left open and what is just inadequately described. As a result, implementers of clients often 'infer' additional properties by testing. Such assumptions might however be broken in new versions of the provided part.

Informal specifications cannot be used as input to tools, such as formal theorem provers, automatic test case generators, or automatic pre- and postcondition checkers. In particular, informal specifications cannot serve as bases for formal refinement proofs that would guarantee the correctness of implementations. In conclusion, informal specifications are insufficient. Formal specifications are needed.

This said, we would like to emphasize that form should follow function. Formality in specifications is a means, not a goal in itself. In practice it often suffices to formally specify only certain aspects of a system and to describe the rest in less formal notations. Furthermore, formal is not a synonym of cryptic. In many cases a few statements in a normal programming language can be considered as formal.

3.2 Algebraic vs model-based specifications

There are two main styles of formal specifications. In the *algebraic* or property-oriented style [41, 2], operations are described solely in terms of each other by relating their input and output values. The most famous example is the stack specification, which is characterized by axioms like $top(push(s, x)) = x$. The latter says that if we push an element x onto an arbitrary stack s and then look at the top element, we get x. Algebraic specifications are best suited for simple abstract data types. They do not scale well to larger systems.

Model-based specifications operate on a state. Therefore, they are closer to implementations. This similarity has provoked criticism of their not really being specifications, but high-level implementations [34]. In practice, however, model-based specifications scale better and and are closer to most people's mind set. Since we are interested in the specification of larger systems, we have chosen the model-based approach (Papers I, II, III, and VII).

Algebraic and model-based specifications can also be combined. For example, basic datatypes of model-based specification methods are commonly described in the algebraic style.

3.3 Formal methods

Formal methods provide the foundations for formal specifications and help us get the most out of the latter. A method is a systematic procedure, technique, or mode of inquiry for doing something. A method is formal if it is rigid, based on a (mathematical) theory, and offers the possibility for mechanical proofs. The term mechanical means that the proofs and calculations follow a fixed set of rules that can be applied in a machinelike manner and can be mechanized with computers.

Formal methods can provide insight, deliver assurance, and help with debugging. The process of formalization often sheds new light on something and triggers off a closer examination. This insight together with concise characterizations of generally desirable features, such as compositionality and type soundness, can also guide us when exploring new mechanisms.

Customers of safety-critical software may demand some assurance that the delivered product actually satisfies its requirements. Formal proofs provide one of the best assurances. Therefore, the use of formal methods has been recommended by regulatory bodies [10].

Formal methods can also be used for debugging, complementing or replacing other techniques like testing. To this aim, formal documents may be used both as input for proof tools and as basis for peer reviews.

3.4 Tools

Paper-and-pencil proofs tend to be error prone and time consuming. Mechanized proof tools, such as proof obligation generators and theorem provers, address these issues.

Usually more errors are made in paper-and-pencil proofs than in mechanized proofs [28]. Small errors are common in the former, and the generally lower level of formality and handwaving arguments for subproofs further reduce the trustworthiness of manual proofs.

Automation in proof tools can also speed up proofs. In practice, the degree of automation largely depends on the kind of properties to be proved. In the case study of Paper I, 83 % of the 1397 proof obligations were automatically discharged by Atelier B [53]. On the other hand, automation largely failed for the type soundness proofs reported in Papers IV and V. In these two cases the tool-based proofs with Isabelle/HOL [49] took longer than manual proofs would probably have. The higher trustworthiness was the main argument for mechanization in these cases.

When automation fails, theorem provers need to be guided by the user. The user invokes a series of proof tactics to prove theorems. The interactively issued commands can be combined into proof scripts. The latter can then be executed in batch mode. Re-execution of scripts is especially useful after changes to the specification.

Systems usually undergo many small changes, both during the initial development and during maintenance. Script-based proof tools allow us to automatically reprove

theorems. This conveys more confidence than the typical adaptation of a paper-and-pencil proof with 'this-should-still-hold' handwaving. In the latter approach, subtle dependencies that cause certain requirements to be invalidated in the modified system often go undetected.

4 Refinement

In the foregoing passages we discussed how to specify parts and prove that the specifications satisfy certain requirements. We now investigate how to make sure that our implementations are correct with respect to their specifications. This can be done by proving refinement between specifications and implementations. Refinement rules allow the part implementer to prove that the part fulfills the specification contract. Refinement between a specification and an implementation preserves the observable behavior and possibly decreases nondeterminism. A refined part (implementation) satisfies any expectation that the client developer can deduce from the specification.

It is desirable that refinement can be established in a modular way. We only want to consider the specification and the implementation, but not the client modules, when proving refinement. If this is possible, we speak of independent refinement.

Many benefits of modularization are lost if the correctness of a refinement depends on the context. A part can no longer be used by looking just at its specification. The client might invalidate premises of the refinement proof. Refinement of a provided part has to be reproved every time a client is changed.

Specifications and refinement can also improve the development process. We can start with a precise statement of *what* the part should do without being bothered by *how* it is to do this. This way, we are more likely to correctly capture the informal requirements. Furthermore, it is simpler to show that certain requirements hold for a concise specification than for a lengthy implementation. By proving refinement between the specification and the implementation we are guaranteed that the latter also satisfies the requirements. Like modularization, program development by refinement gives a separation of concerns, which reduces the detailed reasoning to an achievable amount.

Going directly from a specification to an implementation might sometimes be too big of a step. Too many decisions would have to be made all at once, rather than proceeding step by step. Validation of each step as it is made ensures that we do not waste time making subsequent decisions on wrong premises. Stepwise refinement allows us to make multiple small steps [58, 24]. Starting from the specification, we add details and replace data structures and algorithms by more efficient ones in multiple steps until we arrive at an implementation. By proving refinement between intermediate steps, we are guaranteed that the final implementation is correct with respect to the original specification. This method is adopted in the case study described in Paper I.

There are different forms of refinement. In this thesis, we use algorithmic, data, and greybox, and trace refinement. The first two are described below. The others are discussed in more detail in later sections.

Algorithmic refinement [3, 8] is defined to preserve correctness between statements on the same state space. An algorithmic refinement step may decrease nondeterminism and widen the termination set.

Data refinement [29, 22] is a technique for changing the encapsulated data structures of a module, class, or component. For example, the arrays of pins and balances in Fig. 3 could be replaced by a single array of account objects or by a linked list if the behavior that clients can observe by means of operation calls remains unchanged. Data refinement is used in Papers I, II, III, and VII.

5 Compositional Sharing of Modules in B

If we split up systems into modules, we need mechanisms to recombine modules to systems. Programming languages typically contain a single mechanism for this purpose, e.g. the *IMPORT* statement in Oberon-2. By importing a provider module *P*, a client module gets the right to invoke exported operations of *P*. If several client modules import the same shared module *P*, all importing modules have the same rights. This may make modular reasoning impossible because client modules may invalidate each others' assumptions about the state of the shared module.

We illustrate the problem in B [1], which is strictly modular in order to enable independent refinement. We also sketch our solution from Paper II. Our approach achieves compositionality with access role specifications expressing rely/guarantee conditions.

5.1 A sample problem

Consider the example of a production plant control system. The simplistic plant consists of a conveyor belt and a monitoring console. Both are controlled by their own software modules. Additionally, a database module is used for storing alarms in the variable *alarms*. The conveyor belt should only be running if there are no alarms. This is expressed in the second line of the invariant of the following B specification:

```
MACHINE ConveyorBelt                  /* B specification */
UTILIZES Database                     /* hypothetical shared import */
VARIABLES running                     /* variable declaration */
INVARIANT running∈BOOL ∧              /* typing of variable */
  (running=TRUE ⇒ alarms=∅)           /* discussed invariant property */
                                      /* variable alarms from imported Database */
...
```

Assume that the monitoring console module provides an operation to set an alarm. If the conveyor belt is running at the point where the alarm is set, adding an alarm to the variable *alarms* of the database invalidates the invariant of *ConveyorBelt*: $alarms=\emptyset$ becomes false, the implication holds no longer, and, therefore, the invariant becomes false.

The above specification of *ConveyorBelt* is not suitable if the console can be used to set alarms. Thus, we consider in the sequel the scenario where the console only allows the operator to deactivate, but not to activate, alarms. By deactivating alarms, the invariant of *ConveyorBelt* cannot be invalidated. We now have a system that can be proved consistent and we investigate how this proof can be performed.

Modular means that we only consider the module in question plus modules imported by it. For a modular consistency proof of *ConveyorBelt*, we, therefore, only

consider *ConveyorBelt* and *Database*. By looking only at these two modules, we cannot prove that the invariant of *ConveyorBelt* always holds. We lack an assurance that *Console*, and possible other clients of *Database*, do not set new alarms. Hence, the modular approach fails. Global proofs are required to assert that operations of *Console* cannot invalidate operations of *ConveyorBelt*. Unfortunately, global non-interference proofs make us loose some benefits of modularity, most notably independent refinement.

The B method puts modularity above the possibility to express the above sharing architecture. For B to be modular, sharing is restricted by the single writer/multiple readers paradigm. Thus, the above architecture where both *ConveyorBelt* and *Console* modify the state of the shared *Database* is not possible in B.

5.2 A modular solution based on rely/guarantee conditions

In Paper II we add to B a compositional shared access mechanism based on rely/guarantee conditions [32]. This mechanism removes the single writer/multiple readers restriction and makes the above architecture possible.

To guarantee interference freedom among multiple accessors of a common machine, only the possible modifications to the shared variables are relevant. We define these effects in the form of access *roles* as part of the shared machine. Accessing constructs declare which role(s) they play. The accessors guarantee to perform only modifications allowed by the declared role(s). In return, they can rely on the other accessors adhering to their roles.

We exemplify our mechanism on the specification of the shared *Database*. The database has a contract *SingleDevice* with two roles *Creator* and *Controller*, intended to capture the accesses by *ConveyorBelt* and *Console* respectively. Both role specifications use the *ANY* specification statement. The *ANY* statement chooses nondeterministically a value for *aa* such that the predicate after *WHERE* holds and then executes its body delimited by *THEN* and *END*.

```
MACHINE Database
CONTRACTS                          /* new clause suggested in Paper II */
  SingleDevice ≙                   /* sharing contract with roles Creator and Controller */
    Creator = ANY aa WHERE aa∈NAT THEN NewAlarm(aa) END,
    Controller = ANY aa WHERE aa∈alarmsTHEN ResetAlarm(aa) END
...
```

The machine *ConveyorBelt* accesses *Database* in the role of the *Creator*:

```
MACHINE ConveyorBelt
ACCESSES                           /* new clause suggested in Paper II */
  Database!SingleDevice AS Creator
VARIABLES running
INVARIANT running∈BOOL ∧ (running=TRUE ⇒ alarms=∅)
...
```

Hence, operations of *ConveyorBelt* are permitted to set new alarms by invoking operation *NewAlarm* of *Database*. On the other hand, the above access declaration does not allow them to reset alarms. Dually, operations of *Console*, which accesses *Database* as *Controller*, can only reset alarms.

This new shared access mechanism lets us prove modularly that the invariant of *ConveyorBelt* cannot be invalidated by other clients of *Database*. If the other accessor role *Controller* preserves the invariant of *ConveyorBelt*, then any other client module, such as *Console*, accessing *Database* in this role will do so.

In addition to this modular proof, we need to globally check that *Database* is accessed at most once in each role. If *Console* also accessed *Database* as *Creator*, then operations of *Console* could set alarms. Single access in each role can be checked automatically.

The proposed sharing mechanism works in the same way in specifications, intermediate refinements, and implementations. The modular approach implies that refinement can be proved independently, i.e., without considering the clients.

Our new mechanism can handle many shared access architectures in a modular way. However, it has its limitations. For example, it would be permissible for an operation of *Console* to set an alarm if the operation also stopped the conveyor belt. A system with such an operation could be proved consistent using global proofs. This is, however, not possible with our approach.

The existing composition mechanisms of B are already very complex. Adding yet another mechanism might seem like a step in the wrong direction. However, discussions with developers of Atelier B [53] and engineers of Matra Transport International, who created with B the automatic train operating system for METEOR [9], the first driverless metro in Paris, convinced us that our proposal addresses a real need. Both parties told that they have encountered real life problems that would have benefited from our mechanism. The existing composition mechanisms forced less ideal monolithic or underspecified solutions upon them.

6 Specification and refinement of external calls

In this section we describe how methods with external calls can be specified and refined. We use statements rather than postconditions in our specifications. Because postconditions might be more common for specification purposes, we start with a general discussion of postconditions vs statements.

6.1 Postconditions vs statements

In the model-based approach, the effect of operations can either be expressed with pre- and postconditions or with statements. For example, the implementation of *withdraw* in Fig. 5 can be considered a specification using statements. For comparison, a semantically equivalent specification over the same state space is given in pre-/postcondition style in Fig. 6. The precondition says in which states and with which parameters the operation may be called. The postcondition expresses in which states and with which return values the operation may terminate —provided the precondition was satisfied.

We use priming to denote the initial state of a variable and the syntactic sugar *if a then b else c end* for $(a \Rightarrow b) \wedge (\neg a \Rightarrow c)$, where we assume a to be total. For uniformity, equality is denoted by '=='.

```
public boolean withdraw(int pin, int amount) {
  pre true
  post
    if this.pin==pin ∧ this.balance>=amount then
      this.balance==this.balance′-amount ∧ result==true
    else
      result==false
    end
```

Figure 6: Pre-/postcondition specification of *withdraw*

In this thesis, we write specifications as combinations of preconditions and statements. The precondition has the same function as in pre-/postcondition specifications. The statement replaces the postcondition. It explicitly describes how the state is transformed and the result is computed.

We use the same kinds of statements in specifications as in imperative programs. To leave certain aspects open and write high-level specifications, we additionally use nondeterministic constructs, such as the *ANY* statement introduced in Sect. 5.

Below we investigate the respective advantages of postconditions and statements. The two approaches do not exclude each other. For example, the new version of VDM [48] supports both postcondition-based implicit operation specifications and statement-based explicit operation specifications. An implicit operation specification is basically the same as an explicit specification the body of which is a single specification statement.

Advantages of postconditions

A seeming advantage of postconditions is that various requirements, possibly coming from different sources, can simply be conjoined with '∧'. However, we can always write such a specification as a single specification statement as well.

VDM [33] and Z [52], which both use postconditions, utilize the simplicity of combining requirements to implicitly conjoin the invariant to every postcondition.

On the other hand, in statement-based methods like B or greybox specifications (Paper III) we must prove that operations preserve the invariant if called with values satisfying their precondition. We consider these explicit consistency proofs to be a benefit, rather than a burden. Intuitive and informal specifications capture what an operation should do, regardless of whether the operation might thereby invalidate the invariant. Statement specifications let us express this directly. The consistency proof shows whether this behavior is actually legal. Failure to prove consistency may point to an overly strong invariant. Thus redundancy helps us find specification errors. On the other hand, the implicit conjunction of the invariant to the postcondition in VDM and Z can lead to unintentionally restrictive specifications. This may not be noticed until much later, at a stage when changing the specification may invalidate much work already based on it.

The choice of statements vs postconditions influences whether the invariant is implicitly conjoined or not. VDM illustrates this by conjoining the invariant to the post-

condition of implicit operation specifications and by not conjoining it to statement-based explicit operation specifications. However, the opposite is also possible. The invariant was not implicitly conjoined to the postcondition in earlier versions of VDM [31]. Dunne proposed to implicitly conjoin the invariant to statement-based B specifications [25].

A second advantage of postconditions is that they are declarative, rather than operational, and, therefore, 'truer' specifications than statements. This is, however, a matter of style and taste only.

Advantages of statements

Statement-based specifications have several advantages over postconditions [13, 12]. First, statements allow specifications to be split into multiple sequential steps. Second, they scale better because they may contain calls to other operations. With postconditions, we must either repeat the effect of the call or introduce additional named predicates. Repetition, e.g., duplicating the instantiated postcondition of the called method in the postcondition of *withdraw*, makes postconditions lengthy and creates maintenance problems. Additional predicates complicate the specifications and may require complicated plumbing for use in different scopes. Third, statements give us a uniform notation for specifications and implementations.

As described in Paper III, the main advantage of statements is that they can be used to specify call-backs without complicated encodings. Statement-based specifications can contain calls to other methods. This is discussed below.

6.2 Statement-based specifications of call-backs

Traditional modular systems, such as the bank in Paper I, are layered, and calls only occur from higher to lower layers. In this case, it is sufficient to specify the state changes and return values of methods. There is no need to specify external calls.

If, on the other hand, we also have calls from lower to higher layers, or if our architecture is not layered, just specifying the resulting state transformations of calls might be insufficient. Up-calls from lower to higher levels or more generally call-backs are possible by passing a reference to a procedure or an object with methods. They are common in object- and component-based systems.

A representative application of call-backs is the observer pattern [26]. It allows software components that need to react to certain events, such as particular state changes, to register an observer object with the observed object. The observed object then calls a notify method from each registered observer object upon the occurrence of the respective events.

We extend our banking example to illustrate the observer pattern. The legal department of a bank might like to be informed of transactions on some dubious accounts. When class *Account* is specified and implemented, this and other external methods that should be performed on a withdrawal may not be known. The observer pattern provides a solution: Observer objects can be registered with accounts and they are notified of each transaction.

```
public boolean withdraw(int pin, int amount) {
  if(this.pin==pin && this.balance>=amount) {
    this.balance=this.balance-amount; return true;
    do(int i in 0..this.nofObservers-1)
      registeredObservers[i].withdrawNotification(this, amount);
  } else {
    return false;
  }
}
```

Figure 7: Greybox specification of *withdraw* with notifications

Let *registeredObservers* be an array of references denoting the registered observers for a given account. Then *withdraw* may be specified with statements as in Fig. 7. A *do* loop is used to leave the order, in which the observers are notified, unspecified. The *do* loop is executed once for *i* bound to every value between *0* and *this.nofObservers-1*.

The specification expresses that all observers must be notified. Furthermore, it states that this must happen after the amount has been subtracted from the balance. This is important because an observer might make a call-back to the account object to inquire the current balance. The specification says that this call will return the new rather than the old balance.

When we specify *withdraw*, we do not know how the different observers will react to withdrawal notifications. Thus the specification of *IAccountObserver* (Fig. 8) only states that an account observer has a method *withdrawNotification*.

With postconditions we cannot express calls as such, but only their effects on the state. Since we do not know what the latter are on the unknown state spaces of the observers, we cannot specify them. Hence, the withdrawal notifications cannot be captured with postconditions.

In conclusion, statements let us specify external calls using normal programming language syntax. With postconditions, on the other hand, we cannot specify that calls should be made if the effect of those calls is not known —at least not without complicated encodings. We discuss the specification of external calls with statements, named greybox specifications, in detail in Paper III.

6.3 Greybox refinement

Greybox specifications describe three aspects of a method's behavior: the state transformation, the return value, and the external calls to be made. Data refinement only

```
public interface IAccountObserver {
  void WithdrawNotification(int amount) {
    skip;
  }
}
```

Figure 8: Specification of *IAccountObserver*

```
public boolean withdraw(int pin, int amount) {
  if(this.pin==pin && this.balance>=amount) {
    this.balance=this.balance-amount; return true;
    for(int i=0; i<this.nofObservers; i++)
      registeredObservers[i].withdrawNotification(this, amount);
  } else {
    return false;
  }
}
```

Figure 9: Implementation of *withdraw* with notifications

preserves the first two aspects and is, therefore, insufficient to assure that an implementation satisfies all assumptions that a client may legally deduce from a greybox specification. To fill the gap, we introduce in Paper III the notion of greybox refinement, which preserves all three aspects.

The basic conditions of greybox refinement are as follows: The implementation of a method must make the same sequence of calls to other component instances in the same respective states as the specification, perform the same overall changes to the local state, and return the same value. If the specification is nondeterministic, then the call sequence, local state transformation, and return value of the implementation must correspond to one choice in the specification.

In Paper III we formalize greybox refinement using a trace semantics that records both the state and external method calls. We also show how to prove simple cases with piecewise refinement in context.

Figure 9 gives an implementation of the greybox specification of Fig. 7. The difference is that we have made the order of notifications deterministic by replacing the *do* with a *for* loop. Greybox refinement would also allow us to replace the state space of the part. For example, we could replace the array *registeredObservers* by a *TreeSet*.

The key advantage of greybox refinement over data refinement is that it guarantees a component's properties, that are not described in the common specification with other components, to be preserved when assembled to a system. For example, the specification of *IAccountObserver* only states that account observers have a method *withdrawNotification*, which can be called by accounts. The semantics of this method (on the state space in the scope of the specification) is skip. An instance of a class implementing *IAccountObserver* will, however, do something concrete, such as sending an e-mail to the person monitoring the account.

If the implementation of *Account* is a greybox refinement of its specification, then the *withdraw* method will notify the observer object. Thus, the e-mail gets sent as intended by the implementer of the account observer class.

If, on the other hand, we were to establish only data refinement for the *Account* implementation, then the e-mail might not be sent. The reason for this is that the specification of *withdrawNotification* is skip. Not doing anything is a legal data refinement of a call to a skip method. Hence, with data refinement an implementation of *withdraw* is not forced to notify the observers.

7 Types

Formal semantic specifications together with refinement provide a way for constructing dependable software. Types can enhance this method in several ways:

- Types can serve as labels guaranteeing compliance with standards. When composing components, compatibility can be asserted by a simple comparison of labels. This is especially useful when third parties, possibly users at run-time, combine components. Requiring the user to perform an interactive semantic proof when pasting a part into a compound document would not be practical.
- Types can help to document interfaces. The meaning of a parameter can often be inferred from its name and type.
- Compilers can automatically flag all errors of certain kinds by type checking a program. Thereby certain kinds of errors can be caught before starting more difficult and time-consuming semantic proofs, which usually have to be redone each time after an error has been located and fixed. Type checking can be fully automated because type systems are generally decidable.
- Types can be used at run time to check the validity of operations that could not be proved type correct with the information available at compile time.
- The actual type of an expression can be determined at run time and the behavior of the program can be based on this information.

Types are a powerful instrument for creating semantically consistent and correct programs —whether semantic proofs are performed or not.

7.1 Behavioral typing

Explicitly declared and named types can stand for external semantic contracts. For example, the actual parameters of an operation should satisfy a precondition, which in the object-oriented paradigm may include having associated methods with certain semantics. Consider the method *transferTo* (Fig. 10), which may be added to the class *Account* (Fig. 5). The actual parameter for *to* should be an account with a method *deposit*, adhering to a corresponding specification.

```
public boolean transferTo(Account to, int fromPin, int amount) {
  if(this.withdraw(fromPin, amount)) {
    to.deposit(amount); return true;
  } else {
    return false;
  }
}
```

Figure 10: Method *TransferTo*

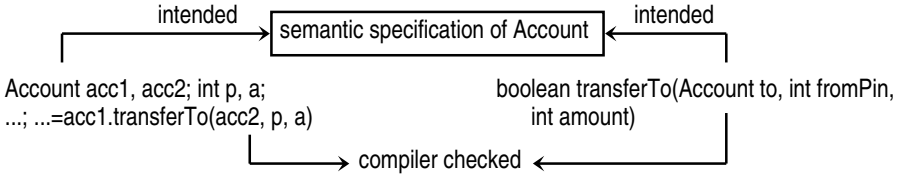


Figure 11: Behavioral typing with compiler checked references to the same standard

Such semantic conditions are in general undecidable and neither automatic compile-time nor run-time checking of them is feasible with state-of-the-art technology. On the other hand, type systems of standard programming languages are —for most practical cases efficiently— decidable. Compilers can check most conformances statically and controlled type casts and type tests can be performed at run time.

Thus, instead of semantic contracts, explicitly declared and named types that stand for the former can be used in programming languages. The behavioral specification is documented separately and linked to the type via the name. Compilers cannot automatically check the compliance with such specifications, but they can verify that references to the same types, and intentionally to the same contracts, have been made (Fig. 11).

Behavioral subtyping

Subsumption lets us consider an instance of a subtype as an instance of a supertype, e.g., a *SavingsAccount* as an *Account* (Fig. 5). Subsumption may not give the desired semantic effects unless the behavior of the subtype instance corresponds to that of a supertype instance. For example, overriding the method *withdraw* (Fig. 5) for *SavingsAccount* so that it adds the amount to the balance would quickly ruin the bank. The semantic conformance of subclasses to their superclasses termed *behavioral subtyping* or *class refinement* is determined by data or greybox refinement of classes.

Standard compilers can use types to check that references to the same standards are made. By declaring *SavingsAccount* as a subclass of *Account*, we state that instances of the former behave like instances of the latter. Thus, the type system and, therefore, the compiler and run-time system can allow variables of declared type *Account* to reference instances of *SavingsAccount* (Fig. 12). At the same time, the type system forbids a variable of type *Account* to reference an instance of any other class that has not been declared to be a subtype of *Account*.

Together with semantic proofs, types are useful because they can be used to automatically detect certain errors before starting costly semantic proofs. If we do not perform the semantic refinement proof, types give us at least some safety.

Standards in other domains: an analogy

Consider an analogy to the well-established component market for mass storage with its interface standards such as SCSI. The manufacturers of host adapters and hard disks verify that their products adhere to the standard. A customer does not have to know the

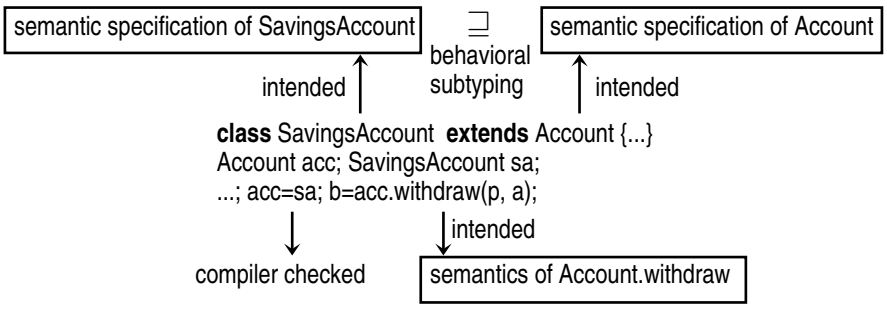


Figure 12: Behavioral subtyping with compiler-checked references to the same standard

SCSI specification. He or she must only check that it says SCSI on both the host adapter and the disk before he or she plugs them together. Likewise component assemblers and users should be able to plug together software components labeled with the same type. Assemblers and users should not be required to study the definitions of any standards and do compliance checking.

The analogy can even be extended to subtyping. Fast SCSI drives can transfer data at twice the speed when used with a Fast SCSI host adapter. Yet, they can also be used like normal SCSI drives on normal SCSI adapters. In analogy, a *SavingsAccount* should be usable like an *Account*. Extended functionality should be exploitable by other components aware of it.

Different forms of behavioral subtyping are discussed in Papers III and VII and used in Papers IV and V.

Behavioral subtyping and encapsulation

Behavioral subtyping is a useful tool for constructing programs. However, proofs of behavioral subtyping require a breach of encapsulation if the superclass contains executable implementations. Behavioral subtyping cannot be proved with the specification of the superclass alone, the latter's implementation is required.

To illustrate this, assume that *sqrt* (Fig. 13) is part of the specification of class *C*. Assume further that the implementation of *sqrt* in *C* actually has a precision of 1 %, rather the 2 % required by the specification. Another developer B may create a subclass *D* of class *C*. Method *sqrt* is overridden in *D* to have a precision of 1.5 %. Because *D* computes the square root with higher precision than the specification of *C*, developer

```
float sqrt(float x) pre x>=1 {
  any(float y: y>=0 && 0.98*y*y<x && x<1.02*y*y) return y;
}
```

Figure 13: Nondeterministic specification of a square root method

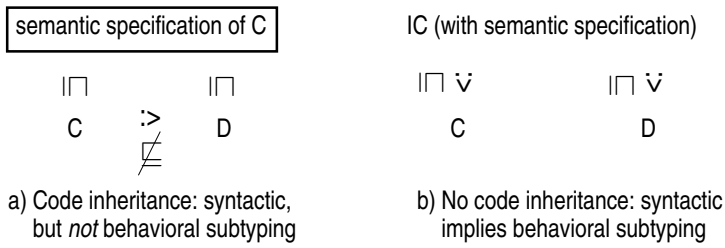


Figure 14: Syntactic and behavioral subtyping with and without code inheritance

B might want to conclude that *D* is a behavioral subtype of *C*. However, this is not true because the implementation *C* calculates the square root with an even higher precision. Figure 14 (a) illustrates the relationships between the specification of *C*, *C*, and *D*. Class *D* being a syntactic subtype of class *C* is denoted by '$C :> D$' and its not being a behavioral subtype is denoted by '$C \not\sqsupseteq D$'.

Developer B would need the implementation of *C* to prove subtyping. However, the latter might not be available to B. Furthermore, forcing B to look at the implementation defeats the purpose of abstraction and information hiding. Even if *D* would refine the current implementation of *C*, a future version thereof that still adheres to its specification might again break refinement by, e.g., also working for input values between 0 and 1. In conclusion, if implementations may refine their specifications, such specifications suffice as contracts for clients, but not for subclasses.

This problem is not restricted to inheritance, rather it extends to any form of subtyping of classes with executable code. If we can override methods, the overriding methods may only refine the specification but not the implementation of the overridden method. If we can add new methods, the additional methods might extend the absolute or relative set of reachable states of the supertype implementation.

Behavioral subtyping is not a goal in itself, but a means of creating semantically correct programs. Hence, we should consider the consequences of *D* being only a behavioral subtype of the specification of *C*, but not of *C*. Clients that do not depend on the implementation of *C* being more deterministic than its specification function properly when using an instance of *D* instead of one of *C*. From a practical point of view, the problem is, therefore, that programmers who are aware of *C* being more precise must not rely on this. Most notably, for self calls in *C*, the developer of *C* must not assume his or her own, more deterministic implementation to be executed.

Although it is possible to write programs in such a way that *D* not being a behavioral subtype of *C* does not cause any harm, we believe that it is better not to rely on the adherence to such coding practice and semantic proofs thereof. It is advisable to avoid code inheritance across component boundaries instead. A safer solution is to introduce an interface type *IC* and attach the semantic specification to this type (Fig. 14 (b)). Paper III advocates such a model and Paper V gives coding conventions to avoid the pitfalls when behavioral subtyping of classes with code is deemed appropriate.

7.2 Type safety

Types provide a means for expressing the set of legal values of a variable. For example by typing a variable *x* as *Account*, we state that *x* may reference at run time an instance of *Account* or a subtype thereof and that the value of *x* may also be *null*. The variable *x* may not have any other value. A programming language must enforce this restriction. For example, it must prevent us from assigning *3* to *x*. Otherwise, types degenerate to mere notes of intention, rather than trustworthy assurances.

A programming language that enforces type restrictions is type sound. Type soundness means that all values produced during any program execution respect their static types. This in turn guarantees that method lookup always succeeds and that no memory corruption can occur through the dereferencing of invalid pointers.

Not all programming languages are type sound. For example C++ allows us to assign the value *3* to variable *x* of type *Account*. The worst part is that this error might go unnoticed for a while and later on cause arbitrary behavior, the source of which is difficult to locate. In the example, method lookup may fail or return an invalid address when trying to invoke a virtual method on *x* and assigning to an instance variable of the 'object' referenced by *x* may lead to memory corruption.

Correct programs can be written in languages that are not type sound. However, it takes a bigger effort and errors in the final product are more likely. Consider an analogy. A carpenter can use either a hammer or a stone to hammer the nails into a dresser. Using the hammer is likely to be easier, more efficient, and less prone to do damage to the wood.

Type soundness is not a trivial property, especially for polymorphic languages [11, 14]. It came to prominence with the discovery that older versions of Eiffel were not type sound [19, 43].

Formal proofs of type soundness, as reported in Papers IV and V, are complex and time consuming. Because they only have to be performed once by the language designer for all the language users to enjoy the benefits, the effort is worth it.

Type safety: a weaker notion used on the binary level

In practice, a weaker notion of type soundness, called type safety, is sometimes favored on the binary level for economical and compatibility reasons. For example, the Java language is type sound and Java compilers must enforce type soundness [27, 57]. However, the verifier of the Java virtual machine [39] is not forced to check or reestablish type soundness of the byte code. The virtual machine only guarantees to protect the user from the consequences of type unsound operations. Exceptions, which can be handled in a controlled manner, are guaranteed to be thrown before method lookup failures and memory corruption could occur. A type unsound operation without such consequences may go unnoticed.

The advantages of this approach are efficiency and improved compatibility. The verifier is simpler and faster. Classes that have undergone incompatible changes may still be loaded together and may work as intended.

The disadvantage is that a type soundness violation may be signaled much later, possibly when the class that caused it isn't even loaded anymore. Thus, it becomes

more difficult to locate the source of the error. Furthermore, it is not sufficient that the type unsound statement is dynamically enclosed by a suitable *catch* clause. The statement that finally causes the virtual machine to throw the exception must be enclosed by a suitable *catch* clause.

It is important to remember that Java only uses the weaker notion of type safety on the binary level. Thus, this example does not go against our belief that strict type soundness should be striven for on the programming language level.

8 Precise typing

To be decidable and safe, type systems have to restrict flexibility. Thereby, type systems are bound to also forbid some sensible combinations. In this section, we illustrate a situation that cannot be properly handled by the type systems of Java and most other languages. We also sketch our solution (Paper IV), which takes a step toward the reconciliation of flexibility and safety.

8.1 A sample problem

We assume two standards that have come into existence independently of each other. The first standard defines an interface *IText*, describing operations, such as insertion and deletion of characters. In particular, *IText* contains a transformation function *displayPoint*, which converts text positions to pixel positions. The second standard defines a compound document framework, like OLE [16], including an interface *IContainer* to be implemented by all classes whose instances may act as compound document containers. The latter must support insertion and removal of document parts. Figure 15 shows portions of these two interfaces in Java.

```
/* as part of a text framework: */
  public interface IText {
    java.awt.Point displayPoint (int textPos);
    /* returns the display position at which the character at textPos is drawn */
    ...
  };

/* as part of a compound document framework: */
  public interface IContainer {
    void insertPart (DocPart part, java.awt.Point xyPos);
    ...
  };
```

Figure 15: The standard interfaces *IText* and *IContainer*

Both standards form individually useful frameworks. Vendors can build components for either of them. The problem emerges with the wish to create components that build on both standards simultaneously. In our example, this would be components that deal with both texts and containers. Classes *ContainerTextA* and *TextContainerB* from vendors A and B may implement both interfaces:

```
public class ContainerTextA implements IContainer, IText {...};
public class TextContainerB implements IText, IContainer {...};
```

These classes exhibit a little nuisance. To insert a document part, one has to pass the graphical coordinates because the container interface must be used. One may prefer to give a text position and have the part inserted after the corresponding character. For this purpose, a generic service can be implemented that maps a text position to the corresponding display position and then inserts the document part there. We assume that a vendor C wants to offer this service within a class *LibraryServices*. Figure 16 shows part of this class and Fig. 17 illustrates the whole scenario.

```
public class LibraryServices {
  public static void insertDocPart(DocPart part, ? into, int textPos) {
    /* the question mark stands for a type saying that interfaces IText
    and IContainer must be implemented */
    into.insertPart(part, into.displayPoint(textPos));
  }
};
```

Figure 16: Vendor C's library services

Vendor C's library service works for instances of classes that implement both interfaces *IText* and *IContainer*. Unfortunately, this cannot be expressed by the type of parameter *into*, as the question mark in Fig. 16 indicates.

The obvious solution is to create a combined interface *ITextContainer*, which extends both *IText* and *IContainer* and does not add or hide anything, and to declare parameter *into* of this type. However, with this solution instances of *ContainerTextA* and *TextContainerB* are not compatible with the library service, because the two classes are declared to implement only the base interfaces but not the combined interface *ITextContainer*. The problem is who is to define the interface *ITextContainer* to be used by all parties. It is not part of either of the two frameworks because they are assumed to be independent. If one of the vendors A, B, or C defines *ITextContainer*, the others would be obliged to use this definition. This contradicts the mutual unawareness postulated for component software vendors. On the other hand, if all vendors declare their own combined interfaces, they are not compatible either.

The alternative is to type parameter *into* as *IText*, check in the body of the method that the actual parameter is also of type *IContainer*, and perform the corresponding cast. The problem with this solution is that we loose static type safety. The compiler cannot warn us if we call *insertDocPart* with an actual parameter that might only implement *IText*, but not *IContainer*.

In summary, the type systems of Java and most other languages do not allow us to precisely express the requested type for parameter *into*.

8.2 Compound types to the rescue

In Paper IV, we introduce compound types, a novel typing mechanism that allows us to state directly that a parameter must implement a set of interfaces, such as *IContainer* and *IText*.

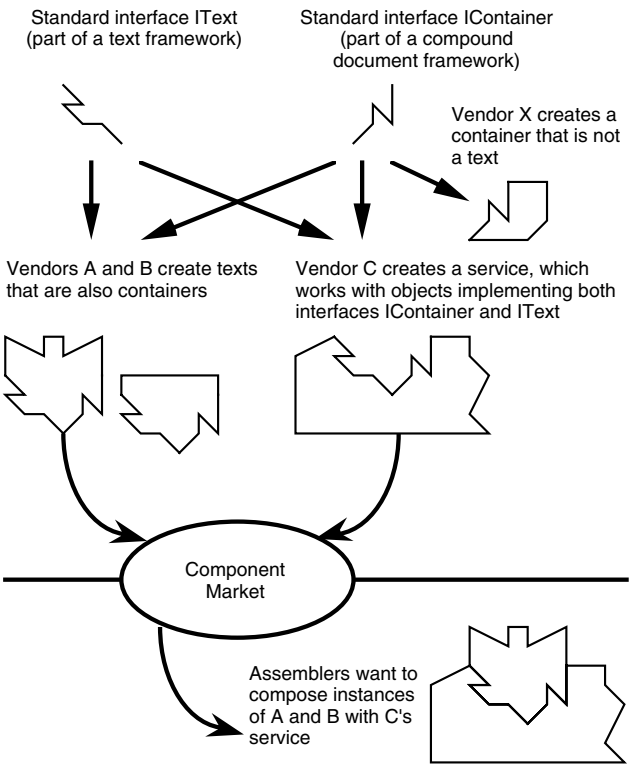


Figure 17: Independent development of classes and insertion service

Compound types are anonymous reference types. They are direct extensions of a set of interface types, referred to as the constituent types. The subtypes of a compound type *S* are exactly those types that are declared to extend or implement all constituent types of *S*. We write compound types as comma separated lists delimited by square brackets.

We can use a compound type to precisely type the parameter *into* of Fig. 16. The compound type *[IContainer, IText]* expresses exactly what values we expect for *into*. With this declaration, we may pass as actual parameter for *into* a reference to an instance of a class declared to implement both *IContainer* and *IText* or the value *null*. The latter case is —as often— undesirable. However, it cannot be excluded without making the type system undecidable or introducing types of non-*null* references.

Most strongly typed object-oriented programming languages would profit from the addition of compound types. In Paper IV, we add compound types as a conservative extension to Java. We have proved type soundness of the resulting type system with Isabelle/HOL.

8.3 Purely structural equivalence for types

Some languages avoid the above problem by using purely *structural* rather than *name* equivalence for types. Subtyping is not based on the declared relationships. Instead, a type T is a subtype of another type S if T contains at least all the methods and public fields contained in S. With structural subtyping, *ContainerTextA* would be a subtype of the combination interface *ITextContainer*.

The problem with purely structural subtyping is that any other type that happens to have the same members is also a subtype —even if it was not intended to be one and has a different behavior. Consider again the above analogy to mass storage interface standards (Sect. 7.1). Purely structural typing would mean that we would plug any other device with a mechanically compatible connector into a SCSI host adapter. It just happens that parallel printer cables fit some SCSI connectors —but the outcome may be disastrous.

Accidental matches of complex interfaces are rare. But many frameworks contain very simple, often even empty base interfaces. For these interfaces, accidental matches are likely.

In conclusion, purely structural subtyping does not support behavioral typing. Our proposal combines the safety of name equivalence with the flexibility of structural equivalence for an important kind of typing problems in component software.

9 Composition and reuse of components

Inheritance and object composition, the two main reuse mechanisms in object-oriented programming, do not fully meet the requirements of component software. Inheritance falls short for three reasons. First, inheritance fixes the superclass at compile time. Thus, it does not satisfy the late binding requirement of component software (Sect. 2.2). Second, inheritance across component boundaries is not safe, as illustrated by the fragile base class problem (Sect. 2.3). Third, it requires a breach of encapsulation to prove behavioral subtyping (Sect. 7.1). This is intolerable for components from different vendors.

Object composition, that is one object holding a reference to another object, is the second reuse mechanism in object-oriented programming. References can be assigned at run time, thereby meeting the late binding requirement. Furthermore, object composition gives a less fragile coupling: Changes to a class are less likely to invalidate classes that contain references to it than subclasses. However, the relationship between an object and another object referenced by the first is rather loose —too lose for certain applications, as illustrated below.

Below we show with an example why neither inheritance nor object composition fully satisfies the needs of component software. We then explain how generic wrappers, introduced in Paper V, solve the problem.

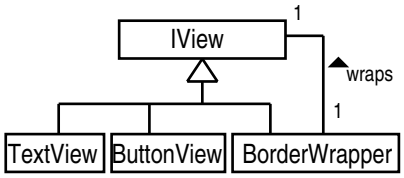


Figure 18: Class diagram of the decorator pattern

9.1 A sample problem

User interface views such as text views may have borders or scroll bars. One way to implement text views with borders is to create a subclass of plain text views. Supposing that there are different kinds of borders, this requires that a subclass for each kind of border is created. In most languages, subclasses can only be defined at compile-time by developers, but not at run time by users. In a component scenario, it may, however, be the user who would like to combine a border and a view from mutually unaware vendors. Even in a closed environment, creating subclasses for all kinds of views with all kinds of borders and scroll bars is problematic because it leads to a combinatorial explosion of classes. Thus, inheritance is not suited for adding borders to views.

Object-composition yields a more flexible solution to this problem [26, Decorator Pattern]. The border is represented by a separate object. It holds a reference to an embedded view, to which it forwards most calls. The border is itself a view. Hence, its presence is more or less transparent to the clients of the embedded view.

Unfortunately, the presence of the border is not fully transparent to the clients of the embedded view. Let *IView* be the base type of all views, such as *TextView* and *ButtonView* (Fig. 18). The *BorderWrapper* class subtypes *IView* and also holds a reference of type *IView*. The problem is that when a border instance decorates a *TextView*, it hides the specific functionality of the latter from its clients because the border is not a *TextView*. This is illustrated in Fig. 19: The type of a reference to a *BorderWrapper* around a *TextView* is not of type *TextView*. The members of *TextView* cannot be accessed directly through such a reference. Furthermore the border cannot be inserted

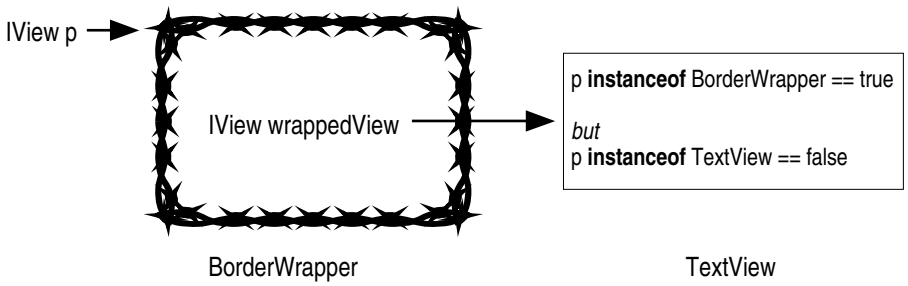


Figure 19: The decorator is not fully transparent to clients of the embedded view

into a list of *TextView*s. The root of these problems is that the type of a *BorderWrapper* does not depend on the actual type of the embedded view. Thus, object composition does not solve the problem of adding borders to views either.

9.2 Generic wrappers as the solution

In Paper V we introduce generic wrappers, a special object-composition mechanism that turns a border wrapping a *TextView* into an element of the latter. Generic wrappers are classes that are declared to wrap instances of a given reference type (class, interface) or of a subtype thereof. Similar to an *extends* clause to specify a super class, we use a *wraps* clause to state the lower type bound for the wrapped object. This also declares the wrapper class to be a subtype of the static type bound. For example, the declaration

```
class GBorderWrapper wraps IView {...}
```

declares the class *GBorderWrapper* to wrap an instance of a class that implements *IView*. When creating a wrapper instance, a reference to the object to be wrapped is passed to the constructor as a special argument delimited by '<>'. For example, the following statement wraps a *TextView* in a *GBorderWrapper*:

```
TextView t = new TextView(...); IView v = new GBorderWrapper<t>(...);
```

The particularity of generic wrappers is that their instances are also of the type of the wrapped object. The *GBorderWrapper* wrapping a *TextView* is of a subtype of both *GBorderWrapper* and *TextView*. Hence, such an aggregate can be assigned to a variable of type *TextView* and the latter's methods can be called on it.

9.3 Design space for generic wrappers

In Paper V we analyze the design space for generic wrappers. One design option is the choice between forwarding and delegation. The difference between forwarding and delegation is the binding of the self parameter in methods of the wrapped object when called through the wrapper. With delegation, the self parameter is bound to the wrapper, and with forwarding it is bound to the wrapped object. Figure 20 illustrates the difference with a client calling method *m* of the wrapped object on a reference to the wrapper. Inheritance is a form of delegation where the wrapper and the wrapped object are merged into one and the binding is fixed at compile time [38].

The advantage of delegation over forwarding is that the wrapper can better modify and customize the behavior of the wrapped object because delegation provides for the specialization of self calls. The main advantage of forwarding is that it eases modular reasoning: As illustrated in Fig. 20, delegation can lead to *up-calls* from the wrapped object to the wrapper. With forwarding, on the other hand, control stays within the wrapped object once a call has been forwarded to it. The wrapper cannot interfere with the flow of control inside the wrapped object [56]. Thus, forwarding gives a looser coupling that does not suffer from the semantic fragile base class problem (Sect. 2.3). This is especially important for component software because the wrapper and the wrapped object may be developed independently and composed at run time.

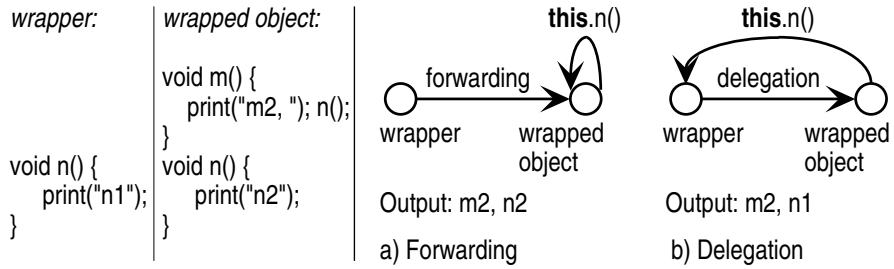


Figure 20: Forwarding vs delegation

Furthermore, forwarding allows programmers to assume self calls to invoke their own implementations. Thus, forwarding eases the problem of syntactic subtypes being behavioral subtypes of their supertypes' specifications, but not of the latter's implementations (Sect. 7.1).

9.4 Generic wrappers in Java

As a proof of concept, we add generic wrappers to Java in Paper V. Our exemplary generic wrappers satisfy the safety and modular reasoning requirements of component software because they use forwarding rather than delegation, fix the wrapped object to the wrapper, and make optimal use of static typing and run-time type information.

We have proved type soundness of the resulting type system with Isabelle/HOL.

9.5 Generic wrappers and compound types

Generic wrappers fit well together with compound types. Compound types let us specify that a parameter must be a text with a label, i.e. of type *[ILabel, IText]*. For example, the method *signText(Key privateKey, [ILabel, IText] idText)* could set the label to the signature of the text calculated with *privateKey* (Fig. 21). Let *LabelWrapper* implement *ILabel* and wrap *IView* and let *TextView* implement *IText*. An instance of *LabelWrapper* wrapping a *TextView* could be passed as second argument to method *signText*.

The type *[ILabel, IText]* precisely expresses the requirements for parameter *idText*. Our generic wrapper *LabelWrapper* wrapping a *Text* fulfills these requirements. With

```
signText(Key privateKey, [ILabel, IText] IdText) {...}

class LabelWrapper implements ILabel wraps IText {...}

Text t; ...;
signText(k, ([ILabel, IText]) new LabelWrapper<t>)
```

Figure 21: Summary of example definitions

normal object composition, on the other hand, it would be impossible to meet the requirements. References to the wrapper are only of type *Label* and references to the wrapped object only of type *Text*. Thus, normal object composition would force us to give up precise typing to make things compatible.

In conclusion, generic wrappers fulfill precise type requirements, even if part of the requirements is met by the wrapper and part by the wrapped object.

9.6 Prototype-based languages

Mechanisms similar to generic wrapping already exist in so-called prototype-based languages. In these languages, class inheritance is replaced by object composition and delegation to parent objects. However, delegation in prototype-based languages does not satisfy the safety requirement of components when used between objects from different components. First, the coupling is too tight leading to difficulties comparable to the fragile base class problem. Second, the parents of a given object can be replaced by other objects, which makes modular reasoning very difficult. Third, static type systems, which in strongly-typed class-based languages are used to detect many errors at the developer's site, do not work well with prototype-based languages. Thus, delegation in prototype-based languages does not satisfy the safety requirements of component software.

10 Concurrency

The second part of this thesis and introduction deals with concurrency and language support for the latter. The themes of specification, refinement, object orientation, and separation of concerns are revisited in the concurrent context.

The physical world is concurrent, in the sense that many things happen at the same time. Computer systems modeling some aspect of the physical world must, therefore, deal with concurrency. Examples of concurrent programs include airline reservation systems and operating systems for multiuser computers. Furthermore, problems without inherent concurrency requirements are often parallelized so that they can be solved in a shorter time using several processors. Finally, reactive systems, which respond to external events and whose input and output are passed during execution, are often most easily understood with concurrent models.

If many tasks of a common problem are performed concurrently, a need naturally arises for coordination and communication between them. At the same time, the tasks should not interfere with each others' work or infinitely wait for each other (deadlock).

When we design a concurrent system, we want to be able to focus on the actual requirements and not be bothered by deadlocks and other artificial problems. For this purpose, we need a suitable conceptual model of concurrency. The spectrum of concurrent programs is so wide that there is no universal panacea. However, we will below try to argue why our model, object-oriented action systems, is very well suited for certain problems.

10.1 Models for concurrency

In this section, we analyze the main design dimensions for a model of concurrency.

True concurrency vs interleaving If two processors execute parts of the same program, they are working *truly concurrently*. Models of true concurrency give the most realistic representations of physical processes and allow for accurate notions of fairness [5]. However, models of true concurrency tend to be very complex.

If, on the other hand, we are only concerned with the functional, untimed correctness of a program, we can use the more abstract *interleaving* model of concurrency [17]. In an interleaving semantics, we think of concurrent atomic transactions to be executed in an arbitrary sequential order. For example, if processor 1 executes S and processor 2 executes T, then we consider the sequential executions $S; T$ and $T; S$ in our reasoning, rather than the parallel $S \parallel T$.

The choice between true concurrency and interleaving illustrates a general phenomenon: A higher abstraction level simplifies reasoning on properties captured by the model. Conversely, certain reasoning cannot be performed in such a model because it does not truthfully represent the necessary details.

Processes vs actions The notion of sequential control flow is pervasive in computing. Turing machines and von Neuman computers are examples of sequential devices. Problem decomposition on the sequencing of tasks is often useful. However, some problems are more easily understood through abstractions unrelated to control flow [15].

A specifications should capture the essence of a problem. If the flow of control is not determined by the problem, the specification thereof should not fix it either. Restricting the control flow when refining a specification for a specific target architecture is simple and will almost trivially preserve correctness. On the other hand, removing or changing control flow restrictions is difficult and often does not preserve correctness.

Most models of concurrency are based on interacting *parallel processes* [23, 45, 30]. To avoid interference and to enable per-process reasoning, various constructs such as semaphores and monitors are introduced. Even with these aids, non-operational reasoning on parallel processes remains difficult and certain properties, such as deadlock freedom, can often only be established globally.

With processes, it may be difficult to get a picture of the overall behavior of a system due to the many possible interactions. This is also reflected in the cumbersome and intricate proof rules for process-based models of concurrency [5].

Instead of adding concurrency to sequential processes, we can make concurrency the foundation of our model. Nondeterministically selected and atomically executed *actions* avoid fixing the flow of control at a high level [35, 4]. With actions, it is usually much simpler to get a picture of the overall behavior. Furthermore, many deadlock and synchronization problems of the process model do not exist in action-based models.

Communication There are three basic models for communication in concurrent systems:

1. In the *shared state* model, the different processes or actions communicate via shared variables.

2. In the *asynchronous message passing* model, processes or actions send messages either to channels or directly to other processes or actions.

3. In the *synchronized event* model two or more processes synchronize and exchange information on a handshake or rendez-vous.

Branching vs linear time The difference between branching-time and linear-time models lies in their treatment of the choices that systems face during their execution. *Linear-time* models describe concurrent systems in terms of the sets of their possible sequences of states. On the other hand, *branching-time* models record the points at which different computations diverge from one another.

Intensionality vs extensionality *Intensional* models focus on describing what systems do. Intensional theories model systems in terms of states and transitions between states.

Extensional models, in turn, are based on what outside observers see. Extensional models first define a notion of observation and then represent systems in terms of the observations that may be made of them.

The dichotomy between intensional and extensional models also exists for sequential theories.

10.2 Action systems

An action system describes the behavior of a concurrent system in terms of the atomic actions that can take place during the execution of the system. Action systems can express both finite and infinite computations. An action system consists of a state, an initialization thereof, and a set of actions. Actions are comprised of a guard and a body. The body can be an arbitrary sequential program. Execution proceeds as follows: First, the initialization is executed. Then, actions are executed repeatedly in a loop until no more action is enabled. The selection of enabled actions is nondeterministic and is not bound to a fairness pledge. The nondeterminism is demonic, in the sense that there is no way of influencing which action is chosen next.

With respect to the above classification, our action systems are extensional and based on an interleaving semantics for actions communicating via shared variables. Action systems provide a simple, yet powerful formal seamless framework from high-level design to implementation. The higher-order logic refinement calculus theory can be used for reasoning about safety properties and about the termination of action systems. In this thesis, we are not concerned with general liveness properties [36], for which a temporal logic would be required.

The action system model for parallel, distributed, and reactive systems was proposed by Back and Kurki-Suonio [4, 5]. The same basic approach has later been used in other models for concurrent and distributed computing, notably in UNITY [15] and

```
MODULE OneFish;

  CONST
    height=10; width=20;

  VAR
    x*, y*: INTEGER;
    right*, up*: BOOLEAN;

  ACTION MoveRight
    WHEN right & (x#width);
  BEGIN INC(x)
  END MoveRight;

  ACTION MoveLeft
    WHEN ~right & (x#0);
  BEGIN DEC(x)
  END MoveLeft;

  ACTION BounceRight
    WHEN right & (x=width);
  BEGIN right:=FALSE
  END BounceRight;

  ... (* bounce left, move up and down *)

BEGIN (* initialization *)
  x:=0; y:=0; right:=TRUE; up:=TRUE
END OneFish.
```

Figure 22: Screen saver *OneFish*

TLA [37]. UNITY and TLA are based on temporal logics and give up some of the generality of action systems in favor of execution fairness.

Figure 22 gives an example of a fish screen saver action system in Action-Oberon, a concrete notation for action systems originally proposed by Back and Sere [6] and extended in Paper VI. A single fish swims around the screen. The current position of the fish is given by Cartesian coordinates x (horizontal axis) and y (vertical axis). The fish is either moving right ($right = TRUE$) or left and either up ($up = TRUE$) or down. When it reaches a border, it changes direction. Note that the lack of a fairness assumption means that the fish might only move along one axis, although the guard for moving along the other axis is infinitely often true.

10.3 Object-oriented action systems

Plain action systems are cumbersome for applications with several similar entities. Consider, for example, a more realistic screen saver with a varying number of fish. With plain action systems we would have to create an array of states and replicate the actions. Still, if the number of fish is not constant, replication as well as run-time creation and destruction of fish are cumbersome and error-prone.

To address this problem, we have added objects with actions to Action-Oberon and written a corresponding compiler and run-time environment for Paper VI. Figure 23 shows part of an object-oriented action system. *POINTER TO RECORD* roughly correspond to *class* in most other languages. The declaration *ACTION (f: Fish) MoveRight* leads to the dynamic creation of an action for each instance of type *Fish*. The bound variable f is called participant and may be used like a variable in the action. It corresponds to the receiver (self) of a method.

Actions with n participants lend themselves to symmetrically express n-ary communication. In most other formalisms communication is asymmetric, that is information is only transferred from the sender to the receiver, or symmetric communication is restricted to two parties.

```
MODULE OOFish;

  TYPE
    Fish=POINTER TO RECORD /* class */
         x, y: INTEGER;
         right, up: BOOLEAN
    END;

  VAR
    fi: Fish; k: INTEGER;

  ACTION (f: Fish) MoveRight
    WHEN f.right & (f.x#width);
  BEGIN INC(f.x)
  END MoveRight;

  ACTION (f: Fish) MoveLeft
    WHEN ~f.right & (f.x#0);
  BEGIN DEC(f.x)
  END MoveLeft;

  ...

BEGIN (* initialization *)
  ... (* create some fishes *)
END OOFish.
```

Figure 23: Screen saver *OOFish*

Suppose we want to program some special behavior if two fish meet. We can do this with *ACTION (f1, f2: Fish) Meet* (Fig. 24). An instance of *Meet* will be created at run time for each pair of fish.

The addition of object-orientation brings to concurrent systems the power of the object-oriented approach, such as encapsulation, subtyping, inheritance, and dynamic binding. Furthermore it removes the semantic gap when using object-oriented analysis and design.

Action systems provide a good model for specifying concurrent, distributed, and reactive systems. However, direct implementations, such as ours, suffer from an inherent inefficiency problem caused by the need to constantly re-evaluate guards. Our Action-Oberon compiler and run-time environment are, therefore, mostly meant to be used for the animation of specifications and for rapid prototyping.

More efficient implementations would be possible by optimizing the re-evaluation of the guards. Applicable techniques include common subexpression elimination and dependency analysis that indicates which actions may change the values of which guards. Further optimization techniques may be borrowed from triggered procedures in databases.

```
ACTION (f1, f2: Fish) Meet WHEN (f1.x=f2.x) & (f1.y=f2.y) & (f1#f2);
BEGIN
  (* do something: e.g.
    - change the directions of the fish
    - create a new fish
    - remove one of the fish *)
END Meet;
```

Figure 24: Type-bound action *Meet* with two participants

10.4 Refinement of object-oriented action systems

In Sect. 7 we have discussed class refinement. The idea was that instances of a subclass should behave like instances of a superclass. In Paper VII we extend this idea to active objects, that is, instances of classes with actions.

Trace semantics

There exists two semantics for action systems. The input/output semantics describes the possible final values for any initial values. It abstracts away from intermediate states. The trace semantics lists the sequences of observable states. Thus, it captures the reactive behavior and is also meaningful for non-terminating systems.

The trace semantics of an action system is given by a set of traces. A trace is a sequence of observable states. Hence, our trace semantics is a linear-time model.

Consider the screen save of Fig. 22 for an example. Representing the state of module *OneFish* as a tuple *(x, y, right, up)*, the following is a possible trace: $<$*(0, 0, TRUE, TRUE), (0, 1, TRUE, TRUE), (0, 2, TRUE, TRUE), (1, 2, TRUE, TRUE),* $\ldots>$.

Trace refinement of action systems [7] is defined so that each trace of the implementation action system is also a trace of the specification action system. Trace refinement implies refinement of the input/output behavior, but not the other way round.

Class refinement

We base our refinement of classes with actions on the trace semantics. A class D refines a class C, if replacing an instance of C by an instance of D in an (almost) arbitrary action system results in a trace refinement of the action system.

For example, a class *Ray* that has identical *MoveRight*, *MoveLeft*, *BounceRight*, and *BounceLeft* actions as *Fish* (Fig. 23), but no actions to move up and down, is a refinement of class *Fish*.

Refinement of classes with actions may also be used for development by refinement. Thus the same separation of concerns between specifications and refinement is possible as for the sequential case.

As for greybox refinement, it is very difficult to prove trace refinement of action systems directly. Therefore, we define in Paper VII a notion of class simulation. Class simulation does not talk about traces and does not contain a quantification over all contexts in which an instance of a class might occur. Hence, class simulation is much easier to establish. The main theorem of the paper states that class simulation implies class refinement.

11 Summary

The dependability of computer-based systems is crucial because they are ubiquitous in safety-critical applications. The dependability of computer programs is largely influenced by the expressiveness and safety of the languages used to create the programs. Expressive languages let us separate concerns and base our reasoning on simple and

powerful conceptual models. A language mechanism can be used to build dependable systems only if it works correctly under all circumstances.

Structure and abstraction allow a separation of concerns and, thereby, reduce the detailed reasoning to an achievable amount. We have performed a large case study in B to explore the virtues of modularization, specification, and stepwise refinement (Paper I). The sharing limitations among modules in B caught our attention. To overcome these restrictions, we have proposed a new compositional symmetric sharing mechanism based on roles expressing rely/guarantee conditions (Paper II).

B and other standard data refinement methods work well for layered systems where calls are only made from higher to lower layers. However, they cannot in a modular way handle call-backs, which are common in object- and component-based systems. To solve this problem, we have introduced greybox specifications and refinement (Paper III). Greybox specifications explicitly prescribe making external calls during the execution of a method and greybox refinement considers the sequence of external calls as part of the observable behavior that is preserved.

Types can enhance the semantic specification and refinement approach. Explicitly named and declared types can be used as labels guaranteeing adherence to semantic contracts. Since type systems are decidable, they can be used in compilers to automatically catch certain kinds of errors before starting difficult semantic proofs. Furthermore, types can be used for run-time compatibility checks. The safety and decidability of type systems comes at a price: Some semantically correct programs are not accepted by the type checker. Such problems are particularly pressing in the realm of component software, where components cannot be changed and recompiled so easily. In Paper IV, we have isolated such a problem and proposed a novel solution, which takes a step toward the reconciliation of safety and flexibility.

Existing composition mechanisms either fail to fully address the safety, modular reasoning, and late composition requirements of component software or allow only limited reuse due to typing restrictions. In Paper V, we have introduced a new composition mechanism that addresses these needs.

Many systems are concurrent. Action systems provide a simple, yet powerful conceptual model for concurrency. In Paper VI, we have added objects to action systems to more easily handle systems with many similar entities and to reduce the semantic gap when using object-oriented analysis. Our compiler and run-time environment allow the animation of action systems in Action-Oberon. Trace refinement of active objects allows us to apply algorithmic, data, and atomicity refinement when creating subclasses (Paper VII).

To assure the safety of both our systems and language mechanisms, we have used tool-based formal methods to achieve the most rigorous proofs. We have used Atelier B to mechanically check the case study of Paper I, and have proved type soundness in Isabelle/HOL for the extended type systems of Papers IV and V.

Acknowledgments Hilkka Yli-Jokipii carefully checked the language of an earlier version of this introduction and suggested many improvements. I am also delighted to acknowledge the help of Marsha Brofka, who proofread the acknowledgments and the abstract.

References

[1] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors. *Algebraic Foundation of System Specification*. IFIP State-of-the-Art Reports. Springer Verlag, 1999.

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

[4] Ralph Back and Reino Kurki-Suonio. Decentralization of process nets with centralized control. In *2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 131–142. ACM Press, 1983.

[5] Ralph Back and Reino Kurki-Suonio. Distributed co-operation with action systems. *ACM Transactions on Programming Languages and Systems 10:513–554*, 1988.

[6] Ralph Back and Kaisa Sere. From action systems to modular systems. In *Proceeding of Formal Methods Europe '94*. LNCS 873, Springer Verlag, 1994.

[7] Ralph Back and Joakim von Wright. Trace refinement of action systems. In *CONCUR 94*, pages 367–384. LNCS 836, Springer Verlag, 1994.

[8] Ralph Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer Verlag, 1998.

[9] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor: A successful application of B in a large project. In *Proceedings of FM'99: World Congress on Formal Methods*, pages 369–387. LNCS 1708, Springer Verlag, September 1999.

[10] Jonathan P. Bowen and Michael G. Hinchey. Formal methods and safety-critical standards. *IEEE Computer*, 27(8):68–71, 1994.

[11] Kim B. Bruce, Robert van Gent, and Angela Schuett. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proceedings of ECOOP '95*, pages 27–51. LNCS 952, Springer Verlag, 1995.

[12] Martin Büchi and Emil Sekerinski. Formal methods for component software: The refinement calculus perspective. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proceedings of the Second Workshop on Component-Oriented Programming (WCOP)*, volume 5 of *TUCS General Publication*, pages 23–32, Short version in ECOOP'97 workshop reader LNCS 1357, June 1997. http://www.abo.fi/~mbuechi/publications/FMforCS.html.

[13] Martin Büchi and Wolfgang Weck. A plea for grey-box components. Technical Report 122, Turku Centre for Computer Science, Presented at the Workshop on Foundations of Component-Based Systems, Zürich, September, 1997. http://www.abo.fi/~mbuechi/publications/GreyBoxes.html.

[14] Luca Cardelli. Type systems. In *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997. http://www.luca.demon.co.uk/Papers.html.

[15] K. M. Chandy and J. Misra. *Parallel Program Design – A Foundation*. Addison Wesley, 1988.

[16] David Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[17] Rance Cleaveland and Scott A. Smolka, editors. Strategic directions in concurrency research. *ACM Computing Surveys*, 28(4):607–625, December 1996.

[18] Component Source. http://www.componentsource.com.

[19] William Cook. A proposal for making Eiffel type-safe. In *Proceedings of ECOOP '89*, pages 57–70. Cambridge University Press, 1989.

[20] Brad Cox. Planning the software industrial revolution. *Software Technologies of the 90's special issue of IEEE Software magazine*, November 1990.

[21] Ole-Johan Dahl, Bjørn Myrhaug, and Kristen Nygård. Simula-67 common base language. Technical Report Publication S-22, Norwegian Computing Centre, Oslo, 1970.

[22] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge Tracts in Theoretical Computer Science, No. 47. Cambridge University Press, 1998.

[23] E.W. Dijkstra. Cooperating sequential processes. In *Programming Languages*, pages 43–112. Academic Press, 1968.

[24] E.W. Dijkstra. Notes on structured programming. In O. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*. Academic Press, 1971.

[25] Steve Dunne. The safe machine: A new specification construct for B. In *Proceedings of FM'99: World Congress on Formal Methods*, pages 472–489. LNCS 1708, Springer Verlag, September 1999.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[27] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[28] John Harrison. Formalized mathematics. Technical Report 36, Turku Centre for Computer Science, 1996. http://www.cl.cam.ac.uk/users/jrh/papers/formmath3.html.

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

[30] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[31] Cliff B. Jones. *Software Development: A Rigurous Approach*. Prentice Hall International, 1980.

[32] Cliff B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North Holland, 1983.

[33] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, 1986.

[34] Cliff B. Jones. Scientific decisions which characterize VDM. In *Proceedings of FM'99: World Congress on Formal Methods*, pages 28–47. LNCS 1708, Springer Verlag, September 1999.

[35] Robert M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, July 1976.

[36] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3:125–143, 1977.

[37] Leslie Lamport. The temporal logic of actions. *ACM Transactions of Programming Languages and Systems*, 16(3):872–923, 1994.

[38] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of OOPSLA '86*, pages 214–223. ACM Press, 1986.

[39] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[40] Lingsoft. Orthografix: Finnish proofing tools for Microsoft Word, 1998. http://www.lingsoft.fi/.

[41] Peter Lucas. On the semantics of programming languages and software devices. In Randall Rustin, editor, *Formal Semantics of Programming Languages (Proceedings of the Courant Computer Science Symposium 2)*, pages 41–57. Prentice Hall, 1972.

[42] McIllroy. Mass-produced software components. In Peter Naur, Brian Randell, and J. N. Buxton, editors, *Software engineering: concepts and techniques: proceedings of the NATO conferences. The Conference on Software Engineering held in Garmisch, Germany, 7th to 11th October 1968*. Petrocelli/Charter, 1976.

[43] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[44] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *Proceedings of ECOOP '98*, pages 355–374. LNCS 1445, Springer Verlag, 1998.

[45] Robin Milner. *A Calculus of Communicating Systems*. LNCS 92, Springer Verlag, 1980.

[46] Object Management Group. CORBA components, 1999. Revision February 15, 1999, formal document orbos/99-02-01, http://www.omg.org.

[47] Peter Mössenböck and Niklaus Wirth. The programming language Oberon-2. *Structured Programming 12:179–195*, 1991.

[48] P. G. Larsen and B. S. Hansen and H. Brunn N. Plat and H. Toetenel and D. J. Andrews and J. Dawes and G. Parkin and others. Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language, December 1996.

[49] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828, Springer Verlag, 1994. See also http://www.cl.cam.ac.uk/Research/HVG/Isabelle/.

[50] Cuno Pfister. Component software: A case study using BlackBox components (online tutorial of the BlackBox Component Builder), 1997. http://www.oberon.ch.

[51] Dale Rogerson. *Inside COM*. Microsoft Press, 1996. See also http://www.microsoft.com/com/.

[52] J.M. Spivey. *The Z Notation*. Prentice Hall, second edition, 1992.

[53] Stéria Méditerranée. *Atelier-B*. France, 1996. http://www.atelierb.societe.com.

[54] Sun Microsystems, Inc. Java Beans, 1997. http://java.sun.com/beans/.

[55] Clemens A. Szyperski. Import is not inheritance — Why we need both: Modules and classes. In *Proceedings of ECOOP 92*, pages 19–32. LNCS 615, Springer Verlag, 1992.

[56] Clemens A. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[57] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, pages 119–156. LNCS 1523, Springer Verlag, 1999.

[58] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14:221–227, 1971.

[59] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, 1982.

Paper I

The B Bank

Martin Büchi

Originally published in: Emil Sekerinski and Kaisa Sere, editors, *Program Development by Refinement: Case Studies Using the B Method*, FACIT series, chapter 4, pages 115–180. Springer Verlag, 1998.[1]

[1]The reference numbers in this reprint differ from those in the original publication because the latter contained a single bibliography for all chapters. The book's Web page referred to in the paper is located at http://www.cs.abo.fi/Bcases/.

4. The B Bank

Martin Büchi

4.1 Introduction

In this chapter we develop a simple banking application with cashier and automated teller machine (ATM) functionality. The cashier can register new customers, create accounts for them, and accept deposits. At the ATM, the customer can withdraw money, query the balance, and change her secret personal identification number (PIN).

We illustrate the combination of structured and formal methods by using object-oriented modelling techniques in the analysis. The communication from B with the environment is exemplified through the development of base machines for persistent storage of objects, string handling, and for interfacing with the Web through HTML and the common gateway interface. The latter permits us to build a uniform graphical interface for both the cashier station and the ATM (Fig. 4.1).

Fig. 4.1. Screenshot of the Final Application

Our aim is to carefully explain design decisions as they come up and to motivate our choices. We stress differences to classical imperative languages and development methods for them.

The sources for both Atelier B and the B-Toolkit can be fetched from the book's Web site. The final application being Web-based, it can also be run over the Internet from the book's Web page without the need for installation.

We start out by rewriting the informal requirements in structured plain English, as is commonly done in practice. This first design document helps to eliminate misunderstandings between the customer and the designer and is often part of a contract. We then proceed to a semi-formal object model using the Unified Modeling Language (UML) [7]. In this step we make the first design decisions by identifying objects, relations, and attributes. This intermediate step bridges the gap between requirement specification and B machine.

Our initial B specification *Bank* encompasses the basic functionality on an abstract level. This is the machine which we animate to find design errors. On top we build a robust graphical user interface. Underneath, we build a foundation for objects and persistent storage. This combination of top-down and bottom up development, where we start with a machine describing the functionality on an abstract level, is very common in B.

On top of the central machine *Bank* we construct a robust interface *RobustBank* with trivial preconditions and error reporting. Using this robust interface and a base machine wrapping a common-gateway interface library, we build a Web-based graphical user interface for our development.

A program consists of an algorithm and communication with the environment. Only the algorithm can be directly implemented in B. Communication is performed using base machines which give a B representation of a resource. A base machine is a machine which is specified in B, but hand coded in C, or another classical language for which a compiler exists. We illustrate the development of a base machine for interfacing with the Web in Sect. 4.7.

The implementation of *RobustBank* shows the principle of structural refinement. An implementation is based on a number of more basic machines, which are in turn based on either more basic or base machines. We discuss the difference between specification and implementation structure. Using a library machine for two-dimensional arrays and a base machine for file access we develop a framework for persistent objects. Another base machine provides persistent strings.

Fig. 4.2 gives an overview of the development process, including section numbers for quick reference. An overview of the implementation of *Bank* will be given in Fig. 4.12.

In the discussion we address the question of proofs in B. What types of properties about our system can we prove within B?

Steria's Atelier B in version 3.2 [23] has been used in this case study. Sect. 4.11 explains the differences in the implementation for B-Core's B-Toolkit 3.4.2 [17]. We briefly discuss a number of interesting differences in the language implementations and provided library constructs.

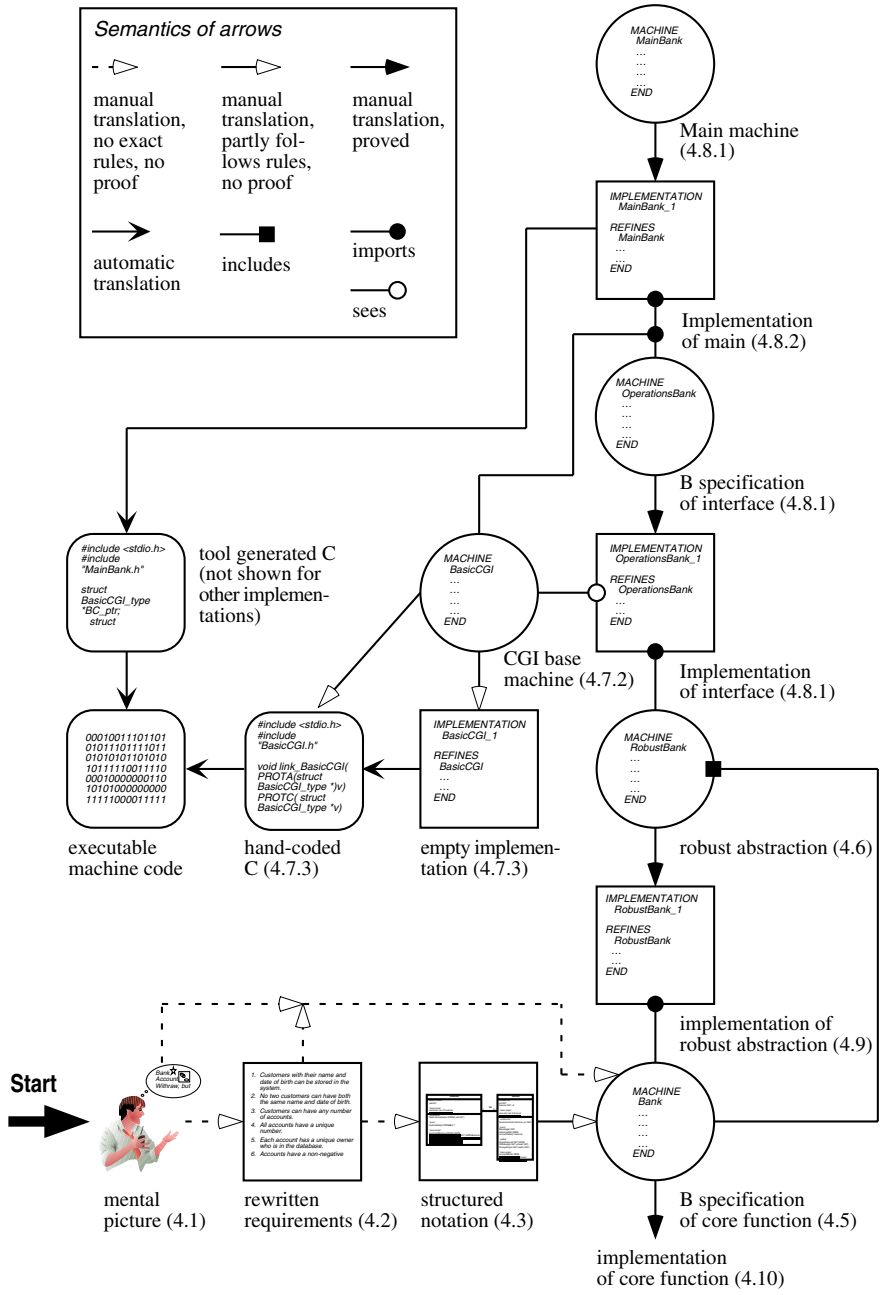


Fig. 4.2. Overview of the Development Process

4.2 Rewriting the Requirements

We start out by making the requirements of the initial application more precise. Such a complete rewrite by the developer of the customer's requirements in a common language provides for a common understanding. It can also eliminate many errors typically introduced by going directly from a mental picture to a specification, or even worse an implementation. Requirements state only what must be achieved, but not how it must be done. Fig. 4.3, an excerpt of Fig. 4.2, shows where in the development process we are.

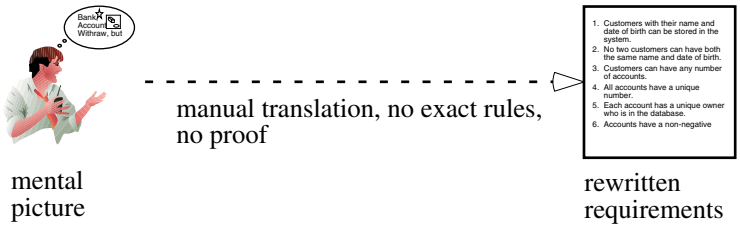


Fig. 4.3. Requirement Analysis

The system should provide for:

1. Customers with their name and date of birth can be stored in the system.
2. No two customers can have both the same name and date of birth.
3. Customers can have any number of accounts.
4. All accounts have a unique number.
5. Each account has a unique owner who is in the database.
6. Accounts have a non-negative balance.
7. Accounts have a secret PIN.

The cashier can perform the following transactions:

8. The cashier can enter new customers into the system by providing their name and year of birth.
9. The cashier can create new accounts with a zero balance providing a customer identification and an initial PIN. The latter can be entered by the customer.
10. The cashier can accept deposits knowing only the number of the account. The secret PIN is not needed for deposits.

The customer can perform the following operation at the ATM, which all require the account number — entered manually rather than read from a chip or magnetic card in our simulation — and the matching secret PIN:

11. The customer can make a withdrawal of at most the current balance.
12. The customer can query the current balance.

13. The customer can change the secret PIN by providing both the old, currently valid, and the new pin. The latter becomes immediately valid and the old PIN can no longer be used.

The user interface should be Web-based and provide access to all the above listed functions of the system. For brevity, we refrain from listing the user interface requirements here. We return to the topic in Sect. 4.8. A more detailed explanation of requirement analysis can be found in software engineering books, such as [19, 22].

4.3 Structured Models

In the next step, analysis, we produce structured models from the problem statement. The structured notations help to produce specifications which are correct with respect to the user requirements. This step is performed manually, following some heuristics. However, it lacks formal rules and, therefore, also a proof. This step could be skipped, going directly to a B specification. However, this would be a rather big step and, hence, also a source of errors. The benefits of integrating formal and structured methods are becoming recognised by many researchers [8, 9]. The IEC 65A 122 standard for safety-critical software also recommends the use of both structured and formal methods for software of the highest integrity level [10]. Often customers can be taught to read structured diagrammatic notations, but not formal AMN specifications. This intermediate step provides a more concise foundation for discussion than the natural language requirements.

The desire to capture all aspects of a problem using graphical models has led to a proliferation of different diagram types. We abstain from using all these — often not very useful — diagrams and do not attempt to capture everything in a graphical notation. We regard graphical models as complimentary to the textual specifications. Not opting for an automatic translation from the graphical model, we can give true abstractions, which quickly convey the main aspects, rather than cluttering the models with implementation details.

For our case study only static structure diagrams are relevant. The large amount of information captured in static structure diagrams is widely acknowledged [11]. Dynamic models are not applicable, because all operations are modeless, for example, the customer enters the account number, the PIN, and the desired amount all at once before asking the system to perform the withdrawal. A functional model would not provide much insight, as all transactions are made against a single database.

We have chosen the Unified Modeling Language (UML) [7]. Fig. 4.4 reminds us again, where in the development process we are.

4.3.1 Class Diagrams

The class diagram shows the static data structure of the real-world system and organises it into workable pieces. It describes real-world object classes and their relationships to each other.

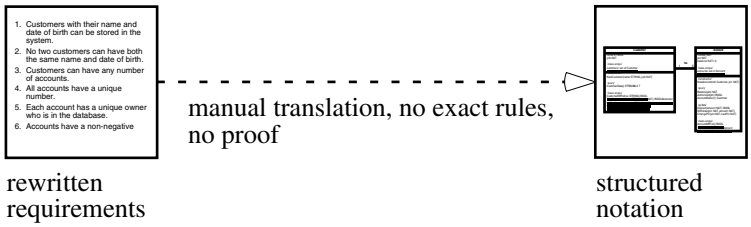


Fig. 4.4. Structured Notation

In our case we identify Customer and Account as object classes (Fig. 4.5). Our simple data dictionary defines them as follows: A Customer is the holder of zero or more accounts. An Account is an entity in our bank against which transactions can be made.

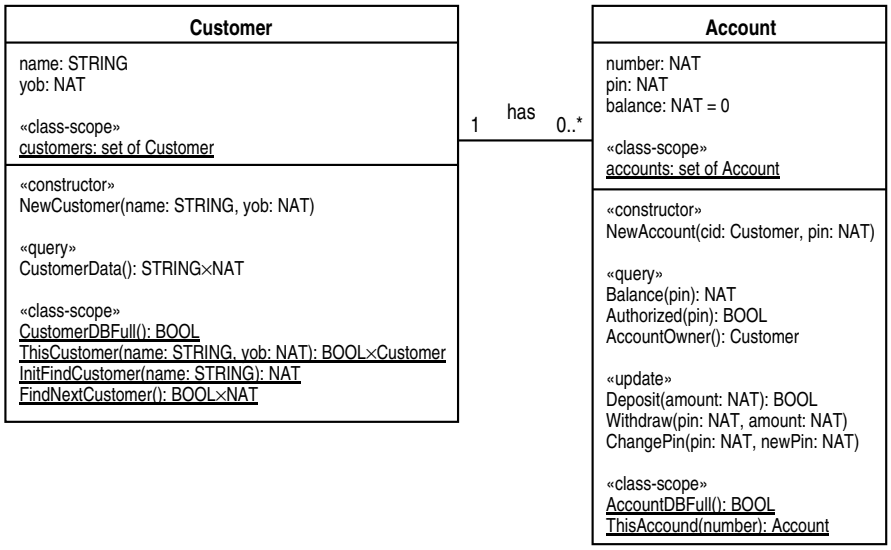


Fig. 4.5. Object Model

Next, we enumerate the attributes, that is, the properties, and the operations of the individual classes. Each Customer has a name and a year of birth (yob). In addition to the instance-scope attributes, of which each instance has its own copy, class Customer has the class-scope attribute customers, the set of all customers in the system. Class-scope members are underlined in the diagram. The class Customer has a single constructor and a single query function. The product type STRING×NAT indicates that CustomerData returns both the name and the year of

birth of a customer. It also has class-scope operations to inquire whether the database is full, to retrieve a customer, and to find all customers with a certain name.

Each Account has a number, a pin, and a balance, which is initially 0. Remember that requirement 4 states that number is an identifier. In entity-relationship models, this would typically be expressed by underlining the attribute — a notation which is used for class-scope attributes in UML. Entity relationship models representing sets, each class must have an identifier. However, in object-oriented systems we can have several objects with the same values for all their attributes. Objects have a system-generated unique identifier. Hence, unlike in multisets, objects with identical attribute values can actually be distinguished. In our example, we do not have multiple objects with identical values for their attributes. A notation for indicating identifiers in class diagrams would add information.

Class Account also has a class-scope attribute accounts, the set of all accounts in the system. Account has a single constructor. The query functions permit the user to query the balance, check whether a pin is valid, and get the owner of an account. The update operations provide functionality to make a deposit or withdrawal and to change the PIN. The class-scope operations allow the user to check whether the database is full and to retrieve an account by its number.

Finally we catalogue the associations, that is, the dependencies between objects. A customer may have any number of accounts; each account has exactly one owner. This association is expressed by the line between the two classes in Fig. 4.5. The multiplicity is expressed using intervals. The '1' next to Customer says that each account is owned by exactly one customer. The '0..*' next to Account expresses that a customer may have any number of accounts. The label has names the association.

4.4 System Design

From the analysis of the system we progress to system design. System design is the high-level strategy for solving the problem and building a solution. During system design, we partition the system into subsystems, decide on what external hard- and software components we use, and establish a conceptual policy.

We start with the middle layer capturing the desired functionality (Fig. 4.6). On top of the basic functionality layer we build a robust abstraction which performs error checking and returns error codes, rather than relying on non-trivial preconditions. The top layer gives us the desired system in the form of a Web interface as defined by the problem statement. Its second foundation is the common gateway interface (CGI) subsystem, which consists of an off-the-shelf CGI library and a B wrapper. The CGI subsystem interfaces to the Web server. The latter communicates via TCP/IP with the Web browsers running on the ATM and the cashier's terminal.

In order to implement the core data, we build a subsystem which supports persistent objects and strings. The former in turn is based on two more basic subsystems, one giving us objects and a second one providing access to the file system. The two bottom layers represent the available resources, namely the hardware and the operating system.

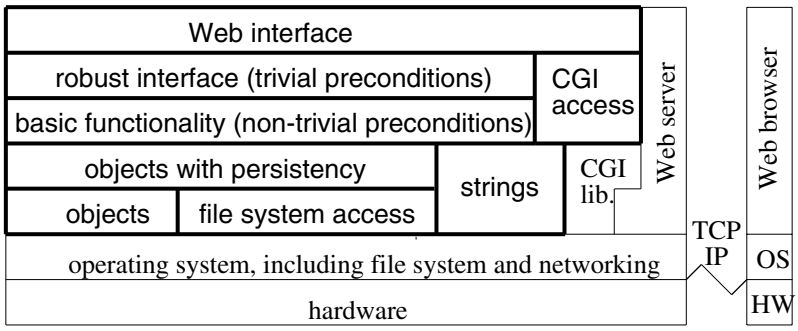


Fig. 4.6. System Design

An alternative would have been to rely on a database management system for persistent storage, giving us such standard features as transaction management, distribution, and crash recovery. We have chosen not to do so in order to maximize the ratio of formally verified software and limit the external dependencies of this case study.

4.5 B Specification

Having outlined the system architecture, we continue by translating the structured model to a B specification, giving the middle layer of basic functionality. First we translate our object model according to fixed rules which gives the state space of the machine and the signature of the operations. Then we add the initialisation and the specification of the operations with help of the rewritten requirements. Fig. 4.7 points again to our current position in the development process.

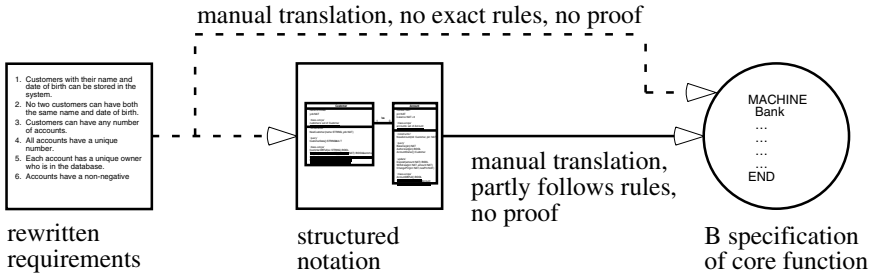


Fig. 4.7. Transformation to B Specification

4.5.1 State

For each object class we introduce a set containing all possible instances. This gives us the sets *CUSTOMER* and *ACCOUNT*. For technical reasons, detailed in Sect. 4.11, we define them as subsets of *NAT* rather than as *SETS*. The cardinalities of the sets, delimiting the maximal number of customers and accounts in the system, are given by the machine parameters *maxCustomers* and *maxAccounts*.

MACHINE
Bank(*maxCustomers*, *maxAccounts*)

CONSTRAINTS
$maxCustomers \in 1 \,..\, 100000 \wedge maxAccounts \in 1 \,..\, 200000$

SEES
StrTokenType

DEFINITIONS
$CUSTOMER == 0 \,..\, maxCustomers\text{-}1$; $ACCOUNT == 0 \,..\, maxAccounts\text{-}1$

Furthermore we introduce the two class-scope variables of Fig. 4.5 *customers* ($\subseteq$ *CUSTOMER*) and *accounts* ($\subseteq$ *ACCOUNT*), which denote the sets of customers and accounts in the system.

Mandatory attributes are modelled as total functions from the set of actual customers, respectively accounts, to the value of the attribute. This gives us variables *customerName*, *customerYob*, *accountNumber*, *accountPin*, and *accountBalance*. Identifiers, for example, *accountNumber* and the product of *customerName* and *customerYob*, are injections, capturing the fact that no two objects with the same values for these attributes can exist.

The seen machine *StrTokenType* defines the set *STRTOKEN* representing strings and the empty string constant *EmptyStringToken* ($\in$ *STRTOKEN*). The rationale behind string tokens will be explained in Sect. 4.7.1.

The relation has can be translated to the total function *accountOwner* from *accounts* to *customer*. It is a function, rather than a general relation, because the maximum multiplicity of Customer is 1; furthermore, it is total because the minimum multiplicity is also 1. The variable *foundCustomers* is used for the implementation of the search-by-name operations for customers as described below.

The last state component is the concrete (also called visible) variable *fileOpen*. It indicates whether the database has been successfully internalised from disk and, thus, whether the machine can actually be used. The difference between a normal (also called abstract or hidden) variable and a concrete variable is that the latter is implemented unchanged and can, therefore, be directly accessed by implementations that import *Bank*.

VARIABLES
customers, *customerName*, *customerYob*,
accounts, *accountNumber*, *accountPin*, *accountBalance*, *accountOwner*,
foundCustomers

CONCRETE_VARIABLES
fileOpen
INVARIANT
$customers \subseteq CUSTOMER \wedge$
$customerName \in customers \rightarrow STRTOKEN \wedge customerYob \in customers \rightarrow \mathbf{NAT} \wedge$
$customerName \otimes customerYob \in customers \rightarrowtail (STRTOKEN \times \mathbf{NAT}) \wedge$
$accounts \subseteq ACCOUNT \wedge$
$accountNumber \in accounts \rightarrowtail \mathbf{NAT} \wedge accountPin \in accounts \rightarrow \mathbf{NAT} \wedge$
$accountBalance \in accounts \rightarrow \mathbf{NAT} \wedge accountOwner \in accounts \rightarrow customers \wedge$
$fileOpen \in \mathbf{BOOL} \wedge foundCustomers \subseteq customers$

4.5.2 Functionality

In the beginning, there are no customers or accounts in the database. Hence, the initialisation assigns the empty set to the sets *customers* and *accounts* and, therefore, also to the functions representing the attributes and relations. As the database has not yet been read from disk *fileOpen* is *FALSE*. We could have designed the system so that internalisation from disk is part of initialisation. Because internalisation can fail, if, for example, the file has been corrupted, a variable indicating its success would have to be set during initialisation and checked by the higher level abstraction. Hence, we would not gain anything. We introduce the abbreviation *RESET* as the same code occurs again later.

DEFINITIONS
RESET ==
$customers := \{\} \,||\, customerName := \{\} \,||\, customerYob := \{\} \,||$
$accounts := \{\} \,||\, accountNumber := \{\} \,||\, accountPin := \{\} \,||$
$accountBalance := \{\} \,||\, accountOwner := \{\} \,||$
$fileOpen := \mathbf{FALSE} \,||\, foundCustomers := \{\}$

INITIALISATION
RESET

The first operation *NewCustomer* creates a new customer object and sets its *name* and *yob* attributes. In order to concentrate on the actual functionality, rather than error checking and reporting, the precondition not only gives a type to the parameters, but also states that there must not be any customer with both the same name and year of birth present in the database, that the database must not be full, and that internalisation (see below) must have succeeded. If these conditions are met, an arbitrary new customer object is selected using the *ANY*-clause. This object is added to *customers* and its *name* and *yob* attributes are set. Note that *customerName(newCustomer)* := *name* is an abbreviation for *customerName* := *customerName* $\cup \{newCustomer \mapsto name\}$.

NewCustomer(*name*, *yob*) =
PRE
$name \in STRTOKEN \wedge yob \in \mathbf{NAT} \wedge$

```
    (name, yob) ∉ ran(customerName ⊗ customerYob) ∧
    customers ≠ CUSTOMER ∧ fileOpen = TRUE
  THEN
    ANY newCustomer WHERE
      newCustomer ∈ CUSTOMER - customers
    THEN
      customers := customers ∪ {newCustomer} ||
      customerName(newCustomer) := name || customerYob(newCustomer):= yob
    END
  END;
```

Any client of *NewCustomer* must be able to verify the precondition. For this purpose we introduce operations *ThisCustomer* and *CustomerDBFull*. Operation *ThisCustomer* checks whether a customer denoted by her *name* and *yob* is present. If this is the case, the operation returns result code *TRUE* and the ID of the customer. Otherwise, the result code is set to *FALSE*. The result code alone would suffice to check the existence; the operation is more general for purposes we shall see later on.

```
  found, cid ← ThisCustomer(name, yob) =
    PRE name ∈ STRTOKEN ∧ yob ∈ NAT ∧ fileOpen = TRUE THEN
      IF (name, yob) ∈ ran(customerName ⊗ customerYob) THEN
        cid := (customerName ⊗ customerYob) ^-1 (name,yob) || found := TRUE
      ELSE
        cid :∈ CUSTOMER || found := FALSE
      END
    END;
```

In practice, databases are assumed to have infinite capacity and their administrators are supposed to add secondary storage as the available storage gets filled. However, the number of incidents of database and buffer overflow problems clearly shows that we should not trust this assumption in a safety-critical system. Operation *CustomerDBFull* allows us to check whether the database is full and, herewith, verify the precondition *customers* $\neq$ *CUSTOMER* of *NewCustomer*. Note that we could prove the invariant of machine *Bank* to be preserved without this precondition. In the case it would not hold, the *ANY*-statement would have to choose an element from the empty set and would therefore be magic. Hence, we could not find any implementation using a finite set *CUSTOMER* which would either always find an unused member or execute magic.

```
  is ← CustomerDBFull =
    PRE fileOpen = TRUE THEN
      is := bool(customers = CUSTOMER)
    END;
```

Operation *NewCustomer* can only be performed if the internalisation of the database from disk has succeeded. This condition is expressed by the last conjunct

of the precondition: *fileOpen* = *TRUE*. A more pragmatic solution would be to assume that any client of *Bank* will terminate with an error message if internalisation fails and not make any calls to *NewCustomer*. However, replacing the formal precondition with this informal assumption would lead to unprovable obligations.

Operation *CustomerData* is an instance-scope operation which returns the name and year of birth of a customer. Self, the identity of the object, is modelled as a normal parameter *cid*. The identity of a customer object can be retrieved using *ThisCustomer*. Atelier B requires the additional typing $cid \in CUSTOMER$.

name, *yob* ← **CustomerData**(*cid*) =
  PRE $cid \in customers \wedge cid \in CUSTOMER \wedge fileOpen = \mathbf{TRUE}$ **THEN**
    $name := customerName(cid) \,||\, yob := customerYob(cid)$
  END;

The find operations give the set of all customers with a certain name. First, operation *InitFindCustomer* must be called. It returns the number of matches and assigns the matching customers to *foundCustomers*. Operation *FindNextCustomer* then returns the matching customers one by one.

nof ← **InitFindCustomer**(*name*) =
  PRE $name \in STRTOKEN \wedge fileOpen = \mathbf{TRUE}$ **THEN**
    $nof, foundCustomers \in (foundCustomers = customerName^{-1}[\{name\}] \wedge$
    $nof = \mathbf{card}(foundCustomers))$
  END;
found, *yob* ← **FindNextCustomer** =
  PRE $fileOpen = \mathbf{TRUE}$ **THEN**
    IF $foundCustomers \neq \{\}$ **THEN**
      ANY *cust* **WHERE** $cust \in foundCustomers$ **THEN**
        $found := \mathbf{TRUE} \,||\, yob := customerYob(cust) \,||$
        $foundCustomers := foundCustomers - \{cust\}$
      END
    ELSE $found := \mathbf{FALSE} \,||\, yob :\in \mathbf{NAT}$
    END
  END;

The triple *NewAccount*, *ThisAccount*, and *AccountDBFull* is similar to the corresponding operations on customers. Operation *NewAccount* expects as parameters the ID of an existing customer and an initial secret PIN. By making the PIN a parameter we favour the scenario where the customer enters the desired PIN when the cashier creates the account. If the ATM card and the PIN are mailed to the customer, a random PIN must be generated in one of the above layers. Operation *AccountOwner* returns the owner of an account.

number ← **NewAccount**(*cid*, *pin*) =
  PRE
    $cid \in customers \wedge cid \in CUSTOMER \wedge pin \in \mathbf{NAT} \wedge$
    $accounts \neq ACCOUNT \wedge fileOpen = \mathbf{TRUE}$
  THEN
    ANY *newAccount*, *newNumber* **WHERE**

```
    newAccount ∈ ACCOUNT - accounts ∧
    newNumber ∈ NAT ∧ newNumber ∉ ran(accountNumber)
  THEN
    accounts := accounts ∪ {newAccount} ||
    accountNumber(newAccount) := newNumber ||
    accountPin(newAccount) := pin || accountBalance(newAccount) := 0 ||
    accountOwner(newAccount) := cid || number := newNumber
  END
END;
found, aid ← ThisAccount(number) =
  PRE number ∈ NAT ∧ fileOpen = TRUE THEN
    IF number ∈ ran(accountNumber) THEN
      aid := accountNumber ^-1 (number) || found := TRUE
    ELSE aid :∈ ACCOUNT || found := FALSE
    END
  END;
is ← AccountDBFull =
  PRE fileOpen = TRUE THEN
    is := bool(accounts = ACCOUNT)
  END;
cid ← AccountOwner(aid) =
  PRE aid ∈ accounts ∧ aid ∈ ACCOUNT ∧ fileOpen = TRUE THEN
    cid := accountOwner (aid)
  END;
```

The operation *Balance* requires the account's PIN. The PIN is only used in the precondition to verify the legitimacy of the client, but not in the body of the operation. Specifying that the entered PIN must match the stored PIN in the precondition, forces us to prove that *Balance* is always called with the correct PIN. Unfortunately, this implies that the parameter *pin* is also present in the actual implementation where it is not used at all. To gain additional security, especially if the upper software levels are not fully proved, the correctness of the PIN could actually be verified in the implementation — contrarily to the standard practice of not verifying preconditions in implementations. Logically, it would be sound to allow implementations to have only a subset of the parameters of the corresponding machine, but in practice this would mean that the client's C code would depend not only on the interface defined by the machine, but also on the actual implementation. The alternative would be to drop the *pin* parameter altogether and trust in the clients always calling an authorisation operation, such as *Authorized*, first. However, such a condition would not create any proof obligations and would, therefore, not be verifiable within B. A model checking solution to the latter approach is documented in [5].

```
bal ← Balance(aid, pin) =
  PRE
    aid ∈ accounts ∧ aid ∈ ACCOUNT ∧
    pin ∈ NAT ∧ accountPin(aid) = pin  ∧ fileOpen = TRUE
  THEN
    bal := accountBalance(aid)
  END;
```

```
is ← Authorized(aid, pin) =
  PRE
    aid ∈ accounts ∧ aid ∈ ACCOUNT ∧ pin ∈ NAT ∧ fileOpen = TRUE
  THEN
    is := bool(accountPin(aid) = pin)
  END;
```

We can enforce that withdrawals and balance queries can only be performed with the correct PIN. On the other hand, secrecy not being a property of behaviors, we cannot ensure it in B. Nothing can prevent an implementation to output secret pins onto a device, the state of which is not captured by the B specification.

The operation *Deposit* credits the amount to the specified account. It cannot verify that the money is actually given to the bank; this is the duty of the cashier.

We have to make sure that the addition *accountBalance(aid)* + *amount* does not create an overflow. There are a number of approaches to this problem:

- One possibility is to blindly assume that no one will ever have this much money and leave the addition unguarded. This will, however, rightfully leave us with an undischargable proof obligation. Even if our assumption holds, a typing error by a cashier could crash the system. The latter could again be caught by a check for a maximum amount in the interface, leaving only a sequence of similar mis-entries as problematic.
- We could strengthen the precondition of *Deposit* with *accountBalance(aid)* $<$ *maxint - amount* and offer an additional operation *MaximalDeposit* returning the biggest possible deposit on a given account. Such an operation could, however, be abused to query the balance without the secret PIN from another software layer. Whether such guarding between software layers is needed in a closed system is debatable. After all, no customer of the bank could abuse this loophole at an ATM. Only programmers writing clients could. Note that introducing such a loophole would not create any unprovable proof obligations in B. We cannot express a property like ‘client machines cannot infer the balance without knowledge of the secret PIN’ in B.
- The third possibility is to let *Deposit* indicate whether the operation has succeeded or not. This cannot as easily be abused to query the balance, because if the operation succeeds a transaction is performed and the money must actually be transferred. Hence, this solution is chosen.

```
status ← Deposit(aid, amount) =
  PRE
    aid ∈ accounts ∧ aid ∈ ACCOUNT ∧ amount ∈ NAT ∧ amount > 0 ∧
    fileOpen = TRUE
  THEN
    IF accountBalance(aid) < MAXINT - amount THEN
      accountBalance(aid) := accountBalance(aid) + amount || status := TRUE
    ELSE status := FALSE
    END
  END;
```

Withdraw(*aid*, *pin*, *amount*) =
  PRE
    $aid \in accounts \wedge aid \in ACCOUNT \wedge pin \in \mathbf{NAT} \wedge amount \in \mathbf{NAT} \wedge$
    $accountPin(aid) = pin \wedge amount \leq accountBalance(aid) \wedge$
    $fileOpen = \mathbf{TRUE}$
  THEN
    $accountBalance(aid) := accountBalance(aid) - amount$
  END;
ChangePin(*aid*, *pin*, *newPin*) =
  PRE
    $aid \in accounts \wedge aid \in ACCOUNT \wedge pin \in \mathbf{NAT} \wedge accountPin(aid) = pin \wedge$
    $newPin \in \mathbf{NAT} \wedge fileOpen = \mathbf{TRUE}$
  THEN
    $accountPin(aid) := newPin$
  END;

Operations *Withdraw* and *ChangePin* follow the same pattern as *Deposit*.

The two final operations *Open* and *Close* concern persistency. An image of the set of customers, accounts, and strings (see below) is stored in the files designated by the parameters *customerFileName*, *accountFileName*, and *stringFileName* between program runs. *Open* is meant to read an arbitrary state satisfying the invariant from secondary storage. If *Open* succeeds, the result code *status* and the status flag *fileOpen* are set to *TRUE*. Note that the new state must satisfy the invariant, even if *status* is *FALSE*. In practice, *status* = *FALSE* means that the aforementioned files do not contain the image of a legal state or that the files cannot be properly accessed. *Close* writes the current state of the machine to the three files.

$status \leftarrow$ **Open**(*customerFileName*, *accountFileName*, *stringFileName*) =
  PRE
    $customerFileName \in STRING \wedge accountFileName \in STRING \wedge$
    $stringFileName \in STRING \wedge fileOpen = \mathbf{FALSE}$
  THEN
    ANY *customersInit*, *customerNameInit*, *customerYobInit*,
      accountsInit, *accountNumberInit*, *accountPinInit*,
      accountBalanceInit, *accountOwnerInit*, *st*
    WHERE
      $customersInit \subseteq CUSTOMER \wedge$
      $customerNameInit \in customersInit \rightarrow STRTOKEN \wedge$
      $customerYobInit \in customersInit \rightarrow \mathbf{NAT} \wedge$
      $customerNameInit \otimes customerYobInit \in customersInit \rightarrowtail (STRTOKEN \times \mathbf{NAT})$
      $\wedge\ accountsInit \subseteq ACCOUNT \wedge$
      $accountNumberInit \in accountsInit \rightarrowtail \mathbf{NAT} \wedge$
      $accountPinInit \in accountsInit \rightarrow \mathbf{NAT} \wedge$
      $accountBalanceInit \in accountsInit \rightarrow \mathbf{NAT} \wedge$
      $accountOwnerInit \in accountsInit \rightarrow customersInit \wedge$
      $st \in \mathbf{BOOL}$
    THEN
      $customers := customersInit \,||\, customerName := customerNameInit \,||$
      $customerYob := customerYobInit \,||$
      $accounts := accountsInit \,||\, accountNumber := accountNumberInit \,||$
      $accountPin := accountPinInit \,||\, accountBalance := accountBalanceInit \,||$

```
      accountOwner := accountOwnerInit ||
      foundCustomers := {} || fileOpen := st || status := st
    END
  END;
status ← Close =
  PRE fileOpen = TRUE THEN
    RESET || status :∈ BOOL
  END
END
```

In B we can only reason about a single program run. We could express as an invariant with auxiliary variables the condition that calling *Close*, then arbitrarily modifying the state, and thereafter calling *Open* should be *skip* on the base state space, if both result codes indicate success. This could be expressed by *Close* creating a snapshot of the current state in a set of auxiliary variables. However, we cannot infer from this that externalisation and internalisation actually work. A meta-language statement (*Close*; *Open*) = *skip* is easier to understand than a similar condition encoded as an invariant. Hence, it might be desirable to have a formal meta language with an associated proof tool for expressing such properties in B, as is done, for example, by the Refinement Calculator [4] for the refinement calculus.

Machine *Bank*, encapsulating the basic functionality, is animated to test whether it satisfies the stated requirements and also to check whether the latter are what we actually want. The proofs for this machine ascertain that the initialisation establishes the invariant and that the operations preserve it. However, the step from the rewritten requirements and the structured notation to the formal B specification cannot be formally proven, as indicated by the arrows in Fig. 4.7.

4.5.3 Discussion

The account number is a unique identifier for accounts. Hence, instead of introducing the system-generated object identifiers *customers* ($\subseteq$ *CUSTOMER*) we could have used account numbers as identifiers, simplifying the specification. The other attributes would then have been functions with domain *accountNumber* rather than *accounts*. In the implementation, we could have still used system-generated identifiers, in order to make references to accounts independent of the chosen pattern for account numbers and to use a generic support machine for persistent objects. The two specifications can be proved to be equivalent by mutual refinement (Exercise 4.3). We decided not to make the simplification in order to better illustrate the general scheme.

In our example, we have only used very simple UML class models. We sketch here briefly the translation of some more advanced elements.

Optional attributes can be modelled by partial functions. Attributes of maximal cardinality greater than one, as allowed in entity-relationship diagrams, can be expressed as general relations. Binary relations between classes with maximum cardinality greater one for both classes are expressed as general relations in B.

Subtypes can be expressed as a subsets. Hence, polymorphism can be expressed in B as 'soft types'. However, dynamic binding must be expressed as case statements. Hence, only closed (complete) systems can be given a B translation. Furthermore, all classes with cyclic references must be specified in the same machine. The transformation is difficult because B prohibits the calling of operations from the same module and the use of sequencing in machines. B is well-suited for the translation of a certain class of object-oriented models.

The combination of B and OMT [18] object models, the predecessors of UML class models, has been pioneered by Lano [13, 12]. Different translations of object diagrams into B have been proposed [6, 21]; the B-Toolkit even offers a tool for automatic translation (Sect. 4.11).

A simple translation of statecharts to B is also given by Lano [13]. A more thorough treatment can be found in Sekerinski [20]. Exercise 4.2 uses dynamic modelling to add online banking with a login to our application.

4.6 Robust Abstraction

To keep the specification simple, the initial machine *Bank* uses non-trivial preconditions rather than elaborate error handling. We could build a graphical user interface directly upon it. However, we opt for an intermediate layer, providing roughly the same functionality but with verification of parameters. Herewith, we effectively split up the task at hand. We avoid duplication of parameter checking for transactions which can be performed in different manners, for example by a cashier or at an ATM, using different interfaces.

We have to decide whether we want to include *Bank* into the robust interface *RobustBank* or not. If we want to reason about the behaviour on the robust level or if we want to be able to do such reasoning on even higher levels, we have to include *Bank*. If, on the other hand, all interesting invariant conditions are provable on the lower level, the inclusion would not make sense. Without including *Bank* we cannot specify under which conditions the operation actually succeeds and which parameters lead to which status code. However, we are guaranteed termination, which means that the corresponding implementation can only call the lower level implementation if the latter's precondition is satisfied. The advantage of the underspecification is that the implementation is also allowed to return an error in cases not explicitly captured by the specification, arising from practical implementation issues. We decide to include *Bank* to be able to perform more reasoning; the alternative approach will be illustrated on the next level up, the user interface layer. Below is the specification of *RobustNewCustomer* in the case where *Bank* would not be included.

$result \leftarrow$ **RobustNewCustomer**($name$, yob) =
  PRE $name \in STRING \wedge yob \in$ **NAT THEN**
    $result :\in \{success, db\_full, db\_error, customer\_already\_present\}$
  END

Although specification and implementation structuring are largely independent in B, the above decision has some practical consequences. If we include *Bank* in *RobustBank*, the latter becomes the focus of refinement and implementation. We only need to implement *Bank* if we opt for importing it in the implementation of *RobustBank*. In the alternate approach of non-inclusion, we implicitly assume that *Bank* is imported in the implementation of the robust level and that the corresponding operations are called.

MACHINE
RobustBank(*maxCustomers*, *maxAccounts*)

CONSTRAINTS
$maxCustomers \in 1 \,..\, 100000 \wedge maxAccounts \in 1 \,..\, 200000$

INCLUDES
BK.Bank(*maxCustomers*, *maxAccounts*)

SEES
StrTokenType

DEFINITIONS
$CUSTOMER == 0 \,..\, maxCustomers\text{-}1$; $ACCOUNT == 0 \,..\, maxAccounts\text{-}1$

SETS
$RESULT = \{success, dbFull, dbError, customerAlreadyPresent,$
$unknownCustomer, negativeAmount, amountTooBig, unknownAccount,$
$AmountGreaterThanBalance, WrongPin\}$

We rename *Bank* in the includes clause so that references to its identifiers must be fully qualified, which increases readability. Note that sets, elements of enumerated sets, and constants do not participate in the renaming.

The robust operations are overly specific with respect to the reported result codes. For example in the case of *RobustNewCustomer* the specification prescribes the result code to be *dbFull* rather than *customerAlreadyPresent* in the case where both are applicable, for example the database is full and the customer passed as parameter is already in the database. This approach is simpler, but constrains the implementation. Exercise 4.6 investigates the more general specification.

OPERATIONS
$result \leftarrow$ **RobustNewCustomer**(*name*, *yob*) =
PRE $name \in STRTOKEN \wedge yob \in NAT$ **THEN**
IF *BK.fileOpen* = **TRUE THEN**
IF $BK.customers \neq CUSTOMER$ **THEN**
IF $(name,yob) \notin \mathbf{ran}(BK.customerName \otimes BK.customerYob)$ **THEN**
$result := success \,||\, BK.NewCustomer(name, yob)$
ELSE $result := customerAlreadyPresent$
END
ELSE $result := dbFull$
END
ELSE $result := dbError$
END
END;

Since a machine is only allowed to change its local state, it is imperative that changes to *Bank*'s state are performed using the latter's operations. However, query operations such as *RobustBalance* could be specified directly and one could argue that it is pointless to write query operations in machines which are included in others. If, however, we have convinced ourselves on the level of *Bank* that any access of an account's balance requires the corresponding PIN, this claim is automatically preserved if we only use operations of *Bank* and do not read its variables directly. This approach also facilitates change. Assume that we introduce a log in *Bank* recording all operations and, thereby, transform *RobustBalance* into a state modifying operation. The operation approach does not require any changes on the robust level indicating better modular continuity. However, since in B we specify behaviour and not call-sequences — as in the realm of component software [3] —, we still might have to adapt the implementation of the robust level, if the implementation does not call the same operation.

```
result, nof ← RobustInitFindCustomer(name) =
  PRE name ∈ STRTOKEN THEN
    IF BK.fileOpen = TRUE THEN
      nof ← BK.InitFindCustomer(name) || result := success
    ELSE result := dbError || nof :∈ NAT
    END
  END;
found, yob ← RobustFindNextCustomer =
  IF BK.fileOpen = TRUE THEN found, yob ← BK.FindNextCustomer
  ELSE found := FALSE || yob := 0
  END;
result, number ← RobustNewAccount(name, yob, pin) =
  PRE name ∈ STRTOKEN ∧ yob ∈ NAT ∧ pin ∈ NAT THEN
    IF BK.fileOpen = TRUE THEN
      IF BK.accounts ≠ ACCOUNT THEN
        IF (name,yob) ∈ ran(BK.customerName ⊗ BK.customerYob) THEN
          result := success ||
          number ← BK.NewAccount((BK.customerName ⊗ BK.customerYob) ^-1
              (name, yob), pin)
        ELSE result := unknownCustomer || number :∈ NAT
        END
      ELSE result := dbFull || number :∈ NAT
      END
    ELSE result := dbError || number :∈ NAT
    END
  END;
result, bal ← RobustBalance(number, pin) =
  PRE number ∈ NAT ∧ pin ∈ NAT THEN
    IF BK.fileOpen = TRUE THEN
      IF number ∈ ran(BK.accountNumber) THEN
        IF pin = BK.accountPin(BK.accountNumber ^-1 (number)) THEN
          bal ← BK.Balance(BK.accountNumber ^-1 (number), pin) ||
          result := success
        ELSE result := WrongPin || bal :∈ NAT
```

```
      END
    ELSE result := unknownAccount || bal :∈ NAT
    END
  ELSE result := dbError || bal :∈ NAT
  END
END;
result, name, yob ← RobustOwner(number) =
  PRE number ∈ NAT THEN
    IF BK.fileOpen = TRUE THEN
      IF number ∈ ran(BK.accountNumber) THEN
        name := BK.customerName(BK.accountOwner (BK.accountNumber ^-1
            (number))) ||
        yob := BK.customerYob(BK.accountOwner (BK.accountNumber ^-1
            (number))) ||
        result := success
      ELSE
        result := unknownAccount || name :∈ STRTOKEN || yob :∈ NAT
      END
    ELSE
      result := dbError || name :∈ STRTOKEN || yob :∈ NAT
    END
  END;
result, dd ← RobustDeposit(number, amount) =
  PRE number ∈ NAT ∧ amount ∈ NAT THEN
    IF BK.fileOpen = TRUE THEN
      IF number ∈ ran(BK.accountNumber) THEN
        IF amount > 0 THEN
          IF BK.accountBalance(BK.accountNumber ^-1 (number)) <
              MAXINT - amount THEN
            dd ← BK.Deposit(BK.accountNumber ^-1 (number), amount) ||
            result := success
          ELSE result := amountTooBig || dd :∈ BOOL
          END
        ELSE result := negativeAmount || dd :∈ BOOL
        END
      ELSE result := unknownAccount || dd :∈ BOOL
      END
    ELSE result := dbError || dd :∈ BOOL
    END
  END;
result ← RobustWithdraw(number, pin, amount) =
  PRE number ∈ NAT ∧ pin ∈ NAT ∧ amount ∈ NAT THEN
    IF BK.fileOpen = TRUE THEN
      IF number ∈ ran(BK.accountNumber) THEN
        IF pin = BK.accountPin(BK.accountNumber ^-1 (number)) THEN
          IF amount > 0 THEN
            IF amount ≤ BK.accountBalance(BK.accountNumber ^-1 (number))
                THEN
              BK.Withdraw(BK.accountNumber ^-1 (number), pin, amount) ||
              result := success
            ELSE result := AmountGreaterThanBalance
            END
```

```
            ELSE result := negativeAmount
            END
          ELSE result := WrongPin
          END
        ELSE result := unknownAccount
        END
      ELSE result := dbError
      END
    END;
  result ← RobustChangePin(number, pin, newPin) =
    PRE number ∈ NAT ∧ pin ∈ NAT ∧ newPin ∈ NAT THEN
      IF BK.fileOpen = TRUE THEN
        IF number ∈ ran(BK.accountNumber) THEN
          IF pin = BK.accountPin(BK.accountNumber ⁻¹ (number)) THEN
            BK.ChangePin(BK.accountNumber ⁻¹ (number), pin, newPin) ||
            result := success
          ELSE result := WrongPin
          END
        ELSE result := unknownAccount
        END
      ELSE result := dbError
      END
    END;
  status ← RobustOpen(customerFileName, accountFileName, stringFileName) =
    PRE
      customerFileName ∈ STRING ∧ accountFileName ∈ STRING ∧
      stringFileName ∈ STRING
    THEN
      IF BK.fileOpen = FALSE THEN
        status ← BK.Open(customerFileName, accountFileName, stringFileName)
      ELSE status := FALSE
      END
    END;
  status ← RobustClose =
    IF BK.fileOpen = TRUE THEN
      status ← BK.Close
    ELSE status := FALSE
    END
END
```

4.7 Base Machines

Before we can build a graphical user interface on top of the robust abstraction, we need to build support for the desired input and output mechanisms. A program consists of two parts: computation and interaction with the environment. The algorithmic aspects of a program can be expressed in B, whereas the input and output must be coded in a traditional language. B does not contain direct language support for

communication with the environment, because input and output is very much dependent on the target architecture (Web, X Windows, disk, audio, etc.).

The B development can be interfaced in two ways with its environment: using base machines or using a main program written in a classical programming language which calls the B development. A base machine is a machine the implementation of which is written in a classical programming language rather than in B. A specification of the desired functionality is given as a regular B machine so that it can be used by other B constructs. The actual implementation, not being expressible in B, is programmed directly in the desired classical language, for example, C or Ada. The alternative approach is to use B to create a service subsystem, a subroutine library, and write the main program which interfaces with the environment and calls the B subsystem in a classical programming language. The two approaches can also be combined, for example, we could write a base machine for file access and still write the main program interfacing with the Web in C. In fact, since only scalars and one-dimensional array are implementable directly in B0 and all other data structures use library machines, which in turn are built on base machines, few interesting developments are possible without base machines at all.

We decided to use base machines rather than writing the main program directly in a classical programming language. Base machines can be reused for other developments. From this perspective, it would be logical to have a standard library of base machines. However, the typical domain of B being embedded systems with custom interfaces, such a library would not be generally usable. Nevertheless, it would be desirable to have for educational purposes.

In many industrial applications, especially in those that build on existing components, B is only used to create the most safety-critical algorithmic part in the middle, building on well-tested databases for persistent storage and complex graphical user interfaces. This often suitable compromise requires a great amount of discipline to be exercised to avoid parts of the algorithm being expressed outside B. We have chosen the all-B approach to illustrate its feasibility.

4.7.1 Strings in Atelier B

Atelier B has a type *STRING* for constant character chains. *STRING* can be used for passing a message like “Hello world” to a terminal output machine or, in our case, to pass the names of the dump files. However, there is no support for non-literal strings as needed for customers’ names. Atelier B does not permit objects of variable length, such as strings, to be passed between operations. Because there is no support for constant-length strings either, we are forced to either use tokens as references to the actual strings, which are stored in a base machine, or pass strings character-by-character with multiple calls. We opt for tokens. Machine *StrTokenType* defines a type of string tokens.

MACHINE
StrTokenType
SETS

STRTOKEN
CONCRETE_CONSTANTS
EmptyStringToken
PROPERTIES
EmptyStringToken ∈ *STRTOKEN*
END

Note that the set *STRTOKEN* is abstract. Therefore, normal B machines cannot simply 'create new string tokens' as would have been the case if we had used a subset of the *NAT* instead. The fact that *STRTOKEN* is valued to a subset of *NAT* in the implementation only helps the C-translator, but cannot be exploited in constructs which see or import *StrTokenType*.

Since string tokens can be compared with '=', we need to have an injection from tokens to strings. To ensure this, only one single machine called *BasicString* is allowed to generate tokens. Input base machines return tokens, not strings. Fig. 4.8 (left side) illustrates string I/O, with *BasicCGI* as an example of an I/O machine. Implementation *MainBank_1* requests a string to be input. *BasicCGI* reads a string from the Web, enters the string in *BasicString* and in return receives a token, which it returns to its client *MainBank_1*. Note that the operations for entering new strings and retrieving strings by token are not specified on the B level, but are only present in the hand-coded C implementation.

The rest of this subsection discusses additional aspects of passing objects of variable size in B. The material is of general interest, but is not necessary for understanding the case study. Hence, it can be skipped on a first reading.

Unfortunately, the token solution has a shortcoming: We cannot ensure in B that no other base machine generates tokens. For example a random base machine could have a machine parameter of set type and provide an operation which returns arbitrary elements of that set. Instanciated with *STRTOKEN*, this machine could generate tokens for which *BasicString* has no corresponding string. We must also ensure that whenever string tokens are externalised, the corresponding strings are also saved.

The obvious, but for other reasons undesirable, remedy to the first problem would be to introduce a set *legalTokens* ⊆ *STRTOKEN* in *BasicString*. Any input operation would then have to modify *legalTokens*. However, only the constructs that includes/imports *BasicString*, but no others that only see *BasicString*, have access to state modifying operations of *BasicString*.[1] As a consequence, input from any source would have to be implemented in a single base machine, contradicting modularity. For example, base machines for input from the Web and from a terminal could not simply be combined by importing both, but would have to be textually merged.

[1] This single writer and multiple readers restriction is due to the visibility of variables of included/imported machines in the invariant of the including/importing implementation. Multiple writers could invalidate each other's invariants.

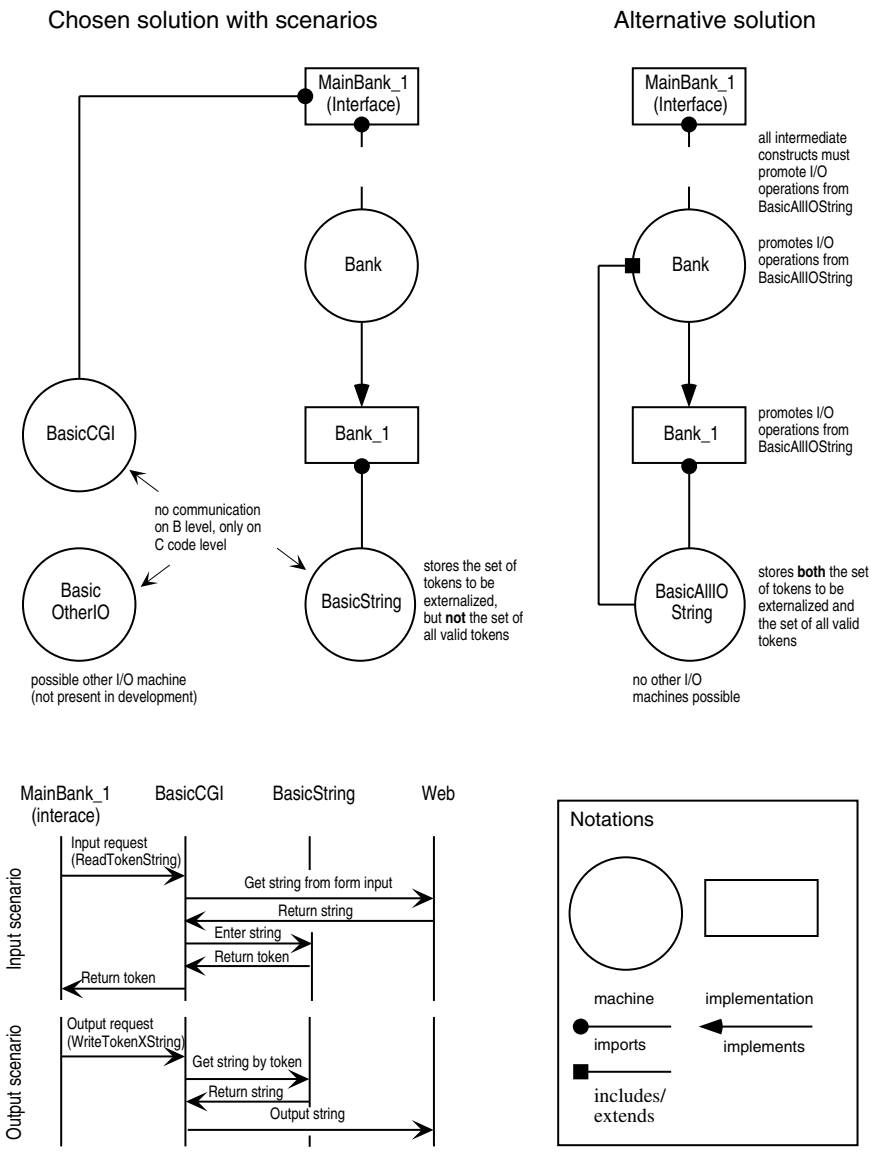


Fig. 4.8. Alternatives for Input/Output and String Storage

The single-writer restriction would complicate the design even if we would limit ourselves to a single input/output (I/O) machine. If we would not want to externalise all strings, but only a selected subset (the names of the customers) that we need again in future program runs, then implementation *Bank_1* would also need write access to *BasicString*'s state because *Bank_1* would have to control the externalisation process. All components accessing *BasicString* in write mode would have to be parents in a straight line, each imported by the next. Hence, the single I/O machine would have to import *BasicString*, respectively be merged into a single machine to also avoid the behind the scene passing of strings. Implementation *Bank_1* would then have to import this machine *BasicAllIOString*. The real inelegance would be that the I/O operations which are accessed from the interface layer would have to be promoted by the specifications *Bank* and *RobustBank*. A similar pollution of the specifications of *Bank* would occur if externalisation of strings were to be controlled by the interface layer and *Bank* would have to provide operations to query the set of strings to be externalised.

Because of the need to combine all I/O into a single I/O machine and the cluttering of specifications with implementation aspects, we do not choose this solution. Rather we accept that we cannot maintain in B a set of all valid tokens. Fig. 4.8 illustrates the two alternatives. The specification of *BasicString* is given on page 161.

4.7.2 Machine *BasicCGI*

In order to input and output data to the Web, we need a machine to access the common gateway interface (CGI), which we call *BasicCGI*. CGI is a standard for interfacing external applications with information servers, such as Web servers. A plain hypertext markup language (HTML) document that the Web daemon retrieves is static, which means it exists in a constant state: a text file that doesn't change. A CGI program, on the other hand, is executed in real-time, so that it can output dynamic information. The user fills out a form in the browser and sends the data to the server which executes the CGI program. The CGI program processes the input, modifies the local database, and generates an output which is sent back to the user's browser for display.

MACHINE
BasicCGI

SEES
StrTokenType

OPERATIONS

status, *num* ← **ReadNat**(*name*) =
PRE *name* ∈ *STRING* **THEN**
status :∈ **BOOL** || *num* :∈ **NAT**
END;

In an HTML form every field has a unique name. Operation *ReadNat* inputs a natural number value of a field, designated by its name, from a form. Since the user can enter an arbitrary number into a given field, we can only assure that *num* is a natural number. The browser, the server, and the connection between them being outside the realm of our specification, we cannot specify that the reported value is actually the one entered by the user. An implementation which always returns 0, independently of the users input would, therefore, be formally correct. Neither can we specify under which circumstances the result code indicates success. Actually, an implementation which always fails would also be correct. The intended meaning of the operation is only captured by its name and the natural language description. The only property guaranteed by the formal specification is termination.

Whether we use result codes or not depends upon how we can react to failure. Consider, for example, a measuring device with an input sensor and a disk to store the values as its only output device. If the disk fails, we can also stop execution. In this case an abstraction specifying the disk as reliable leads to a simpler system. Alternatively, we might specify the disk as unreliable, but simply ignore the result codes in the higher layers, leading to unprovable obligations. On the other hand, if we can react to failure by, for example, storing the current state on a spare disk and showing an error message on the screen, return codes are desirable. In non safety-critical systems, operations with a very high success probability are often assumed to be fully reliable, because little can be done in case of failure and the resulting system is much simpler.

To be more precise, the return codes in our example indicate whether the Web server has indicated an error or not. If, for example, the underlying hardware has malfunctioned in a way not traced by the operating system or Web server, for example, a communication error resulting in a correct checksum, the error goes unnoticed. Building up a system from components, we specify each component separately and reason about the whole system using composition rules assuming the implementations to adhere to the specification. If a specification is too weak, the corresponding component cannot be used intelligently. Although more truthful, a specification saying that the CGI functions might have failed even if the result indicates success, is useless, because we cannot build on it. Risk estimates using probabilistic reasoning would need to complement a development in B [24, 15, 16].

Operation *ReadTokenString* reads a string from a form field. As described above, the string is stored in machine *BasicString* and only a token is returned. If the string contained in the field is longer than *maxLength*, the operations returns failure.

status, *str* ← **ReadTokenString**(*name*, *maxLength*) =
  PRE *name* ∈ *STRING* ∧ *maxLength* ∈ **NAT THEN**
    status :∈ **BOOL** || *str* :∈ *STRTOKEN*
  END;

The remaining five operations are concerned with outputting a new document in response to the user's request. Each document has a MIME (Multipurpose Internet Message Extension) type which tells the browser the format of the remaining data

stream. In our case, the type is always “text/html”. Operation *WriteLiteralContentType* lets us send the MIME type to the browser. Parameter *mimeType* is of type *STRING* as a constant literal string is envisaged to be used as an actual parameter.

WriteLiteralContentType(*mimeType*) =
  PRE *mimeType* ∈ *STRING* **THEN skip END**;

In HTML, certain characters such as ‘<’ are reserved for markup purposes. Additionally, 8-bit characters must be encoded using either their mnemonic or their decimal codes in the Latin-1 character set. For example, the letter ‘ü’ can be encoded as either ‘ü’ or as ‘ ü’. Operation *WriteLiteralString* outputs a string without any conversions; hence, the string can contain HTML tags, but special characters must already be encoded. Operation *WriteLatin1TokenString* converts a string from the Latin-1 character set to its HTML encoding.

WriteLiteralString(*str*) =
  PRE *str* ∈ *STRING* **THEN skip END**;
WriteLatin1TokenString(*str*) =
  PRE *str* ∈ *STRTOKEN* **THEN skip END**;

In arguments to CGI programs, certain reserved characters as well as 8-bit characters must be encoded as their hexadecimal codes in the Latin-1 character set. The letter ‘ü’, for example, is represented as ‘%FC’. Since such argument strings may not contain any spaces, the latter are converted to ‘+’s. This type of conversion is performed by operation *WriteURLString* before outputting its argument.

WriteURLTokenString(*str*) =
  PRE *str* ∈ *STRTOKEN* **THEN skip END**;
WriteNat(*num*) =
  PRE *num* ∈ **NAT THEN skip END**
END

The actual output operations are specified as skip as the output is not part of the state captured by the B specification. Although the output operations can also fail in practice, we have chosen the less safe, but more convenient approach of specifying them as reliable.

A partial modelling of the output would also have practical consequences. The operations of *BasicCGI* might be called from different implementation constructs. As long as the operations are inquiry operations they can be called from implementations which see *BasicCGI*. If, on the other hand, the output operations modify the state, the lowest machine in the hierarchy using *BasicCGI* must import the latter and promote the operations.

Machine *BasicCGI* does not enforce its output to be correct HTML, for example, there is no check for matching markup tags. Although desirable, such checks would make the machine much more cumbersome to use as tags could not be embedded in strings and the machine would have to be updated to use new HTML tags.

4.7.3 Implementing *BasicCGI*

To implement *BasicCGI* we first write an 'empty' B implementation the C translation of which gives us a C code skeleton conforming to the coding standards of Atelier B's translator. This skeleton is then filled in with the actual code. The implementation *BasicCGI_1* contains only the minimal information to conform to B and be translatable. We have to value every set and constant, initialize concrete variables of the specification, and list all the operations. Operations are simply specified as skip, if they have no return parameters and otherwise as dummy assignments to the return parameters. We do not prove anything about this empty implementation. Note that *BasicCGI_1* sees *BasicString* to force the latter being imported somewhere in the development.

```
IMPLEMENTATION
  BasicCGI_1
REFINES
  BasicCGI
SEES
  StrTokenType, BasicString
OPERATIONS
  status, num ← ReadNat(name) =
    BEGIN
      status := TRUE; num := 0
    END;
  status, str ← ReadTokenString(name, maxLength) =
    BEGIN
      status := TRUE; str := EmptyStringToken
    END;
  WriteLiteralContentType(mimeType) = skip;
  WriteLiteralString(str) = skip;
  WriteURLTokenString(str) = skip;
  WriteLatin1TokenString(str) = skip;
  WriteNat(num) = skip
END
```

Rather than implementing CGI access from scratch we build upon the public domain ANSI C library cgic version 1.05 from Thomas Boutell [2]. This library provides for comfortable parsing of form input. The second included header file *trad_ctx.h* defines some macros such as *PROTx* to make the source code portable between ANSI C and K&R compilers.

In a project, a machine can be imported several times with different instance names. Different instances represent different data. Implementing a base machine, we have to decide whether multiple instantiation is permitted or not. If, for example, a base machine represents a physical device such as an LED only one copy of the corresponding base machine should be included in a development. If a base machine

does not allow for multiple instantiations, we have to verify that the project adheres to this rule. The restriction cannot be expressed in AMN; depending upon the target language and the translator it is possible to write C code which fails to compile, respectively link if the rule is violated. If, as in our case, this is not possible, manual inspection is necessary. On the other hand, if we allow multiple instantiations, the state of an instantiation must be included into the struct *BasicX_type*. As discussed above, we do not need to make our machine *BasicCGI* instanciable, even if we use it from more than one implementation construct. Hence, we opt for this simpler approach which also corresponds more closely to the reality we model. Our third base machine *BasicFile* (Sect. 4.10.4) illustrates multiple instantiation. The hand-coded additions and modifications are set in italics in the C source files.

```
#include"cgic.h"
#ifndef trad_ctx_include_def
   #include "trad_ctx.h"
#endif

/* Links to machines from the SEES clause */
#ifndef StrTokenType_include_def
   #include "StrTokenType.h"
#endif

/* Structure associated to component (instance record) */
struct BasicCGI_type {
   int BasicCGI_init_already_done;
} ;

#define BasicCGI_include_def

/* Reference to machines from the SEES clause */
EXTERN struct StrTokenType_type *StrTokenType_ptr;

/* Prototypes of translated operations */
EXTERN void link_BasicCGI PROTF((struct BasicCGI_type *v));

EXTERN void init_BasicCGI PROTF((struct BasicCGI_type *v));

/* Type of name changed manually from INT32 to char* */
EXTERN void ReadNat_BasicCGI PROTF((struct BasicCGI_type *v,
      INT32 *status, INT32 *num, char *name));

/* The other operations can be found on the book's Web page. */
```

In its original implementation, cgic provides itself a main function and expects the user to write a function called *cgiMain* which is called after initialisation. By changing a handful of lines as indicated in the online source code, we turn cgic's *main* function into a function *cgiInit* which we call from *init_BasicCGI*. The specifications does not allow the initialisation to fail. In practice, if the initialisation fails we write a message to *stderr* and abort execution. Since we cannot perform any transaction anyhow, abortion at startup is the simplest solution. The operations

are simply calls to the corresponding procedures of cgic, respectively *fprintf* commands.

```
#include <stdio.h>
#include "BasicCGI.h"
#include "BasicString.h"

void link_BasicCGI(PROTA(struct BasicCGI_type *)v)
     PROTC(struct BasicCGI_type *v)
{}

void init_BasicCGI(PROTA(struct BasicCGI_type *)v)
     PROTC(struct BasicCGI_type *v)
{
  if (StrTokenType_ptr->StrTokenType_init_already_done &&
       (v->BasicCGI_init_already_done==0)) {
     if(cgiInit()!=0){
       fprintf(stderr, "Initialization of BasicCGI failed.\n"); exit(-1);
     }
     v->BasicCGI_init_already_done=1;
  }
}


void ReadNat_BasicCGI(PROTA(struct BasicCGI_type *)v, PROTA(INT32 *)status,
       PROTA(INT32 *)num, PROTA(char *)name)
     PROTC(struct BasicCGI_type *v) PROTC( INT32 *status)
       PROTC(INT32 *num) PROTC(char *name)
{
  int s;

  s=cgiFormInteger(name, num, 0);
  if((s==0)&&(*num>=0)) {
     *status=TRUE;
  } else {
     *num=0; *status=FALSE;
  }
}

/* The other operations can be found on the book's Web page. */
```

We prove in B that all calls to operations of *BasicCGI* satisfy the respective preconditions. Hence, there is no need to write checks for the preconditions in the C code of *BasicCGI*. The hand-coded C implementation is a refinement of its B specification. The validity of the refinement has to be asserted using normal verification techniques, for example, testing and third party code inspection.

We make a separate project out of *BasicCGI*, *BasicString*, and *StrTokenType* to facilitate reuse in other projects. This also prevents us from accidentally overwriting the hand-coded implementation. The files *cgic.c* and *cgic.h* must be manually added to the Makefile, copied from the data base to the code directory. Additionally, the

target *BasicCGI* must be removed from the Makefile, as we only want to create a library and make would produce an error because of the missing *main* function.

For didactic reasons, we have presented the implementation of the base machine directly following its specification. In practice, we often write the implementation only after we have actually used its specification in other constructs and, thereby, convinced ourselves of its appropriateness. The disadvantage of this is that the specification might not be implementable on the target system, causing a rework of all dependent constructs.

4.8 User Interface

The user interface presents an entry mask to the user, parses the input with the help of *BasicCGI*, sends the request to the robust interface *RobustBank*, and presents the results using again the CGI machine. We first prototype this interaction using static HTML code with normal links between the pages rather than calls to our CGI application. Once we are satisfied with the look and feel, we write the user interface

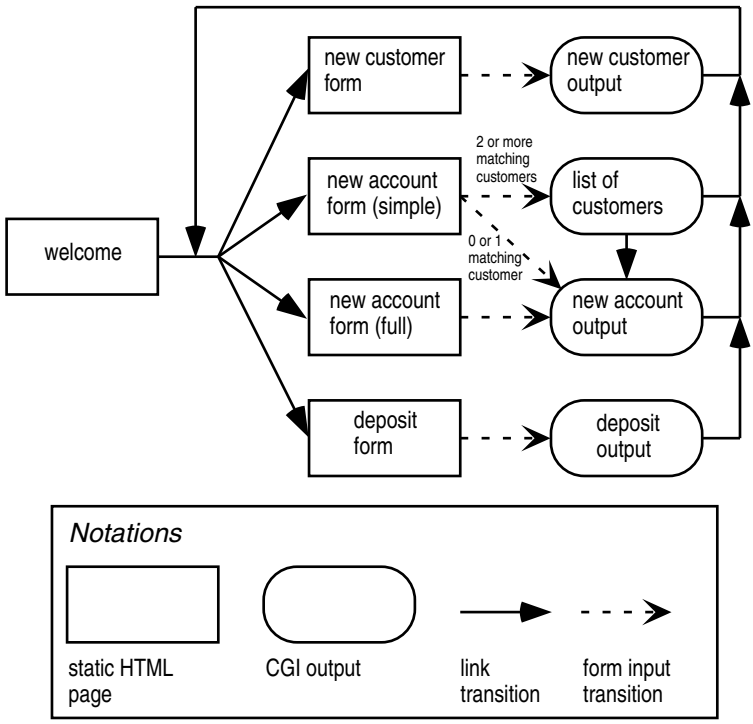


Fig. 4.9. Cashier Interaction

which generates the same HTML code based upon CGI requests. Static information, such as the input forms, remains in the form of normal HTML files.

The cashier is presented with a menu on the bottom of her terminal, from which she can choose a form to enter a new customer, create a new account for an existing customer, or make a deposit. In the 'new customer' form, the cashier enters the name and year of birth of the customer and clicks on a button to send the data to the CGI application. In response, the cashier gets a screen saying that the operation has succeeded or that an error has occurred. These messages are all generated by the same CGI program, but for prototyping we need to create different HTML pages. After reading the output message, the cashier clicks on another menu choice.

When creating a new account, the cashier has the option of entering both the customer's name and year of birth or only the name. If there is only one customer with the given name in the system, an account is created. On the other hand, if there is more than one customer with this name, the cashier is presented with a list. She then simply clicks on the desired customer to create the account. In the latter case, the links contain all the parameters, which would usually be entered into the form by the cashier. For example for customer 'Garfield', born in '1978', and PIN '2001' the URL of the link would be 'http://.../cgi-bin/AB/MainBank?command=1&name=Garfield&yob=1978&pin=2001'. The CGI

```
<HTML>
  <HEAD><TITLE>B Bank: New Customer</TITLE></HEAD>
  <BODY BGCOLOR="#228B22">
    <H1>B Bank: New Customer</H1>
    <FORM ACTION="http://www.tucs.abo.fi/cgi-bin/mbuechi/AB/MainBank"
        METHOD="POST">
      <INPUT TYPE="HIDDEN" NAME="command" VALUE="0">
      <TABLE BORDER="0">
        <TR ALIGN="Center" VALIGN="Middle">
          <TD ALIGN="RIGHT">Customer name:</TD>
          <TD ALIGN="LEFT"><INPUT NAME="name" SIZE="18"></TD>
        </TR>
        <TR ALIGN="Center" VALIGN="Middle">
          <TD ALIGN="RIGHT">Year of birth:</TD>
          <TD ALIGN="LEFT"><INPUT NAME="yob" SIZE="4"></TD>
        </TR>
      </TABLE>
      <P>
        <INPUT TYPE="submit" VALUE="Add customer"><BR>
        <INPUT TYPE="reset" VALUE="Reset input form">
      </P>
    </FORM>
  </BODY>
</HTML>
```

Fig. 4.10. HTML Source Code of 'New Customer' Form

program doesn't have to store any temporary information. 'Deposit' leads to simple one-step interaction sequences like 'new customer', as depicted in Fig. 4.9.

For brevity's sake, we do not list all the HTML pages. We assume the reader to be familiar with basic HTML. In Fig. 4.10, the *FORM* tag introduces the actual entry form. Its attribute *ACTION* states the URL of the CGI program, to which the input data is sent upon pressing the submit button. The input field 'name' takes the customer name. Rather than creating a separate CGI application for each entry form, we use a hidden input field 'command' which selects the desired operation. The CGI program is our final B applications, which we copy to the CGI directory of the Webserver and give the suitable execution rights.

The user interaction at the ATM and the corresponding HTML pages are similar. On a standard ATM, the account number is read from a card. To run our simulation without any special hardware, the user is also requested to enter the account number. A typical ATM interface is modal, that is, one first inserts the card, then enters the PIN, and finally performs the desired transaction. In our simulation, the user is requested to enter all information in a single modeless dialog. Exercise 4.2 shows how to model a modal interface using the idea of links generated by the program.

In order to make navigation easier in the simulation, we add a frame set with a meta menu which lets us easily switch between the cashier terminal and the ATM, displayed with different background colour in the right-hand side frame.

4.8.1 Main Program

To keep the size of the individual operations small, we create one operation per transaction type. Since in B operations from the same construct cannot be called, we divide the user interface into two machines. Machine *MainBank* contains the main program. It reads the 'command' field and calls the selected operation of machine *OperationsBank*, which does the actual work.

We do not duplicate the state on the user interface level in *OperationsBank*, as we do not want to perform any reasoning. Hence, the specification of the transaction operations is simply skip.

MACHINE
OperationsBank
OPERATIONS

NewCustomer = skip;
NewAccount = skip;
Deposit = skip;
Withdraw = skip;
Balance = skip;
ChangePin = skip;
Error(*number*) =
PRE *number* ∈ **NAT THEN skip END**;

```
  status ← Open(customerFileName, accountFileName, stringFileName) =
    PRE
      customerFileName ∈ STRING ∧ accountFileName ∈ STRING ∧
      stringFileName ∈ STRING
    THEN
      status :∈ BOOL
    END;
  status ← Close =
    BEGIN
      status :∈ BOOL
    END
END
```

The machine *MainBank* is also stateless. The specification of its single operation *main* is skip, guaranteeing only termination. Since the persistent state, existing beyond a single program run, cannot be modelled, skip is in fact the only reasonable specification for a main program.

```
MACHINE
  MainBank

OPERATIONS
  main = skip
END
```

4.8.2 Implementations

The implementation *MainBank_1* first opens the database. Then it reads the value of the 'command' input field, calls the selected operation, and closes the database.

```
IMPLEMENTATION
  MainBank_1

REFINES
  MainBank

IMPORTS
  BC.BasicCGI, OB.OperationsBank , StrTokenType

OPERATIONS
  main =
    VAR dbst, st, res IN
      dbst ← OB.Open("/tmp/customer", "/tmp/account", "/tmp/strings");
      IF dbst = TRUE THEN
        st, res ← BC.ReadNat("command");
        IF st = TRUE THEN
          CASE res OF
            EITHER 0 THEN OB.NewCustomer
            OR 1 THEN OB.NewAccount
```

```
            OR 2 THEN OB.Deposit
            OR 3 THEN OB.Withdraw
            OR 4 THEN OB.Balance
            OR 5 THEN OB.ChangePin
            ELSE OB.Error(0)
            END
          END
        ELSE OB.Error(1)
        END;
        dbst ← OB.Close
      ELSE OB.Error(2)
      END
    END
END
```

The implementation *OperationsBank_1* imports *RobustBank*. The operation *NewCustomer* first outputs the header of the result screen, which is independent of the outcome of the operation. Then it reads the value of the 'name' field, calls *RobustNewCustomer* and presents the result.

The loop in operation *NewAccount* shows the advantage of not just using B to create a subroutine library. In this case, loops on the user interface level would not be proved to terminate. For brevity's sake, some operations are omitted in the listing below. They can, as all other constructs, be found on the book's Web page.

```
IMPLEMENTATION
  OperationsBank_1

REFINES
  OperationsBank

IMPORTS
  RB.RobustBank(100, 200)

SEES
  BC.BasicCGI

CONCRETE_CONSTANTS
  False1

PROPERTIES
  False1 ∈ BOOL ⤖ NAT

VALUES
  False1 = {(TRUE ↦ 0), (FALSE ↦ 1)}

DEFINITIONS
  CASHIER_HEADER(title) == HEADER(title, "#228B22");
  ATM_HEADER(title) == HEADER(title, "#DC143C");
  HEADER(title,color) == (
    BC.WriteLiteralContentType("text/html");
    BC.WriteLiteralString("<HTML>\n<HEAD><TITLE>B Bank: ");
    BC.WriteLiteralString(title); BC.WriteLiteralString("</TITLE></HEAD>\n");
    BC.WriteLiteralString("<BODY BGCOLOR="); BC.WriteLiteralString(color);
```

```
    BC.WriteLiteralString(">\n<H1>B Bank: ");
    BC.WriteLiteralString(title); BC.WriteLiteralString("</H1>\n"));
  FOOTER == BC.WriteLiteralString("</BODY></HTML>\n");
  DB_FULL_MSG == BC.WriteLiteralString("<P>Sorry. The database is full.</P>");
  DB_ERR_MSG ==
    BC.WriteLiteralString("<P>Sorry. The databse is not working.</P>");
  UNK_ACC_MSG(num) == (
    BC.WriteLiteralString("<P>Account "); BC.WriteNat(num);
    BC.WriteLiteralString(" is not in database.</P>"));
  CGI_SCRIPT == "http://www.tucs.abo.fi/cgi-bin/mbuechi/AB/MainBank";
  MAX_NAME_LENGTH == 256

OPERATIONS
  NewCustomer =
    VAR st, name, yob, result IN
      CASHIER_HEADER("New Customer");
      st, name ← BC.ReadTokenString("name", MAX_NAME_LENGTH);
      IF st = TRUE THEN
        st, yob ← BC.ReadNat("yob");
        IF st = TRUE THEN
          result ← RB.RobustNewCustomer(name, yob);
          CASE result OF
            EITHER success THEN
              BC.WriteLiteralString("<P>Customer ");
              BC.WriteLatin1TokenString(name);
              BC.WriteLiteralString(" ("); BC.WriteNat(yob);
              BC.WriteLiteralString(") has been added.</P>")
            OR customerAlreadyPresent THEN
              BC.WriteLiteralString("<P>Customer ");
              BC.WriteLatin1TokenString(name);
              BC.WriteLiteralString(" ("); BC.WriteNat(yob);
              BC.WriteLiteralString(") is already in database.</P>")
            OR dbFull THEN DB_FULL_MSG
            OR dbError THEN DB_ERR_MSG
            END
          END
        ELSE BC.WriteLiteralString("<P>Could not get year of birth.</P>")
        END
      ELSE BC.WriteLiteralString("<P>Could not get name.</P>")
      END;
      FOOTER
    END;
  NewAccount =
    VAR st, name, yob, pin, result, number, nof, found, ii IN
      CASHIER_HEADER("New Account");
      st, name ← BC.ReadTokenString("name", MAX_NAME_LENGTH);
      IF st = TRUE THEN
        st, pin ← BC.ReadNat("pin");
        IF st = TRUE THEN
          st, yob ← BC.ReadNat("yob");
          IF st = FALSE THEN
            result, nof ← RB.RobustInitFindCustomer(name);
            IF result = success THEN
```

```
    IF nof = 0 THEN
      BC.WriteLiteralString("<P>Customer ");
      BC.WriteLatin1TokenString(name);
      BC.WriteLiteralString(" is not in database.</P>")
    ELSIF nof = 1 THEN
      found, yob ← RB.RobustFindNextCustomer;
      st := TRUE
    ELSE BC.WriteLiteralString("<P>Choose from list:</P><UL>");
      ii := 0; found, yob ← RB.RobustFindNextCustomer;
      WHILE found = TRUE DO
        BC.WriteLiteralString("<LI><A HREF=");
        BC.WriteLiteralString(CGI_SCRIPT);
        BC.WriteLiteralString("?command=1&name=");
        BC.WriteURLTokenString(name);
        BC.WriteLiteralString("&yob=");
        BC.WriteNat(yob);
        BC.WriteLiteralString("&pin=");
        BC.WriteNat(pin);
        BC.WriteLiteralString(">");
        BC.WriteLatin1TokenString(name); BC.WriteLiteralString(" (");
        BC.WriteNat(yob); BC.WriteLiteralString(")</A></L>");
        found, yob ← RB.RobustFindNextCustomer
      INVARIANT
        yob ∈ NAT ∧
        RB.BK.foundCustomers ∈ 𝔽 (RB.BK.foundCustomers)
      VARIANT
        card(RB.BK.foundCustomers)+1-False1(found)
      END;
      BC.WriteLiteralString("</UL>")
    END
  ELSE DB_ERR_MSG
  END
END;
IF st = TRUE THEN
  result, number ← RB.RobustNewAccount(name, yob, pin);
  CASE result OF
    EITHER success THEN
      BC.WriteLiteralString("<P>New account number ");
      BC.WriteNat(number);
      BC.WriteLiteralString(" has been created for customer ");
      BC.WriteLatin1TokenString(name);
      BC.WriteLiteralString(" ("); BC.WriteNat(yob);
      BC.WriteLiteralString(").</P>")
    OR unknownCustomer THEN
      BC.WriteLiteralString("<P>Customer ");
      BC.WriteLatin1TokenString(name);
      BC.WriteLiteralString(" ("); BC.WriteNat(yob);
      BC.WriteLiteralString(") is not in database.</P>")
    OR dbFull THEN DB_FULL_MSG
    OR dbError THEN DB_ERR_MSG
    END
  END
END
```

```
        ELSE BC.WriteLiteralString("<P>Could not get pin.</P>")
        END
      ELSE BC.WriteLiteralString("<P>Could not get name.</P>")
      END;
    FOOTER
  END;
  Deposit =
    VAR st, number, amount, result, dd, name, yob IN
      CASHIER_HEADER("Deposit");
      st, number ← BC.ReadNat("number");
      IF st = TRUE THEN
        st, amount ← BC.ReadNat("amount");
        IF st = TRUE THEN
          result, dd ← RB.RobustDeposit(number, amount);
          CASE result OF
            EITHER success THEN
              BC.WriteLiteralString("<P>A deposit of ");
              BC.WriteNat(amount);
              BC.WriteLiteralString(" has been made on account ");
              BC.WriteNat(number);
              result, name, yob ← RB.RobustOwner(number);
              IF result = success THEN
                BC.WriteLiteralString(" belonging to ");
                BC.WriteLatin1TokenString(name);
                BC.WriteLiteralString(" ("); BC.WriteNat(yob);
                BC.WriteLiteralString(")")
              END;
              BC.WriteLiteralString(".</P>")
            OR negativeAmount THEN
              BC.WriteLiteralString("<P>Amount must be greater than 0.</P>")
            OR amountTooBig THEN
              BC.WriteLiteralString("<P>Amount too big. ")
              BC.WriteLiteralString("No deposit has been made.</P>")
            OR unknownAccount THEN UNK_ACC_MSG(number)
            OR dbError THEN DB_ERR_MSG
            END
          END
        ELSE BC.WriteLiteralString("<P>Could not get amount.</P>")
        END
      ELSE BC.WriteLiteralString("<P>Could not get number.</P>")
      END;
      FOOTER
    END;
/* Operations Withdraw, Balance, and ChangePin and Error omitted. Check the book's Web
page. */
  status ← Open(customerFileName, accountFileName, stringFileName) =
    status ← RB.RobustOpen(customerFileName, accountFileName, stringFileName);
  status ← Close =
    status ← RB.RobustClose
END
```

4.9 Implementation of the Robust Abstraction

The missing piece is the implementation of the robust layer *RobustBank*. We have to make a choice as to whether we want to import *Bank* in the implementation *RobustBank_1* or whether we want to build directly on lower level abstractions.

Often, a more abstract specification is included into the robust level and a similar, more concrete specification is imported in the implementation. The machine that is included in the specification should be as abstract as possible to avoid overspecification. The machine that is imported in the implementation should be quite concrete to make it more useful. The use of two different constructs solves this dilemma. However, in our case we can include, respectively import the same ma-

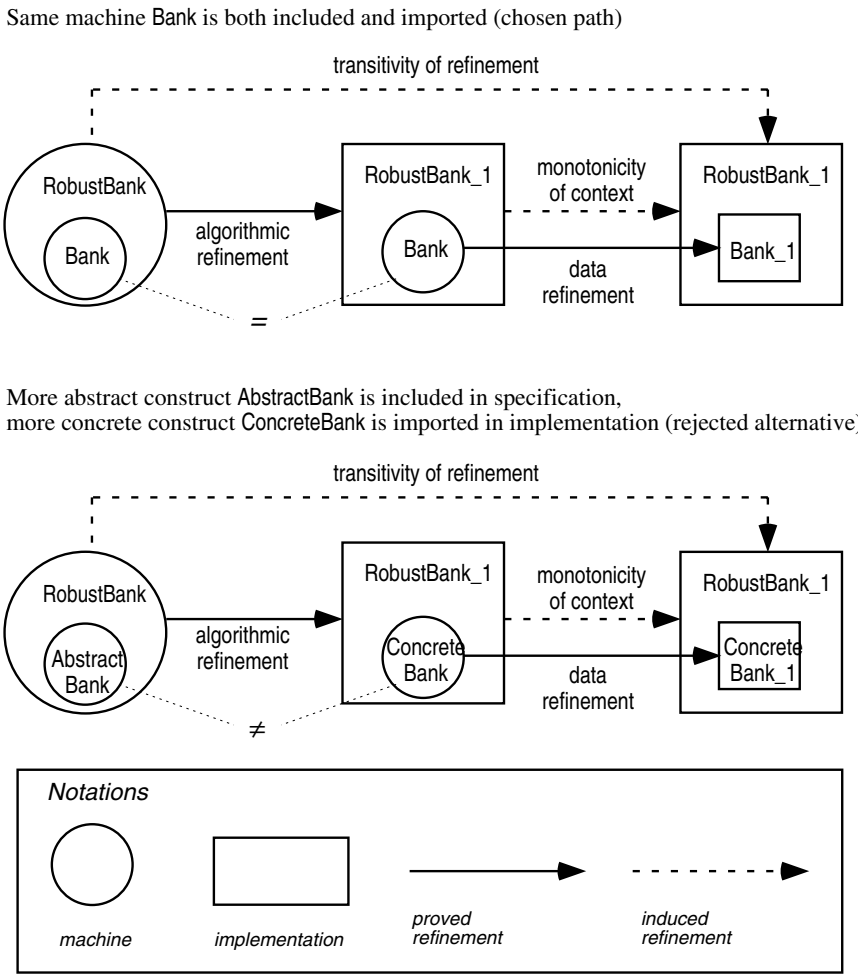


Fig. 4.11. Import of Included Machine vs. Import of More Concrete Construct

chine *Bank* in both the specification and the implementation, avoiding a proliferation of constructs. In the alternative case *Bank*, respectively a more abstract version *AbstractBank*, would have been used only in the specification, but would not have to be refined to an implementation. Fig. 4.11 shows the two options.

Importing an already included machine without renaming, respectively renaming it identically both times, constitutes an algorithmic refinement. The identity mapping invariant is implicitly added.

The operations first check whether the parameters and the current state satisfy the preconditions of the corresponding operations in *Bank* and then call them, or report an error if the conditions do not hold.

```
IMPLEMENTATION
  RobustBank_1(maxCustomers, maxAccounts)

REFINES
  RobustBank

IMPORTS
  BK.Bank(maxCustomers, maxAccounts)

SEES
  StrTokenType

OPERATIONS

  result ← RobustNewCustomer(name,yob) =
    VAR status, cid IN
      IF BK.fileOpen = TRUE THEN
        status ← BK.CustomerDBFull;
        IF status = FALSE THEN
          status, cid ← BK.ThisCustomer(name,yob);
          IF status = FALSE THEN
            BK.NewCustomer(name,yob); result := success
          ELSE result := customerAlreadyPresent
          END
        ELSE result := dbFull
        END
      ELSE result := dbError
      END
    END;
  result, nof ← RobustInitFindCustomer(name) =
    IF BK.fileOpen = TRUE THEN
      nof ← BK.InitFindCustomer(name); result := success
    ELSE
      result := dbError; nof := 0
    END;
  found, yob ← RobustFindNextCustomer =
    IF BK.fileOpen = TRUE THEN
      found, yob ← BK.FindNextCustomer
    ELSE
      found := FALSE; yob := 0
    END;
```

```
result, number ← RobustNewAccount(name, yob, pin) =
  VAR status, cid IN
    number := 0;
    IF BK.fileOpen = TRUE THEN
      status ← BK.AccountDBFull;
      IF status = FALSE THEN
        status, cid ← BK.ThisCustomer(name, yob);
        IF status = TRUE THEN
          number ← BK.NewAccount(cid, pin); result := success
        ELSE result := unknownCustomer
        END
      ELSE result := dbFull
      END
    ELSE result := dbError
    END
  END;
result, bal ← RobustBalance(number, pin) =
  VAR status, aid IN
    bal := 0;
    IF BK.fileOpen = TRUE THEN
      status, aid ← BK.ThisAccount(number);
      IF status = TRUE THEN
        status ← BK.Authorized(aid, pin);
        IF status = TRUE THEN
          bal ← BK.Balance(aid, pin); result := success
        ELSE result := WrongPin
        END
      ELSE result := unknownAccount
      END
    ELSE result := dbError
    END
  END;
result, name, yob ← RobustOwner(number) =
  VAR status, aid, cid IN
    yob := 0;
    IF BK.fileOpen = TRUE THEN
      status, aid ← BK.ThisAccount(number);
      IF status = TRUE THEN
        cid ← BK.AccountOwner(aid);
        name, yob ← BK.CustomerData(cid);
        result := success
      ELSE
        result := unknownAccount; name := EmptyStringToken; yob := 0
      END
    ELSE
      result := dbError; name := EmptyStringToken; yob := 0
    END
  END;
/* Operations RobustBalance, RobustOwner, RobustDeposit, RobustWithdraw,
and RobustChangePin omitted. Check on the book's Web page. */
status ← RobustOpen(customerFileName, accountFileName, stringFileName) =
  IF BK.fileOpen = FALSE THEN
```

```
      status ← BK.Open(customerFileName, accountFileName, stringFileName)
    ELSE status := FALSE
    END;
  status ← RobustClose =
    IF BK.fileOpen = TRUE THEN
      status ← BK.Close
    ELSE status := FALSE
    END
END
```

4.10 Implementation of *Bank*

Our next task is to refine *Bank* to an implementation, because we have chosen to import it into *RobustBank*. In Sect. 4.4 we have already outlined the basic structure of this implementation. Now we have to take a closer look at our requirements on one hand and the available resources, that is, the B library machines and the operating system of the target computer, on the other. This is the gap we have to bridge.

The data structures we need to implement are object classes with attributes as well as relations. We need to be able to create new objects, read and modify their attributes, and externalise and internalise them. All our attributes are of types *NAT* and *STRTOKEN*. If we provide a possibility to reference string tokens with natural numbers, strings, respectively references to string tokens can also be stored like *NAT*s. Functional relations (*accountOwner*) can also be modelled as *NAT* attributes if *NAT* is also chosen as the identifier type for objects.

Atelier B provides a base machine *BASIC_ARRAY_RGE* for two dimensional array. This could be used to store objects with their *NAT* attributes by letting the first index select the object and the second the desired attribute or vice versa. If the number of fields is known, we could alternatively use a number of one-dimensional arrays which can be directly implemented in B0.

A simple machine for file access named *BASIC_FILE_VAR*, originating from the data-base example of the B Book [1], is also provided. This machine permits objects with attributes of identical type to be stored and retrieved from file. Using it to externalise strings would be very cumbersome. Also, it does not provide for persistency between program runs as the name of the file is generated at random. Neither does it perform any error handling; file system errors cause it to abort.

We could implement *Bank* directly on our own base machines *BasicFile* and *BasicString* and on *BASIC_ARRAY_RGE*. However, it seems to be wiser to introduce a middle layer, which encapsulates general support for objects. This simplifies the implementation of *Bank* and gives us a reusable subsystem. It also frees us from hardwiring whether we want to internalise the complete database at startup or whether we only want to keep the currently accessed object in the main memory.

We implement *Bank* using a machine *Object* providing the aforementioned support for objects and *BasicString*. The specification still leaves it open whether the

complete database is kept in main memory or not. In the implementation we can no longer postpone the decision. We decide to read the whole database at startup; the other solution for a similar object-support machine is developed by Abrial in the aforementioned data-base example. Fig. 4.12 shows the structure of the intended development with section numbers for reference. We create a separate project for the object support and string machines to facilitate reuse.

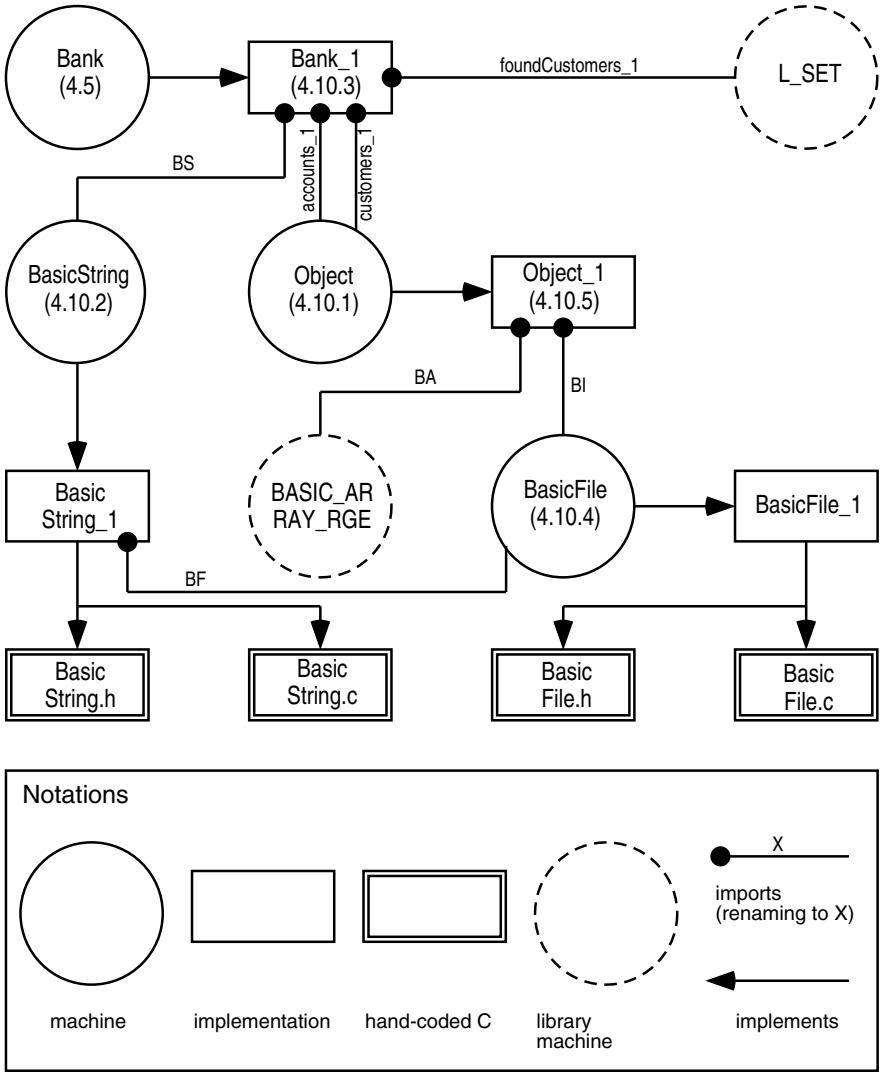


Fig. 4.12. Implementation of *Bank*

We proceed in a top-down fashion. We first identify the required functionality for implementing *Bank*, specify the necessary machines *Object* and *BasicString*, and then implement *Bank*. We then repeat the same sequence of steps for *Object* and *BasicString*.

4.10.1 Machine *Object*

As stated above, *Object* must be able to store a set of objects, each having a given number of attributes of identical type. We need to create new objects, modify and read their fields, search for an object by the value of one of its fields, and check whether the database is full or not.

Object has four parameters. The first parameter *maxNofObjs* denotes the maximal number of objects, which the machine can store. As discussed in Sect. 4.5, such an upper bound is needed in a safety-critical system in order to avoid overflows. The question remains, however, how we should constrain the maximal value of objects. This value is determined by the available main memory storing the objects and the available disk space for externalisation. This contradicts our aim to make the specification independent of the target computer. Even if we know our target architecture, the available memory at run time depends also upon which other processes are running and how many instantiations of the *Object* machine are present. Obviously, we cannot formally prove the instantiations to work for any value — except for 0. Such a proof would not be within B. In practice, we have to reason for the complete system that the chosen instantiations are permissible for the given resources. We implement our machine so that it allocates all the required memory at startup. Although failure during initialisation also violates the specification, it is usually less harmful than at run time. For the second resource, the disk storage, we take the more optimistic and less safe assumption that the disk always has at least as much free space as we have main memory.

MACHINE
Object(*maxNofObjs*, *nofFields*, *VALUE*, *valueElement*)

CONSTRAINTS
$maxNofObjs \in \mathbf{NAT1} \land nofFields \in \mathbf{NAT1} \land valueElement \in VALUE$

DEFINITIONS
$FIELD == 0 \,..\, nofFields\text{-}1$; $OBJECT == 0 \,..\, maxNofObjs\text{-}1$

VARIABLES
object, *objectSequence*, *field*, *foundObjects*

CONCRETE_VARIABLES
fileOpen

INVARIANT
$object \subseteq OBJECT \land \mathbf{card}(object) \leq maxNofObjs \land$
$objectSequence \in \mathbf{perm}(object) \land field \in FIELD \rightarrow (object \rightarrow VALUE) \land$
$fileOpen \in \mathbf{BOOL} \land foundObjects \subseteq object$

INITIALISATION

object := {} || *objectSequence* := [] || *field* := *FIELD* × { {} } ||
fileOpen := **FALSE** || *foundObjects* := {}

The second parameter *nofFields* takes the number of fields per object. It would be desirable to use a machine parameter of set type to designate the fields rather than the integer range 0 . . *nofFields*-1. Using such a branded type, certain errors could be flagged by the type checker rather than resulting in unprovable obligations at a later stage of the development. The reason why we do not use a machine parameter of set type is that it is not possible in B to iterate over an arbitrary set in an implementation as will be required in the implementation of *Object*. An iterator base machine cannot be implemented either because of an unfortunate C encoding decision in Atelier B.

The third parameter *VALUE* is the domain of the fields. The fourth parameter *valueElement* takes an arbitrary element of *VALUE*. It is required for the deterministic initialisation of a concrete variable of type *VALUE* in the implementation *Object_1*.

OPERATIONS

obj ← **CreateObject**(*initValue*) =
  PRE
    initValue ∈ *VALUE* ∧ **card**(*object*) < *maxNofObjs* ∧ *fileOpen* = **TRUE**
  THEN
    ANY *newObj*, *objSeq* **WHERE**
      newObj ∈ *OBJECT* - *object* ∧ *objSeq* ∈ **perm**(*object* ∪ {*newObj*})
    THEN
      object := *object* ∪ {*newObj*} || *objectSequence* := *objSeq* ||
      field := λ *ii*.(*ii* ∈ *FIELD* | *field*(*ii*) ∪ {*newObj* ↦ *initValue*}) ||
      obj := *newObj*
    END
  END;

vv ← **GetField**(*oo*, *ff*) =
  PRE *oo* ∈ **NAT** ∧ *oo* ∈ *object* ∧ *ff* ∈ *FIELD* ∧ *fileOpen* = **TRUE THEN**
    vv := *field*(*ff*)(*oo*)
  END;

SetField(*oo*, *ff*, *vv*) =
  PRE
    oo ∈ **NAT** ∧ *oo* ∈ *object* ∧
    ff ∈ *FIELD* ∧ *vv* ∈ *VALUE* ∧ *fileOpen* = **TRUE**
  THEN
    field(*ff*)(*oo*) := *vv* || *foundObjects* :∈ $\mathcal{P}$ (*object*)
  END;

is ← **Full** =
  PRE *fileOpen* = **TRUE THEN**
    is := **bool**(**card**(*object*) = *maxNofObjs*)
  END;

nof ← **NofObjects** =
  PRE *fileOpen* = **TRUE THEN**
    nof := **card**(*object*)
  END;

Operation *GetSequenceObj* permits the traversal of all objects. For this purpose we have introduced the variable *objectSequence*, which is always a permutation of the set of objects. Operation *CreateObject* reshuffles the sequence to allow for more implementation freedom. Exercise 4.5 shows how this, without the provision for deleting objects overly general specification, allows the simple addition of object deletion.

obj ← **GetSequenceObj**(*index*) =
  PRE
    index ∈ **NAT** ∧ *index*+1 ∈ **dom**(*objectSequence*) ∧ *fileOpen* = **TRUE**
  THEN *obj* := *objectSequence*(*index*+1)
  END;

The find operations follow the same pattern as their correspondences in *Bank*.

InitFind(*ff*, *vv*) =
  PRE *ff* ∈ *FIELD* ∧ *vv* ∈ *VALUE* ∧ *fileOpen* = **TRUE THEN**
    foundObjects := *field*(*ff*) $^{-1}$ [{*vv*}]
  END;
found, *oo* ← **FindNext** =
  PRE *fileOpen* = **TRUE THEN**
    IF *foundObjects* ≠ {} **THEN**
      ANY *obj* **WHERE** *obj* ∈ *foundObjects* **THEN**
        found, *oo*, *foundObjects* := **TRUE**, *obj*, *foundObjects* - {*obj*}
      END
    ELSE *found* := **FALSE** || *oo* :∈ *OBJECT*
    END
  END;

Internalizing objects with references to other objects (relations), we have to be able to verify whether the references denote valid objects. Operation *InDomain* serves this purpose.

is ← **InDomain**(*obj*) =
  PRE *obj* ∈ **NAT** ∧ *fileOpen* = **TRUE THEN**
    is := **bool**(*obj* ∈ *object*)
  END;

If the file denoted by parameter *name* of *Open* does not exist a new file is created.

status ← **Open**(*fileName*) =
  PRE *fileName* ∈ *STRING* **THEN**
    ANY *obj*, *objSeq*, *st* **WHERE**
      obj ⊆ *OBJECT* ∧ **card**(*obj*) ≤ *maxNofObjs* ∧
      objSeq ∈ **perm**(*obj*) ∧ *st* ∈ **BOOL**
    THEN
      object := *obj* || *objectSequence* := *objSeq* || *foundObjects* :∈ $\mathcal{P}$ (*obj*) ||
      field :∈ *FIELD* → (*obj* → *VALUE*) || *status* := *st* || *fileOpen* := *st*
    END
  END;

status ← **Close** =
  PRE *fileOpen* = **TRUE** **THEN**
    fileOpen := **FALSE** || *status* :∈ **BOOL**
  END

END

4.10.2 Machine *BasicString*

As explained in Sect. 4.7.1, machine *BasicString* stores all strings in the system. Because of the single writer restriction, this cannot be reflected in the B specification. The latter only represents the mapping from natural number indices to string tokens and the registration of strings to be externalised.

Machine *BasicString* can store at most *maxNofStrings* persistent strings. Operation *AddString* can be specified without any precondition that enough memory is available for a string of a certain size as the memory allocation has already taken place upon token generation.

MACHINE
  BasicString(*maxNofStrings*)

CONSTRAINTS
  maxNofStrings ∈ **NAT1**
SEES
  StrTokenType
VARIABLES
  regStrings, *bsFileOpen*
INVARIANT
  regStrings ∈ **NAT** ⤔ *STRTOKEN* ∧ **card**(*regStrings*) ≤ *maxNofStrings* ∧
  bsFileOpen ∈ **BOOL**
INITIALISATION
  regStrings := {} || *bsFileOpen* := **FALSE**
OPERATIONS
  index ← **AddString**(*ss*) =
    PRE
      ss ∈ *STRTOKEN* ∧ **card**(*regStrings*) ≠ *maxNofStrings* ∧
      bsFileOpen = **TRUE**
    THEN
      IF *ss* ∈ **ran**(*regStrings*) **THEN** *index* := *regStrings* $^{-1}$ (*ss*)
      ELSE
        ANY *newId* **WHERE** *newId* ∈ **NAT-dom**(*regStrings*) **THEN**
          index, *regStrings* := *newId*, *regStrings* ∪ {(*newId* ↦ *ss*)}
        END
      END
    END;
  is ← **IsFull** =
    is := **bool**(**card**(*regStrings*)=*maxNofStrings*);

```
  bb ← InDomain(index) =
    PRE index ∈ NAT THEN
      bb := bool(index ∈ dom(regStrings))
    END;
  ss ← GetString(index) =
    PRE
      index ∈ NAT ∧ index ∈ dom(regStrings) ∧ bsFileOpen = TRUE
    THEN
      ss := regStrings(index)
    END;
  found, index ← FindString(ss) =
    PRE ss ∈ STRTOKEN THEN
      IF ss ∈ ran(regStrings) THEN
        found, index := TRUE, regStrings ^-1 (ss)
      ELSE
        found := FALSE || index :∈ NAT
      END
    END;

  status, nof ← BsOpen(fileName) =
    PRE fileName ∈ STRING THEN
      ANY res, regStringsInit WHERE
        res ∈ BOOL ∧ regStringsInit ∈ NAT ⤖ STRTOKEN ∧
        card(regStringsInit) ≤ maxNofStrings
      THEN
        regStrings := regStringsInit || bsFileOpen := res ||
        status := res || nof := card(regStringsInit)
      END
    END;
  status ← BsClose =
    PRE bsFileOpen = TRUE THEN
      bsFileOpen := FALSE || status :∈ BOOL
    END
END
```

The empty implementation as well as the hand-coded C source are available from the book's Web page.

4.10.3 Implementation *Bank_1*

Using *Object*, *BasicString*, and *L_SET* we can now implement *Bank*. We instantiate the *Object* machine twice to implement the customer and account objects. Library machine *L_SET* is used for temporary storage of the not yet retrieved set of customers from the find operations.

```
IMPLEMENTATION
  Bank_1(maxCustomers, maxAccounts)

REFINES
  Bank
```

IMPORTS
BASIC_BOOL, *BASIC_ARITHMETIC*,
BS.BasicString(*maxCustomers*),
customers_1.Object(*maxCustomers*, 2, **NAT**, 0),
accounts_1.Object(*maxAccounts*, 4, **NAT**, 0),
foundCustomers_1.L_SET(*maxCustomers*, 0 . . *maxCustomers*-1)

SEES
StrTokenType

Constant *False1* is introduced for expressing variant functions in operations *ThisCustomer* and *InitFindCustomer*.

CONCRETE_CONSTANTS
False1

PROPERTIES
$False1 \in \mathbf{BOOL} \rightarrowtail \mathbf{NAT}$

VALUES
$False1 = \{(\mathbf{TRUE} \mapsto 0), (\mathbf{FALSE} \mapsto 1)\}$

During internalisation, we have to check whether all references from accounts to customers captured by *accountOwner* reference existing customers and whether all references to strings from *customerName* are in the domain of the internalised strings. Hence, internalisation fails if it fails in one of the three instantiated machines or the consistency check fails. Rather than resetting the already internalised parts if an error is detected, the linking invariant separates two cases. If internalisation succeeded, the state is represented by the state of the imported machines. Otherwise, it is the initial state. Implementation *Bank_1* is a data refinement of machine *Bank* as specified by the linking invariant.

DEFINITIONS
$customerName\_1 == 0$;
$customerYob\_1 == 1$;
$accountNumber\_1 == 0$;
$accountPin\_1 == 1$;
$accountBalance\_1 == 2$;
$accountOwner\_1 == 3$

CONCRETE_VARIABLES
nextAccountNumber_1

INVARIANT
$nextAccountNumber\_1 \in \mathbf{NAT} \wedge$
$((fileOpen = \mathbf{TRUE}) \Rightarrow$
$customers = customers\_1.object \wedge$
$(\forall ll.(ll \in customers\_1.object \Rightarrow$
$customerName(ll) = BS.regStrings(customers\_1.field(customerName\_1)(ll)))) \wedge$
$customers\_1.field(customerName\_1) \in customers\_1.object \rightarrow \mathbf{dom}(BS.regStrings) \wedge$
$\mathbf{card}(BS.regStrings) \leq \mathbf{card}(customers) \wedge$
$customerYob = customers\_1.field(customerYob\_1) \wedge$

```
    accounts = accounts_1.object ∧
    accountNumber = accounts_1.field(accountNumber_1) ∧
    accountPin = accounts_1.field(accountPin_1) ∧
    accountBalance = accounts_1.field(accountBalance_1) ∧
    accountOwner = accounts_1.field(accountOwner_1) ∧
    ( ∀ ll.(ll ∈ accounts_1.object ⇒
      accounts_1.field(accountNumber_1)(ll) < nextAccountNumber_1)) ∧
    nextAccountNumber_1 < MAXINT - maxAccounts + card(accounts) ∧
    customers_1.fileOpen = TRUE ∧
    accounts_1.fileOpen = TRUE ∧
    BS.bsFileOpen = TRUE ∧
    foundCustomers = ran(foundCustomers_1.set_vrb)) ∧
  ((fileOpen = FALSE) ⇒
    customers = {} ∧ customerName = {} ∧ customerYob = {} ∧
    accounts = {} ∧ accountNumber = {} ∧
    accountPin = {} ∧ accountBalance = {} ∧
    accountOwner = {} ∧
    foundCustomers = {} )

INITIALISATION
  nextAccountNumber_1 := 0; fileOpen := FALSE

OPERATIONS
  NewCustomer(name, yob) =
    VAR cid, ii IN
      cid ← customers_1.CreateObject(0);
      ii ← BS.AddString(name);
      customers_1.SetField(cid, customerName_1, ii);
      customers_1.SetField(cid, customerYob_1, yob)
    END;

  name, yob ← CustomerData(cid) =
    VAR sn IN
      sn ← customers_1.GetField(cid, customerName_1);
      name ← BS.GetString(sn);
      yob ← customers_1.GetField(cid, customerYob_1)
    END;

  is ← CustomerDBFull =
    BEGIN
      is ← customers_1.Full;
      IF is=FALSE THEN
        is ← BS.IsFull
      END
    END;

  found, cid ← ThisCustomer(name, yob) =
    VAR sindex, curYob IN
      cid := 0; curYob := 0; found, sindex ← BS.FindString(name);
      IF found = TRUE THEN
        customers_1.InitFind(customerName_1, sindex);
        found, cid ← customers_1.FindNext;
        IF found=TRUE THEN
          curYob ← customers_1.GetField(cid, customerYob_1)
        END;
```

```
      WHILE (found = TRUE) ∧ (yob ≠ curYob) DO
        found, cid ← customers_1.FindNext;
        IF found = TRUE THEN
          curYob ← customers_1.GetField(cid, customerYob_1)
        END
      INVARIANT
        cid ∈ 0 .. maxCustomers-1 ∧ found ∈ BOOL ∧
        customers_1.foundObjects ⊆ customerName ^-1 [{name}] ∧
        (found = FALSE ⇒ yob ∉ customerYob[customerName ^-1 [{name}]]) ∧
        (found = TRUE ⇒
          (cid ∈ customerName ^-1 [{name}] ∧ curYob = customerYob(cid) ∧
          (yob=curYob ⇒ cid=(customerName ⊗ customerYob) ^-1 (name,yob)) ∧
          (yob ≠ curYob ⇒ yob ∉ customerYob[customerName ^-1 [{name}]-
              customers_1.foundObjects])))
      VARIANT
        card(customers_1.foundObjects) + 1 - False1(found)
      END
    END
  END;
nof ← InitFindCustomer(name) =
  VAR found, index, sindex IN
    foundCustomers_1.CLR_SET;
    nof := 0;
    found, sindex ← BS.FindString(name);
    IF found = TRUE THEN
      customers_1.InitFind(customerName_1, sindex);
      found, index ← customers_1.FindNext;
      WHILE found = TRUE DO
        foundCustomers_1.INS_SET(index);
        nof := nof + 1;
        found, index ← customers_1.FindNext
      INVARIANT
        (found = TRUE ⇒
          customerName ^-1 [{name}] = ran(foundCustomers_1.set_vrb) ∪
            customers_1.foundObjects ∪ {index}) ∧
        (found = FALSE ⇒
          customerName ^-1 [{name}] = ran(foundCustomers_1.set_vrb) ∧
          customers_1.foundObjects = {} ) ∧
        nof = card(foundCustomers_1.set_vrb)
      VARIANT
        card(customers_1.foundObjects)+1-False1(found)
      END
    END
  END;
found, yob ← FindNextCustomer =
  VAR nof, cid IN
    nof ← foundCustomers_1.CARD_SET;
    IF nof = 0 THEN
      found := FALSE; yob := 0
    ELSE
      found := TRUE;
      cid ← foundCustomers_1.VAL_SET(1);
```

```
      foundCustomers_1.RMV_SET(cid);
      yob ← customers_1.GetField(cid, customerYob_1)
    END
  END;
```

We assign consecutive account numbers to newly created accounts, where *nextAccountNumber* contains the next account number which is the greatest number in the system plus one. We do, however, not blindly trust that the internalised file adheres to this convention, that is, we do not simply set *nextAccountNumber* to number of accounts plus one, which would lead to an undischargable proof obligation.

```
number ← NewAccount(cid, pin) =
  VAR aid IN
    aid ← accounts_1.CreateObject(0);
    accounts_1.SetField(aid, accountNumber_1, nextAccountNumber_1);
    accounts_1.SetField(aid, accountPin_1, pin);
    accounts_1.SetField(aid, accountBalance_1, 0);
    accounts_1.SetField(aid, accountOwner_1, cid);
    number := nextAccountNumber_1;
    nextAccountNumber_1 := nextAccountNumber_1 + 1
  END;
bal ← Balance(aid, pin) =
  bal ← accounts_1.GetField(aid, accountBalance_1);
is ← Authorized(aid, pin) =
  VAR actualPin IN
    actualPin ← accounts_1.GetField(aid, accountPin_1);
    is := bool(pin = actualPin)
  END;
cid ← AccountOwner(aid) =
  cid ← accounts_1.GetField(aid, accountOwner_1);
status ← Deposit(aid, amount) =
  VAR bal, xx IN
    bal ← accounts_1.GetField(aid, accountBalance_1);
    xx := MAXINT - amount;
    IF bal < xx THEN
      accounts_1.SetField(aid, accountBalance_1, bal+amount);
      status := TRUE
    ELSE
      status := FALSE
    END
  END;
```

Operation *Deposit* introduces the local variable *xx* only because in B0 the arguments of a comparison cannot contain arithmetic expressions.

```
/* Operations Withdraw, ChangePin, AccountDBFull, and ThisAccount omitted. Check on
the book's Web page. */
  status ← Open(customerFileName, accountFileName, stringFileName) =
    VAR nofAccounts, ii, jj, aid, owner, nbr, nofStrings, nofCustomers, cid, nameNr, yob,
        curAid, curNbr, curCid, curNameNr, curYob, xx, yy IN
```

```
fileOpen := FALSE; nextAccountNumber_1 := 0;
status ← customers_1.Open(customerFileName);
IF status = TRUE THEN
  status ← accounts_1.Open(accountFileName);
  IF status = TRUE THEN
    nofAccounts ← accounts_1.NofObjects;
    ii := 0;
    WHILE (ii < nofAccounts) ∧ (status = TRUE) DO
      aid ← accounts_1.GetSequenceObj(ii);
      owner ← accounts_1.GetField(aid, accountOwner_1);
      status ← customers_1.InDomain(owner);
      nbr ← accounts_1.GetField(aid, accountNumber_1);
      IF nbr ≥ nextAccountNumber_1 THEN
        xx := MAXINT - nbr;
        yy := maxAccounts - nofAccounts + 2;
        IF xx < yy THEN
          status := FALSE
        ELSE nextAccountNumber_1 := nbr + 1
        END
      END;
      jj := ii + 1;
      WHILE (jj < nofAccounts) ∧ (status = TRUE) DO
        curAid ← accounts_1.GetSequenceObj(jj);
        curNbr ← accounts_1.GetField(curAid, accountNumber_1);
        IF nbr = curNbr THEN
          status := FALSE
        END;
        jj := jj + 1
      INVARIANT
        jj ∈ ii+1 .. nofAccounts ∧
        status ∈ BOOL ∧
        (status = TRUE ⇒
          ( ∀ kk.(kk ∈ ii+2 .. jj ⇒ nbr ≠ accounts_1.field
              (accountNumber_1)(accounts_1.objectSequence(kk))) ∧
          owner ∈ customers_1.object ∧
          ¬ (nbr ≥ nextAccountNumber_1 ∧
              MAXINT-nbr<maxAccounts-nofAccounts+2)))
      VARIANT
        nofAccounts - jj
      END;
      ii := ii + 1
    INVARIANT
      ii ∈ 0 .. nofAccounts ∧
      status ∈ BOOL ∧
      nextAccountNumber_1 ∈ NAT ∧
      (status = TRUE ⇒
        ( ∀ kk.(kk ∈ 1 .. ii ⇒
          accounts_1.field(accountNumber_1)(accounts_1.objectSequence(kk)) <
              nextAccountNumber_1 ∧
          accounts_1.field(accountOwner_1)(accounts_1.objectSequence(kk)) ∈
              customers_1.object ∧
           ∀ ll.(ll ∈ 1 .. nofAccounts ∧ kk ≠ ll ⇒
            accounts_1.field(accountNumber_1)(accounts_1.objectSequence(kk))
```

```
                    ≠ accounts_1.field(accountNumber_1)
                        (accounts_1.objectSequence(ll))))) ∧
              nextAccountNumber_1 < MAXINT - maxAccounts + nofAccounts)
          VARIANT
            nofAccounts - ii
          END;
          /* Consistency check of customers_1 and BS ommitted. Check on the Web. */
          foundCustomers_1.CLR_SET; fileOpen := status
        END
      END
    END;
  status ← Close =
    BEGIN
      status ← customers_1.Close;
      IF status = TRUE THEN
        status ← accounts_1.Close;
        IF status = TRUE THEN status ← BS.BsClose END
      END;
      nextAccountNumber_1 := 0; fileOpen := FALSE
    END
END
```

4.10.4 Machine *BasicFile*

In order to permanently store objects on disk, as required for the implementation of *Object*, we need a base machine to access the file system, which we call *BasicFile*. It should let us open a file in different modes, access the file, and provide operations to delete a file and check for the existence of a file. We want to store both natural numbers as well as elements of a given set, passed as a machine parameter. An instance of *BasicFile* represents a single file.

The variables *fileName* and *fileMode* denote the name and mode of the currently open file. The name of the file has been specified as an arbitrary string, although certain characters might not be permitted in file names and certain names might denote special resources.

```
MACHINE
  BasicFile(VALUE)

SETS
  FILE_MODE = {READ_WRITE, TRUNCATE_WRITE, READ, WRITE}

DEFINITIONS
  READ_MODE == {READ_WRITE, READ};
  WRITE_MODE == {READ_WRITE, TRUNCATE_WRITE, WRITE}

VARIABLES
  fileMode, fileOpen

INVARIANT
  fileMode ∈ FILE_MODE ∧
```

```
  fileOpen ∈ BOOL

INITIALISATION
  fileMode :∈ FILE_MODE || fileOpen := FALSE

OPERATIONS
  status ← Open(fileName, mode) =
    PRE fileName ∈ STRING ∧ mode ∈ FILE_MODE THEN
      ANY rr WHERE rr ∈ BOOL THEN
        fileMode := mode || fileOpen := rr || status := rr
      END
    END;
  status ← Close =
    PRE fileOpen = TRUE THEN
      fileOpen := FALSE || status :∈ BOOL
    END;
  status ← Delete(fileName) =
    PRE fileName ∈ STRING THEN
      status :∈ BOOL
    END;
  exists ← FileExists(fileName) =
    PRE fileName ∈ STRING THEN
      exists :∈ BOOL
    END;
```

The read operations are specified as returning an arbitrary value, not linking write and read at all. Such a specification would be very difficult to capture in B, too cumbersome to apply in reasoning in clients, and impossible to satisfy in the implementation.

```
  status ← WriteNat(num) =
    PRE num ∈ NAT ∧ fileOpen = TRUE ∧ fileMode ∈ WRITE_MODE THEN
      status :∈ BOOL
    END;
  status, num ← ReadNat =
    PRE fileOpen = TRUE ∧ fileMode ∈ READ_MODE THEN
      status :∈ BOOL || num :∈ NAT
    END;
  status ← WriteVal(val) =
    PRE val ∈ VALUE ∧ fileOpen = TRUE ∧ fileMode ∈ WRITE_MODE THEN
      status :∈ BOOL
    END;
  status, val ← ReadVal =
    PRE fileOpen = TRUE ∧ fileMode ∈ READ_MODE THEN
      status :∈ BOOL || val :∈ VALUE
    END
END
```

The C implementation, which is based on the code skeleton generated from the empty B implementation, consists mostly of straightforward calls of the corresponding functions of *stdio.h*. The procedure *ReadVal_BasicFile* also checks whether the read value actually represents an element of the machine parameter *VALUE*. Unfortunately, Atelier B's C translator only passes the upper bound of the representing integer range in the ill-named parameter *size_VALUE* in the initialisation. This suffices for enumerated sets that are represented as consecutive integer constants starting from 0. However, for instantiations of *VALUE* with integer ranges with a lower bound other than 0 we cannot test whether the read value is below the indicated range. The sources of *BasicFile_1.imp*, *BasicFile.h*, and *BasicFile.c* can be found online.

4.10.5 Implementation *Object_1*

Using the base machine *BasicFile* and the library machine *BASIC_ARRAY_RGE* we can now implement *Object* and, herewith, finish the development.

BASIC_ARRAY_RGE models a two dimensional array with the total function $arr\_rge \in RANGE \rightarrow (INDEX \rightarrow VALUE)$, where *INDEX*, *VALUE*, and *RANGE* are machine parameters. We instantiate *RANGE* with the set of fields and *INDEX* with the object numbers. For example, *arr_rge(0)(7)* denotes the 0th field of the 7th object. We use the variable *nofObjs_1* to denote the number of objects and link it to *object* with $object = 0 \,..\, nofObjs\_1\text{-}1$. This gives us also the linking invariant for *field* as $\forall\, ii.(ii \in FIELD \Rightarrow field(ii) = 0 \,..\, nofObjs\_1\text{-}1 \lhd arr\_rge(ii))$.

IMPLEMENTATION
Object_1(*maxNofObjs*, *nofFields*, *VALUE*, *valueElement*)

REFINES
Object

IMPORTS
BI.BasicFile(*VALUE*),
BA.BASIC_ARRAY_RGE(0 .. *maxNofObjs*-1, *VALUE*, 0 .. *nofFields*-1)

DEFINITIONS
FIELD == 0 .. *nofFields*-1; *OBJECT* == 0 .. *maxNofObjs*-1;
READ_MODE == {*READ_WRITE*, *READ*};
WRITE_MODE == {*READ_WRITE*, *TRUNCATE_WRITE*, *WRITE*}

CONCRETE_VARIABLES
nofObjs_1, *findField*, *findValue*, *findMax*, *findNext*

INVARIANT
$nofObjs\_1 \in 0 \,..\, maxNofObjs \wedge object = 0 \,..\, nofObjs\_1\text{-}1 \wedge$
$\mathbf{size}(objectSequence) = nofObjs\_1 \wedge$
$(\forall\, ii.(ii \in 0 \,..\, nofObjs\_1\text{-}1 \Rightarrow objectSequence(ii{+}1) = ii)) \wedge$
$(\forall\, ii.(ii \in FIELD \Rightarrow field(ii) = 0 \,..\, nofObjs\_1\text{-}1 \lhd (BA.arr\_rge(ii)))) \wedge$
$(fileOpen = \mathbf{TRUE} \Rightarrow (BI.fileOpen = \mathbf{TRUE} \wedge BI.fileMode \in WRITE\_MODE)) \wedge$
$findField \in FIELD \wedge findValue \in VALUE \wedge$
$findMax \in \text{-}1 \,..\, nofObjs\_1\text{-}1 \wedge findNext \in 0 \,..\, nofObjs\_1 \wedge$

foundObjects = (*field*(*findField*) $^{-1}$ [{*findValue*}]) ∩ *findNext* .. *findMax*

INITIALISATION
nofObjs_1 := 0; *fileOpen* := **FALSE**;
findField := 0; *findValue* := *valueElement*; *findMax* := -1; *findNext* := 0

OPERATIONS

obj ← **CreateObject**(*initValue*) =
  VAR *fld* **IN**
    fld := 0;
    WHILE *fld* < *nofFields* **DO**
      BA.STR_ARR_RGE(*fld*, *nofObjs_1*, *initValue*);
      fld := *fld* + 1
    INVARIANT
      fld ∈ 0 .. *nofFields* ∧ *BA.arr_rge* ∈ *FIELD* → (*OBJECT* → *VALUE*) ∧
      (∀ *ii*.(*ii* ∈ *FIELD* ⇒ *field*(*ii*) = 0 .. *nofObjs_1*-1 ◁ (*BA.arr_rge*(*ii*)))) ∧
      (∀ *ii*.(*ii* ∈ 0 .. *fld*-1 ⇒ *BA.arr_rge*(*ii*)(*nofObjs_1*) = *initValue*))
    VARIANT
      nofFields - *fld*
    END;
    obj := *nofObjs_1*; *nofObjs_1* := *nofObjs_1* + 1
  END;

vv ← **GetField**(*oo*, *ff*) =
    vv ← *BA.VAL_ARR_RGE*(*ff*, *oo*);

SetField(*oo*, *ff*, *vv*) =
  BA.STR_ARR_RGE(*ff*, *oo*, *vv*);

is ← **Full** =
  IF *nofObjs_1* = *maxNofObjs* **THEN** *is* := **TRUE**
  ELSE *is* := **FALSE**
  END;

nof ← **NofObjects** =
    nof := *nofObjs_1*;

obj ← **GetSequenceObj**(*index*) =
    obj := *index*;

InitFind(*ff*, *vv*) =
  BEGIN
    findField := *ff*; *findValue* := *vv*; *findMax* := *nofObjs_1*-1; *findNext* := 0
  END;

found, *oo* ← **FindNext** =
  VAR *val*, *maxObj*, *findStart* **IN**
    found := **FALSE**; *oo* := 0;
    IF *findNext* ≤ *findMax* **THEN**
      val ← *BA.VAL_ARR_RGE*(*findField*, *findNext*);
      findStart := *findNext*;
      WHILE (*findNext* < *findMax*) ∧ (*val* ≠ *findValue*) **DO**
        findNext := *findNext* + 1;
        val ← *BA.VAL_ARR_RGE*(*findField*, *findNext*)
      INVARIANT
        findNext ∈ *findStart* .. *findMax* ∧
        (∀ *ll*.(*ll* ∈ *findStart* .. *findNext*-1 ⇒ *BA.arr_rge*(*findField*)(*ll*) ≠ *findValue*)) ∧

```
      val = BA.arr_rge(findField)(findNext)
    VARIANT
      findMax-findNext
    END;
    IF val = findValue THEN
      found := TRUE; oo := findNext
    END;
    findNext := findNext + 1
  END
 END;
is ← InDomain(obj) =
  is := bool(obj < nofObjs_1);
status ← Open(fileName) =
 VAR st, ii, fld, vv IN
  status ← BI.FileExists(fileName);
  IF status = TRUE THEN
    status ← BI.Open(fileName, READ);
    IF status = TRUE THEN
      status, nofObjs_1 ← BI.ReadNat;
      IF (status = TRUE) ∧ (nofObjs_1 ≤ maxNofObjs) THEN
        ii := 0;
        WHILE (status = TRUE) ∧ (ii < nofObjs_1) DO
          fld := 0;
          WHILE (status = TRUE) ∧ (fld < nofFields) DO
            status, vv ← BI.ReadVal;
            BA.STR_ARR_RGE(fld, ii, vv);
            fld := fld + 1
          INVARIANT
            fld ∈ 0 .. nofFields ∧ status ∈ BOOL
          VARIANT
            nofFields - fld
          END;
          ii := ii + 1
        INVARIANT
          ii ∈ 0 .. nofObjs_1 ∧ status ∈ BOOL
        VARIANT
          nofObjs_1 - ii
        END;
        IF status = TRUE THEN
          status ← BI.Close;
          IF status = TRUE THEN
            status ← BI.Open(fileName, TRUNCATE_WRITE)
          END
        END
      ELSE
        nofObjs_1:=0;
        status := FALSE
      END
    ELSE
      nofObjs_1 := 0
    END
  ELSE
    nofObjs_1 := 0; status ← BI.Open(fileName, TRUNCATE_WRITE)
```

```
      END;
      findMax := -1; findNext := 0;
      fileOpen := status
    END;
  status ← Close =
    VAR ss, ii, fld, vv IN
      ss ← BI.WriteNat(nofObjs_1);
      IF ss = TRUE THEN
        ii := 0;
        WHILE (ss = TRUE) ∧ (ii < nofObjs_1) DO
          fld := 0;
          WHILE (ss = TRUE) ∧ (fld < nofFields) DO
            vv ← BA.VAL_ARR_RGE(fld, ii);
            ss ← BI.WriteVal(vv);
            fld := fld + 1
          INVARIANT
            fld ∈ 0 .. nofFields
          VARIANT
            nofFields - fld
          END;
            ii := ii + 1
        INVARIANT
          ii ∈ 0 .. nofObjs_1
        VARIANT
          nofObjs_1 - ii
        END
      END;
      IF ss = TRUE THEN
        status ← BI.Close
      ELSE
        status := FALSE; ss ← BI.Close
      END;
      fileOpen := FALSE
    END
END
```

At this point we can translate the complete project.

4.11 B-Toolkit Implementation

In this section we list some of the changes necessary to port the case study from Atelier B to the B-Toolkit. The point of this section is to illustrate the large differences between the two tools —even on the language level!— which make porting a non-trivial task. The magnitude of such a port can be compared to the translation of an X Window program written in K&R C to ANSI C on the Apple Macintosh: both require some little changes on the language level and the use of a different base library. The rest of this section is mainly targeted at B-Toolkit users who are interested in a description of the adaptations made in the B-Toolkit version of the ATM.

4.11.1 Differences in the Supported Language

The following 'syntactic' differences can be compensated for with simple rewrites:

- In the B-Toolkit, machine parameters are not repeated in refinements and implementations.
- In the B-Toolkit, lowercase machine parameters are implicitly constrained to be of type *SCALAR*.
- Ordered pairs must be written as $(a \mapsto b)$ rather than (a,b) in the B-Toolkit, whereas both notations are allowed in Atelier B.
- Sets and constants are valued in the *PROPERTIES* clause; there is no special values *VALUES* clause as in Atelier B.
- The constant *MAXINT*, the greatest representable natural number, is not predefined in the B-Toolkit.
- In the B-Toolkit, the subset $0 \,..\, MAXINT$ is denoted by *SCALAR* rather than *NAT*. The type *SCALAR* is defined in machine *Scalar_TYPE*, which must be imported if scalars are used.
- In the B-Toolkit, booleans are defined as enumerated type in the library machine *Bool_TYPE*, which must be imported if booleans are used.
- In the B-Toolkit, strings are defined as sequences in the library *String_TYPE*, which must be imported if strings are used.
- In the B-Toolkit, there can only be one *DEFINITION* clause per construct. Definitions are visible in the whole construct, not just from the syntactic introduction point on forward as in Atelier B. Parameters of definitions are restricted to single-letter identifiers (jokers). Definitions containing the parallel operator ('$\|$') must be parenthesised.
- In the B-Toolkit, renamed variables must be parenthesised if the inverse is taken.
- In the B-Toolkit, the *bool(P)* operator, which converts the value of a condition to a *BOOL*, is not available in implementations. An if-clause must be used instead.
- The B-Toolkit C translator does not accept arithmetic expressions as actual parameters. The values of arithmetic expressions must be evaluated and stored in local variables, which can then be passed as parameters.
- The C translator does not accept read access to output parameters, even if they have been properly initialized. Local variables, which are at the end of the operation assigned to the output parameters, must be used within the operation in place of output parameters appearing on the right hand side of assignments or in conditions.
- Whereas the Atelier B translator creates only few name clashes, which lead to errors at link time, its correspondent in the B-Toolkit cannot even handle operations on different layers with identical names. Hence, one is forced to invent new names and, thereby, pollute the name space.

The following differences make porting from the Atelier B to the B-Toolkit difficult:

- The B-Toolkit does not support dot renaming in implementations. This means that renamed textual copies of multiple used constructs must be made. In our

case, *Object* and all the constructs it needs would have to be textually present with different name prefixes. This also requires identical proofs to be performed for each copy. This restriction in the B-Toolkit is due to the fact that all constructs are single instance only which is also exhibited by the C translator putting implementation data into global variables rather than instantiation records. On the level of base machines, which reside in the standard library, textual renaming is performed automatically upon configuration. The team library does not provide for renaming.

- In the B-Toolkit all constants are abstract, whereas Atelier B has both concrete and abstract constants. The B-Toolkit translator decides which constants can be used in implementations.
- Concrete variables and variables in implementations are not supported. All global variables, such as *nextAccountNumber_1*, must be implemented using library machines. Sets which are both included and imported lead to name clashes. Different renaming does not help because sets do not participate in renaming. Hence, sets must be factored out into separate machines which are only seen in the specification. Third-party constructs which do not respect this design pattern, such as the library machines in B-Toolkit prior to version 4, can, therefore, not be easily extended as extension is performed by both including and importing the same machine.

The following differences would make porting from the B-Toolkit to Atelier B difficult. Some of these 'additional features' are used in the B-Toolkit version:

- Machines can contain the *VAR* clause. Hence, we can use it to hide the return parameter *dd* from *Deposit* in *RobustDeposit*.
- Machine parameters are visible in the *PROPERTIES* clause. Hence, we could model the set *CUSTOMER* of machine *Bank* as an abstract set with cardinality *maxCustomers* and value it to *CUSTOMER = 0..maxCustomers-1* in the implementation. To rule out any circular definitions, Atelier B does not permit this in accordance with [1, Chapter 12.1.7].
- The B-Toolkit allows strings to be passed as parameters. Hence, there is no need to introduce string tokens. Strings being sequences implies that functions such as *size* are applicable. Porting a construct which makes use of this from B-Toolkit would be difficult. In general, string support in the B-Toolkit is better. Unfortunately, B-Toolkit's C translator creates fixed length arrays for local string variables and does not perform any overflow tests.
- Sets of imported or seen machines can be used in the instantiation of other machines.
- Set machine parameters can be instantiated with 'unions' ('$\cup$') of sets. This is not described in the B Book [1]; it could be understood as type sums. Unfortunately, on the implementation level, where sets are represented as (initial) intervals of natural numbers, operations on such sets are based on the natural number projections only, leading, in our opinion, to ill-typed expressions and wrong results. Thus, for sets *COLORS* = {*red, blue, green*} and *FRUITS* = {*apple, banana, grape*} we can calculate {*red, blue*} $\cap$ {*banana, grape*} = {*blue*} = {*banana*} as

both *blue* and *banana* are represented by 2. Union of sets is used extensively by the base generator (see below).

4.11.2 Differences in the Provided Base Machines and Libraries

In the B-Toolkit, all provided library machines are base machines, whereas Atelier B comes only with a small set of base machines and numerous extensions in the form of normal B developments. In the B-Toolkit, base machines reside in the standard library (SLIB).

The B specification of base machines must be given in a separate project, otherwise the linker requires an implementation in B and does not use the hand-coded C source. After successful analysis and compilation, the configured construct along with its C implementation is copied to the SLIB, to which one needs write permission. The main differences in the C encoding are the representation of machine data in global variables rather than in instance records and the division of header information into the '.h' and a '.g' file. Note that when introducing a construct from the SLIB, the C sources are copied. Thus, if the (implementation of the) base machine is changed, it must be removed and reintroduced into projects using it.

Compilation and linking is under the control of the tool. Hence, external source files such as *cgic* cannot simply be added manually to the Makefile as in Atelier B. Instead, they need to be introduced as so-called lower-level SLIB constructs. Lower-level SLIB constructs have no B specification and can only be accessed from the C code of other SLIB constructs. Instead of a lower-level SLIB, a normal C library can be created out of the legacy code and included manually in a normal SLIB construct.

4.11.3 Adapting the Development

The B-Toolkit implementation takes the above listed language differences into account. Additionally, supplied base machines have been used in place of the self-developed persistent object machines. The B-Toolkit provides base machines for objects and string objects. Library machine *Bank_str_obj*, where *Bank* is the rename prefix for the instanciation, provides for string objects, like our own base machine *BasicString*. *Rename_ffnc_obj* provides for two dimensional arrays; it replaces *Object* of our Atelier B development. We introduce two copies called *CUSTOMER_ffnc_obj* and *ACCOUNT_ffnc_obj* for storing customers and accounts respectively.

In combination with machine *file_dump*, the multiple object machines also provide for persistency. A file is opened with *file_dump* into which all machines can externalise their state. Unfortunately, the code contains no error or consistency checking. Atelier B's library also contains a machine *BASIC_SAVE* which roughly corresponds to *file_dump*; however, it does not function anymore and the corresponding procedures have been removed from the B specification of the other library machines.

4.11.4 Automatic Translation of Object Models

The B-Toolkit acknowledges the fact that object models can be automatically translated to B machines. From a textual description of the object model a set of machines and corresponding implementations is generated. The base description (Fig. 4.13) lists global variables (customers and accounts) as well as the object classes (CUSTOMER and ACCOUNT) with their attributes and the relations. Relations can be expressed asymmetrically by being part of one of the participating object classes, as done in the example, or as separate entities.

From the base construct, a list of operations on the global variables and on objects of the listed classes is generated. After optional manual filtering of the operations' list, a set of machines and implementations is generated. The implementations are based on constructs from the standard library described above. Based on BankFoundation it would then be possible to implement Bank. Editing the generated machines and implementations directly is not recommended because of the lack of backward propagation to the base construct; it would result in breaking the link and the possibility to regenerate the constructs after changing the base.

It is doubtful whether using the base generation tool would be justified in our case. Even if certain aspects are actually formally proved and the code is automatically generated, added complexity is a source for errors. Manual reuse of those library constructs that are actually needed seems to be better suited in our case.

```
SYSTEM
  BankFoundation

IS

  GLOBAL
    customers : SET(CUSTOMER)[100];
    accounts : SET(ACCOUNT)[200]
  END;

  BASE
    CUSTOMER
  MANDATORY
    name : STRING [256]; yob : NAT
  END;

  BASE
    ACCOUNT
  MANDATORY
    number : NAT; pin : NAT;
    balance : NAT; owner : CUSTOMER
  END

END
```

Fig. 4.13. Base Construct for Automatic Generation

The B-Toolkit comes with three small data base like examples, called PERSON1, PERSON2, and PERSON3, which illustrate the differences between the manual use of the standard library constructs and the application of the base generator.

4.12 Discussion

4.12.1 Related Work

The B Book [1] contains a much smaller example of a database application. The database example as well as an ATM case study are included in the Atelier B distribution. The documentation of the ATM, which is in French only, provides an exemplary requirement specifications, a traceability matrix, and a set of test scenarios. On the other hand, it lacks a description of the construction process as well as a detailed explanation of the produced code. The ATM relies on a Tcl/Tk graphical interface as main program and delegates more work to unverified base machines.

A comprehensive B bibliography is maintained by the B users group on the Web at http://estas1.inrets.fr:8001/ESTAS/BUG/WWW/BUGhome/BUGhome.html.

4.12.2 Metrics

Fig. 4.14 provides some metrics of the development. The empty implementations of the base machines, the hand-coded C sources, and the HTML pages are not included.

4.12.3 What Have We Proved?

We would like to conclude with a few remarks on proofs. What have we actually proved in our development? We have proved that all operations of the machines respect their invariants and that the implementations are refinements of their specifications, provided that the B theory is correct, the tools generated all necessary obligations, and the tools did not discharge any false obligations.

What haven't we proved? We haven't proved that the specification corresponds to the informal requirements; especially, that we have captured all requirements as invariants. Furthermore, we haven't proved that the hand-coded base machines actually satisfy their specifications. We are also at the mercy of the B to C translator, the C compiler, and the used computers with their operating systems.

In conclusion, the many unprovable and unproved aspects even of a formal development in B are a clear sign, that good engineering practices, including animation, peer code review, and testing, are also important in a 'proved' development.

4.13 Exercises

Exercise 4.1 (Search operations). Give the cashier the possibility to display all customers who have their 20th birthday this year and are entitled to a present. Use the pattern of *SetFindCustomer* and *FindNextCustomer* of machine *Bank*.

Machines

	total length	obvious proof obligations	proof obligations	percent auto proved
MainBank	9 lines	3	0	100
OperationsBank	49 lines	19	0	100
RobustBank	239 lines	101	10	100
Bank	288 lines	394	49	95
Object	171 lines	125	17	100
BasicFile	102 lines	26	0	100
BasicString	98 lines	41	6	100
BasicCGI	72 lines	15	0	100
StrTokenType	14 lines	1	0	100
Total	1042 lines	725	82	98

Implementations (without base machines)

	total length	obvious proof obligations	proof obligations	percent auto proved
MainBank_1	52 lines	16	4	100
OperationsBank_1	334 lines	1028	285	99
RobustBank_1	206 lines	856	27	85
Bank_1	305 lines	526	643	71
Object_1	204 lines	291	230	70
StrTokenType_1	10 lines	3	2	100
Total	1111 lines	2720	1191	78

Fig. 4.14. Statistics of the Development

Exercise 4.2 (Online banking). Extend the bank so that customers can transfer money from one account to another over the Internet. The customer logs in using the account number, a password, and a one time code. The latter can for simplicity be chosen to be the login number. After login, the customer can make any number of transfers from her accounts to any accounts. The session is terminated by an explicit logout or after a fixed timeout. Withdrawals must now also be authorisable using the customer's password rather than the secret PINs of the individual accounts. Tool generated forms, similar to the lists generated by 'new account', which contain hidden information, like the 'command' field, can be used so that the password and one time code must only be entered once. For the timeout, a base machine giving the time must be added and the time when a one time code was first used must be stored on disk between program runs.

Exercise 4.3 (Simplified specification of accounts). As noted in Sect. 4.5.3, account numbers being unique they could be used as object identifiers for accounts in machine *Bank*. Remove the sets *ACCOUNT* and *accounts*, change the type of

accountNumber to *NAT* and the domain of the other account attributes to *accountNumber*, and constrain the cardinality of *accountNumber* to *maxAccounts*. Introduce the current specification as a refinement of the new one. Optionally, introduce the simplified specification as refinement of the current specification to gain an equivalence proof by mutual refinement.

Exercise 4.4 (Subtyping). Use subtyping modelled by subsetting to introduce two kinds of accounts. Savings accounts which get interest and cheque accounts without interest, but with the advantage that they allow overdrafts up to a certain limit.

Exercise 4.5 (Deleting customers and accounts). Provide for the deletion of customers and accounts. Be careful not to allow the deletion of accounts with non zero balance and of customers with accounts. Which invariants of the current system depend on the fact that deletion of customers and accounts is not possible?

Exercise 4.6 (Non-deterministic choice of error codes). If several preconditions of a transaction are not satisfied, the robust operations prescribes exactly which result code must be returned. For example, if *RobustNewAccount* is called with a non-existent customer when the account data base is full, *dbFull* rather than *unknownCustomer* must be reported. Respecify the robust operations so that the choice of the reported violated condition is arbitrary, thus avoiding overspecification.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. Th. Boutell. cgic: an ANSI C library for CGI programming. http://www.boutell.com/cgic/.
3. M. Büchi and W. Weck. A plea for grey-box components. In *Foundations of Component-Based Systems '97*, 1997. http://www.abo.fi/~mbuechi/.
4. M. Butler, J. Grundy, T. Långbacka, R. Ruksenas, and J. von Wright. The Refinement Calculator: Proof support for program refinement. In Lindsay Groves and Steve Reeves, editors, *Formal Methods Pacific'97: Proceedings of FMP'97*, Discrete Mathematics and Theoretical Computer Science, pages 40–61, Wellington, New Zealand, July 1997. Springer-Verlag.
5. M.B. Dwyer, V. Carr, and L. Hines. Model checking graphical user interfaces using abstractions. In *Proceedings of ESEC/FSE '97*, LNCS 1301, pages 244–261. Springer–Verlag, September 1997.
6. Ph. Facon et al. Mapping object diagrams into B specification. In *Methods Integration Workshop*, 1996.
7. I. Jacobson G. Booch, J. Rumbaugh. *Unified Modeling Language User Guide*. Addison-Wesley, 1998. http://www.rational.com/uml/.
8. D. Coleman, F. Hayes, and S. Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1), 1992.
9. J. V. Hill. *Microprocessor Based Protection Systems*. Elsevier, 1991.
10. IEC. Software for computers in the application of industrial safety-related systems, 1992. IEC 65A 122.
11. C.B. Jones. A rigorous approach to formal methods. *IEEE Computer*, pages 20–21, April 1996.
12. K. Lano. *The B Language and Method: A guide to Practical Formal Development*. Springer Verlag, 1996.
13. K. Lano. Integrating formal and structured methods in object-oriented system development. In S.J. Goldsack and S.J.H. Kent, editors, *Formal Methods and Object Technology*. Springer Verlag, 1996.
14. K. Lano and H. Haughton. *Specification in B: An Introduction Using the B Toolkit*. Imperial College Press, London, 1996.
15. Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Trans. Program. Lang. Syst.*, 18(3):325–353, May 1996.
16. Carroll Morgan. The generalised substitution language extended to probabilistic programs. In *Proceedings of B'98: the 2nd International B Conference*. LNCS, Springer Verlag, 1998. http://www.comlab.ox.ac.uk/oucl/groups/probs/bibliography.html.
17. D. S. Neilson and I. H. Sorensen. *The B-technologies: a system for computer aided programming*. B-Core (UK) Ltd., Oxford, U.K., 1996. Including the *B-Toolkit User's Manual, Release 3.2*.

18. J. Rumbaugh, M. Blaha, W. Premerlani, and F. Eddy. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
19. R. S. Pressman. *Software Engineering : A Practitioner's Approach*. McGraw Hill, 4th edition, 1996.
20. E. Sekerinski. Statecharts in B. In *Proceedings of the second B conference*, pages 182–197. LNCS 1393, Springer Verlag, April 1998.
21. R. Shore. Object-oriented modelling in B. In *Proceedingsof 1st Conference on the B method*, pages 133–154, 1996.
22. I. Sommerville. *Software Engineering*. Addison-Wesley, 5th edition, 1995.
23. Stéria Méditerranée. *Atelier-B*. France, 1996.
24. N. Storey. *Safety-Critical Computer Systems*. Addison-Wesley, 1996.
25. J. Wordsworth. *Software Engineering with B*. Addison-Wesley, September 1996.

Paper II

Compositional Symmetric Sharing in B

Martin Büchi and Ralph Back

Originally published in: Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *Proceedings of FM'99: World Congress on Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 431–451. Springer Verlag, September 1999.

Compositional Symmetric Sharing in B

Martin Büchi and Ralph Back

Åbo Akademi University
Turku Centre for Computer Science
Lemminkäisenkatu 14A, 20520 Turku, Finland
{Martin.Buechi, Ralph.Back}@abo.fi,
http://www.abo.fi/~{mbuechi, backrj}

Abstract. Sharing between B constructs is limited, both on the specification and the implementation level. The limitations stem from the single writer/multiple readers paradigm, restricted visibility of shared variables, and structural constraints to prevent interference. As a consequence, applications with inherent sharing requirements have to either be described as large monolithic constructs or be underspecified, leading to a loss of modularity respectively certain desirable properties being unprovable.
We propose a new compositional symmetric shared access mechanism based on roles describing rely/guarantee conditions. The mechanism provides for multiple writers on shared constructs, visibility of shared variables in the accessors' invariants, and controlled aliasing. Use is uniform in machines, refinements, and implementations. Sharing is compositional: all proof obligations are local and do not require knowledge of the other accessors' specifications, let alone their or the shared construct's implementation.
Soundness of the mechanism is established by flattening.

1 Introduction

The B method provides support for modularization and, herewith, for information hiding, compositionality of module operations, reusability of modules, and decomposition of proofs [4, 5]. Modules can be combined using a number of different mechanisms. Refinement being 'almost' monotonic with respect to the composition mechanisms, most proof obligations arise on a per module base. The few additional restriction on the global structure can be checked automatically. In this compositional approach, we can focus on a part of a large system, establish desired properties for this part, and be guaranteed that these properties hold in the complete system.

To achieve compositionality [8] and independent refinement, sharing is restricted in B. Sharing is based on the single writer/multiple readers paradigm. If several constructs access a shared construct, only one accessor (writer) can modify the state of the shared construct. The other accessors (readers) are limited to read-only access, respectively to calling inquiry operations. To ensure that invariants cannot be invalidated, only the single writer is allowed to reference variables of the shared construct in its invariant. Because of these limitations, applications with inherent sharing requirements cannot be handled satisfactorily, as described in Sect. 2.2.

We introduce a new sharing mechanism that overcomes the single writer and the variable visibility restrictions. Multiple constructs can have write access to a shared construct and reference shared variables in their invariants. The mechanism is compositional: all proof obligations arise on a per module base and only a few automatically checkable restrictions on the sharing graph are required for global correctness. The key element are freely specifiable accessor roles, which determine how the different accessors can use the shared construct. Adherence to these role specifications guarantees that the accessors do not invalidate each others invariants.

Role specifications can be considered as guarantee conditions in the sense of Cliff Jones [15] with the rely conditions being given by the other roles. Rely/guarantee conditions (also known as assumption/commitment specifications) have been developed as a compositional proof method for shared variable and message-passing concurrency with interleaving semantics. This paper shows that the same theory is also applicable to modular sequential systems with sharing.

Section 2 reviews the existing sharing mechanisms and illustrates a shortcoming on a concrete example. Throughout the paper we use numbered variations of the same example. In Sect. 3 we take the problem to its roots, analyzing the reasons for the existing restrictions. Section 4 introduces the new mechanism. We provide further details of the sharing mechanism in Sect. 5. In Sect. 6 we list the complete syntax, the proof obligations, the visibility rules, and the well-formedness criteria for the composition graph. Using flattening of constructs, we prove the soundness of the proposed mechanism in Sect. 7. Sect. 8 lists related work and draws the conclusions.

2 The Problem

2.1 Review of Existing Composition Mechanisms

B has three different constructs: machines, refinements, and implementations, distinguished by different syntactic restrictions. Machines express original specifications. Refinements are intermediate constructs. An implementation denotes the end of a refinement chain and contains executable code. In addition to behavioral specifications, refinements and implementations contain data refinement relations in form of gluing invariants. Machines can be parameterized. Parameters are instantiated by the single writer.

The B method has four mechanisms to compose constructs. The different mechanisms can be used in different constructs. The target of a composition is always a machine.

INCLUDES The *INCLUDES* clause can appear in machines and refinements. It can be understood as textual inclusion with the restriction that variables of the included machine can only be modified indirectly through operations of the included machine so that the invariant of the included machine is preserved. The including construct instantiates the parameters of the included machine and can reference variables of the included machine in the invariant. The including construct becomes the focus of refinement, the included construct doesn't have to be implemented unless it is also imported.

USES The *USES* clause can only appear in machines. It provides for limited sharing on the specification level. Any number of machines can use a shared machine. All using and the used machine must be included into a common machine, which becomes the focus of refinement. The using machines cannot be refined. They have read-only access to the shared machine and can reference shared variables in their invariants. To guarantee that the including machine, which is the only writer, does not invalidate the invariants of the using machines, the using machines' invariants have to be proved upon inclusion.

SEES The *SEES* clause can appear in any construct. It provides for read only access to a shared machine. Variables of the seen machine cannot be referenced in the invariant of the seeing construct. Without this restriction, the invariant of the seeing machine could be invalidated by the construct with write access to the seen machine.

IMPORTS The *IMPORTS* clause can only appear in implementations. The importing machine instantiates the parameters of the imported machine, can call both inquiry and modification operations and can reference variables of the imported machine in its invariant. Imported machines can be seen by any number of other constructs.

Summary The *INCLUDES* and *USES* clauses can be considered as weak or syntactic relations [5]. Their aim is to combine text of machine specifications; this structure is not reflected in subsequent refinements or in the final implementation. *SEES* and *IMPORTS* on the other hand are strong relations as the shared code will remain visible as a module in the final implementation.

2.2 A Problem with the Existing Mechanisms

In this subsection, we illustrate a shortcoming of the existing sharing mechanisms with a concrete example. The example's main characteristic are its inherent sharing requirements.

We consider a control system for a manufacturing plant consisting of various devices, such as robot arms and conveyor belts. Each device is controlled by its own software module, the central *Run* operation of which is periodically called by a scheduler. Whenever a device controller notices an error, the device is stopped and an alarm is registered in a central database. The plant operator can list the active alarms on the screen and deactivate alarms after fixing their cause. The devices check whether all their alarms have been deactivated and if so resume work.

The database is shared between all the device controllers and the monitoring console. They all need both read and write access: the device controllers need to check for active alarms and enter new alarms, the monitoring console needs to list active alarms and change the activity status of alarms.

How do we specify, refine, and implement such an architecture using B's existing composition mechanisms? Let us first look at the specification. Since the devices are to a certain degree independent and since multiple instances of the same device type can

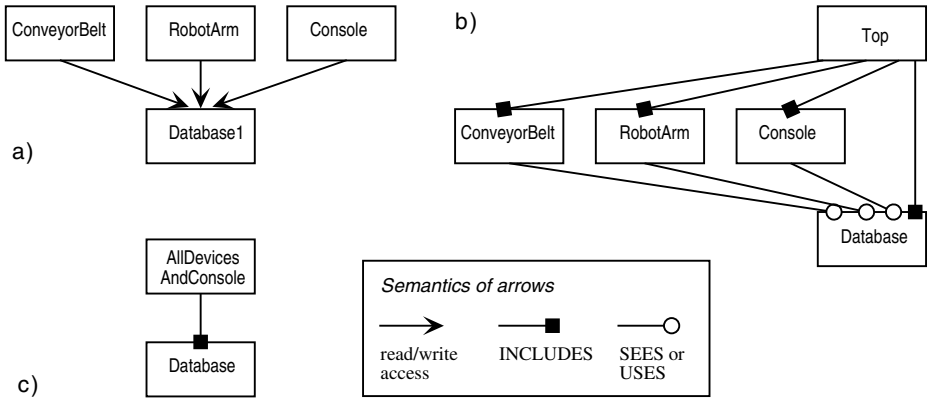


Fig. 1. Specification Approaches

exist, it makes sense to specify them modularly using separate machines. Likewise, the database is captured by a separate machine, which is accessed by the device controllers and the monitoring console (Fig. 1 a). As remarked above, all accessors need both read and write access to the shared database. However, the existing sharing mechanisms are limited by the single writer paradigm (Sect. 2.1).

This sharing architecture is possible with *SEES* or *USES*, if the called operations, such as *NewAlarm* in *Database1*, are specified as inquiry only – although their implementations modify the concrete state:

MACHINE Database1
OPERATIONS
NewAlarm(type) $\widehat{=}$ **PRE** type $\in$ NAT **THEN skip END**;
bb $\leftarrow$ ActiveAlarms $\widehat{=}$ **BEGIN** bb :$\in$ BOOL **END**;
...
END

Unfortunately, this underspecification precludes any sensible reasoning. Because there is no set of active alarms in *Database1*, we cannot express the fact that the conveyor belt is only running if none of its alarms are active. Likewise, we cannot prove that an alarm will remain active until acknowledged by the operator. In conclusion, this architecture cannot be applied satisfactorily in B.

An alternative architecture is based on a top machine *Top* as single writer to the database, to which the device controllers and the monitoring console have only read access employing *SEES* or *USES* (Fig. 1 b). In this scenario, the device controllers cannot register alarms in the database directly. Instead, they have to return the corresponding information upon being called by *Top*, which in turn enters the alarms into the database. This becomes rather cumbersome if there are intermediate machines through which the information has to be passed or if the information does not have a constant length.

An additional problem becomes apparent when looking at the active alarms display operation of the monitoring console. This operation has to get a list of all active alarms.

This list not being constant in size, it cannot be returned by a single operation call. Instead, the elements have to be retrieved one by one –like the results of an SQL query in C. It is often simpler if the database maintains a set of already returned elements (See e.g. [6] for details of such a retrieval operation.) rather than requiring the search criteria together with a resume index to be passed with every call. However, if the retrieval operation updates a variable, it cannot be called by the monitoring console with read-only access. This problem could again be 'solved' by an undesirable underspecification.

Employing *SEES* in the device controllers we would be forced to specify properties relating devices and the database (e.g., the conveyor belt is only running if none of its alarms are active) in the *Top* machine rather than in the device controllers. With *USES* on the other hand, we would be allowed to reference variables of the database in the invariants of the device controllers, but would not be allowed to refine the latter leading either to a monolithic implementation of *Top* or a difficult to manage almost duplication of constructs.

In any case, the all including construct *Top* becomes the single focus of refinement without any direct support for architectural structure preserving refinement. Whereas it is definitely beneficial that B does not force the specification and implementation structures to be identical, the example shows that there are cases where more support for structure preserving refinement would be needed.

The problems of access restrictions can be overcome by merging all device controllers and the monitoring console into one big machine (Fig. 1 c). This, however, leads to a loss of modularity and, herewith, of information hiding, compositionality, reusability (e.g. multiple instantiation if we have several conveyor belts), and decomposition of proofs.

The same problems reoccur at the refinement and implementation levels, where we are also restricted by the single writer approach and the limitations of shared variables visibility. The above problems are not limited to our specific example. Further motivation to analyze the reasons for the current restrictions and to suggest new mechanisms are, e.g., given by [22, Chapters 4, 5, 6, and 7].

3 Analysis of the Problem

In this section, we analyze the reasons for the single writer and shared variable visibility restrictions. In a nutshell, the restrictions are due to interference that would contradict compositionality and independent refinement by invalidating local proofs.

Consider a variation of the plant control system where alarms are set on the console. The conveyor belt simply adjusts its execution status based on whether there are active alarms in the system. The shared machine *Database2* contains a variable $activeAlarms \subseteq NAT$. The conveyor belt is specified as follows, using the keyword *UTILIZES* to indicate some sort of sharing access:

MACHINE ConveyorBelt2
UTILIZES Database2
VARIABLES running
INVARIANT running$\in$BOOL $\wedge$ (running=TRUE $\Rightarrow$ activeAlarms=$\emptyset$)

When the console sets an alarm it invalidates the invariant of *ConveyorBelt2* if *running* is *TRUE*. A construct with write access to a shared machine may invalidate any other accessor's invariant, if the latter references variables of the shared machine.

Such undesirable interferences cannot be ruled out with local proofs for any of the three machines *Database2*, *ConveyorBelt2*, or *Console2*. They require either a global approach or a modular approach with noninterference proofs like [20]. In both cases we would loose the benefits of independent refinement provided by a compositional theory [25].

Hence we have to choose between having multiple writers without the possibility to reference variables of the shared machine in any of the accessors' invariants or the current single writer paradigm. Not being able to reference variables of the shared machine in any of the accessors' invariants is too restrictive, precluding the proving of many properties. For example, we cannot prove that the conveyor belt is only running if there are no active alarms because the variable *activeAlarms* of *Database2* is not visible in the invariant of *ConveyorBelt2*.

The same problems of destroying each other's invariants exist on the refinement and implementation levels. In addition to the local invariant, also the gluing invariant, expressing the data refinement relation, could be invalidated if we were to allow multiple writers [23].

On the positive side, we can note that multiple writers never invalidate the invariant of the shared machine as all modifications are done through operation calls.

4 Role-Based Access

To guarantee interference freedom among multiple accessors of a common machine, only the possible modifications to the shared variables are relevant. We define these effects in form of access *roles* as part of the shared machine. Accessing constructs declare which role(s) they play. The accessors guarantee to perform only modifications allowed by the declared role(s). In return, they can rely on the other accessors adhering to their roles. Let construct A accesses a shared machine in role R_1. If the other roles $R_2, \ldots, R_n$ maintain the invariant of A, then any accessor in role R_i $(i \in 2..n)$ maintains the invariant of A. Thus, we can both specify and refine accessor A without knowing the other accessors' specifications or implementations.

Because a library machine might foresee multiple sharing scenarios and because a custom machine might be used in multiple instances with different sharing, we allow the definition of multiple *contracts* with different roles.

4.1 Role Specifications

We illustrate the concept on our plant control system. This time, the conveyor belt creates the alarms reacting to sensors and an emergency stop button. The monitoring console is used to deactivate alarms. *Database3* defines a contract *SingleDevice* with two roles *Creator* and *Controller*, intended to capture the accesses by *ConveyorBelt3* and *Console3* respectively. An instance of *Database3* with contract *SingleDevice* can have at most two accessors, one for each role.

```
MACHINE Database3
CONTRACTS
  SingleDevice ≙
    Creator = ANY type WHERE type∈NAT THEN NewAlarm(type) END,
    Controller = ANY aa WHERE aa∈activeAlarmsTHEN ResetAlarm(aa) END
VARIABLES alarms, activeAlarms, alarmType
INVARIANT alarms⊆NAT ∧ activeAlarms⊆alarms ∧ alarmType∈alarms→NAT
INITIALISATION alarms, activeAlarms, alarmType:=∅, ∅, ∅
OPERATIONS
  aa ← NewAlarm(type) ≙
    PRE type∈NAT THEN
      ANY nn WHERE nn∈NAT-alarms THEN
        aa, alarms:=nn, alarms∪{nn} ||
        activeAlarms, alarmType(nn):=activeAlarms∪{nn}, type
      END
    END;
  ResetAlarm(aa) ≙ PRE aa∈activeAlarms
    THEN activeAlarms:=activeAlarms-{aa} END;
  nof ← NofActiveAlarms ≙ nof:=card(activeAlarms);
  ...
END
```

We specify the set of alarms as a subset of *NAT* and the active alarms as a subset of all alarms. The attribute *alarmType* is a functions from *alarms* to *NAT*. More on this approach of mapping records/classes to B, including proper treatments of finiteness, can be found in [18, 6].

4.2 Accesses

The machine *ConveyorBelt3* declares that it accesses the *Database3* as *Creator* in a *SingleDevice* contract. We use a '*!*' as separator of the qualified identifier because the dot is reserved for possible renaming.

```
MACHINE ConveyorBelt3
ACCESSES Database3!SingleDevice AS Creator
VARIABLES running
INVARIANT running∈BOOL ∧ (running=TRUE ⇒ activeAlarms=∅)
INITIALISATION running:=TRUE
OPERATIONS
  rr ← Run ≙
    CHOICE
      ANY type WHERE type∈0..8
        THEN NewAlarm(type) || running, rr:=FALSE, FALSE END
    OR running, rr:=bool(activeAlarms=∅), bool(activeAlarms=∅)
    END;
  EmergencyStop ≙ BEGIN NewAlarm(9) || running:=FALSE END
END
```

The operations *Run* and *EmergencyStop* have to act as refinements of *Creator* [] *skip* on the state space of the *Database*, which is clearly the case. This also implies, that inquiry operations can be freely called by any accessor.

Furthermore, we need to show that another construct, accessing *Database3* in the second role *Controller* cannot invalidate the invariant of *ConveyorBelt3*. To this aim, we show that the role specification *Controller* executed like an operation on the combined state space of *Database3* and *ConveyorBelt3* maintains the latter's invariant. This is the case, because *Controller* can only deactivate alarms. Deactivation is unproblematic because the second conjunct of the invariant of *ConveyorBelt3* is an implication and not an equality.

4.3 Refining and Implementing Accesses

Refinements and the implementation of *ConveyorBelt3* have to make the same changes to the variables of the shared machine *Database3*.

We assume *Motor3* to be a machine controlling the power of the motor and *Sensor3* a sensor that is activated if a load on the conveyor belt is about to fall off the edge.

```
IMPLEMENTATION ConveyorBelt3′ REFINES ConveyorBelt3
ACCESSES Database3!SingleDevice AS Creator
IMPORTS M.Motor3, S.Sensor3
INVARIANT running=M.on
OPERATIONS
  rr ← Run ≙
    VAR ss, nof IN
      ss ← S.ReadSensor;
      IF ss=TRUE THEN M.ShutOff; NewAlarm(0)
      ELSE nof ← NofActiveAlarms; IF nof=0 THEN M.TurnOn END
      END
    END;
  EmergencyStop ≙ BEGIN M.ShutOff; NewAlarm(9) END
END
```

Instead of accessing the database itself, the implementation *ConveyorBelt3′* could also import another machine that accesses the database and performs the changes.

4.4 Instantiation

In the existing composition mechanisms, the single writer also instantiates the machine parameters of the utilized machine. In our new mechanism, instantiation is separate from access using the *INSTANTIATES* clause, which specifies the machine, the contract, and the values of the parameters, if any. For example, we might have an implementation *Main3′*, which imports the accessors and instantiates the shared database:

```
IMPLEMENTATION Main3′ REFINES Main3
INSTANTIATES Database3!SingleDevice
IMPORTS ConveyorBelt3, Console3
```

Every accessed copy of a shared machine must be instantiated exactly once in an implementation, naming the same contract as all accessors. Renaming can be performed in the *ACCESSES* and *INSTANTIATES* clauses, thus allowing multiple instances with possibly different contracts. The renaming of the construct containing the *INSTANTIATES* clause determines the number of instances.

5 Further Aspects of Role-Based Access

5.1 Replicated Roles

In the previous section we have used role-based access for a plant control with a single device. In reality, we have many devices, which all have almost identical role specifications. Rather than requiring textual duplication, we introduce a replication mechanism over a constant set. Thus, we can define the role *Creator* of *Database4* as follows:

MACHINE Database4
CONTRACTS
MultipleDevices $\widehat{=}$
Creator(no$\in$0..19) $=$
ANY type **WHERE** type$\in$10$\times$no..10$\times$no+9 **THEN** NewAlarm(type) **END**,

This example definition allows 20 accessors in the role of creators, one for each value between 0 and 19. The replicator *no* may be used inside the scope of the role definition like a constant. A construct that accesses a shared machine in a replicated role has to indicate its replication value. The conveyor belt could be defined as:

MACHINE ConveyorBelt4
ACCESSES Database4!MultipleDevices **AS** Creator(0)

For the non-interference proofs the other replicated roles have to be considered like different role specifications. In the example, we would have to prove that *Creator(nn)* for *nn*$\in$*1..19* maintains the invariant of *ConveyorBelt4*. This is the case if we adapt the invariant of *ConveyorBelt3* as below and adjusting the *Run* operation correspondingly.

INVARIANT
running$\in$BOOL $\wedge$ (running=TRUE $\Rightarrow$ activeAlarms$\cap$(alarmType^{-1}[0..9])=$\emptyset$)

5.2 Form of Role Specifications

As shown in the examples, role specifications take the format of normal B operations. Traditionally, rely/guarantee conditions are expressed as predicates over the current and the next state of variables. However, we feel that operation-like specifications are more in line with B.

As a guiding principle, we allow the same statements as in operations of a refinement that includes the accessed machine. Thus, multiple substitutions, sequencing, and nondeterministic choice are all allowed, but loops are not. Variables can be read directly, but modified through operation calls only. To gain sufficient expressiveness, either loops or direct modifications should be made legal.

We do not have to prove that operation calls in role specifications satisfy the preconditions of the called operations, because we perform these proofs for the actual accessors. As an engineering aid, the tools should nevertheless support conformance proofs for role specifications. Precondition violating role specifications do not give the accessor more freedom, but make the non-interference proofs for other accessors more difficult.

The role specifications, like any module interfaces, should be very simple compared to the code of the actual accessors. Hence, roles are described like operations, rather than full machines with variables that maintain their values between calls. Such specifications would require full-blown construct refinement with gluing invariants between role and local variables in accessors and also more complex non-interference proofs. The simple format suffices in most cases and is, combined with some coding tricks modifying the operation specifications, as general as full machines.

The relative simplicity of the role specifications compared to the code of the actual accessors reduces the complexity of the non-interference proofs. The simplification of the non-interference proofs for all other accessors on all refinement levels by far outweighs the additional burden of the single role adherence proof.

An overly weak role specification makes it easy to prove role adherence, but impossible to guarantee non-interference for other accessors. An overly strong specification causes the opposite problem. Writing the role specifications is a design step, like any other definition of module interaction.

5.3 Adherence to Role Specifications

An operation of an accessor adheres to a role specification if it either acts as a refinement of the specification or as *skip* on the state space of the accessed machine. We add *skip* as an implicit choice to every role specification. This corresponds to the guarantee condition being reflexive [15], respectively the stuttering transition being built into the semantics [3, 17].

Whether an operation O that refines a role specification R is called multiple times or whether O acts as a refinement of $R^{\wedge}$ $(=skip[]R[](R;R)[]\ldots)$ has the same effect for the other accessors. This corresponds to the guarantee condition being transitive [15], respectively mumbling being built into the semantics. Explicitly allowing mumbling makes the proofs more difficult and can –provided we allow direct write access or loops in role specifications– always be replaced with a weaker role specification. For simplicity, we do not consider mumbling in this paper.

In the initialization of the accessors we only allow inquiry operations of the accessed machine to be called. Otherwise we would have to define an order in which the accessors are initialized and could not assume the shared machine to be in its initial state when the accessors are initialized. The initializations acting as *skip* on the accessed machine, they automatically adhere to the role specifications.

5.4 Sharing Structure

Sharing is used to get multiple access paths to the same data. In the presence of independent refinement, we need some structural restrictions to control aliasing. In this section, we give two examples of what could go wrong without such structural restrictions. A full account of the restriction is given in Sect. 6.4.

We adopt the following notation in figures: The primed constructs are the implementations refining the unprimed machines. Multiple instances of an accessed machine –if present– are graphically indicated by duplication to make collaborations clear. We

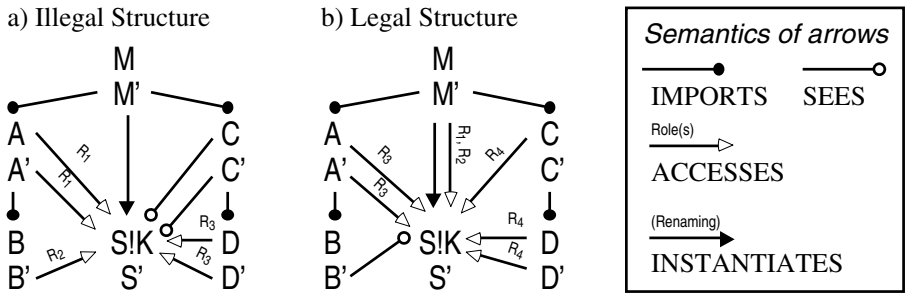


Fig. 2. Illustration of Legal and Illegal Composition Graphs

append the name of the actual contract to the name of the accessed machine. The access arrows are adorned by the role(s) and possibly the replication values, the instantation arrow by the possible renaming. In a slight abuse of notation, it would also be possible to visualize roles as UML style interfaces (circles) attached to the shared machine. However, our notion of roles and that of UML interfaces is not identical because our roles contain guarantee conditions rather than the signatures of callable operations.

Consider a machine A that accesses a shared machine S (left branch of Fig. 2 a). The implementation A' also accesses S and furthermore imports B. Machine B does not access S, but the implementation B' does. Even if we locally prove that the operations of A' act as a refinements of the operations of A, this property might not hold in the complete system. A' may call operations of B and, thereby, unknowingly modify S. This might lead to S being modified differently than specified in A, B' observing S in a state where the gluing invariant of B' does not hold, and A' violating preconditions of operations of S. This problem is due to A' accessing S both traceably and untraceably. The problem is not bound to B only accessing S in the implementation. An invisible access could also be created if B' would not access S directly, but import a machine E that accesses S.

Without constraints on the composition graph, also the interaction between the old and the new composition mechanism can lead to problems. A seeing construct C' (right branch of Fig. 2 a) assume that the state of the seen machine S does not change during the execution of an operation of C'. To enforce this, the seeing construct C' can only call inquiry operations. In proofs of C', no substitutions are made on the state of the seen machine S. If C' could indirectly modify S, global correctness would be invalidated. This could, for example, happen if the seeing machine imports a third machine D, the implementation of which accesses S. Thus, we need restrictions on the structure of the development to ban such architectures.

5.5 Emulating the Existing Composition Mechanisms

The existing composition primitives *IMPORTS* and *SEES* can be emulated using *ACCESSES* and *INSTANTIATES* as follows: A contract permitting a single writer with full access rights and an infinite number of readers with a *skip* role specification is added

to the shared construct. Then, *IMPORTS* can be replaced with an access in the writer role and an instantiation. The *SEES* clauses are replaced with accesses in the replicated reader role. Because the existing mechanisms *IMPORTS* and *SEES* capture a frequent special case and because abolishing them would require more complicated global restrictions based not only on the structure but also the semantics of roles, it makes sense to have all mechanisms at our disposition.

Promotion of operations (turning operations of a utilized machine into proper operations of the utilizing construct) would only be possible in combination with *ACCESSES* in trivial cases where the other accessors do not make any observable modifications (e.g., for the single writer in the above emulation contract). Furthermore, promotion of operations is much less important with *ACCESSES*, because the latter provides for multiple writers. Therefore, we do not consider the promotion of operations from accessed machines, but rather count promotion as a further reason to also keep the existing import mechanism.

The *USES* mechanism cannot be emulated because it dictates that the used machine be included into another machine whereas an accessed machine cannot be included. *INCLUDES*, being a copying rather than a sharing mechanism, cannot be emulated with *ACCESSES*.

6 Formal Definitions

6.1 Syntax

We give the following extended syntax definitions [2, p 715ff] for machines, refinements, and implementations:

MACHINE
  Machine_Header
CONSTRAINTS
  Predicate
CONTRACTS
  Contract_List
ACCESSES
  Access_List
INSTANTIATES
  Inst_List
USES
...

REFINEMENT
  Machine_Header
REFINES
  Id_List
ACCESSES
  Access_List
INSTANTIATES
  Inst_List
SETS
...

IMPLEMENTATION
  Machine_Header
...
VALUES
  Predicate
ACCESSES
  Access_List
INSTANTIATES
  Inst_List
IMPORTS
...

Contract_List, *Access_List*, and *Inst_List* are defined as follows:

Syntactic Category	Definition
Contract_List	Contract; Contract_List Contract
Contract	Contract_Name $\widehat{=}$ Role_List

Syntactic Category	Definition
Role_List	Role, Role_List Role
Role	Role_Name = Statement Role_Name(Replicator ∈ Set) = Statement
Access_List	Access; Access_List Access
Access	Machine_Name!Contract_Name **AS** Acc_Role_List Renamed_Name.Machine_Name!Contract_Name **AS** Acc_Role_List
Acc_Role_List	Acc_Role, Acc_Role_List Acc_Role
Acc_Role	Role_Name Role_Name(Simple_Term)
Inst_List	Inst, Inst_List Inst
Inst	Machine_Name!Contract_Name Machine_Name!Contract_Name(Expression_List) Renamed_Name.Machine_Name!Contract_Name Renamed_Name.Machine_Name!Contract_Name(Expression_List)

Contract_Name, *Role_Name*, *Replicator*, *Machine_Name*, and *Renamed_Name* all stand for *Identifier*. The form of the role specification is discussed in detail in Sect. 5.2.

6.2 Proof Obligations

We give the proof obligations for machines and implementations containing an *ACCESSES* clause. The rules for refinements are analogous. As noted in Sect. 5.2, the *CONTRACTS* clause does not give rise to any proof obligations. We leave out sets, constants and assertions as the respective proof obligations are unchanged. Figure 3 gives an overview of the proof obligations.

MACHINE $M_s(P_s)$
CONSTRAINTS C_s
CONTRACTS
  $K \mathrel{\widehat{=}}$
    $R_1 = F_1$,
    $R_2 = F_2$
VARIABLES X_s
INVARIANT I_s
INITIALISATION U_s
OPERATIONS
  $u_s \leftarrow O_s(w_s) \mathrel{\widehat{=}}$
    PRE Q_s **THEN** V_s **END**
END

MACHINE $M_1(P_1)$
CONSTRAINTS C_1
ACCESSES
  M_s!K **AS** R_1
VARIABLES X_1
INVARIANT I_1
INITIALISATION U_1
OPERATIONS
  $u_1 \leftarrow O_1(w_1) \mathrel{\widehat{=}}$
    PRE Q_1 **THEN** V_1 **END**
END

IMPLEMENTATION
  $M'_1(P_1)$
REFINES M_1
ACCESSES
  M_s!K **AS** R_1
CONCRETE_VARS
  X'_1
INVARIANT I'_1
INITIALISATION U'_1
OPERATIONS
  $u_1 \leftarrow O_1(w_1) \mathrel{\widehat{=}}$
    BEGIN V'_1 **END**
END

We use the abbreviations A_1 for $P_1 \in \mathcal{P}_1(\mathrm{INT})$ and A_s for $P_s \in \mathcal{P}_1(\mathrm{INT})$. Occurrences of O_s in U_1, V_1, U'_1, V'_1, F_1, and F_2 should be replaced by V_s with the parameters substituted accordingly [2, p 314ff]. As in [2] we do not make this substitution explicit in the proof obligations.

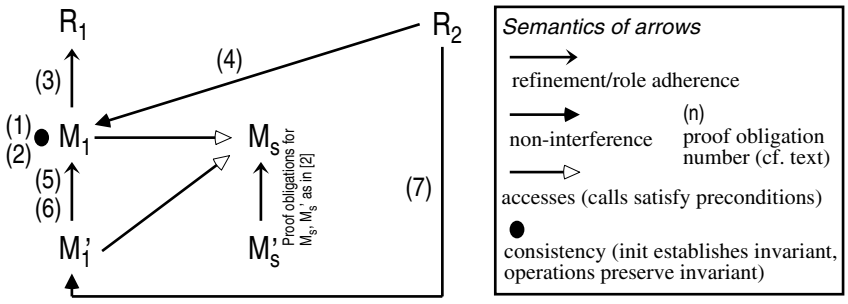


Fig. 3. Proof Obligations for an Accessing Machine and Implementation

Machine M_1 The first proof obligation of M_1 states that the initialization must establish the invariant. The role of the accessed machine is similar to the one of an included machine, except that its parameters are not actualized [2, p 331ff].

$$A_1 \wedge C_1 \wedge A_s \wedge C_s \Rightarrow [U_s][U_1]I_1 \tag{1}$$

The next obligation concerns the preservation of the invariant of the accessing machine by its operations:

$$A_1 \wedge C_1 \wedge I_1 \wedge Q_1 \wedge A_s \wedge C_s \wedge I_s \Rightarrow [V_1]I_1 \tag{2}$$

The third obligation states that the operations of the accessor must conform to the declared role. Note that there is no corresponding obligation for the initialization because the latter may not call modification operations of the accessed machine. Because both the role specification F_s and the operation O_1 operate on X_s, renaming must be performed. Let $\hat{X}_s$ be a fresh set of variables, then we get

$$A_1 \wedge C_1 \wedge I_1 \wedge Q_1 \wedge A_s \wedge C_s \wedge I_s \wedge \hat{X}_s{=}X_s \Rightarrow [[X_s{:=}\hat{X}_s]V_1]\neg[F_1 \,[\!]\, \mathbf{skip}]\neg(\hat{X}_s{=}X_s) \tag{3}$$

If a construct accesses a machine in multiple roles, its operations have to conform to the nondeterministic choice of the two roles. Thus, if M_1 were to access M_s as R_1 and R_2, then F_1 would have to be replaced by $F_1 [\!] F_2$.

The fourth obligation concerns the interference freedom by all other roles, which in our case is only R_2.

$$A_1 \wedge C_1 \wedge I_1 \wedge A_s \wedge C_s \wedge I_s \Rightarrow [F_2]I_1 \tag{4}$$

For replicated roles, we have to prove non-interference for all replication values except for the one of the accessor in question. Let us assume the following replications $R_1(g_1{\in}G_1)$ and $R_2(g_2{\in}G_2)$ and let M_1 access M_s as $R_1(h_1)$. Then we get the following three obligations:

$$h_1{\in}G_1 \tag{4'}$$
$$A_1 \wedge C_1 \wedge I_1 \wedge A_s \wedge C_s \wedge I_s \wedge g_1{\in}G_1\text{-}\{h_1\} \Rightarrow [F_1]I_1$$
$$A_1 \wedge C_1 \wedge I_1 \wedge A_s \wedge C_s \wedge I_s \wedge g_2{\in}G_2 \Rightarrow [F_2]I_1$$

Replication not only avoids duplication of role specifications, it also leads to a reduction of the overal proof burden by combining many similar non-interference obligations.

The proof obligations for operation calls (satisfy precondition) are unchanged.

Implementation M'_1 For the implementation M'_1 we have 3 proof obligations. The first two proof obligations concerning initialization and operation refinement are similar to those of an implementation that imports another machine [2, p 597ff].

$$\mathsf{A}_1 \wedge \mathsf{C}_1 \wedge \mathsf{A}_s \wedge \mathsf{C}_s \Rightarrow [\mathsf{U}_s][\mathsf{U}'_1]\neg\,[\mathsf{U}_1]\neg\mathsf{I}'_1 \tag{5}$$

The second proof obligation is for the operation refinement. The 1-to-1 data refinement of the shared variables is explicit in this obligation *($\hat{X}_s$=X_s)*.

$$\begin{aligned}&\mathsf{A}_1 \wedge \mathsf{C}_1 \wedge \mathsf{I}_1 \wedge \mathsf{I}'_1 \wedge \mathsf{Q}_1 \wedge \mathsf{A}_s \wedge \mathsf{C}_s \wedge \mathsf{I}_s \wedge \hat{X}_s{=}\mathsf{X}_s \Rightarrow \\ &\quad [[\mathsf{u}_1{:=}\hat{u}_1][\mathsf{X}_s{:=}\hat{X}_s]\mathsf{V}'_1]\neg[\mathsf{V}_1]\neg([\mathsf{X}_s{:=}\hat{X}_s]\mathsf{I}'_1 \wedge \hat{u}_1{=}\mathsf{u}_1 \wedge \hat{X}_s{=}\mathsf{X}_s)\end{aligned} \tag{6}$$

For sharing in the implementation only, we have to prove adherence rather than 1-to-1 data refinement in the implementation. The third and last obligation concerns the interference freedom. As noted above for machines, it should be replicated if some of the roles are.

$$\mathsf{A}_1 \wedge \mathsf{C}_1 \wedge \mathsf{I}_1 \wedge \mathsf{I}'_1 \wedge \mathsf{A}_s \wedge \mathsf{C}_s \wedge \mathsf{I}_s \Rightarrow [\mathsf{F}_2]\mathsf{I}'_1 \tag{7}$$

If M or M'_1 also instantiates M_s, say P_s with N_s, then A_s can be replaced by the stronger predicate P_s=N_s in the above proof obligations. In this case we have the additional proof obligation that the actual parameters satisfy the constraints, as for *INCLUDES* and *IMPORTS*.

6.3 Visibility Rules

For brevity, we only summarizes some key aspects of the visibility rules here. In the *CONTRACTS* clause we allow only read access to variables. A construct's own sets and constants as well as those of seen machines are allowed as parameters of instantiations. To prevent cyclic dependencies, sets and constants of included, used, imported, and accessed machines are, on the other hand, not visible in the *INSTANTIATES* clause.

Only a construct's own sets and constants and those of seen machines, but not those of imported machines may be used as parameters of instantiations. In their initializations, accessors can call only inquiry operations of an accessed construct.

If a construct A only instantiates, but does not access B, then none of the objects of B are visible in A. Like *SEES*, but unlike *INCLUDES*, *ACCESSES* is not transitive. If machine A includes, uses, sees, imports, or accesses B and B accesses C, then the objects of C are not visible in A. It is, however, possible that A also accesses C (Fig. 2 b).

6.4 Well-Formedness of the Composition Graph

The well formedness criteria for the composition graph concerning *ACCESSES* to guarantee global correctness are presented below. They are simple enough to be checked automatically.

Similar checks are already performed for the existing composition mechanisms [2, 21]. For simplicity, we talk about 'accessed machines' instead of renamed instances thereof.

The following conditions can be verified by the type checker on a per-construct base:

1. If a machine, a refinement, or an included machine thereof accesses a machine M_s as R of contract K then this construct's implementation must either access M_s as R of contract K or import exactly one machine that contains such an access.
2. If a machine, a refinement, or an included machine thereof accesses a machine M_s as $R_1, \ldots, R_i$ of contract K then this construct's implementation may not access M_s in any other roles nor import a machine accessing M_s in any other roles. (The proof obligation for operation refinement would not allow modifications not covered by $R_1, \ldots, R_i$ anyhow.)
3. A construct and one of its included machines cannot access the same machine in the same role.
4. If a machine, a refinement, or an included machine thereof contains an instantiation, then all further refinements and the implementation must either contain the same instantiation with the same parameters or include/import without renaming a machine containing such an instantiation.

The following conditions must be checked globally for complete projects:

1. Every accessed machine is instantiated exactly once in an implementation.
2. Every shared machine is accessed at most once in each role, respectively for each replication value, by an implementation
3. All accesses and the instantiation of a machine are for the same contract.
4. An accessed machine cannot be included or imported. This also implies that neither a used nor a using machine can be accessed.
5. A seen machine must either be instantiated or imported.

To present the remaining architectural condition, we extend the notation of [21]. The relational notation is as in B: '$^+$' denotes the transitive non-reflexive closure, '*' the transitive and reflexive closure, and ';' composition.

1. M_1 *sees* M_2 iff the implementation of M_1 sees the machine M_2.
2. M_1 *m_sees* M_2 iff machine M_1 sees machine M_2.
3. M_1 *imports* M_2 iff the implementation of M_1 imports the machine M_2.
4. M_1 *accesses* M_2 iff the implementation of M_1 accesses the machine M_2.
5. M_1 *depends_on* M_2 iff the implementation of M_1 is built utilizing M_2: $depends\_on \mathrel{\widehat{=}} (sees \cup imports \cup accesses)^+$.
6. M_1 *can_alter* M_2 iff the implementation of M_1 can alter the variables of M_2: $can\_alter \mathrel{\widehat{=}} depends\_on^*; (imports \cup accesses)$.

7. M_1 *any_accesses* M_2 iff M_1, one of its refinements, or its implementation accesses the machine M_2.
8. M_1 *(imp_acc* M_s*)* M_2 iff the implementation of M_1 imports the machine M_2 and M_2 accesses M_s.
9. M_1 *traceably_accesses* M_2 iff M_1 accesses M_2 through a chain of imports, in which all machines access M_2:
   M_1 *traceably_accesses* $M_2 \mathrel{\widehat{=}} M_1$ *(imp_acc* M_2*)***; accesses* M_2.
10. M_1 *untraceably_accesses* M_2 iff M_1 indirectly accesses M_2 in a way other than an imports chain, in which all machines access M_2:
    untraceably_accesses $\mathrel{\widehat{=}}$ *(depends_on; accesses)-traceably_accesses*.
11. M_1 *instantiates* M_2 iff the implementation of M_1 instantiates the machine M_2.
12. M_1 *references* M_2 iff the implementation of M_1 references the machine M_2: *references* $\mathrel{\widehat{=}}$ *(sees*$\cup$*imports*$\cup$*accesses*$\cup$*instantiates)*$^+$.
13. *id* is the identity relation.

The composition graph must then satisfy the following condition:

$$((\text{sees} \cup \text{imports} \cup \text{accesses}); \text{can\_alter}) \cap \qquad \text{(i)}$$
$$(((\text{imports} \cup \text{accesses}); \text{m\_sees}^+) \cup (\text{sees}; \text{m\_sees}^*)) = \emptyset \wedge$$
$$\text{any\_accesses} \cap \text{untraceably\_accesses} = \emptyset \wedge \qquad \text{(ii)}$$
$$\text{references} \cap \text{id} = \emptyset \qquad \text{(iii)}$$

The first conjunct states that a seen machine must not be modified. The second conjunct asserts that no construct accesses the same machine directly and untraceably. The third conjunct excludes cyclic dependencies.

The right branch of Fig. 2 a) violates the first conjunct of the above condition, the left branch violates the second conjunct. M' not accessing S has nothing to do with the violations; the corresponding access in Fig. 2 b) is just shown as an additional option.

7 Soundness

In this section we give a partial proof of the soundness of our new shared access mechanism. We syntactically merge a shared machine and all its accessing machines into a new machine and the implementation of the shared machine along with the implementations of the accessors into a new implementation. Then we show that all the proof obligations of these two constructs, which do not contain the new mechanism, are implied by the obligations of the individual constructs. Namely, the invariant of the merged machine holds and the implementation is a correct refinement.

Because we have substitutions of both V_s and V'_s for O_s, we have to indicate which body is used. We write $[O_s \backslash V_s]$ for this extended substitution which includes the parameters, e.g., $[O_s \backslash V_s]$ $(a \leftarrow O_s(b))$ equals $[u_s, w_s{:=}a, b]V_s$ if u_s is the output and w_s the input parameter of O_s. We assume here that operations are not recursive.

Let M_s, M_1, and M'_1 be as in Sect. 6.2. Furthermore, let M_2 and M'_2 be like M_1 and M'_1 respectively, but with index '2'. With M'_S as in Fig. 4, we get the two merged constructs M and M' (Fig. 4). Note that V'_s gets substituted for O_s in the implementation M'.

IMPLEMENTATION
  $M'_s(P_s)$
REFINES M_s
CONCRETE_VARS
  X'_s
INVARIANT I'_s
INITIALISATION U'_s
OPERATIONS
  $u_s \leftarrow O_s(w_s) \mathrel{\widehat{=}}$
    BEGIN V'_s **END**
END

MACHINE $M(P_1, P_2, P_s)$
CONSTRAINTS
  $C_1 \wedge C_2 \wedge C_s$
VARIABLES X_1, X_2, X_s
INVARIANT $I_1 \wedge I_2 \wedge I_s$
INITIALISATION
  $U_s; [O_s \backslash V_s](U_1 \parallel U_2)$
OPERATIONS
  $u_1 \leftarrow O_1(w_1) \mathrel{\widehat{=}}$
    PRE Q_1 **THEN**
      $[O_s \backslash V_s]V_1$
    END;
  $u_2 \leftarrow O_2(w_2) \mathrel{\widehat{=}}$
    PRE Q_2 **THEN**
      $[O_s \backslash V_s]V_2$
    END
END

IMPLEMENTATION
  $M'(P_1, P_2, P_s)$
REFINES M
CONCRETE_VARS
  X'_1, X'_2, X'_s
INVARIANT $I'_1 \wedge I'_2 \wedge I'_s$
INITIALISATION
  $U'_s; [O_s \backslash V'_s](U'_1; U'_2)$
OPERATIONS
  $u_1 \leftarrow O_1(w_1) \mathrel{\widehat{=}}$
    BEGIN $[O_s \backslash V'_s]V'_1$ **END**;
  $u_2 \leftarrow O_2(w_2) \mathrel{\widehat{=}}$
    BEGIN $[O_s \backslash V'_s]V'_2$ **END**
END

Fig. 4. Flattened Constructs

Theorem 1. *If all proof obligations of* M_s, M'_s, M_1, M'_1, M_2, *and* M'_2 *are true ([2, p 763ff], Sect. 6.2), then all proof obligations of* M *and* M′ *hold.*

Several soundness proofs of the rely/guarantee method for shared variable systems have been given in the literature for different formalisms [24, 27, 1, 12]. The proof of this theorem is very similar.

8 Summary

8.1 Related Work

The use of assumptions and commitments to achieve compositionality in program verification was first proposed by Francez and Pnueli [11]. Jones introduced rely/guarantee conditions as a method for top-down program development [15]. Ketil Stølen has added wait-conditions to handle synchronization and auxiliary variables to increase expressiveness [24]. Jones himself applied the idea to object-oriented systems [16]. Rely/guarantee specifications have also been incorporated into temporal logic-based formalisms, thereby also capturing certain liveness properties: Collete added them to UNITY [7] and Abadi and Lamport to TLA [1]. Misra and Chandy have first used assumption/commitment specifications for message passing systems [19]. A unifying overview of shared variable and message passing assumption/commitment specifications is given by [26].

Neither VDM nor Z have an equally powerful modularization mechanism as B, although some constructions have been suggested [9, 13]. RAISE, Cogito, and other related formalisms provide different forms of modularization. However, we are not aware of any compositional symmetric shared access mechanism comparable to ours.

Both Jones [15] and Stølen [24] combined rely/guarantee specifications with a VDM like logic and syntax. However, their aim was to reason about concurrent programs only and they have not investigated rely/guarantee specifications in VDM for modular sharing. Whereas existing work has mostly focused on the use of assumption/commitment specifications for concurrent system, this paper has applied them to achieve compositionality in sequential systems with shared components.

Role-based contracts for different forms of collaborations have been proposed, e.g., by Helm et al for object-oriented systems [14] and by Francez and Forman for interacting processes [10]. Role-based specifications expressing rely/guarantee conditions as part of the shared construct are believed to be new. Traditionally, a rely/guarantee pair is part of each component to be composed. Centralization of all rely/guarantee specifications is possible in our case because only a single component is shared, whereas most other approaches handle mutual sharing. Our benefit is that all proofs for an accessor can be performed without knowing the other accessors.

Pioneering work in explaining the existing B composition mechanisms and their interplay with refinement has been done by Bert, Potet, and Rouzaud [5, 21].

8.2 Conclusions

We have extended the B method with a compositional symmetric shared access mechanism that overcomes the limitations of the single-writer restriction and the limited visibility of shared variables of the existing mechanisms. Based on rely/guarantee conditions expressed as accessor roles of the shared construct, the new mechanism is compositional, providing for independent refinement without the need to know the other accessors. The abstraction of possible modifications into compact role specifications simplifies the non-interference proofs. The new mechanism provides for flexible sharing on all levels; applications with sharing requirements can be specified, refined, and implemented without loss of modularity or underspecification as has been the case with the existing mechanisms. Uniform applicability in all constructs, replicated roles, multiple contracts, and good integration with existing composition mechanism add to the flexibility of the new mechanism.

For the new mechanism, we have given formal definitions of the syntax, the proof obligations, the visibility rules, and the restrictions on the composition graph. A partial soundness proof completes the paper.

Acknowledgments. Marina Waldén and Emil Sekerinski provided detailed comments on earlier drafts. We would also like to thank Wolfgang Weck for a number of fruitful discussions on the topic. The referees' comments are gratefully acknowledged.

References

1. Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.
2. J. R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

3. R. Back and J. von Wright. Trace refinement of action systems. In *CONCUR 94*, pages 367–384. LNCS 836, Springer Verlag, 1994.
4. J. A. Bergstra, J. Heering, and P. Klint. Module algebra. *Journal of the ACM*, 37(2):335–372, 1990.
5. Didier Bert, Marie-Laure Potet, and Yann Rouzaud. A study on components and assembly primitives in B. In *Proceedings of the first B conference*, pages 47–62, 3 rue du Maréchal Joffre, BP 34103, 44041 Nantes Cedex 1, 1996. IRIN Institut de recherche en informatique de Nantes.
6. Martin Büchi. The B Bank. In Emil Sekerinski and Kaisa Sere, editors, *Program Development by Refinement: Case Studies Using the B Method*, chapter 4, pages 115–180. Springer Verlag, 1998. http://www.abo.fi/~mbuechi/publications/BBook.html.
7. Pierre Collette. Application of the composition principle to UNITY-like specifications. In *Proceedings of TAPSOFT 93*, pages 230–242. LNCS 668, Springer Verlag, 1993.
8. Willem-Paul de Roever. The quest for compositionality—a survey of assertion-based proof systems for concurrent programs, part I: Concurrency based on shared variables. In F.J. Neuhold and G. Chroust, editors, *Proceedings of the IFIP Working Conference "The role of abstract models in computer science"*, pages 181–205. North-Holland, 1985.
9. J.S. Fitzgerald and C. B. Jones. Modularizing the formal description of a database system. In *VDM'90: VDM and Z – Formal Methods in Software Development*, pages 189–210. LNCS 428, Springer Verlag, 1990.
10. N. Francez and I. Forman. *Interacting Processes: A Multiparty Approach to Coordinated Distributed Programming*. ACM Press, 1996.
11. Nissim Francez and Amir Pnueli. A proof method for cyclic programs. *Acta Informatica*, 9:133–157, 1978.
12. Peter Grønning, Thomas Qvist Nielsen, and Hans Henrik Løvengreen. Refinement and composition of transition-based rely-guarantee specifications with auxiliary variables. In *Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 332–348. LNCS 472, Springer Verlag, 1990.
13. I. J. Hayes and L. P. Wildman. Towards libraries for Z. In J. P. Bowen and J. E. Nicholls, editors, *Z User Workshop: Proceedings of the Seventh Annual Z User Meeting*, Workshops in Computing. Springer Verlag, 1993.
14. Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of OOPSLA/ECOOP '90*, pages 169–180, 1990.
15. Cliff B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North Holland, 1983.
16. Cliff B. Jones. Accomodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.
17. Leslie Lamport. The temporal logic of actions. *ACM Transactions of Programming Languages and Systems*, 16(3):872–923, 1994.
18. Kevin Lano. Integrating formal and structured methods in object-oriented system development. In S.J. Goldsack and S.J.H. Kent, editors, *Formal Methods and Object Technology*. Springer Verlag, 1996.
19. J. Misra and M. Chandy. Proofs of networks of processes. *IEEE Software Engineering*, 7(4):417–426, 1981.
20. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.
21. Marie-Laure Potet and Yann Rouzaud. Composition and refinement in the B-method. In *Proceedings of the second B conference*, pages 46–65. LNCS 1393, Springer Verlag, 1998.
22. Emil Sekerinski and Kaisa Sere, editors. *Program Develoment by Refinement: Case Studies Using the B Method*. FACIT. Springer Verlag, 1998.

23. Kaisa Sere and Marina Waldén. Data refinement of remote procedures. In *Proceedings of TACS 97*, pages 267–294. LNCS 1281, Springer Verlag, 1997.
24. Ketil Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, University of Manchester, 1990. Available as technical report UMCS-91-1-1.
25. Qiwen Xu. On compositionality in refining concurrent systems. In J. He, J. Cooke, and P. Wallis, editors, *Proceedings of the BCS FACS 7th Refinement Workshop*. Electronic Workshops in Computing, Springer Verlag, 1996.
26. Qiwen Xu, Antonio Cau, and Pierre Collette. On unifying assumption-commitment style proof rules for concurrency. In *Proceedings of CONCUR 94*, pages 267–282. LNCS 836, Springer Verlag, 1994.
27. Qiwen Xu, Willem-Paul de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

Paper III

The Greybox Approach: When Blackbox Specifications Hide too Much

Martin Büchi and Wolfgang Weck

Submitted for publication.

The Greybox Approach: When Blackbox Specifications Hide Too Much

Martin Büchi[1] and Wolfgang Weck[2]

[1] Åbo Akademi University, Turku Centre for Computer Science,
Lemminkäisenkatu 14A, FIN-20520 Turku, Martin.Buechi@abo.fi
[2] Oberon microsystems Inc., Technoparkstrasse 1, CH-8005 Zürich, weck@oberon.ch

Abstract. Development of different parts of large software systems by separate teams, replacement of individual software parts during maintenance, and marketing of independently developed software components require behavioral interface descriptions. Interoperation and reuse are impossible without sufficient description; only abstraction leaves room for alternate implementations.
Specifications that only relate the state prior to service invocation (precondition) to that after service termination (postcondition) do not sufficiently capture external calls made during operation execution. If other methods called in the specification cannot be fully specified, it is not sufficient that the implementation only performs the specified state transformation. The implementation must also make the prescribed external calls in the respective states.
We show how to specify both state change and external call sequences using simple extensions of programming languages. Furthermore, we give a formal definition of the correctness of implementations with respect to such specifications. Finally, we give two theorems stating that all functional properties of components are preserved in systems assembled thereof.

1 Introduction

Independently developed and marketed software components and individually replaceable software parts quickly gain importance. The interfaces between those have to be specified so that independent readers arrive at the same conclusions. The necessity for complete interface descriptions is especially big in the realm of independently developed software components. Hence, we expect current bad experiences with fuzzy specifications to raise mainstream acceptance of the overhead it takes to write more systematic and formalized specifications of software component interfaces. Broad acceptance, however, can only be expected, if the required extra effort, both in time and intellectual, does not outweigh the experienced —or expected— gain.

The contributions of this paper are an interface specification approach that captures both state transformations and component interactions via method calls as well as accompanying refinement rules. The latter can be used both informally in the back of the head as well as for formal proofs. Even for projects where the (expected) cost savings in testing and maintenance don't outweigh the cost of proofs, safety is not critical, and the time is tight, we believe that it's worthwhile to at least have these rules in mind when coding. They may be taken like loop invariants and termination functions, which

are rarely written down —let alone formally proved—, yet are in some programmers' minds when coding. The refinement calculus [6] gives us a solid semantic foundation.

To enhance practical applicability and acceptance, our specification language is defined as a slight extension of an imperative programming language, in the case of this paper Java. Considering that even pre- and postconditions, for which some tool support already exists [35], are rarely adopted, we put special emphasis on bringing specifications closer to the mind setting of an imperative programming language user.

Software components are binary units of possibly independent production, acquisition, and deployment that interact to form a functioning system [53]. In this paper, we take the simplifying view that each component consists of exactly one class. For brevity, we talk about 'component instances' rather than of 'instances of the single class of the component'.

1.1 Interface specifications are abstractions

It is an old observation that there is a need to present abstractions of software building blocks to make them reusable by third parties and to allow for alternate implementations. A long time ago, programming languages started to provide means for syntactic encapsulation, but even today only few can represent semantic abstractions. Almost all approaches to the latter rely on relating the system's state prior to an operation invocation to that after termination. Pre- and postconditions are the most prominent example here. In this paper, we discuss a situation in which these approaches are unsatisfactory and suggest to draw on the theory of program refinement and to use abstract programs as specification formalism.

Already modular programming introduced by Parnas in 1972 [45, 44] includes information hiding or encapsulation to separate concerns between implementing and using a module. This simplifies the analysis of complex software systems, because software using a specific module M can be described without explaining M's implementation details. As a further consequence, the implementation of M can be changed later on, as long as it still meets the same abstraction. On the syntactic level, modules are supported by several programming languages, such as Modula-2 [56], Modula-3 [41], and Ada [54]. Programmers decide which identifiers (variables, procedures) are visible (exported) to clients of their module and which are hidden and can be accessed by code within the same module only.

Encapsulation is also one of several pillars on which object-oriented programming (OOP) rests. As with modules, object implementations are decoupled from their interfaces. In addition, however, the interfaces may be changed or extended by subclasses. The separation between specification and implementation is also crucial to achieve polymorphism, another main pillar of object orientation. Only because an object's client does not depend on one specific implementation, it can work with instances of subclasses as well.

Syntax level encapsulation is provided by most OOP languages, for example, C++ [51], Eiffel [34], and Java [22]. Modula-3 [41], Oberon [57], and Component Pascal [43] combine object-orientation with modules.

As a combination of modular and object-oriented programming, component software relies on encapsulation and abstraction as well [53]; and again, syntax level ab-

straction is supported by interface description languages (IDLs), as defined for Microsoft's COM [48] and OMG's CORBA [23] standards.

Syntax level encapsulation is extremely effective when it comes to ensuring that certain internal invariants are never invalidated by client modules or classes. Not granting access to data or functionality means that all access is under local control. If things go wrong, the reason must be in the own software building block.

1.2 From syntactic to semantic abstractions

Syntax level encapsulation and abstraction is not enough. For those identifiers that are visible to client programmers we often need to describe how and under which circumstances they are to be used.

Syntactic abstraction must, therefore, be complemented with semantic abstraction. If variables can be accessed, assignments may need to be constrained with invariants. If operations can be invoked, it must be said under which circumstances they may be invoked and what they then can be expected to do. Prominent languages and formalisms for this are Eiffel's pre- and postconditions [34, 35], Larch C++ [30], and Parnas' tables [27].

All these approaches consider everything between an operation's invocation and termination as completely hidden. All available information deals with what is before and after an operation's execution. No information is given about what happens in-between. We call such specifications *blackbox specifications* (Fig. 1).

Often, blackbox specifications are sufficient, but there are cases in which they are too black and more information is needed. Therefore, some software libraries are provided with complete implementation source code for last reference. Unfortunately, source code spoils most advantages of encapsulation. We refer to source code also as *whitebox specification*. In this paper we discuss situations in which blackboxes are too dark and propose a way to lighten up blackboxes to become greyboxes, which combine the advantages of black- and whiteboxes.

Overview. Section 2 illustrates a typical component instance interaction case where blackbox specifications hide too much. Furthermore, it introduces the observer-pattern, which is used as an example throughout the paper. In Sect. 3 we show a first specification approach that indicates mandatory call-backs without giving up abstraction. In the following section we iron out the remaining problems and present the final greybox specification style. Section 5 talks about implementations. Section 6 presents the full refinement rules to assert correctness of implementations with respect to specifications.

Fig. 1. Blackbox Specification

Aiming for practical applicability, we show how to prove correctness of frequent special cases of greybox refinement in Sect. 7. Readers who are content with an informal, intuitive explanation of greybox refinement may skip Sects. 6.5 and 7. In Sect. 8, we show what greybox refinement of components implies for systems assembled thereof. Section 9 summarizes the required additions to imperative programming languages for the purpose of greybox specification. Finally, Sect. 10 discusses related work and Sect. 11 draws the conclusions.

2 The Problem: When Black is Too Dark

Above, we defined *blackbox specifications* as descriptions that only relate the states before and after an operation (Fig. 1). It is impossible to draw conclusions about what happens between these two observation points unless there is some trace left in the observable state.

Such blackbox specifications are insufficient when it comes to call-backs. Call-backs activate functionality external to the specified component instance (Fig. 2). Typically such functionality is installed by the calling client or even third party software, as it is the case with the observer example detailed later in this section.

That blackboxes do not provide enough information to deal with call-backs has been stated and discussed in [53]. In the following we shall briefly recapitulate that discussion.

2.1 Call-backs

A call-back mechanism allows clients of a library to register operations for activation under certain circumstances. Figuratively, the client instructs the library *to call it back* upon the occurrence of a certain event (Fig. 2).

Call-backs are used to make systems extensible. In layered system architectures, they occur as calls from lower into higher layers in which case they are known as up-calls [14]. Up-calls allow the programmers of higher layers to modify the behavior of the lower layers of which they are clients.

In a similar way, methods implemented in a subclass but called in the respective superclass can be interpreted as call-backs from reusable into reusing software components.

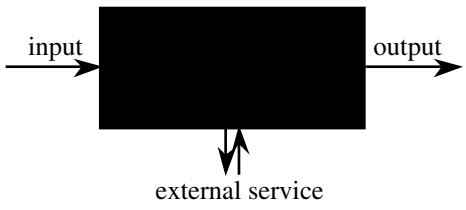


Fig. 2. Blackbox Specification with External Call

A representative application of call-backs is the observer design pattern [21]. It allows software components that need to react to certain events, such as particular state changes, to register an *observer* object with the *observed* object. The observed object then calls a notify method of each registered observer object upon the occurrence of the respective events.

A prominent application of the observer pattern is the Model-View-Controller architecture (MVC), developed originally for Smalltalk [29, 21]. The MVC architecture provides a separation of concerns between internal representation and manipulation of data (model), data presentation to the user (view), and command interpretation (controller). Because of that separation, the way of presenting the data to users can be changed by replacing the view component while keeping —or reusing— the model component. Most implementations of the MVC architecture allow more than one view to present the same model at a time. These views may even display the data differently, for instance, as a spreadsheet and as a pie chart. In this paper, we use a simplified version of the MVC pattern without a separate controller.

Instance of the model component must work with zero, one, or more simultaneous views and must not depend on the actual form of presentation by the views. Hence, the observer pattern is used. The model is the observer. The views are registered as observers and notified on changes to the model.

2.2 Example: part of a text system

Szyperski [53] presents a simple text system as an example of the above. In this paper we shall only look at those functions needed to delete characters. As the form of presentation we use Java syntax together with pre- and postconditions (Fig. 3).

All our specifications are model based. That is, we add a model state, such as registeredObservers and text, to express the specifications. We use the modifier **private** to mark model fields and methods, i.e., members that are only present for specification purposes and are not accessible to clients and classes implementing the interface.

We give the specification additions directly rather than as comments with a special start symbol [31]. A pre-parser can easily remove the additions so that the stripped version can still be processed with a normal Java compiler.

We use the abstract data types **setof** and **seqof** for sets and sequences of objects. The empty set is denoted by '{}' and the empty sequence by '<>'. The length of a sequence is given by len, the ith element of sequence s can be accessed as s[i] with indices ranging from 0 to len(s)-1. In accordance with Java, we use '==' for equality and '=' for assignment. The keyword **all** denotes universal quantification. We use the keyword '@pre' in postconditions to refer to the value of a variable at the start of the operation. The keyword **result** denotes the result value. In discussions, we use the notation C.m to refer to the specification/implementation of the instance method m in interface/class C.

Instances of classes implementing ITextModel are models. Changes, such as deleting a character, can be initiated by clients calling the respective methods, e.g. deleteCharAt. Views need to call the register method once to subscribe to notifications about text changes.

```
interface ITextModel {
  private setof ITextObserver registeredObservers={};
  private seqof char text=<>;

  int length();
    pre true
    post result==len(text)

  char charAt(int pos);
    pre 0<=pos && pos<len(text)
    post result==text[pos]

  void deleteCharAt(int pos);
    pre 0<=pos && pos<len(text)
    post (all int i: 0<=i && i<pos: text[i]==text@pre[i]) &&
           (all int i: pos<=i && i<len(text): text[i]==text@pre[i+1]) &&
           len(text)==len(text@pre)-1

  ...

  void register(ITextObserver obs); // specification omitted
  void unregister(ITextObserver obs); // specification omitted
}
```

Fig. 3. Pre-/Postcondition Specification of Interface ITextModel

The above specification of ITextModel.deleteCharAt does not say that and when observers are notified. These notification calls cannot be expressed with pre- and postconditions because they do not change the the state of the text model. We could only add a textual comment like 'call deleteNotification of all registered observers after deleting character'. Additionally, we could put the corresponding comment to ITextObserver.deleteNotification (Fig. 4). However, none of these plain English comments are machine checkable and enforceable.

```
interface ITextObserver {
  void deleteNotification(int pos);
    // pre character that was at pos has just been deleted
    // (informal comment, not tool checkable)
}
```

Fig. 4. Specification of Interface ITextObserver

2.3 Analysis of the example

We focus our analysis on the specification of the notification of the observers. How can we specify in ITextModel.deleteCharAt that the model must notify all registered views exactly once in an arbitrary order after making the modifications.

Blackbox specifications are about state changes only The problem is rooted in the fact that we are specifying an interface of an open, extensible system. While we know and can prescribe many aspects of the model's state, we want to retain full flexibility for the observer. It is the very idea of this pattern that one does not have to define what observers may do upon a notification. This openness presents a dimension of extension.

When we specify ITextModel, we don't know how the different views, registered as observers, will react to delete notifications. We don't even know the state spaces of the different views. With postconditions we cannot express calls as such, but only the effect of calls on the state. Since we don't know the effects of deleteNotification on the different views and because the calls do not modify the model's state, the notification calls cannot be captured with postconditions. Blackbox specifications are about state changes only. They cannot describe the notification calls.

The next five paragraphs contain a rather technical refutation of encoding attempts in blackbox specifications. They may be safely skipped by readers interested only in how the greybox approach works, but not in the details of why blackbox specifications don't.

Theoretically, we could encode the calls made by an operation into history/trace variables. For example, each observer could be equipped with a counter to be implicitly incremented during each notification call. With this we could add a conjunct to the postcondition of ITextModel.deleteCharAt expressing that the counters of all registered observers have been incremented by 1. However, with only these counter we could still not specify in which order, in which states, and with which parameter values the calls are to be made.

It is possible to encode all this information with traces. However, such an encoding is very complicated and unusable for practical specifications. Furthermore, it does not give the desired results in combination with abstract data type refinement based on observational substitutability: Without inquiry operations for all trace variables, the latter can just be 'forgotten'. With inquiry operations, their values could be computed differently. Only a mandatory 1-to-1 data refinement relation on trace variables and syntactic rules to forbid explicit assignment to these variables could, theoretically, rest the case.

Another seeming workaround is to keep a copy of ITextModel.text in ITextObserver and strengthen the postcondition of ITextModel.deleteCharAt to require the copy to be synchronized. The same synchronization condition is added to the postcondition of ITextObserver.deleteNotification. The idea is to force the implementation of deleteCharAt to call deleteNotification after changing the local state and to have the implementation of deleteNotification synchronize the states by calls to ITextModel.charAt. Like the trace variables above, the state copy in ITextObserver would have to be directly or indirectly observable to force it being preserved in standard data refinements.

In addition to its practical clumsiness, this approach fails to capture certain aspects: First, the values of the actual parameters of ITextObserver.deleteNotification are not

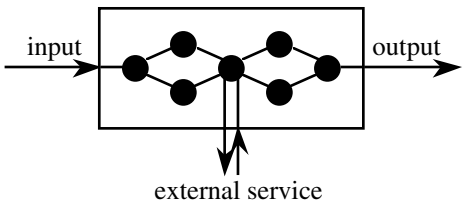


Fig. 5. Whitebox Specification

recorded. Second, sending multiple delete notifications or sending an insert notification —assuming that the specification of ITextObserver.insertNotification has the same postcondition as deleteNotification— instead would be correct refinements. Third, this approach wouldn't allow us to specify that in the method deleteCharsBetween(int from, int to), which deletes all characters between positions from and to, multiple delete notifications must be sent to the same observer. To address these issues, this approach would have to be augmented by a trace encoding as described above, rendering the state copy superfluous.

In conclusion, blackbox specification of call-backs through trace encoding and/or a copy of the other component instances' states is not practical.

Whitebox specifications would work but aren't abstract enough Whitebox specifications, that is source code, show exactly when and in which order the observers are notified (Fig. 5). However, they ruin abstraction by fixing too many details. For example, we might want to leave the notification order open so it can be changed in future versions.

Informal specifications are not good enough Pure informal specifications might be clear enough in very simple cases, but they also have their share of disadvantages. They cannot be used as input to any tool, such as automatic test case generation [13], automatic pre- and postcondition checking [35, 18], or formal theorem provers [1].

Informal sentences are subject to interpretation, which in turn depends on the particular context of the reader. Often, these contexts vary, resulting in mismatches of interpretations by independent vendors and eventually leading to incompatible software components. For example, it is likely that an informal specification would not unambiguously answer the following questions: In what order are the observers notified? Is it the same order every time? Can the observers register additional observers upon being notified of a state change? Are these newly registered observers notified in the current round?

This problem is aggravated by the fact that component interfaces need to abstract and for this often intentionally leave certain aspects undefined. However, with informal specifications it is often not clear what is intentionally left unspecified.

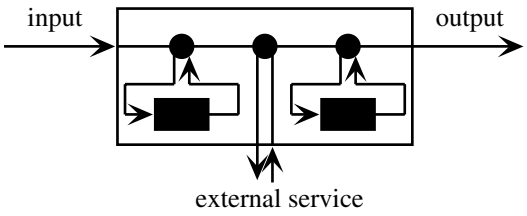


Fig. 6. White-on-Black Layered Specification

3 A Pragmatic Approach: Layering White on Black

In this section we discuss a simple and pragmatic way to specify the circumstances under which call-backs are to be made. The idea is to decompose an operation with call-backs into a set of private operations and a public operation calling the former. None of these private operations contain any call-backs. Hence, they can be specified as complete blackboxes. On top of that layer of blackbox specification we put a single whitebox specification of the original operation (Fig. 6). The latter contains only calls to the blackboxes of the lower layer, the call-backs, and eventual loops and conditionals in which call-backs occur. In our ITextObserver example, we decompose the deleteCharAt method into just one blackbox operation changing the text data and a whitebox operation. The latter calls the blackbox operation and notifies the observers (Fig. 7).

```
interface ITextModel {
  private ITextObserver[] registeredObservers;
  private int nofObservers=0;
  private seqof char text=<>;

  private void removeCharacter(int pos);
    //Blackbox specification:
    post (all int i: 0<=i && i<pos: text[i]==text@pre[i] &&
            (all int i: pos<=i && i<len(text): text[i]==text@pre[i+1]) &&
            len(text)==len(text@pre)-1

  void deleteCharAt(int pos) pre 0<=pos && pos<len(text) { // Whitebox specification:
    removeCharacter(pos);
    for(int i=0; i<nofObservers; i++){
      registeredObservers[i].deleteNotification(pos);
    }

  ...
}
```

Fig. 7. Layered Specification of Interface ITextModel

```
interface LayeredSpecPattern {
  private void partOne(...);
    // Blackbox specification:
    // pre Pre1
    // post Post1

  private void partTwo(...);
    // Blackbox specification:
    // pre Pre2
    // post Post2
```

```
  void publicService(...) {
    // Whitebox specification:
    partOne(...);
    callBack(...);
    partTwo(...);
  }
}
```

Fig. 8. Pattern for Layered Specifications

The general idea used in the example can be summarized as follows. To make clear, under which circumstances the observer is notified, we need to give a whitebox specification of all services that contain such calls; in our case this is the deleteCharAt method. Specifying the entire service as a whitebox, however, would be too detailed. Hence, wherever we want to give an abstraction instead of an actual implementation, we include a call to another private service, which we specify as a blackbox. Without loss of generality we assume a single call-back only in the general specification pattern illustrating the above idea (Fig. 8).

The specification could also be given as a partly abstract class instead of an interface. The public method could be implemented and the blackbox methods would be left abstract. A subclass would then simply have to implement the two blackbox methods. Figure 9 givew the UML class diagram of this approach, which corresponds to the template method pattern [21].

The main advantages of this layered approach is that it can readily be deployed with formalisms such as Eiffel [34] and JML [31] that allow both black- and whitebox specifications. Although external calls can be specified, neither of these approaches comes with refinement rules that preserve the external call sequence.

This layered approach also has some disadvantages. We need to make every data structure, such as registeredObservers, used in the whitebox part concrete. Leaving certain aspects, such as the notification order, unspecified may be difficult. Also, the need to introduce a blackbox operation for every block makes specifications less readable.

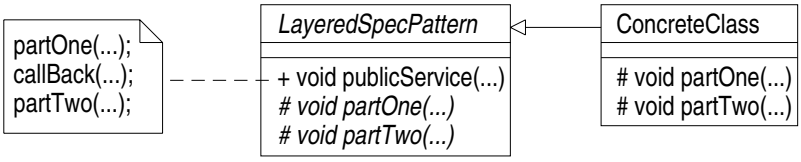


Fig. 9. Template Method Pattern for White-on-Black Layering

4 Abstract Programs: Shades of Grey

Specification statements [3, 6] can be used instead of auxiliary blackbox methods. A specification statement is of the form **any**(T y: P) {S} where T is a type, y a (bound) variable, P a predicate, and S a statement. Upon execution, an arbitrary value is chosen for y such that P holds and then S is performed. For example, **any**(float y: y>=0 && 0.98*y*y<x && x<1.02*y*y) {s=y;} assigns the square root of x, computed with a precision of 2 % to y and subsequently to s. If, for example, the value of x is 16, then that of s will be between 3.96 and 4.04 after the statement has been executed.

If there are no values for the bound variable y such that the predicate P holds, then the **any** statement aborts. Specification statements permit us to write specifications that are as high level as their pre-/postcondition counterparts.

In Fig. 7, we can replace the call to removeCharacter in deleteCharAt by

```
any(seqof char txt: (all int i: 1<=i && i<pos: txt[i]==text[i]) &&
    (all int i: pos<i && i<=len(text): txt[i]==text[i]) && len(txt)==len(text)-1)
        {text=txt;}
```

Whereas the above square root example was truly nondeterministic, the text example isn't because there is exactly one possible outcome for every initial value of text and pos satisfying the precondition. Hence, provided that concatenation of strings is defined, the specification (Fig. 10) can in this case be written as:

```
text=text[0..pos-1]+text[pos+1..len(text)-1]
```

As another specification construct, we add a **do** loop over sets. The body of a **do** loop is executed exactly once with the iterator bound to each element of the initial value of the set. We use a **do** loop to express that all observers must be specified in an arbitrary order exactly once in deleteCharAt (Fig. 10).

Our focus has been on the notification calls, which we want to specify so that they must be made in an implementation. However, a specification may also make optional calls. Say, we use a call to a square root function of a math component. An implementation should be free to either make the same call or compute the result itself. To distinguish between mandatory and optional calls, we use the following approach. *Inquiry methods* are declared with the modifier **inquiry** in the specification. All other methods are called *modification methods* —whether they actually modify the state or not. Inquiry methods may not modify the state or contain any calls to modification methods. In our example, length and charAt are inquiry methods. Calls to modification methods in specifications are referred to as *mandatory calls*; calls to inquiry methods as *optional calls*. Implementations must make the same mandatory calls as their specifications and may not make any additional calls to modification methods of the mentioned component instances. Additionally, they can make arbitrary calls to inquiry operations.

As a final element, we allow constructors in specifications. This is necessary for cases where the initialization must contain external method calls. For example, we might want that every view is always registered as an observer with a model. In this case, we need to specify a constructor for ITextObserver that makes a registration call.

```
interface ITextModel {
  private setof ITextObserver registeredObservers={};
  private seqof char text=<>;
  public int maxObservers=10;

  invariant // denoted by I in the text
    card(registeredObservers)<=maxObservers

  inquiry int length() {
    return len(text);
  }

  inquiry char charAt(int pos) pre 0<=pos && pos<len(text) {
    return text[pos];
  }

  void deleteCharAt(int pos) pre 0<=pos && pos<len(text) {
    text=text[0..pos-1]+text[pos+1..len(text)-1];
    do(ITextObserver o in registeredObservers) {
      o.deleteNotification(pos);
    }
  }

  ...

  void register(ITextObserver obs) pre obs!=null && not(obs in registeredObservers)
        && card(registeredObservers)<maxObservers {
    registeredObservers=registeredObservers+{obs};
  }
}
```

Fig. 10. Greybox Specification of Interface ITextModel

Figure 11 gives the general pattern for greybox specifications with the same semantic meaning as Fig. 8. Post1′ and Post2′ correspond to Post1 and Post2 with u_i@pre replaced by u_i and u_i by w_i. We have here generalized the **any** statement for multiple

```
interface GreyboxSpecPattern {
  void publicService(...) {
    any(T1 w1, ..., Tm wm: Post1′) {u1=w1; ...; um=wm;};
    callBack(...);
    any(T1 w1, ..., Tn wn: Post1′) {u1=w1; ...; un=wn;};
  }
}
```

Fig. 11. Pattern for Greybox Specifications

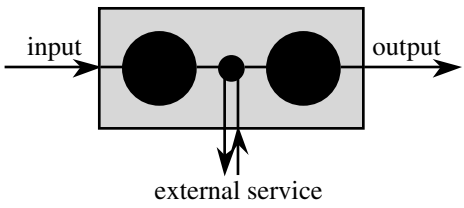


Fig. 12. Greybox Specification

variables. That is, an arbitrary tuple of values will be chosen for $w_1, \ldots, w_m$ such that the predicate holds. Figure 12 illustrates the greybox specification approach.

4.1 Consistency of specifications

Specifications that may abort or invalidate their own invariant when executed are of questionable use. Therefore, we require the following five consistency conditions to hold for greybox specifications: If an operation is started in a state and with actual parameters that satisfy both the invariant and the precondition, then

1. The method may not abort, i.e., try to access a **null** reference (For simplicity, we consider throwing an exception as abortion.).
2. The invariant is guaranteed to hold after termination.
3. All external calls are made with parameters that satisfy the respective precondition.
4. The own invariant holds whenever an external call is made. This simplifies reentrance, as explained in Sect. 6.1.
5. An operation either terminates or makes infinitely many external calls to modification operations. The second option allows us to specify nonterminating methods such as the scheduler of an operating system.

We follow the B method [1] by requiring consistency proofs to show that operations preserve the invariant. This contrasts implicit operation specifications in VDM [28] and Z [50], where the invariant is implicitly conjoined to every postcondition. We consider these explicit consistency proofs to be a benefit, rather than a burden. Intuitive and informal specifications capture what an operation should do, regardless whether it might thereby invalidate the invariant. Statement specifications let us express this directly. The consistency proof shows whether this behavior is actually legal. Failure to prove consistency may also point to an overly strong invariant. Thus redundancy helps us find specification errors. On the other hand, the implicit conjunction of the invariant to the postcondition in VDM and Z can lead to unintentionally restrictive. Without the need for a consistency proof, however, this may not be noticed until much later, when changing the specification may invalidate work already based on it.

Constructors have to satisfy the above five consistency requirements when called with parameters satisfying the precondition.

```
class CTextModel implements ITextModel {
  private ITextObserver[] regObs=new ITextObserver[maxObservers];
  private int nofObs=0;
  private StringBuffer t=new StringBuffer();
  invariant // denoted by I′ in the text
    0<=nofObs && nofObs<=maxObservers &&
    ITextModel.registeredObservers==regObs[0..nofObs-1] &&
    (String)ITextModel.text==(String)t

  public void deleteCharAt(int pos) {
    ... // remove character from t
    for(int i=0; i<nofObs; i++) {
      regObs[i].deleteNotification(pos);
    }
  }

  public void register(ITextObserver obs) {
    regObs[nofObs]=obs;
    nofObs++;
  }
  ...
}
```

Fig. 13. Implementation CTextModel

5 Implementing Greybox Specifications

Implementations are coded as normal Java classes (Fig. 13). The only addition is the invariant. The latter consists of two —possibly intermingled— parts, the local and the gluing invariant (abstraction relation). The local invariant restricts the 'legal' values of the local variables. Its aims are the documentation of desired properties and the simplification of proofs. The gluing invariant links the values of the concrete variables in the implementation to the abstract variables of the specification. For example, ITextModel contains a set of observers. This set is implemented with an array (Fig. 13). Simulation-based correctness proofs of an implementation with respect to its specification (Sects. 6 and 7) require us to state how the concrete and the abstract variables, e.g., the values of the set and the array, are related. The invariant conjunct ITextModel.registeredObservers==regObs[0..nofObs-1], where regObs[0..nofObs-1] is the set {regObs[0]} ∪ {regObs[1]} ∪ ... {regObs[nofObs-1]}, states this.

6 Refinement of Greybox Specifications

A specification can be viewed as a contract between a client and a provider who implements it. To refine a specification means to improve it from the client's point of view. That is, the provider can deliver more, but not less. Intuitively, refinement is defined by

observational substitutability: The user must not be able to observe any new behavior of specified attributes if a component is replaced by a refinement thereof in any system.

For practical purposes, a proof rule involving a quantification over all possible systems, in which the component might be placed, is not useable. Hence, we develop simulation-based rules instead and account not only for the behavior observable by users, but also for that only observable by client software components.

A client can observe three functional aspects of another component instance A: The return values from method calls to A, the public variables of A (if any), and calls made by A to modification methods of other component instances. The latter are observable because they may change the states of the called component instances. These states are, in turn, observable by the first two means.

The basic conditions of greybox refinement are simple: The implementation of every method must make the same sequence of mandatory calls to other component instances in the same respective states as the specification, perform the same overall changes to the local state, and return the same value. If the specification is nondeterministic, then the call sequence, local state transformation, and return value of the implementation must correspond to one choice in the specification. The refinement calculus [3, 6, 40] provides a formal base, but not yet any rules, to prove this conformance.

In this section we first clear the field by discussing a number of critical issues surrounding greybox refinement and then formalize the latter. We end the section with two theorems on the refinement of systems with multiple components.

6.1 Reentrance

Reentrance occurs if method m of an instance of class A calls method n of an instance of class B and n calls —directly or indirectly— method o of A (Fig. 14) [53, 36]. For example, an observer could in its implementation of deleteNotification call ITextModel.charAt to enquire the current value of text.

Both for the consistency of specifications and for the refinement rules (Sect. 6.5) we assume the invariant of an object to hold whenever one of its methods is called. Hence, we need to establish the invariant before making calls to other component instances, so that the own invariant holds upon reentrant calls.

We follow [15] in banning reentrant calls to modification methods. Although such calls can be handled in theory, they make components very difficult to understand. Assume, for example, that in Fig. 14 method o would be a modification method. Then the

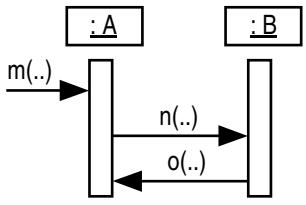


Fig. 14. Reentrance Scenario

effect of method m on the local state of an instance of A would not be described by m alone. Instead all called external modification methods would have to be examined for possible reentrant calls to modification methods.

Mutual recursion between inquiry methods of two instances of A and B could happen if A.m calls B.n and B.n calls A.m. Mutual recursion between inquiry methods is only problematic, if it is infinite. Because the calls to inquiry methods are not visible from specifications, such mutual recursions cannot be detected by modular reasoning. Possible infinite mutual recursions can, however, be detected at the time when components are combined or at run-time with special tests.

The impossibility of detecting all possible infinite mutual recursions at compile time is a consequence of the axioms of component software, namely late composition of components from mutually unaware vendors. Any compile-time solution has to give up the full flexibility of the components software model. One solution to the mutual recursion problem is to impose a partial order on inquiry methods and to allow only calls to 'smaller' methods. Considering that the order must be over components from mutually unaware vendors, this leads to serious restrictions.

6.2 Unmentioned component instances

Often implementations use additional component instances not mentioned in the specification. For example, an observer may use a window instance, in which it paints the text and a model may use a set from a collection framework to manage the references to registered observers. This use of additional component instances is necessary, but comprises a variety of problems.

We call component instances referred to in the specification as *mentioned component instances* and all others as *unmentioned component instances*. For example, from the perspective of ITextModel and of classes implementing only the former, any instance of a component CSet is classified as unmentioned, assuming that CSet doesn't implement ITextObserver. Furthermore, any instance of ITextObserver not registered as an observer is labeled unmentioned.

As motivated above, allowing calls to modification methods of unmentioned component instances is absolutely necessary. However, this may lead to reentrant calls of modification methods. Consider the following scenario: Method CTextModel.deleteCharAt calls a modification method event of a (unmentioned) instance of a CLog component, the specification of which does not mention any other components. Since, event is a modification method, it may itself call other modification methods, including modification methods of our CTextModel instance and the latter's registered observers. In both cases, refinement of ITextModel by CTextModel may be violated. This problem is similar to the one of mutual recursion above. A static solution has to give up part of the flexibility of component software.

6.3 Self calls

Self calls, that is calls to methods of the same component instance, deserve special treatment. The idea of greybox specifications is to prescribe what calls have to be made between different component instances in addition to the changes of the instance state.

If a specification contains self calls, it is simply for the purpose of factoring out parts that —ignoring recursion— could be textually substituted for the call. Hence, self calls are never mandatory. An implementation of a specification method containing self calls must simply make the same external calls and local state modification as the specification method including the own called methods. It is not required, but often beneficial, to establish the invariant before making self calls.

6.4 Additional methods and constructors

Classes may have more public methods and constructors than one of their implemented interfaces. For example, CTextModel could also have a method deleteCharsBetween(int from, int to), which deletes all characters between positions from and to. Every possible execution of an additional method must constitute a refinement of the external call sequence and state modifications performed by a finite sequence of interface methods with possible local computation of parameters in-between. In this case, other component instances cannot tell the difference between the single call to the new method and the sequence of calls (mumbling invariance) [9]. Additional inquiry methods can be added freely (stuttering invariance), as covered by the above definition with the empty sequence.

A class may implement multiple interfaces. In this case, the methods prescribed by other interfaces have to be considered as additional methods. For example, if CTextModel were also to implement an interface ILog, all implementations of methods prescribed by ILog would have to satisfy the above criteria with respect to ITextModel and vice versa.

Especially if a class implements several interfaces, the above requirement for additional methods sometimes turns out to be too strict. To overcome this problem, we allow interfaces to contain a special method others. This method is not to be implemented by classes, it simply shows what other modifications —with accompanying external calls— are allowed in additional methods.

The rules for additional constructors are anologous. Every possible execution of an additional constructors must constitute a refinement of the external call sequence and state modifications performed by an interface constructor followed by a finite sequence of interface methods with possible local computation of parameters in-between.

6.5 Formalizing greybox refinement

In this subsection, we give the general formalization of greybox refinement using a trace semantics, partly inspired by the trace semantics for action systems [5]. We formalize the conformance of call sequences and state changes. Predicate transformer semantics for basic programming language constructs can be found in the literature [6, 12].

First we review some fundamentals of predicates, relations, and product and sum types.

Predicates and relations. Predicates (boolean expressions) on a type Γ are functions from elements of type Γ to boolean. For example, for variable x of type int, x>0 is a predicate on int. A predicate determines a subset, e.g. the positive integers. Thus we

use $\subseteq$ as order between predicates, e.g. x>5 $\subseteq$ x>0. Predicates being functions, they can be applied to values, e.g. (x>5)(6) is true.[1]

Relations of type $\Sigma \leftrightarrow \Gamma$ are functions of type $\Sigma \rightarrow \Gamma \rightarrow$ boolean. An invariant I′ of an implementation with state space Γ refining a specification with state space Σ can also be considered as a relation of type $\Sigma \leftrightarrow \Gamma$. We can apply such a relation to a state σ of the specification and a state γ of the implementation: I′(σ, γ). For relation R and predicate p the relational image im(R, p) is defined as {y | (**exists** x: : R(x,y) && p(x))}. We use && for conjunction of predicates and relations.

Product and sum types. The product type int $\times$ char denotes the type of tuples of integers and characters. The tuple (5, 'a') is an element of it. We use the projection functions fst and snd for tuples, e.g. fst((5, 'a')) is 5 and snd((5, 'a')) is 'a'.

The sum type int $\oplus$ char denotes a disjoint union of an integer and a character (corresponding to a variant record in Pascal or a union with a type tag in C). A variable of this type has either an integer or a character value. We leave the injection and projection functions for sum types implicit.

Definition of greybox refinement. With these notions we can define greybox refinement. Without loss of generality, we only allow constants and variables as actual parameters.

When we animate (execute) method m of the sample interface ITest, it generates a behavior. A behavior is a sequence of states, where the successor state is computed by executing the next atomic statement.[2] Let Σ be the type of the state space of the specification, for example **setof** ITextObserver $\times$ **seqof** char for ITextModel. When executing ITextModel.deleteCharAt(2) the following is a possible behavior: <({o1, o2}, [texZt]), ({o1, o2}, [text]), ({o1, o2}, [text]), ({o1, o2}, [text]), ({o1, o2}, [text]), ({o1, o2}, [text])>.

As defined so far, we do not record which modification methods of which objects are called with which parameters. Let Ω_i for i **in** 1..e denote the types of references to mentioned component instances in our specification ITest. Furthermore, let $\Delta_{i,j}$ for j **in** 1..f_i denote the parameter types of the modification methods of component type i. Finally, let Ω_0 denote the type Unit with the single element unit and let f_0==0. By extending every state in a sequence by an element of type

$$\bigoplus_{i=0}^{e} \left(\Omega_i \times \bigoplus_{j=1}^{f_i} \Delta_{i,j} \right)$$

we can indicate for every state, whether in this state a mandatory call is made and if so to which method of which object and with which parameters. That is, an element of the above type is a tuple of a reference to a component instance and a parameter value for one of the modification methods of the referenced component instance. Different methods with identical parameter types are correctly distinguished by the sum type.

[1] We omit the explicit λ if the binding is clear from the names or the position as in the example. I.e., x gets bound to 6.

[2] External method calls are split into two atomic statements as explained below. Otherwise, the grain of atomicity is not important.

A *simple behavior* is of type:

$$\textbf{seqof}\left(\Sigma \times \left[\bigoplus_{i=0}^{e}\left(\Omega_i \times \bigoplus_{j=1}^{f_i}\Delta_{i,j}\right)\right]\right)$$

A mandatory method call like w=e.m(c) generates two states σ_i and σ_{i+1}, such that σ_i==(fst(σ_{i-1}), (e,c)) and σ_{i+1}==(x, (unit, unit)), where x stands for fst(σ_{i-1}) with the value of w replaced by the return value of e.m(c). A state is a *call state* if the second component is not (unit, unit).

For every method ITest.m we generate the set sbeh(ITest.m) of all simple behaviors where the first state and the value of the parameter satisfies the invariant and the precondition.

Let Π be the type of the input parameter of the method in question, for example int for ITextModel.charAt, and Φ the result type, that is char for the aforementioned method. A *full behavior* additionally contains the values of the parameters and the result. A full behavior is of type:

$$\textbf{seqof}\left(\Sigma \times \left[\bigoplus_{i=0}^{e}\left(\Omega_i \times \bigoplus_{j=1}^{f_i}\Delta_{i,j}\right)\right]\right) \times (\Pi \times \Phi)$$

The set of full behaviors of method ITest.m is denoted by beh(ITest.m). For full behavior b, the corresponding simple behavior is fst(b).

We distinguish three kinds of behaviors: (normally) terminating, aborting, and infinite behaviors. A terminating behavior is generated if the method returns control to its caller via a return statement or if its result type is void also by executing the last statement. An aborting behavior is generated if the method aborts, e.g., tries to access an array at an index outside its boundaries. Since aborting behaviors are undesirable, we actually require both specifications and implementations to be free of them. An infinite behavior is generated if the operation neither terminates nor aborts. The terminating (beh_+(ITest.m)), aborting ($\text{beh}_\perp$(ITest.m)), and infinite behaviors (beh_∞(ITest.m)) form a partitioning of all full behaviors. The same holds for their simple counterparts.

Traces are the observable parts of behaviors. They are generated by removing non-observable states, that is states where no external calls are made, from behaviors. We get the set of full traces tr(ITest.m) (respectively simple traces str(ITest.m)) by performing the following two operations on each behavior b in beh(ITest.m) (resp. sbeh(ITest.m)):

1. If b **in** beh_∞(ITest.m) ends in an infinite sequence of non-call states, then we remove b from beh_∞(ITest.m) and add it with the infinite sequence of non-call states removed to $\text{beh}_\perp$(ITest.m).
2. Remove all non-call states, except for the first state and if b is finite the last state, from b.

Simple/full traces are of the same type as simple/full behaviors. The trace set is the disjoint union of the terminated, aborted, and infinite traces.

Likewise we generate the set of traces of the implementation CTest.m under question. We assume that the state space of CTest is Γ. The initial states are given by the

ITest	interface
CTest	class
Σ	type of state space of ITest
Γ	type of state space of CTest
I'	invariant of CTest
Ω_i	types of references to mentioned components
$\Delta_{i,j}$	parameter types of modification methods of Ω_i
$\oplus$	sum type
$\times$	product type
fst, snd	first and second projection of tuple
m	prescribed method
n	additional method
K	prescribed constructor
L	additional constructor
sbeh, beh	set of simple/full behaviors
str, tr	set of simple/full traces
b	behavior
s, s′	simple traces
t, t′	full traces
$\frown$	concatenation of simple traces
str(ITest)*	transitive closure of all simple traces of ITest
str($\widehat{\text{ITest}}$)	union of simple traces of constructors
im(I', I)	relational image of I under I'

Fig. 15. Summary of Notation and Conventions

predicate im(I', I && p), where I' is the invariant of CTest, I the invariant of ITest and p the precondition of ITest.m. When computing traces for implementations, we only consider mentioned component instances and their types. Thus the types of the traces of ITest.m and CTest.m$_{\text{ITest}}$ only differ in the first component of the sequences, which is Σ, respectively Γ. The subscript for CTest.m is necessary because CTest may implement multiple interfaces (Sect. 6.4).

We say that two simple traces s and s′ correspond at position i (writtn s $\propto^i_{I'}$ s′), if the state parts of the ith sequence elements are related by the invariant of the implementation I', taken as a relation, and the external call parts are identical:

$$\text{s} \propto^i_{I'} \text{s}' \quad \stackrel{\text{def}}{=} \quad I'(\text{fst(s[i]), fst(s'[i])) \&\& snd(s'[i])==snd(s[i])}$$

We define *trace approximation* under the refinement relation I' as follows: The simple trace s approximates s′ under I' (written s $\preceq_{I'}$ s′) if one of the following conditions holds:

- s and s′ are terminating, len(s)==len(s′), and (**all** i: 0<=i && i<len(s): s $\propto^i_{I'}$ s′).
- s and s′ are infinite and (**all** i: 0<=i: s $\propto^i_{I'}$ s′).

Greybox refinement for method m holds if for every trace of CTest.m$_{\text{ITest}}$ there is a corresponding trace of ITest.m:

$$\mathsf{ITest.m} \preceq_{I'} \mathsf{CTest.m}_{\mathsf{ITest}} \stackrel{\text{def}}{=} (\textbf{all } \mathsf{t}' \text{: } \mathsf{t}' \textbf{ in } \mathsf{tr}(\mathsf{CTest.m}_{\mathsf{ITest}}) \text{: } \textbf{exists } \mathsf{t} \text{: } \mathsf{t} \textbf{ in } \mathsf{tr}(\mathsf{ITest.m}) \text{:}$$
$$\mathsf{fst(t)} \preceq_{I'} \mathsf{fst(t')} \;\&\&\; \mathsf{snd(t)}{=}{=}\mathsf{snd(t')})$$

To summarize: This condition requires that CTest.m preserves the invariant, refines its specification (call sequence, state transformation, return value), and establishes the invariant before each external call.

Additional methods. If CTest contains an additional public method n beyond those prescribed by ITest, there must for every simple trace of CTest.n be a finite sequence of concatenateable traces of methods of ITest that approximates the former. For the starting states we distinguish two cases: If n is not prescribed by any interface implemented by CTest and it has the precondition p in the implementation, then the starting states are im(I′, I) && p. If n is prescribed by another interface ITest2 with invariant I_2 and state space Σ_2, then I′ is actually a function of type $\Sigma \to \Sigma_2 \to \Gamma \to$ boolean. The starting states are $\{\gamma \mid$ **exists** σ, σ_2: I′(σ, σ_2, γ) && I(σ) && $I_2(\sigma_2)$ && p(σ_2)$\}$.

We define concatenation ($\frown$) of simple traces as follows: Two simple traces u and v can be concatenated if u is terminating and if fst(u[len(u)-1])==fst(v[0]). In this case we first add the simple traces and then remove the two intermediary non-call states. The concatenated trace belongs to the same kind (terminating, aborted, infinite) as v. We define str(ITest) to be the union of all simple traces of its modification methods — including others (Sect. 6.4)—, and str(ITest)* the transitive closure thereof with respect to concatenation (including the empty trace).

With these definitions we can formally express the refinement condition for additional methods:

$$\mathsf{ITest}^* \preceq_{I'} \mathsf{CTest.n}_{\mathsf{ITest}} \stackrel{\text{def}}{=} (\textbf{all } \mathsf{s}' \text{: } \mathsf{s}' \textbf{ in } \mathsf{str}(\mathsf{CTest.n}_{\mathsf{ITest}}) \text{:}$$
$$\textbf{exists } \mathsf{s} \text{: } \mathsf{s} \textbf{ in } \mathsf{str}(\mathsf{ITest})^* \text{: } \mathsf{s} \preceq_{I'} \mathsf{s}')$$

Note that for additional methods there is no equivalence criteria for parameters or return values.

Constructors. The conditions for constructors are analogous to those for methods. However, the initial states are arbitrary. For simplicity, we consider the initialization in the field declaration as part of every constructor. The condition for prescribed constructor K[3] is as follows:

$$\mathsf{ITest.K} \preceq_{I'} \mathsf{CTest.K}_{\mathsf{ITest}} \stackrel{\text{def}}{=} (\textbf{all } \mathsf{t}' \text{: } \mathsf{t}' \textbf{ in } \mathsf{tr}(\mathsf{CTest.K}_{\mathsf{ITest}}) \text{: } \textbf{exists } \mathsf{t} \text{: } \mathsf{t} \textbf{ in } \mathsf{tr}(\mathsf{ITest.K}) \text{:}$$
$$\mathsf{fst(t)} \preceq_{I'} \mathsf{fst(t')} \;\&\&\; \mathsf{snd(t)}{=}{=}\mathsf{snd(t')})$$

Additional constructors are like additional methods, except that the first simple trace of the concatenated sequence must be that of a constructor. We define str($\widehat{\mathsf{ITest}}$) to be the union of all simple traces of the constructors of ITest. The rule for additional constructor L then becomes:

$$\mathsf{ITest}^* \preceq_{I'} \mathsf{CTest.L}_{\mathsf{ITest}} \stackrel{\text{def}}{=} (\textbf{all } \mathsf{s}' \text{: } \mathsf{s}' \textbf{ in } \mathsf{str}(\mathsf{CTest.L}_{\mathsf{ITest}}) \text{: } \textbf{exists } \mathsf{s} \text{:}$$
$$\mathsf{s} \textbf{ in } (\mathsf{str}(\widehat{\mathsf{ITest}}) \frown \mathsf{str}(\mathsf{ITest})^*) \text{: } \mathsf{s} \preceq_{I'} \mathsf{s}')$$

[3] In most languages, constructors have the same name as the containing classes. To avoid overloading, we use different identifiers.

Combination If all the above conditions hold for all methods and constructors of CTest, then CTest refines/implements ITest (written ITest $\leq_{I'}$ CTest). Furthermore, we write ITest $\leq$ CTest for (**exists** I': : ITest $\leq_{I'}$ CTest).

7 Refinement Proofs in Practice

The above trace refinement rules are difficult to apply directly. In cases where the mandatory external calls in the specification and the implementation are embedded in similar control structures (loops, conditionals), we can use simpler data refinement in context rules for corresponding blocks (Fig. 16).

7.1 Data refinement

We review some fundamentals of weakest preconditions and refinement following [6] and of data refinement following [4].

Weakest precondition. For statement S and predicate q, wp(S,q) denotes Dijkstra's weakest precondition, that is the set of states from which S is guaranteed to terminate in q. For S and predicate p the strongest postcondition sp(S, p) denotes the smallest set of states in which S may terminate if started from p.

Assert and guard. The assert statement **assert** p skips if the boolean expression p holds and aborts otherwise. The guard statement is the dual of the assert. For predicate p, **guard** p skips if p holds and magically establishes any postcondition if p does not hold.

Algorithmic refinement. Statement S′ refines statement S, written S $\sqsubseteq$ S′, if S′ establishes any postcondition q from any state where S establishes q:

S $\sqsubseteq$ S′ $\stackrel{\text{def}}{=}$ (**all** q: : wp(S, q) $\subseteq$ wp(S′, q))

Data refinement. Data refinement is a general technique by which one can change the data representation in a refinement. Any observable behavior of the refined component must also be observable on the abstract component. Note that unlike for greybox refinement, the observable behavior does not include the return values of method calls or the external calls made by the statement. The return value is usually included in data refinement of abstract data types, but the external calls in combination with the state transformation is believed to be unique to greybox refinement.

Our formal definition of data refinement is based on the simpler to apply, but incomplete proof technique of forward simulation. For relation $R : \Sigma \leftrightarrow \Gamma$ let [R] denote a nondeterministic relational update, that is a statement from Σ to Γ such that the states are related by R. It is the same as **any**(γ': R(σ, γ')) $\{\gamma = \gamma';\}$ except that it also changes the state space. Statement S′ data refines statement S under relation R, written S $\sqsubseteq_{\mathsf{R}}$ S′:

S $\sqsubseteq_{\mathsf{R}}$ S′ $\stackrel{\text{def}}{=}$ S; [R] $\sqsubseteq$ [R]; S'

```
interface IB {
  private E e; // reference to other object
  invariant I
  int m(Z z) pre p {
    W w; C c; int r;
    S;
    w=e.n(c);
    T;
    return r;
  }
}
```

```
class CB implements IB {
  private E e′; // reference to other object
  invariant I′ // implies IB.e==e′
  public int m(Z z) {
    W w'; C c′; int r';
    S′;
    w′=e′.n(c′);
    T′;
    return r′;
  }
}
```

Fig. 16. Structure-Preserving Refinement

7.2 Piecewise data refinement in context

In case of structural similarity (Fig. 16), we can establish greybox refinement by proving data refinement in context of the corresponding blocks (S, S′ and T, T′), the parameters of external calls (r, r′), and of the result values (r, r′). Let S, S′, T, and T′ be statements without any calls to modification operations of mentioned component instances. Let E.n denote a modification operation. Furthermore, let _c, _c′, _r, and _r′ be fresh variables of the same types as their correspondents without the initial underscore.

We consider separately the first block and the following blocks. For each we develop a sequence of increasingly more general, but also more complex rules. Especially if proofs are done informally, it is often easier to apply a less general rule, provided that it suffices.

First block. The first block S including the value of the method parameter has to be data refined by S′:

$$S;\ \_c{=}c \sqsubseteq_{I' \,\&\&\, \_c==\_c'} S';\ \_c'{=}c' \tag{1a}$$

The assignments to _c and _c′ and the extension of the refinement relation by _c==_c′ guarantee that the values of the actual parameters of the method calls e.n and e′.n are the same.

This condition is sufficient, but too strong. We only require data refinement in a context where the invariant I and the precondition p hold. We express the context with an **assert** statement:

$$\textbf{assert}\ I \,\&\&\, p;\ S;\ \_c{=}c \sqsubseteq_{I' \,\&\&\, \_c==\_c'} S';\ \_c'{=}c' \tag{1b}$$

That we only consider contexts where I′ holds is already determined by the refinement relation.

Following blocks. In the rules for the second and the following blocks, it is a bit more complicated to express the minimal context in which data refinement must hold. The first rule does not limit the context:

$$\mathsf{w{=}e.n(c);\ T;\ \_r{=}r} \sqsubseteq_{\mathsf{I'}} \mathsf{\ \&\&\ \_r{=}{=}\_r'\ w'{=}e.n(c');\ T';\ \_r'{=}r'} \tag{2a}$$

Here we use _r and _r′ to assure refinement of the return values. The preceding method calls e.n, respectively e′.n, are required because the refinement relation is only required to hold after termination of S and S′, but not after the method calls.

Condition (2a) is sufficient, but too strong. We only require refinement to hold in contexts that are reachable by executing S in states where I && p holds. This gives us the sharper condition:

$$\textbf{assert}\ \mathsf{sp(S,\ p\ \&\&\ I);\ w{=}e.n(c);\ T;\ \_r{=}r} \sqsubseteq_{\mathsf{I'}} \mathsf{\ \&\&\ \_r{=}{=}\_r'\ w'{=}e.n(c');\ T';\ \_r'{=}r'} \tag{2b}$$

This is the sharpest context we can describe with asserts; using guards we can express an even more general rule. Often, S′ is more deterministic and, therefore, gives us more context information than S. Consider the following correct refinement example:

I: true	p: true	S: **any**(int y: true) {x=y;}	T: x=1
I′: x==x′ && e==e′		S′: x′=1	T′: skip

This cannot be proved correct with the above rule, but with the following more general rule:

$$\mathsf{w{=}e.n(c);\ T;\ \_r{=}r} \sqsubseteq_{\mathsf{I'}} \mathsf{\ \&\&\ \_r{=}{=}\_r'\ }\textbf{guard}\ \mathsf{sp(S',\ im(I',\ I\ \&\&\ p));\ w'{=}e.n(c');\ T';\ \_r'{=}r'} \tag{2c}$$

This rule is sufficiently complete for most practical applications. A complete rule, requiring additional concepts and notation, as well as as a proof of completeness are beyond the scope of this paper.

Let condition (1) be true if (1a) or (1b) holds and let condition (2) be true if (2a), (2b), or (2c) holds. Then we get the following theorem:

Theorem 1 (Soundness of piecewise data refinement). *If conditions (1) and (2) hold, then* $A.m \leq_{I'} B.m$.

The semi-commuting diagram in Fig. 17 shows this piecewise data refinement in context, where solid lines mean 'for every choice' and dashed lines 'there exists a choice'. The strongest postcondition expressions are remarks to illustrate conditions (2b) and (2c).

Insufficient conditions. To sharpen the intuition, we also list the following two alternatives for condition (2), which are sometimes wrongfully believed to be sufficient. In the first case we require data refinement of the complete method:

$$\mathsf{S;\ w{=}e.n(c);\ T;\ \_r{=}r} \sqsubseteq_{\mathsf{I'}} \mathsf{\ \&\&\ \_r{=}{=}\_r'\ S';\ w'{=}e.n(c');\ T';\ \_r'{=}r'} \tag{f1}$$

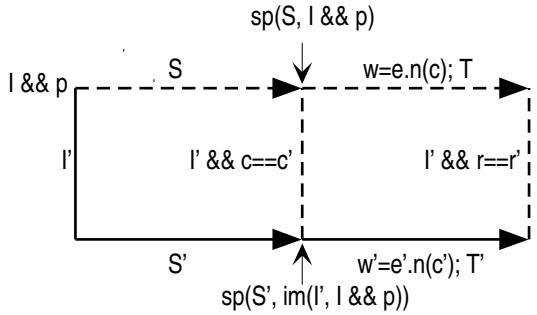


Fig. 17. Piecewise Data Refinement

The following counterexample shows that the above condition (f1) together with (1) is insufficient. The conditions hold, but greybox refinement doesn't because $<0, 1>$ is not a legal sequence of states for x.

I: true	p: true	S: **any**(int y: true) {x=y;}	T: skip
I′: x==x′ && e==e′		S′: x′=0	T′: x′=1

The second insufficient replacement for condition (2) is 'derived' from the layered specification pattern (Fig. 8). It says that partTwo, standing for T, must be data refined by partTwo′, that is T′. This condition, $\mathsf{T} \sqsubseteq_{\mathsf{I}'} \mathsf{T}'$, as well as condition (1) hold in the following example:

I: true	p: true	S: skip	w=e.n(c): w=1	T: w=0; x=1
I′: w==w′ && w′==0 && x==x′		S′: skip	w′=e′.n(c′): w′=1	T′: w=0

However, trace refinement does not hold. The problem of this condition is that it wrongfully assumes I' to hold *after* the call to e.n. If we impose and prove the additional, unnecessarily restricting consistency requirement on specifications that the invariants also hold after the external method calls, then we can use this rule. For layered specifications (Sect. 3) this means that with this additional consistency requirement, we can produce a correct implementation by implementing —or if we write the specification as an abstract class inheriting— the whitebox layer unchanged and proving data refinement in context (without any calls) of the blackboxes.

Sufficient conditions for additional methods and for constructors using data refinement in context follow the same pattern as conditions (1) and (2).

8 Refinement of Component-Based Systems

In this section, we show what greybox refinement of individual components implies for systems assembled thereof.

We assume that components only reference interfaces and use factories [21, Factory Pattern] to create instances of components implementing the respective interface. Let

CA and CB be two components implementing interfaces IA and IB, respectively. We define ◇(IA:=CA, IB:=CB) to be the composition of components CA and CB, that is an environment in which CA is used for the instantiation of elements of IA and CB for IB. Conceptually, but not for actual execution, we can also use interfaces for instantiation. For example, ◇(IA:=IA, IB:=IB) denotes the environment in which IA and IB are used for creating instances of themselves.

Greybox refinement extends naturally to compositions of components. If such a composition forms a complete system in the sense that it doesn't make any external calls, then greybox refinement coincides with more traditional forms of abstract data type refinement (e.g. [17]). For compositions, we get the following theorem:

Theorem 2 (Preservation of observational behavior). *We assume that there are no infinite mutual recursions (Sect. 6.1) and no reentrant calls to modification methods via unmentioned component instances (Sect. 6.2). Furthermore, we assume that no run-time type test or other form of reflection is used. If IA ≤ CA and IB ≤ CB then the following five greybox refinements hold:*

(1) ◇(IA:=IA, IB:=IB) ≤ ◇(IA:=CA, IB:=IB)
(2) ◇(IA:=IA, IB:=IB) ≤ ◇(IA:=IA, IB:=CB)
(3) ◇(IA:=IA, IB:=IB) ≤ ◇(IA:=CA, IB:=CB)
(4) ◇(IA:=CA, IB:=IB) ≤ ◇(IA:=CA, IB:=CB)
(5) ◇(IA:=IA, IB:=CB) ≤ ◇(IA:=CA, IB:=CB)

Especially properties (4) and (5), which do not hold in other approaches, are interesting. Instantiate IA by ITextObserver, CA by VendorATextObserver, IB by ITextModel, and CB by VendorBTextModel and assume that the hypothesis of Theorem 2 holds. Assume furthermore, that vendor A has programmed the observer —in combination with ITextModel— to display the beginning of the model's text in a 80 * 25-character size window. Assume that we select this observer, because it displays the text in the desired format, and combine it with VendorBTextModel. Property (4) then guarantees that the combined system actually displays the model's text in the aforementioned format.

Property (3) does not guarantee this because the specification of ITextObserver does not even prescribe the observer to display the text. It could as well read it out, spell check it, or scan it for credit card numbers. Properties (4) and (5) are, therefore, crucial for modular and component-based development.

Run-time type tests and other forms of reflection have been explicitly excluded in Theorem 2, because they would give additional observational power that could be exploited to make any refinement impossible. For example, property (3) would not hold if IB and CB contained a statement like:

```
IA a=IAFactory.create(); if(a instanceof CA) {... /* change some state */}
```

It is, of course, possible to use reflection and invoke additional methods in a way that preserves refinement. Namely, clients may invoke additional methods to achieve the same effect as multiple calls to old methods. We illustrate this with a component CB that uses a run-time type test to check whether a referenced object is of a certain type. If it is, CB.call invokes an additional method.

```
interface IB {
  void callM(IA a) {
    any(int no: no>=0) {
      for(int i=0; i<no; i++) {
        choose {
          if(exists X1 x1: : p1(a, x1)) {
            any(X1 x1: p1(a, x1)) {
              Y1 y1=a.m1(x1);
            }
          }
        | x.others();
        }
      }
    }
  }
}
```

```
class CB implements IB {
  public void callM(IA a) {
    if(a instanceof CA &&
        (exists X2 x2: : p2((CA)a, x2))) {
      any(X2 x2: p2((CA)a, x2)) {
        Y2 y2=((CA)a).m2(x2);
      }
    }
  }
}
```

Fig. 18. Use of Reflection and Calls to Additional Methods

Let IA have method m1 of the following form:

Y1 m1(X1 x1) **pre** p1(this, x1) {...}

Method callM of IB invokes an arbitrary number of times methods m1 and others (Fig. 18). This is expressed with a **choose** statement, which nondeterministically selects and executes one of its alternatives separated by '|'. The implementation of callM in CB invokes the additional method m2 of CA instead.

Imposing again the same restrictions on the other components as above, we get the following theorem:

Theorem 3 (Reflection and additional methods). *We assume that there are no infinite mutual recursions (Sect. 6.1) and no reentrant calls to modification methods via unmentioned component instances (Sect. 6.2). If only CB uses reflection and if IA $\leq$ CA then the same five greybox refinements as in Theorem 2 hold.*

9 Towards a Greybox Specification Language

In this paper we have used invariants, preconditions, specification statements, abstract data types, and loops over sets as extensions to Java for formulating component contracts. The final definition of a greybox specification language is subject to future research. To get some feedback what constructs such a language needs to include, we have conducted a number of small to medium sized case studies within our group. The most notable example [52] is the specification of a part of the text subsystem of the commercial BlackBox component framework [42]. The language used for this was an extension of Component Pascal [43], the implementation language of the BlackBox

component framework. The greybox specification was shown to be consistent with the original blackbox specification provided with the product.

There are two reasons why it is important to fix a greybox language and not just use ad-hoc notations. First, tool support can only be provided for a clearly defined language. Second, ad-hoc notations are subject to different interpretations. Below we summarize the specification extensions used in our case studies:

- Invariants and method preconditions (Sects. 2.2, 4) with universal and existential quantifications.
- Fields, private members (methods, fields), and constructors in interfaces (Sects. 2.2, 4).
- The modifier **inquiry** for inquiry only methods (Sect. 4).
- The special method others to provide for more modifications in additional methods (Sect. 6.4).
- Abstract data types set and sequence of objects together with the usual operations (Sect. 2.2).
- Loops (**do**) to iterate over sets and sequences (Sect. 4).
- Specification statement **any** (Sect. 4).
- The non-deterministic control structure **choose** (Sect. 8).

10 Related Work

In this section, we discuss whether and how other specification methods can be utilized to (1) specify both state transformations and mandatory calls and (2) prove refinement of both aspects in implementations.

Pre/post specifications. As discussed in Sect. 2.3, methods that are based on pre/post specifications (without an explicit encoding of the external call trace) cannot specify mandatory external calls that the component must make. Thus Z [50] does not satisfy our requirements. Neither do Meyer's design by contract [33, 35], the Object Constraint Language (OCL) [55], or the Java Modeling Language (JML) [31], although they are especially targeted at object- and component-based development, respectively a language for these paradigms. As pointed out (Sect. 3), we can express layered specifications with notations, such as Eiffel, that express both white- and blackbox specifications. However, none of these approaches comes equipped with refinement rules that also preserve the external call sequence.

We are not aware of any pre/post specification-based methods that actually encode the call sequence with the respective states into trace variables (Sect. 2.3) to achieve the same expressiveness as greybox specifications.

B The B method [1] uses a combination of preconditions and abstract statement sequences to specify operations. The operation bodies can contain calls to operations of imported modules. However, in refinement steps only the overall state transformation, but not the external call sequence needs to be preserved. Thus, B does not solve the problem at hand. Related methods such as VDM [28], VDM++ [16], and RAISE [46] do not give a better grip on the problem.

Class refinement. Mikhajlova and Sekerinski [38, 37, 39] also use a refinement calculus-based extension of an object-oriented language with nondeterministic constructs and abstract data types for their treatment of class refinement. Their definition of refinement only requires data refinement of the state transformations, but does not include refinement of external call sequences. Because they equal non-termination with abortion, they do not have a practically useful treatment of methods that make infinitely many external calls. Their condition for additional methods is weaker than ours: Additional methods must simply preserve the strongest invariant implied by existing methods; thus, preserve absolute rather than relative (from the current) state reachability as we demand. This would not be a sensible option for greybox refinement because no refinement of external call sequences could be demanded. No provisions, such as our special others method, are made for additional methods to perform supplementary modifications. Their refinement theorems do not take reflection into consideration.

Behavioral subtyping Behavioral subtyping, a related notion to class refinement, establishes data refinement between types that are meant to stand for classes and specifications thereof. It has been studied in semi-formal settings guaranteeing only partial correctness by America [2], by Liskov and Wing [32], and by Dhara and Leavens [19]. However, because pre/post specifications are used, these approaches are not suitable for the problem at hand either.

Components with reentrance Mikhajlov et al. [36] have developed a method for semi-modular refinement of components in the presence of reentrance. Calls to other components can be specified, but only the resulting state transformations have to be refined. Hence, they cannot handle the kind of systems exemplified in this paper by the observer pattern. Specifically, if we replace greybox refinement by their definition of refinement in Theorem 2,[4] then only properties (1) – (3), but not the crucial properties (4) and (5) hold. Additional minor differences include them not handling additional methods and equaling non-termination with abortion.

Refinement calculi. As exemplified by our own proposal, the refinement calculi of Back [3, 6] and Morgan [40] with their abstract statement notation can be used to specify external calls. However, their refinement rules only take the state transformations, but not the calls, into account. The semantics of method calls is defined by reduction, which is inappropriate for the problem at hand.

Contracts of Helm, Holland, and Gangopadhyay. Helm et al. define a mostly syntactic notion of interaction contracts [24, 25] for the object-oriented design of components. External calls can be explicitly specified. However, lacking the distinction between modification and inquiry operations, all specified calls are mandatory. As a consequence, not only the call to deleteNotification in ITextModel.deleteCharAt, but also the call corresponding to our charAt in ITextObserver.deleteNotification is explicitly mentioned to be mandatory in their treatment of the observer pattern.

[4] This is only meaningful if the combined systems, like ◇(IA:=CA, IB:=CB), make no external calls.

Call sequences can be specified using sequential and parallel composition as well as conditionals. The local state change is indicated by a combination of postcondition and place where the modification should be executed:

SetValue(Value val) {Δvalue; Notify();} [value==val]

This means that first, indicated by the Δvalue, value should be set so as to satisfy the postcondition, that is to val. This notation does not work if several changes with external calls in-between should be made. For example, the following specification, presented in our notation, can not be expressed in theirs:

SetTwoValues(Value val1, Value val2) {value=val1; Notify(); value=val2;}

Hence, their notation cannot satisfactorily be used to express the states in which external calls have to be made. Furthermore, no operation preconditions can be expressed.

Holland's thesis [25] gives part of an operational semantics for an object-oriented programming language and for interaction contracts. However, no semantic reasoning is done. There is a notion of contract refinement to design specialized contracts; however, no clear semantic conditions are listed and the examples are such that not even the state transformation aspect can be captured by any standard notion of data refinement. No conditions for the correct implementation of specified call sequences and state transformations is given. The notion of contract refinement is not applicable for proving implementation correctness, because there is a fundamental dichotomy between contracts (specifications) and class implementations in their work. The OOram method [47] partly expanded on these ideas, but does not solve the problems discussed in this paper.

UML. Different kinds of diagrams from the Unified Modeling Language (UML) [49] can be used to specify both state changes and call sequences. Sequence and collaboration diagrams —collectively called interaction diagrams— show interactions of fixed sets of objects, including the messages sent among them. The more common instance form describes one actual sequence of message interchanges; thus, it is not appropriate for general specifications. On the other hand, the generic form describes all possible sequences using loops and branches. Using loops, we can also indicate messages sent to an a priori unknown set of objects, such as our observers. There is no notation to distinguish between mandatory and optional calls. The main focus of interaction diagrams are the possible message sequences. State changes can be indicated by placing a copy of an object icon showing those modifications. No invariants or operation preconditions can be expressed in collaboration diagrams. Thus no consistency check (Sect. 4.1) is possible. Figure 19 shows the collaboration diagram approximating ITextModel.deleteCharAt. UML does not prescribe a fixed format for repetition expressions or state changes; hence, we use our own notation.

In our experience, interaction diagrams that make use of loops, branches, and object icon duplication to express greybox specifications quickly become crowded and unreadable. We are not aware of any use of interaction diagrams in the sense of greybox specifications. Formal semantics for UML is still work in progress. Furthermore, UML lacks a notion of refinement and, therefore, cannot be used to assert the correctness of

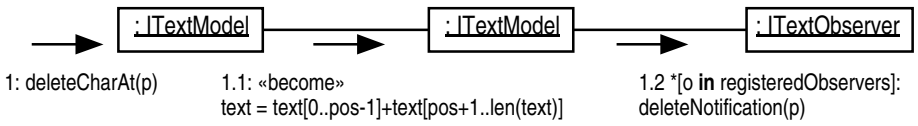


Fig. 19. Collaboration Diagram for ITextModel.deleteCharAt

implementations based on UML diagrams. The object message sequence chart [11] and the message sequence chart notations [26], from which UML sequence diagrams are derived, have the same limitations.

Activity diagrams can also be used to model operations. They give flowchart-like representations as used in visual programming languages. However, even the principal authors of UML admitted that this is usually more cumbersome than a textual representation [8]. Furthermore, there is no clear notation to indicate external calls and activity diagrams lack both a formal semantics and refinement rules.

Catalysis. Catalysis [20], a method specifically targeted at the development of components and frameworks, contains several possibilities to indicate what external calls must be made during the execution of a method. For most cases, the preferred way is to use UML statechart diagrams. Statecharts show state machines that emphasize the flow of control from state to state. Although state changes, external calls, conditionals, and loops can all be encoded in the Catalysis version of statecharts, they are really meant for higher levels of abstraction and, therefore, quite cumbersome to use for greybox-like specifications. Catalysis does not provide a clear semantics and refinement rules that preserve all relevant aspects.

Sequence expressions can express sequencing constraints on external method calls using sequential composition, alternative, arbitrary iteration, and concurrency, but no conditionals. Time indexes (e.g. x@i for the value of x at time i) can be used for parameters of calls in message sequences, but it is impossible to indicate in which states external calls have to be made.

Catalysis differentiates between optional and mandatory calls. Unlike in our approach where this distinction is based on the kind of the called method, Catalysis lets the specifier of the caller decide individually for every call. Catalysis lacks a formal semantics and has only vague informal refinement rules for asserting the correctness of an implementation.

No other surveyed method, such as Fusion and OOAD, gives a better grip on the problem.

Algebraic specifications. Algebraic specifications suffer from the same deficiencies as pre/post specifications. Consider the typical stack example. For stack s and element e, s==pop(put(s, e)). Here we cannot specify what external method calls methods pop and put must make.

Grey box data refinement. The terms greybox specification and refinement have been coined by the authors of this article [10]. Boiten and Derrick later introduced the similar names grey box data refinement and types for an unrelated form of data refinement [7].

11 Conclusions

Specification approaches that only relate the state prior to operation invocation to the state after operation termination are insufficient to cope with call-backs in extensible systems: The sequence of external calls and the respective states in which the latter must be made cannot be specified. An encoding with auxiliary trace variables could theoretically solve this problem. However, in practice such an encoding would be very complex and almost unreadable.

Specifications that only relate pre- and post-operational states are called blackboxes. As the other extreme, whitebox specifications contain all implementation details which often makes them too restrictive. On the middle ground there are two possibilities. In a discrete combination of the concepts, one can layer a whitebox on blackboxes. On a continuous scale, we recommend a new method which we call greybox specification.

To specify external calls, we proposed component interface specifications to draw on abstract programs rather than on pure blackbox views. Formally, this approach has a sound basis in the refinement calculus. Practically, abstract programs are very close to the programmers' intuition. To increase acceptability further, we recommend to define a greybox specification language as a natural extension of an implementation language. In this paper we use such an extension of Java.

Greybox refinement preserves or refines the observable behavior of components. Unlike in normal data refinement, external calls, and not just their state transformation effect in the specification, are also considered part of the observable behavior. As a result, properties of component implementations beyond those in the specifications are preserved when combined with other components into a system. The given refinement rules can be used to establish the correctness of implementations with respect to specifications. These rules can be used for fully formal reasoning and also give an intuition for informal justifications.

Greybox specifications also have a number of 'soft' advantages [10]: They are (usually) shorter than source code, tend to be more readable than large postconditions — even without trace encoding—, scale better, and allow to indicate enough detail for resource-efficient reuse. Here, we have on purpose not discussed these advantages in order not to distract from the fundamental problem solved by greybox specifications.

Acknowledgments. We would like to thank Ralph Back, Dominik Gruntz, Cuno Pfister, Clemens Szyperski, Anna Mikhajlova, and Leonid Mikhajlov for a number of fruitful discussions.

References

1. Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

2. Pierre America. Designing an object-oriented programming language with behavioral subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop*, pages 60–90. LNCS 489, Springer Verlag, 1991.
3. Ralph Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.
4. Ralph Back. Changing data representation in the refinement calculus. In *21st Hawaii International Conference on System Sciences*. IEEE Press, 1989.
5. Ralph Back and Joakim von Wright. Trace refinement of action systems. In *CONCUR 94*, pages 367–384. LNCS 836, Springer Verlag, 1994.
6. Ralph Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer Verlag, 1998.
7. E.A. Boiten and J. Derrick. Grey box data refinement. In J. Grundy, M. Schwenke, and T. Vickers, editors, *International Refinement Workshop & Formal Methods Pacific '98*, Discrete Mathematics and Theoretical Computer Science, pages 45–59. Springer-Verlag, September 1998.
8. Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley, 1998.
9. Martin Büchi and Emil Sekerinski. Formal methods for component software: The refinement calculus perspective. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proceedings of the Second Workshop on Component-Oriented Programming (WCOP)*, volume 5 of *TUCS General Publication*, pages 23–32, Short version in ECOOP'97 workshop reader LNCS 1357, June 1997. http://www.abo.fi/~mbuechi/publications/FMforCS.html.
10. Martin Büchi and Wolfgang 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, Zürich, September, 1997. http://www.abo.fi/~mbuechi/publications/GreyBoxes.html.
11. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley & Sons, 1996.
12. Ana Cavalcanti and David A. Naumann. A weakest precondition semantics fo an object-oriented language of refinement. In *Proceedings of FM'99: World Congress on Formal Methods*, pages 1439–1459. LNCS 1709, Springer Verlag, September 1999.
13. Marsha Chechik and John Gannon. Automatic analysis of consistency between requirements and designs. In *Proceedings of COMPASS'95*, pages 123–132, 1995.
14. D.D. Clark. Structuring a system using up-calls. In *Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP)*, ACM Operating System Review, 19(5), pages 171–180, 1985.
15. Derek Coleman et al. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.
16. E.H. Dürr and J. van Katwijk. VDM++ — a formal specification language for object-oriented designs. In *Computer Systems and Software Engineering, Proceedings of CompEuro'92*, pages 214–219. IEEE Computer Society Press, 1992.
17. Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge Tracts in Theoretical Computer Science, No. 47. Cambridge University Press, 1998.
18. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical report, Compaq SRC Research Report 159, 1998.
19. Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings 18th International Conference on Software Engineering*, pages 258–267. IEEE Press, 1996.
20. Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison Wesley, 1998. http://www.catalysis.org.

21. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
22. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.
23. Object Management Group. The common object request broker: Architecture and specification, 1997. Revision 2.0, formal document 97-02-25, http://www.omg.org.
24. Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of OOPSLA/ECOOP '90*, pages 169–180, 1990.
25. Ian M. Holland. *The Design and Representation of Object-Oriented Components*. PhD thesis, Northeastern University, 1993.
26. International Telecommunication Union. Z.120: Message sequence chart (MSC), October 1992. http://www.itu.int.
27. R. Janicki, D.L. Parnas, and J. Zucker. Tabular representations in relational documents. In C. Brink, W. Kahl, and G. Schmidt, editors, *Relational Methods in Computer Science (Advances in Computing Science)*, chapter 11, pages 184–196. Springer Verlag, 1997. Also as CRL Report 313, McMaster University.
28. Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, 1986.
29. G.E. Krasner and S.T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.
30. Gary T. Leavens. An overview of Larch/C++ behavioral specifications for C++. In Haim Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 121–142. Kluwer Academic Publishers, 1996.
31. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06d, Iowa State University, Department of Computer Science, April 1999.
32. Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
33. Bertrand Meyer. Applying 'design by contract'. *IEEE Computer*, 25(10):40–51, October 1992. See also http://www.eiffel.com/doc/manuals/technology/contract/index.html.
34. Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, second edition, 1992.
35. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
36. Leonid Mikhajlov, Emil Sekerinski, and Linas Laibinis. Developing components in the presence of re-entrance. In *Proceedings of FM'99: World Congress on Formal Methods*, pages 1301–1320. LNCS 1709, Springer Verlag, September 1999.
37. Anna Mikhajlova. *Ensuring Correctness of Object and Component Systems*. PhD thesis, Turku Centre for Computer Science, October 1999. http://www.tucs.fi.
38. Anna Mikhajlova and Emil Sekerinski. Class refinement and interface refinement in object-oriented programs. In *Proceedings of FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, pages 82–101. LNCS 1313, Springer Verlag, 1997.
39. Anna Mikhajlova and Emil Sekerinski. Ensuring correctness of Java frameworks: A formal look at JCF. Technical Report 250, Turku Center for Computer Science, March 1999. http://www.tucs.fi.
40. Carroll Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.
41. Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall Series in Innovative Technology, 1991.
42. Oberon microsystems, Inc. BlackBox Component Builder, 1997. http://www.oberon.ch/.
43. Oberon microsystems, Inc. Component Pascal, 1997. http://www.oberon.ch/docu/component_pascal.html.

44. David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
45. David Lorge Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.
46. The RAISE Language Group. *The RAISE Specification Language*. Prentice Hall, 1992.
47. Trygve Reenskaug, Per Wold, and Odd Arild Lehne. *Working with Objects: The OOram Software Engineering Method*. Manning, 1995.
48. Dale Rogerson. *Inside COM*. Microsoft Press, 1996.
49. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
50. J.M. Spivey. *The Z Notation*. Prentice Hall, second edition, 1992.
51. Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.
52. Petri Suni. Grey-box specification seems to work — a case study. LuK-tutkielma, University of Turku, Department of Computer Science, 1999.
53. Clemens A. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1997.
54. S. Tucker Taft and Robert A. Duff, editors. *Ada 95 Reference Manual: Language and Standard Libraries (International Standard ISO/IEC 8652:1995(E))*. LNCS 1246, Springer Verlag, 1997.
55. Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language : Precise Modeling With UML*. Addison-Wesley, 1999.
56. Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, 1982.
57. Niklaus Wirth. The programming language Oberon. *Software – Practice and Experience*, 18(7):671–690, 1988.

Paper IV

Compound Types for Java

Martin Büchi and Wolfgang Weck

Originally published in: *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) '98*, pages 362–373. ACM Press, 1998.[2]

[2]This paper has been reformatted for the different page size.

Compound Types for Java

Martin Büchi and Wolfgang Weck

Åbo Akademi University, Turku Centre for Computer Science,
Lemminkäisenkatu 14A, FIN-20520 Turku
Martin.Buechi@abo.fi, Wolfgang.Weck@abo.fi

Abstract. Type compatibility can be defined based on name equivalence, that is, explicit declarations, or on structural matching. We argue that component software has demands for both. For types expressing individual contracts, name equivalence should be used so that references are made to external semantical specifications. For types that are composed of several such contracts, the structure of this composition should decide about compatibility.
We introduce compound types as the mechanism to handle such compositions. To investigate the integrability into a strongly typed language, we add compound types to Java and report on a mechanical soundness proof of the resulting type system.
Java users benefit from the higher expressiveness of the extended type system. We introduce compound types as a strict extension of Java, that is without invalidating existing programs. In addition, our proposal can be implemented on the existing Java Virtual Machine.

1 Introduction

One of several reasons to use Java is its support of component-oriented programming, the creation of compiled building blocks to be used in different contexts, and the assembly of systems from such components. JavaBeans [34], Java's component model, competes with other component software standards, such as CORBA [13] and Microsoft's COM [32], but the language itself may also be used to program to these language independent standards.

Type systems, such as Java's, help to document and safeguard component interfaces. By annotating inter-component call parameters with types, one provides some primitive documentation on how to use a service and at the same time expresses a statically checkable precondition: the object passed must implement certain methods, as stated by the type.

Explicitly declared and named types can stand for contracts about services. The behavioral specification is documented separately and linked to the type via the name. A compiler can, of course, not check compliance with such a specification, but it can verify that references to the same types, and intentionally the same contracts, have been made. Explicitly stated contracts are particularly important in the component software realm [35].

Frequently, classes need to conform to more than one contract. For instance, Microsoft's OLE [6] defines ActiveX control containers via a bundle of contracts to be

implemented. Java supports multiple subtyping to this end. Similarly, one may want to declare variables or method parameters of a type comprising several contracts. This is not supported to the same degree by Java. On a first glance it may seem to be no problem because one only would have to declare the right subtype. We will demonstrate, however, that this may not be possible with independently developed software components.

This problem is explained in Sect. 2. In Sect. 3 we take the problem to its root, the question whether type compatibility is being decided by name equivalence or structural match. We will show that we need a mixture of both, and therefore, we propose compound types in Sect. 4.[1] In Sect. 5 we show how to add compound types to Java. They are a strict extension, that is, existing Java programs need not be changed. This also holds for the run-time support, the byte code and the virtual machine in particular. We illustrate the latter in Sect. 6. In Sect. 7 we report on a mechanically verified soundness proof for the extended type system. Section 8 relates to other work and Sect. 9 summarizes our conclusions.

2 The Problem

The problem with Java's type system is explained in this section. In 2.1 we briefly review the relevant aspects of Java's type system. In the second subsection, we introduce the essentials of component software, the domain in which the problem mainly surfaces. With these preliminaries we show that it may be impossible to sharply type a parameter to demand a specific combination of interfaces, so that existing or independently developed classes need not be modified to be compatible.

2.1 Java's type system

An essential ingredient of object-oriented programming and component software is polymorphism. In Java, subtyping relationships can be declared in three ways: a class can subclass another class, a class can implement an interface, and an interface can extend another interface. Subclassing provides for code inheritance in addition to subtyping, whereas interfaces are pure types, that is, no code is attached to them. Typically, multiple subtyping needs less extra conflict resolution rules than multiple subclassing. Thus, Java's designers decided that a class can have only a single direct superclass but implement several interfaces. Because a class without an explicitly declared superclass implicitly inherits from a predefined class Object, every class —except Object— subtypes exactly one class directly and an arbitrary number, possibly zero, of interfaces, as illustrated in Fig. 1.

If a class C is declared to implement an interface I, all methods defined by I exist in C, too. Thus, it is type-safe to assign instances of class C to variables of type I.

We have to take a closer look at the situation in which a class C implements several interfaces, say I and J, that both declare a method with the same name, m. If I and J both

[1] The compound types defined in this paper are not to be confused with structured types (records, arrays, functions), which are sometimes also called compound types.

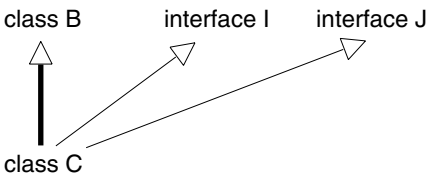


Fig. 1. A class extending a base class and implementing two interfaces

declare m with exactly the same signature and return type, C defines m only once and binds this to both interfaces. If the parameter lists differ, Java's overloading mechanism takes care of the situation. Both versions exist within C and upon a call the one with the best fitting signature is selected.[2] The case in which the parameter lists are equal but the return types differ is not permitted by the overloading rules. Consequently, the compiler rejects the declaration of C if I and J conflict in this way.

Subtyping relations can also be established between interfaces. An interface I can be declared to extend other interfaces J0, ..., Jn. A class implementing I, implicitly implements all interfaces J0, ..., Jn, but not vice versa. Interfaces cannot subtype (extend) classes.

Whereas subclassing is basically a mechanism for code reuse, multiple subtyping offered via interfaces can be used to express different aspects of objects. For instance, a class D of objects being an applet, runnable in a separate thread, and wanting to be informed of changes in observable objects, would subclass the class Applet and implement the interfaces Runnable and Observer.

These combinations are stated without the need to declare a new type first. The class D automatically constitutes such a new type of its own. Another class, E, also extending Applet and implementing the same interfaces as D, establishes a new type too, but a different one. Java uses name equivalence of types, that is, two types are compatible only if declared so. With classes D and E being declared as separate types, instances of one cannot be assigned to variables of the other's type. Figure 2 summarizes the discussed aspects of Java's type system.

- code inheritance via single subclassing
- multiple subtyping via interfaces (without code inheritance)
- conflict resolution (partially) via overloading
- only types declared to be compatible are compatible (name equivalence)

Fig. 2. Relevant aspects of Java's type system

[2] This may not be decidable, in which case an error is flagged, as defined by Java's rules about overloading.

2.2 Component software

One purpose of using object-oriented technology is to create building blocks to be used in several systems. To support such building blocks, Java features, for instance, separate compilation of classes and bundling of related classes as packages.

Component software tries to move the idea of building blocks to an industrial scale. Like in other engineering disciplines, software systems shall be assembled from premanufactured components rather than crafted individually by hand. This is an old idea, dating back to the NATO conference on software engineering in 1968 [22]. In 1990 Brad Cox even advocated an industrial revolution in the software realm [10], observing that software components are not just a technological issue but a cultural one as well. In particular, as also stressed in a recent book by Clemens Szyperski [35], the true potential of component software comes from establishing component markets. If system assemblers can acquire individual components from several vendors, they can actually combine the many special skills, ideas, and inventions each vendor has to offer. The number of possibly interesting combinations grows rapidly as vendors can join an open market to offer components in the field of their special competence.

An example, frequently used to illustrate this, are spell-checking components that can be composed with a text editor to provide for in-place checking and correction. For instance, a vendor with special competence in linguistics and a specific language, Finnish, offers an add-on spell checker [18] to be used with Microsoft Word. Offering a component rather than a complete word processor allows the company to concentrate on what their staff is good at. Also, the market for Finnish language checkers is probably not extremely large and may not give enough revenue to finance the construction of a competitive editor.

Furthermore, Microsoft's editor serves as an integration platform with other add-ons offered by further vendors. For instance, a user requiring not only well performed checking of Finnish but also needing to include some special type of diagrams into documents, may not find an of-the-shelf program with this particular feature combination. Custom programming of such a system would be too expensive in most cases, but a custom assembly from standard components may be achievable at a reasonable price. Because of this, component software is also described as a path between standard and custom software [29].

Using industrial components as described above has two requirements. Firstly, using a premanufactured component must be easy enough, compared to programming it from scratch, to balance the cost of acquiring it. Otherwise, component reuse is simply not going to happen because system development costs would increase instead of decrease.

The most inexpensive way to compose components is by plug-and-play. This means that neither the components need to be adapted nor any programming is required to glue the components together. The problem may not even be the programming itself, but the required reengineering and detailed understanding of the components or at least their interfaces. Plug-and-play in turn may sometimes be left to the end user. Netscape's Communicator Plug-Ins [7] are an example of this.

As a second requirement, vendors must be able to produce compatible components despite being mutually unaware. Considering a large component market, it is impossible to demand any vendor to know about all other products and to adapt to them.

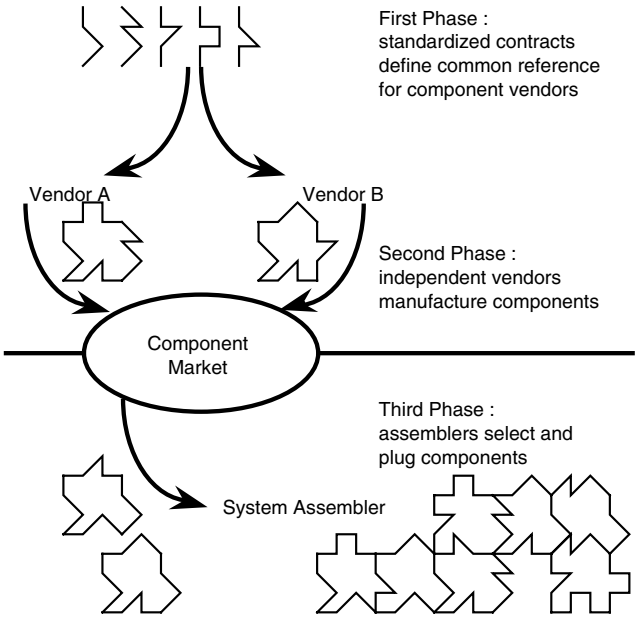


Fig. 3. The three phases of component-oriented programming

It seems, as if the latter requirement of independent component development would contradict the former requirement of plug-and-play, but fortunately this is not the case. Component vendors cannot be expected to synchronize their work with each other, but they can build on common standards. If the latter are properly designed, the independently produced components will still interoperate in a plug-and-play manner.

It is not sufficient, however, to use a common wiring (or plumbing) standard, such as COM, CORBA, or a specific programming language, such as Java. These standards only define the calling conventions for procedures or methods respectively. In addition, component plug-and-play requires application domain dependent standard contracts, specified both syntactically and semantically.

The three phases of creating component systems are shown in Fig. 3, omitting feedback loops, which drive the evolution of standards and components. During the first phase, different standard interfaces are designed and described in public. In the second phase, vendors program towards these standards and place the resulting components on the market. In the third phase, finally, system assemblers select and acquire components from the market and plug them together. Note, that assemblers do not need to analyze *what* a standard interface actually specifies. It suffices to know *that* two components refer to a common standard.

In Java, types are used to support standard contracts. By types we understand both classes and interfaces. Like a plug-and-play system assembler, the loader can check that two components refer to the same type(s) and are thus compatible. A type's name

designates the standard. The full behavioral specification associated with it must be stated outside the language in the documentation for component manufacturers.

2.3 A scenario exemplifying a problem with Java

The following example describes a situation that cannot be properly handled by Java's type system. We assume two different and independent standards, which have come into existence entirely unrelated. One of them defines an interface Text, describing operations, such as insertion and deletion of characters. We also assume a transformation function which converts text positions to pixel positions. The second standard defines a compound document framework, like OLE [6], including an interface Container to be implemented by all objects that may act as compound document containers. The latter must support insertion and removal of document parts. Figure 4 shows portions of these two interfaces in Java.

Both standards form individually useful frameworks. Vendors can build components for either of them. The problem comes with the wish to create components that build on both standards simultaneously. In our example, this would be components that deal with both texts and containers.

In object-oriented programming, combining independently emerged frameworks has been described as an open problem [21]. This is not a problem with component software, where components just implement standard interfaces but do not reuse code from the framework. A component can implement several interfaces belonging to different standard collections. (A common plumbing standard, for instance Java, is still helpful but not strictly necessary.)

Figure 5 shows portions of two sample classes, TextContainerA of Vendor A and ContainerTextB of vendor B, both implementing the interfaces Text and Container of our sample standards. These classes exhibit a little nuisance. To insert a document part

```
/* as part of a text framework: */
  public interface Text {
    void insert (char ch, int textPos);
    /* insert character ch at position textPos */
    ...
    java.awt.Point displayPoint (int textPos);
    /* returns the display position at which the character at textPos is drawn */
    ...
  };

/* as part of a compound document framework: */
  public interface Container {
    void insertPart (DocPart part, java.awt.Point xyPos);
    ...
  };
```

Fig. 4. The standard interfaces Text and Container

```
/* Vendor A's component: */
  public class TextContainerA implements Text, Container {...};

/* Vendor B's component: */
  public class ContainerTextB implements Text, Container {...};
```

Fig. 5. Classes offered by vendors A and B

one has to pass the graphical coordinates because the container interface must be used. One may prefer to give a text position and have the part inserted after the corresponding character. For this purpose, a generic service can be implemented that maps the text position to a display position and then inserts the document part there. We assume that a Vendor C wants to offer this service within a class LibraryServices. Figure 6 shows part of this class and Fig. 7 illustrates that whole scenario.

Vendor C's library service works only for instances of classes that implement both interfaces Text and Container. Unfortunately, this cannot be expressed by the type of parameter into, as the question mark in Fig. 6 indicates.

The obvious solution is to create a combined interface TextContainer, which extends both Text and Container and does not add or hide anything, and to declare parameter into of this type. However, instances of neither TextContainerA nor ContainerTextB are compatible with the library service, as they are both declared to implement only the base interfaces but not the combined interface TextContainer. The problem is who is to define the interface TextContainer to be used by all parties? It is not part of either of the two frameworks because they are assumed to be independent. If one of the vendors A, B, or C defines TextContainer, the others would be obliged to use this definition. This contradicts mutual unawareness postulated for component software vendors. On the other hand, if all three vendors declare their own combined interfaces, they are not compatible either.

The problem can be partly tackled by conventions. A class never implements more than one interface directly; an interface never extends any of its superinterfaces by more than one base interface. Instead, combined interfaces named as concatenation of the fully qualified names of the two direct superinterfaces in alphabetical order are introduced into a package CombinedInterfaces. In our example, all three vendors would create and use interface com_X_Text_com_Y_Container, assuming that Text is part of

```
public class LibraryServices {
  public static void insertDocPart (DocPart part, ? into, int textPos) {
    /* the question mark stands for a type saying that interfaces
     Text and Container must be implemented */
    into.insertPart(part, into.displayPoint(textPos));
  }
};
```

Fig. 6. Vendor C's library services

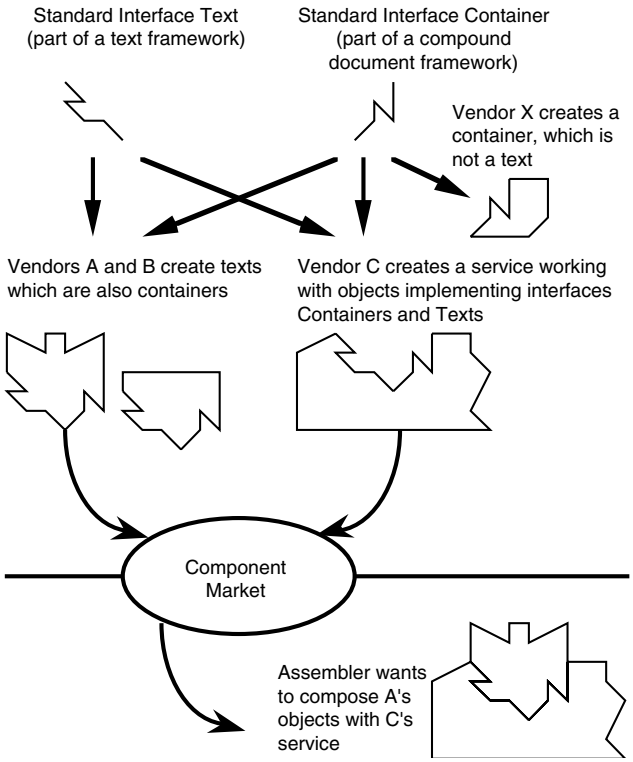


Fig. 7. Independent development of classes and insertion service

standard X and Container of Y. The system assembler then deletes all but one of the equivalent definitions. Unfortunately, this renders plug-and-play less feasible.

Furthermore, the conventions-based approach suffers from the combinatorial explosion of the number of interfaces. For the combination of three interfaces, the three pair interfaces have to be created and the combined interface has to be defined as the extension of all three pair interfaces as to make its implementations compatible with the pair interfaces as well. The overhead of this solution and, herewith, the pollution of the name space grows exponentially with the number of combined interfaces. Furthermore, legacy classes that do not abide by this convention are left out.

Finally, this approach fails completely, if we try to combine three or more types one of which is a class. Assume that we have a class C and interfaces I and J. According to the above convention, this would give us the abstract classes CI and CJ as well as the combined interface IJ. For classes implementing all three types C, I, and J to be compatible with the three pair types CI, CJ, and IJ, the triple type would have to extend all three pair types. This, however, is impossible because Java does not support multiple class inheritance. We can define a class D which extends CJ and implements IJ, but instances of D —or D's subclasses— cannot be assigned to variables of type CI

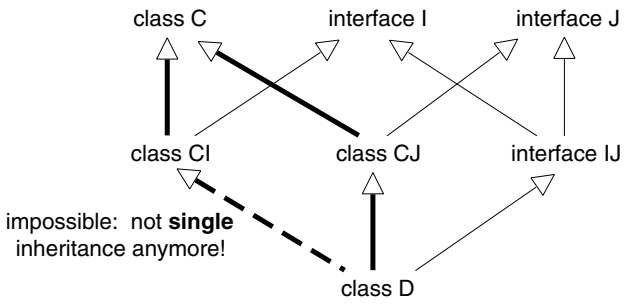


Fig. 8. Impossibility of compatibility with all subsets of supertypes

(Fig. 8). Java does not permit us to declare a class to be compatible with all subsets of its supertypes. This is an additional problem, not bound to component software.

As a different approach, we can resort to run-time tests and textual annotations. We declare parameter into of InsertDocPart (Fig. 6) to be of type Text, add a comment that it must also implement Container, and cast the parameter's value to Container when accessing the latter's members. In this approach, we loose static type checking.

Yet another possibility would be to use two parameters, one of type Text and one of type Container and require them to reference the same object. Again, we need a less desirable run-time test instead of compile-time type checking.

3 Structure vs. Name Equivalence of Types

The problem described above can be attributed to Java's use of name equivalence of types. Types are compatible only if explicitly declared so. A radical cure would be to use structure equivalence instead, as for instance proposed in [16]. All types that look alike would be considered compatible in this case. From the modeling perspective of object-oriented programming, however, name equivalence is more expressive. In this section we review the advantages of both structural and name equivalence, before we introduce compound types as a beneficial combination of these two in the next section.

3.1 Structure equivalence of types

With structure equivalence, any two types containing methods and fields with the same names and signatures are equivalent. Likewise, subtyping is based on structure. A type T is assumed to be a subtype of another type S if T contains at least all the methods and fields contained in S. This is the principle; there are more elaborated rules, for instance, allowing for co- or contravariant parameters.

The goal is an as big as possible type matching relation and, therefore, a maximum of flexibility. Types and the relations between them may even be inferred automatically by the compiler and thus need not be declared explicitly by the programmer.

If Java would use structure equivalence between types, the problems described in the previous section would not exist. Vendor C could define a combined interface TextContainer, extending Text and Container and use this to type the parameter into. As both implementations TextContainerA and ContainerTextB contain (at least) all methods named in TextContainer, they would be structural subtypes of TextContainer and, thus, compatible with the library service.

The fundamental purpose of a type system is to prevent the occurrence of run-time errors [4]. On a quasi syntactical level structural type equivalence suffices. Types prevent, for instance, 'method not understood' and memory access violation errors that could occur if, for example, an integer could be assigned to a pointer.

3.2 Name equivalence of types

When using objects as a modeling aid, we would like to eliminate errors beyond those covered by structural type equivalence. Whenever an object of a specific type is required, say as a parameter, we actually intend to require a specifically behaving object. Behavioral subtyping [19] and class refinement [24] formalize this idea of behavior associated to types and of subtypes having to refine the behavior of their supertype(s).

From this point of view, types stand for semantical specifications. While the conformance of an implementation to a behavioral specification cannot be easily checked by current compilers, type conformance is checkable. By simply comparing names, compilers can check that several parties refer to the same standard specification (Fig. 9).

Similarly, Microsoft's COM [32] uses interface identifiers (IIDs) to give each interface its own 'name.' IIDs become, like numbers of ISO standards, abstractions of specifications.

Consider also the analogy to the well-established component market for mass storage with its interface standards such as SCSI, IDE, etc. The buyer of a new hard drive simply ensures that she buys a SCSI drive, if that is what it says on her disk controller.

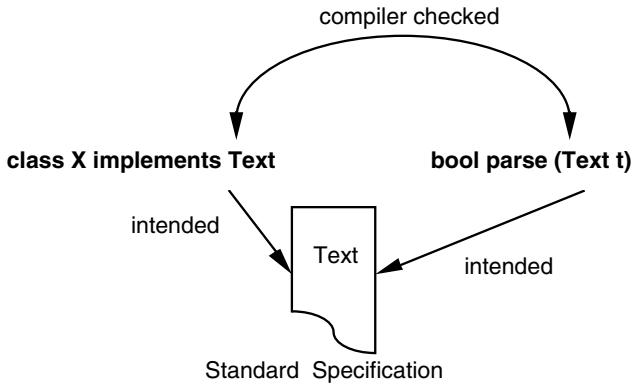


Fig. 9. Compiler checked reference to same standard specification

For her, the term SCSI represents a common reference made by the manufacturers of the controller and the drive.

The buyer would not be served well, if she would simply shop for a hard disk with matching mechanical connectors, as a drive that adheres to another, incompatible logical signaling standard might also fit this criterion.

In the same way, even if the projections of two unrelated semantical specifications by coincidence result in the same structure, the two should not be considered equal. As an extreme example of such accidental matches, borrowed from [20], consider two classes: Rectangle with operations Move and Draw and a class Cowboy with operations Move, Draw, and Shoot. Looking only at the structure, Cowboy is a subtype of Rectangle.

This can be ruled out only by forcing programmers to be explicit about their intentions. In other words, type equivalence and subtype relations must be declared rather than be inferred.

To this purpose, several languages use name equivalence. They consider two types compatible only if the declaration of either type explicitly refers to the name of the other. Java is one example. Modula-3 [5] uses structural type equivalence by default, but allows the programmer to explicitly demand name equivalence by assigning a unique brand to a type.

4 Compound Types

Both structural and name equivalence offer benefits as discussed above. Structure equivalence gives more flexibility when composing software, name equivalence allows programmers to better express their intentions. To combine these advantages, we introduce a light-weight construction to explore the middle ground between exclusive use of structure or name equivalence: compound types.

To begin with, let us analyze the respective advantages of name and structure equivalence in the context of our initial example of TextContainers introduced in Sect. 2.3. Name equivalence allows us to explicitly state that objects compatible with interface Text are supposed to adhere to the respective specification, in our example partially provided in the form of comments. A similar statement holds with respect to interface Container. Structure equivalence, on the other hand, would allow us to type the service defined in Fig. 6 more reasonably.

The behavioral specification of that service refers to the two specifications associated with the types Text and Container, not just to the union of the methods and fields defined by these types. In the service's implementation, this shows whenever the parameter into is used as if being of the (behavioral) type Text or Container.

We call a type that combines the behavioral specifications of several other types the *compound type* of these. In the following, we denote a compound type as a list of its constituent types in square brackets. In our example, the service's parameter into would be typed as [Text, Container].

Neither in a language based only on structure equivalence nor in one using only name equivalence such a type can be expressed. With structure equivalence more types than wanted would be compatible because of possible accidental, purely syntactical

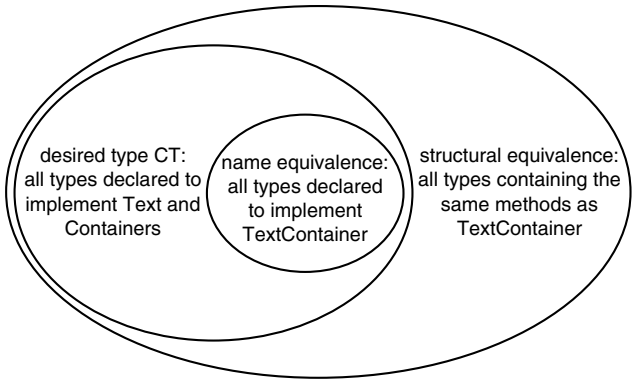


Fig. 10. Type compatibility with name equivalence, structure equivalence, and compound types

matches. With name equivalence, different types declared with the same constituent types remain incompatible. Fig. 10 visualizes the different sets defined when using name equivalence, structure equivalence, and compound types.

Compound types, composed from the same behavioral types can be treated as equal even with respect to behavioral specification. Any type subtyping both Text and Container, such as TextContainerA, must respect both semantical specifications at the same time. Consequently, it can be safely cast to either of its constituent types and therefore it is compatible with the corresponding compound type [Text, Container].

We conclude that type equivalence of compound types can and should be defined based on the structure of the composition. We thus speak of structure equivalence of compositions of name equivalent types. Compound types combine the best of two worlds.

Using compound types, we can solve our typing problem of the parameter into from Fig. 6. We give it the type [Text, Container]. Since both classes TextContainerA and ContainerTextB implement the two constituent interfaces Text and Container, instances of them can be passed as actual parameters to the library service. Variable into having all members of its constituent interfaces, the required method calls can be made without any additional casts or run-time validity tests. Thanks to the combination of structural and name equivalence, instances of other classes that just happen to declare methods with the same names and signatures as Text and Container, rather than implement the two interfaces, are rightfully rejected at compile time as values for parameter into. Figure 11 illustrates the subtype relationships, omitting transitive arrows.

Compound types also solve the other typing problem pointed out in Sect. 2.3 and illustrated in Fig. 8. Java's type system does not allow a programmer to declare a class that is assignment compatible with all subsets of extended, respectively implemented types, if the class implements more than one interface and does not have Object as its direct superclass. Consider now the case with compound types. Let class G extend class C and implement interfaces I and J. Instances of G can be assigned to variables of types C, I, J, [C, I], [C, J], [I, J], and [C, I, J].

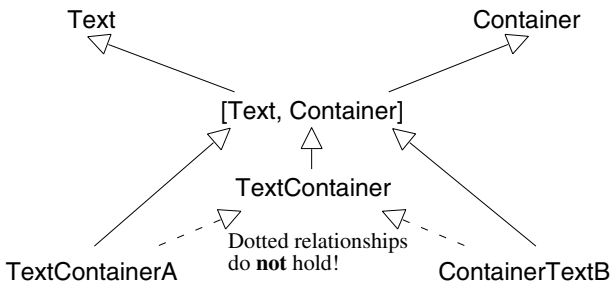


Fig. 11. Subtype relationship (transitive arrows omitted)

Compound types have underpinning in the theory of intersection types ([9, 33], see [30] for a recent overview). The intersection of two types S and T is the type of all elements belonging to both S and T. The new idea is to use defined types, which represent behavioral specifications, rather than the members of these types as atoms.

To mark this specific choice of atoms and to emphasize the intuition of combining specifications, rather than that of intersecting sets of possible values, we have decided to use the new name compound type.

5 Compound Types in Java

In this section we discuss a number of details showing how compound types are integrated into Java. We investigate the conditions for well-formedness and a number of interesting properties.

We define compound types in Java as anonymous reference types. A compound type is a direct extension of a set of interfaces and a non-final class, collectively referred to as constituent types. The members (methods, fields) of a compound type are the members of its constituent types with their respective accessibility. The compound type does not add any additional members, redefine any members, or hide any constants. If no constituent class type is explicitly given, Object is implicitly assumed acknowledging that any reference type can be converted to Object by assignment conversion.

Compound types can be used as parameter types, variable types, return types of methods, cast operators, and operands of the instanceof operator. They are not permitted in the **extends** or **implements** clauses of interface and class declarations.

A variable, the declared type of which is a compound type, may have as its value a reference to an instance of a class declared to extend the constituent class and implement the constituent interfaces, or the value of the variable may be null. In other words, the legal values of a variable, the declared type of which is a compound types, are those that could also be assigned to variables of all constituent types of the variable's type.

Compound types are written as comma separated lists delimited by square brackets. The order of constituent types is not relevant, e.g., [Text, Container] and [Container, Text] denote the same type.

As a guiding principle, a compound type [C, I1, I2, ..., In] is well-formed, if the abstract class definition **abstract class** D **extends** C **implements** I1, I2, ..., In {}, where D is a fresh name, would be acceptable in the same package. Thus, no two constituent types may define a method with the same name and signature but different return types.

If the rare and, therefore, comparatively minor problem of method clashes were solved in the Java base language, e.g., by qualified names, it would also automatically disappear for compound types.

Including more than one class type is pointless, as no compatible objects could ever be created in Java's single class inheritance system, unless the classes are in a subclass relationship. For simplicity and consistency, we do not allow more than one class to be included.

On the other hand, coherence with Java's design principles dictates that both an interface and one of its superinterfaces may be included. Assume that TrivText is a superinterface of Text. Then, instances of TextContainerA and ContainerTextB can also be assigned to variables of type [Text, Container, TrivText] as their classes indirectly also implement TrivText. However, [Text, Container] and [Text, Container, TrivText] do — due to an in our opinion unfortunate feature of Java — not denote the same type. In Java interfaces may shadow constants defined in their superinterfaces. Let TrivText define a constant **int** k = 21 and Text shadow it by defining **boolean** k = **true**. Then a.k denotes the boolean expression **true** for a of type [Text, Container], but is ambiguous for a of type [Text, Container, TrivText]. In the latter case, a qualification such as Text.a.k would be required. Including an interface that is already implemented by the constituent class is analogous. As in a class declaration, the same interface may not be included more than once in a compound type.

We have introduced compound types as anonymous types only. They could, however, also be given names for documentation purposes. Of course, partially structure equivalence as described above would also apply to the named variant. A named compound type would only be visible where all its constituting interfaces are visible.

The changes we propose to Java's language specification [12] can be found in [3, Appendix].

6 Emulating Compound Types on the Virtual Machine

Our proposed extension requires modifications of the Java compiler, but programs with compound types can be executed on an unchanged virtual machine [17]. The latter is significant because many of the security and portability properties of Java are tied to the virtual machine, as remarked by Agesen et al. [1].

There are —at least— two ways of emulating compound types. Both of them are hinted at in Sect. 2.3. One idea is to use one of the constituent types in place of the compound type and to employ explicit casts to access members of the other constituent types. Most of these casts require run-type validity checks. However, if the byte-code has been generated by a correct compiler all these casts will always succeed at run time, as they have already at compile time been proven correct.

These superfluous run-time tests are also needed when using current Java as only one of the constituent types can be asserted statically. Removing unnecessary tests automatically requires a flow analysis of the complete system. Already expensive for closed systems, this is entirely impossible for extensible systems that by definition are never complete.

Thus, the use of compound types on an existing virtual machine without any adaptation does not incur any performance penalty over a solution in current Java. Rather, if the virtual machine would be adapted to support compound types, a performance increase over the state-of-the-art would result.

The other way is to use multiple variables, one for each constituent type. These variables all contain the same value but have different types. The compiler takes care that the variables are changed in lockstep; a run-time check that all refer to the same object is not necessary. This solution also comes with some overhead in space and time compared to an adapted virtual machine, but it is as efficient as a solution in current Java.

The result of the instanceof expression E **instanceof** [C, L1, L2, ..., Ln] is true if the value of E is not **null** and the reference could be cast to [C, L1, L2, ..., Ln] without raising a ClassCastException. Let o be a fresh variable of type Object. Then

```
    E instanceof [C, L1, L2, ..., Ln]
≡   (o=E) instanceof C && o instanceof L1 && ...
    && o instanceof Ln
```

In spite of the emulation option on the existing virtual machine, compound types cannot be mapped to plain Java without loosing static safety on the language level.

7 Type Soundness

In this section, we report on a mechanically verified formal proof of type soundness of Java with compound types. Type soundness intuitively means that all values produced during any program execution respect their static types. An immediate corollary of type soundness is that method calls always execute a suitable method, that is, there are no 'method not understood' errors at run time. Type soundness is not a trivial property, especially for polymorphic languages [2, 4]. It came to prominence with the discovery of the failure of its application to older versions of Eiffel [8, 23].

Our proof of type soundness for compound types is based on the work of von Oheimb and Nipkow [36], a much extended version of [26], in which they have formalized and proved type soundness of a large subset of Java. They verified the proof mechanically with the theorem prover Isabelle/HOL [27].

To this formalization, we added compound types as reference types, appended the widening and casting relations with compound types, and defined the members of the latter. Finally, we adapted the proofs and ran them through Isabelle/HOL.[3] The definition of compound types adds 131 lines to the existing 1371 lines, approximately 10 %.

[3] At http://www.abo.fi/~mbuechi/publications/CompoundTypes.html the Isabelle theories are available.

Here, we present the extensions to the widening and casting relations, which are interesting in their own rights. A full report of all the mechanical details is beyond the scope of this paper.

The Java language specification introduces identity and irreflexive widening conversions separately. Since in all conversion contexts permitting widening identity conversions are possible as well, the two are merged in the formalization. The expression $\Gamma \vdash S \preceq T$ says that in program environment Γ objects of type S can be transformed to type T by identity or widening conversion. In particular, expressions of type S can be assigned to variables of type T and expressions of type S can be passed for formal parameters of type T. Widening can be understood as a syntactic, declared form of subtyping.[4] Unlike subtyping in most type theoretic frameworks, the Java language specification does not say that widening is transitive. Hence, transitivity is a proved property rather than an axiom.

We use the following naming conventions:

C, D	classes	M, L	sets of interfaces
I, J	interfaces	S, T	arbitrary types
R	reference type	Γ	program, environment

Likewise, $\Gamma \vdash C \prec_{\mathsf{C}} D$ expresses that C is a subclass of D, $\Gamma \vdash C \rightsquigarrow I$ that class C implements interface I, and $\Gamma \vdash I \prec_{\mathsf{I}} J$ that I is a subinterface of J. Furthermore, $\mathsf{is\_type}\ \Gamma\ T$ expresses that T is a legal type in Γ, $\mathsf{RefT}\ R$ denotes reference type R, and NT stands for the null type. With this, we can express the following two typing judgments, which are also applicable to compound types:

$$\frac{\mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash T \preceq T} \qquad \frac{\mathsf{is\_type}\ \Gamma\ (\mathsf{RefT}\ R)}{\Gamma \vdash \mathsf{NT} \preceq \mathsf{RefT}\ R}$$

Further $\mathsf{Class}\ C$ stands for the class type C, $\mathsf{Iface}\ I$ for the interface type I, and $\mathsf{T}[\cdot]$ for an array type with elements of type T. $\mathsf{Compound}\ (C, L)$ denotes the compound type with class C and interfaces $L_i \in L$. The discriminators $\mathsf{is\_class}\ \Gamma\ C$, $\mathsf{is\_iface}\ \Gamma\ I$, and $\mathsf{is\_compound}\ \Gamma\ (C, L)$ are also used. The latter is true, if the compound type is well formed, that is, all constituent types are accessible, C is not final, and there is no method name p such that two constituent types define a method named p with identical signature but different return types. We assume that $\mathsf{is\_class}\ \Gamma\ \mathsf{Object}$ and $\mathsf{is\_iface}\ \Gamma\ \mathsf{Cloneable}$ holds for all Γ. In Java, $\mathsf{Cloneable}$ is the only interface implemented by arrays. With this we can define the remaining widening rules involving compound types:

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D; \\ \mathsf{is\_compound}\ \Gamma\ (D, M);\ \forall J \in M.\, \Gamma \vdash C \rightsquigarrow J\end{array}}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Compound}\ (D, M)}$$

$$\frac{\begin{array}{c}\mathsf{is\_iface}\ \Gamma\ I; \\ \mathsf{is\_compound}\ \Gamma\ (\mathsf{Object}, M);\ \forall J \in M.\, \Gamma \vdash I \prec_{\mathsf{I}} J \vee I = J\end{array}}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Compound}\ (\mathsf{Object}, M)}$$

[4] For simplicity, the term 'subtyping' is used in the other sections of this paper in place of the formally correct notion of 'widening'.

$$\frac{\mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash \mathsf{T}[\cdot] \preceq \mathsf{Compound}\ (\mathsf{Object}, \{\})}$$

$$\frac{\mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash \mathsf{T}[\cdot] \preceq \mathsf{Compound}\ (\mathsf{Object}, \{\mathsf{Cloneable}\})}$$

$$\frac{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D;\ \mathsf{is\_compound}\ \Gamma\ (C, L)}{\Gamma \vdash \mathsf{Compound}\ (C, L) \preceq \mathsf{Class}\ D}$$

$$\frac{\mathsf{is\_compound}\ \Gamma\ (C, L);\ \Gamma \vdash C \rightsquigarrow J \vee (\exists I \in L.\Gamma \vdash I \prec_{\mathsf{i}} J \vee I = J)}{\Gamma \vdash \mathsf{Compound}\ (C, L) \preceq \mathsf{Iface}\ J}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D; \\ \mathsf{is\_compound}\ \Gamma\ (C, L);\ \mathsf{is\_compound}\ \Gamma\ (D, M); \\ \forall J \in M.\Gamma \vdash C \rightsquigarrow J \vee (\exists I \in L.\Gamma \vdash I \prec_{\mathsf{i}} J \vee I = J)\end{array}}{\Gamma \vdash \mathsf{Compound}\ (C, L) \preceq \mathsf{Compound}\ (D, M)}$$

The casting relation $\Gamma \vdash S \preceq_{\mathsf{?}} T$ states, that a cast from type S to type T is permissible at compile time, that is, the type cast '(T)e', where e is of type S, might succeed at run-time. If it can be proven to always fail, the compiler can already flag an error.

If $\Gamma \vdash S \preceq T$ holds, the cast can be proven to always succeed. Otherwise, a run-time validity test must be performed to check whether $\Gamma \vdash R \preceq T$ holds for the run-time type R of the cast operand. The following general casting conversions are applicable to compound types as well:

$$\frac{\Gamma \vdash S \preceq T}{\Gamma \vdash S \preceq_{\mathsf{?}} T} \qquad \frac{\Gamma \vdash \mathsf{RefT}\ S \preceq_{\mathsf{?}} \mathsf{RefT}\ T}{\Gamma \vdash (\mathsf{RefT\ S})[\cdot] \preceq_{\mathsf{?}} (\mathsf{RefT\ T})[\cdot]}$$

In the rules below, 'no_conflict $\Gamma\ (I, D, M)$' means that there is no method name p such that both I and D or one of the interfaces in M declare a method named p with the

$\Gamma \vdash S \preceq T$	S widens to ('is subtype of') T in Γ
$\Gamma \vdash C \prec_{\mathsf{c}} D$	C is a subclass of D in Γ
$\Gamma \vdash C \rightsquigarrow I$	C implements I in Γ
$\Gamma \vdash I \prec_{\mathsf{i}} J$	I is a subinterface of J in Γ
$\Gamma \vdash S \preceq_{\mathsf{?}} T$	cast from S to T permissible at compile time in Γ
'no_conflict $\Gamma\ (I, D, M)$'	in Γ interface I, class D, and constituent interfaces of M do not define a method with the same name and the same signature but different return types

Fig. 12. Summary of notation

same signature but different return types.[5] We use this abbreviation freely for different combinations of classes, interfaces, and sets of interfaces to indicate the absence of a method clash in place of the actual predicates, which are lengthy and technical.

$$\frac{\Gamma \vdash D \prec_{\mathsf{C}} C;\ \mathsf{is\_compound}\ \Gamma\ (D,M)}{\Gamma \vdash \mathsf{Class}\ C \preceq_? \mathsf{Compound}\ (D,M)}$$

$$\frac{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D;\ \mathsf{is\_compound}\ \Gamma\ (D,M);\ \neg(\mathsf{is\_final}\ \Gamma\ C);\ \text{`}\mathsf{no\_conflict}\ \Gamma\ (C,M)\text{'}}{\Gamma \vdash \mathsf{Class}\ C \preceq_? \mathsf{Compound}\ (D,M)}$$

$$\frac{\mathsf{is\_iface}\ \Gamma\ I;\ \mathsf{is\_compound}\ \Gamma\ (D,M);\ \text{`}\mathsf{no\_conflict}\ \Gamma\ (I,D,M)\text{'}}{\Gamma \vdash \mathsf{Iface}\ I \preceq_? \mathsf{Compound}\ (D,M)}$$

$$\frac{\Gamma \vdash D \prec_{\mathsf{C}} C;\ \mathsf{is\_compound}\ \Gamma\ (C,L);\ \neg(\mathsf{is\_final}\ \Gamma\ D) \vee (\forall I \in L.\Gamma \vdash D \rightsquigarrow I);\ \text{`}\mathsf{no\_conflict}\ \Gamma\ (D,L)\text{'}}{\Gamma \vdash \mathsf{Compound}\ (C,L) \preceq_? \mathsf{Class}\ D}$$

$$\frac{\mathsf{is\_compound}\ \Gamma\ (C,L);\ \mathsf{is\_iface}\ \Gamma\ J;\ \text{`}\mathsf{no\_conflict}\ \Gamma\ (C,L,J)\text{'}}{\Gamma \vdash \mathsf{Compound}\ (C,L) \preceq_? \mathsf{Iface}\ J}$$

$$\frac{\mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash \mathsf{Compound}\ (\mathsf{Object},\{\}) \preceq_? \mathsf{T}[\cdot]}$$

$$\frac{\mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash \mathsf{Compound}\ (\mathsf{Object},\{\mathsf{Cloneable}\}) \preceq_? \mathsf{T}[\cdot]}$$

$$\frac{\Gamma \vdash D \prec_{\mathsf{C}} C;\ \mathsf{is\_compound}\ \Gamma\ (C,L);\ \mathsf{is\_compound}\ \Gamma\ (D,M);\ \text{`}\mathsf{no\_conflict}\ \Gamma\ (C,L,D,M)\text{'}}{\Gamma \vdash \mathsf{Compound}\ (C,L) \preceq_? \mathsf{Compound}\ (D,M)}$$

$$\frac{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D;\ \mathsf{is\_compound}\ \Gamma\ (C,L);\ \mathsf{is\_compound}\ \Gamma\ (D,M);\ \text{`}\mathsf{no\_conflict}\ \Gamma\ (C,L,D,M)\text{'}}{\Gamma \vdash \mathsf{Compound}\ (C,L) \preceq_? \mathsf{Compound}\ (D,M)}$$

Whereas the widening rules can be considered as a particular instantiation of subtyping for intersection types, the rules for casts are believed to be new.

The currently by von Oheimb and Nipkow formalized subset of Java, on which we build, still does not capture all features. Of them final classes, modifiers, class variables, static methods, interface fields, and methods of the class Object would be relevant for compound types.

The main advantages of a mechanized over a paper-and-pencil proof are additional confidence and support for extensions. We would like to stress the second aspect. Not

[5] In what is believed to be an omission from the specification [28], Java checks at compile time only for clashes between methods contained in interfaces, but not for clashes between methods contained in classes and interfaces. Opting for maximum static detection of errors, casts involving compound types are defined to check for all kinds of clashes.

only did the formalization result in a soundness proof, but the proof tool also reminded us of what all needed to be defined about compound types before the desired properties could be established. Most proof scripts worked without modifications. The fact that all theorems were reproved mechanically for the extended language definition conveys more confidence than the typical adaptation of a paper-and-pencil proof with 'this-should-still-hold' handwaving.

8 Related Work

Our analysis leading to the observation that component software demands a combination of named, behavioral types and structure equivalence for compositions of those, was inspired by Microsoft's binary standard COM. To our knowledge, however, it has never been presented on the programming language level so far. Without this underpinning, some existing programming languages offer similar or related constructions. In this section we review in brief Microsoft's COM, the languages Objective-C, Sather, and Modula-3, the theory of intersection types, and —as a quite different technology— binary component adaptation.

Microsoft's COM: The principle idea of using structural type equivalence with named types as atomic building blocks, each presenting a behavioral contract, is very much inspired by Microsoft's Component Object Model (COM) [32, 6]. In COM, objects cannot be accessed directly but through interfaces only. These interfaces have globally unique identifiers (GUID) as names. It is the intention that with each interface also goes a behavioral specification, to be documented separately.

An object's type is defined as the set of the interfaces implemented by it. The COMEL language [14], built to formalize COM, consequently uses interface sets, similar to our compound types, to type objects. This compositional definition of an object's behavior is heavily used, for instance, by the ActiveX framework [6], which defines, for example, an ActiveX control container as any object implementing a specific set of interfaces.

Here the parallel ends, however. Clients of a COM object need to use a separate reference variable to each interface through which they want to interact with the object, because each interface may be implemented by a separate node and thus have a different address in memory. This is acceptable as memory layout under the hood and may be hidden by a proper programming language. We expect such a language to build heavily on compound-typed variables.

To determine whether an object is of a given type, queries must be issued for each interface being part of that type. This may seriously impact a system's performance, in particular, if an object is situated remotely. Therefore, distributed COM (DCOM) introduced a service to retrieve sets of interfaces.

Alternatively, categories could be used. Membership of classes in a certain category can describe, beyond other, that a specific set of interfaces is supported. In this sense, categories can be compared to explicitly declared subtypes. Only if a class makes an explicit reference, that is, registers as a category member, the information can be exploited.

Objective-C: Objective-C [25], an object-oriented extension of C, first introduced the dual class and interface hierarchies. Entities can be typed with a combination of a class type and one or more protocol types (Objective-C's name for interfaces), much like our compound types. Objective-C's type system is not sound; for example, the validity of casts is not checked at run time. Introducing and verifying compound types as part of a type-sound language, such as Java, still remained to be done.

Modula-3: Modula-3 [5] is another language which combines name equivalence and structure equivalence of types. This combination, however, is different than what we proposed. In Modula-3 structure equivalence is the default for all types, unless declared as branded, which makes them clearly distinguishable. Also, Modula-3 supports only single subtyping and thus compound types cannot contribute anything.

Sather: Sather [11], an object-oriented programming language featuring multiple subtyping and subclassing in separate hierarchies, allows the programmer to introduce types as supertypes of already existing classes. That way it offers two symmetric possibilities to introduce a subtype relationship: it can be declared with either the sub- or the supertype. Most other languages require a declaration with the subtype.

The compatibility problem described in Sect. 2.3 may be solved partially by that. Even if vendor C creates the library service after vendor A creates his TextContainer component, C can still declare the type of the parameter into (Fig. 6) in such a way that A's implementation becomes a supertype. This requires, of course, that C is aware of A's component.

Sather allows subtype relationships to be introduced in the source code by programmers of either type, but not by third parties, such a system assemblers, who only have access to the binary components. In our above example, the library service is still not

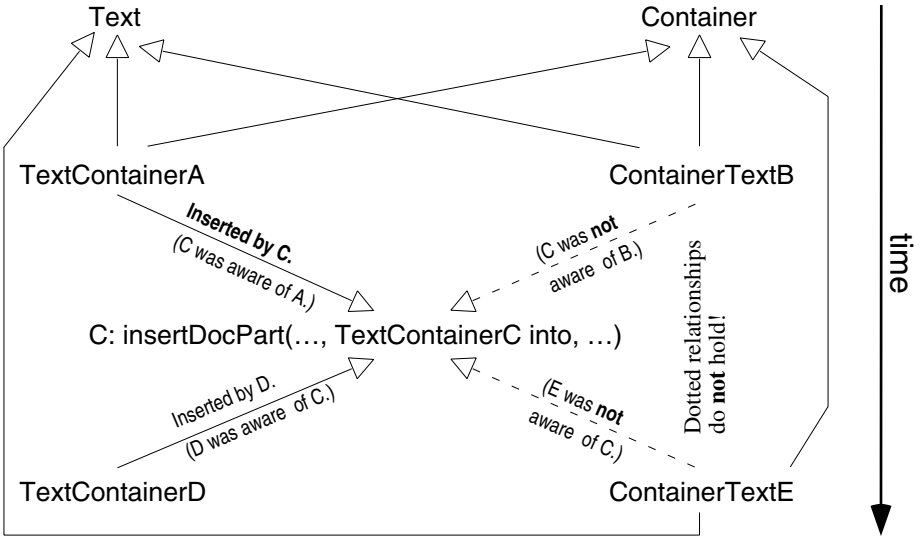


Fig. 13. Scenario in Sather where supertyping solves part of problem

compatible with vendor B's component, as C was not aware of B's implementation and did thus not explicitly declare it to be a subtype. Likewise, any components created after the library service, the manufacturers of which were not aware of the combined type introduced by C, are incompatible with the library service (Fig. 13). With the mutual unawareness postulate for a large component market, Sather's supertyping does, therefore, not solve the problem at hand.

Pure structure equivalence in Java: The use of pure structural type equivalence between classes and interfaces in Java to increase compatibility has been suggested by Läufer et al. [16]. In their suggestion, any instance of a class that provides an implementation for each method in an interface can be used where a value of the interface type is expected. Thus, classes declared to implement several interfaces directly, such as TextContainerA, are compatible with interfaces, such as TextContainer, combined of the base interfaces implemented by the class. However, also classes that by coincidence happen to contain methods with matching signatures but that are not meant to adhere to the associated semantics are assignment compatible. As explained in Sect. 3.2, pure structure equivalence ignores the modeling aspect of types resulting in too large a compatibility relation.

Using only structure equivalence to decide compatibility between classes and interfaces, as proposed by Läufer et al., it is not possible to express that a parameter must also subclass a certain class in addition to implementing some interfaces. As a case in point, structural conformance between classes and interfaces does not solve the problem of compatibility with all subsets of supertypes for the case of three or more types including a class other than Object (Fig. 8).

Furthermore, the proposal requires changes to the Java Virtual Machine, possibly introducing some security problems. In addition, the existing Java language is changed, rather than extended as by our compound types.

Intersection types: As pointed out in Sect. 3, intersection types with classes and interfaces as atoms are the theoretical foundation for our approach. Intersection types were introduced into the λ-calculus in the late 70's by Coppo and Dezani-Ciancaglini [9] and independently by Sallé [33]. The original motivation for introducing intersection types was the desire for a type-assignment system in which the typing of terms is invariant under β-expansion and in which every term with a normal form has a meaningful typing.

In the past twenty years, intersection types, infinite intersections, and the dual notion of union types have been studied extensively in type theory. Pierce and others have also studied the combination of intersection types with bounded polymorphism and other object-oriented concepts (see [30] for a summary of his thesis and an overview of recent work in the field). In contrast to our work, these studies all take the 'type' rather than the 'modeling' view. Thus, they use pure structure equivalence, not taking semantical soundness into account.

Forsythe [31], a descendant of Algol 60, is the only programming language that explicitly uses intersection types and that we are aware of. Forsythe is based on pure structure equivalence, rather than on a combination of name and structure equivalence as our approach. 'Objects' exist in the form of function records only, not allowing for co-variant specialization of the self parameter.

Binary component adaptation (BCA): BCA allows components to be adapted in binary form and during program loading [15]. BCA rewrites class files before or while they are loaded without requiring source code access. Thus, modifications described by delta files can be applied by third-parties. Adding an interface to the implements clause, one of the supported modifications, could be used to solve the compatibility problem described in Sect. 2.3: Vendor C, the creator of the library service, declares a combined interface, which is used to type the parameter into (Fig. 6). Even if vendors A and B have not declared their components to implement this interface, a component integrator can add it to the lists of implemented interfaces using BCA.

BCA adds further flexibility because it can be used to glue classes that are not based on common standard interfaces. Unfortunately, it also burdens the person assembling the system with the task of figuring out how to do this correctly. That is, the system assemblers need to understand the interfaces' semantics and program the adaptation. Plug-and-play with made-to-fit components, as enabled by compound types, is the more economical alternative wherever applicable. Furthermore, BCA makes systems harder to understand as delta files must also be taken into account.

BCA does not solve the problem of compatibility with all subsets of supertypes for the case of three or more types including a class other than Object (Fig. 8), because BCA does not add any new kind of types or modify any conversion rules.

9 Conclusions

We have exhibited a shortcoming of Java's current type system. In a programming language for extensible component software, substitutability of typed objects should neither be decided by the types' name nor just by the structural compatibility of signatures exclusively. Name equivalence, as offered by Java, is too restrictive when composing independently evolved standards or frameworks. Structure equivalence, on the other hand, does not support behavioral typing, that is, to associate semantical specifications with type names.

We concluded that one needs both. On the level of declared types, name equivalence is to be used. A behavioral contract can be associated with each type. When composing these types, however, we want separately declared compositions to be compatible if they have the same structure, that is if they consist of the same types.

To this end, we propose compound types as structurally matched compositions of named types, considered to match only if declared so.

We showed how to add compound types as anonymous compositions of named types to Java, an example of a practical, type-sound programming language. To a variable of a compound type one can assign any object with the same structure in terms of implemented interfaces and extended classes.

Java is well suited to host compound types. Building on multiple inheritance of interfaces, we integrated our proposal smoothly. The resulting language is a strict extension and thus backward compatible. Java programs with compound types can be executed on an unchanged virtual machine.

A mechanical soundness proof gives additional confidence in the well-definedness of the extended type system. The relative ease of adapting a formalization of the existing

Java language further illustrates the orthogonality of our proposal. The changes we propose to Java's language specification [12] can be found in [3, Appendix].

We believe that compound types can contribute to any typed language with multiple subtyping and name equivalence of types.

Acknowledgments David von Oheimb and Tobias Nipkow provided us with their formalization of Java and helped us with our extensions. We would like to thank Ralph Back, Dominik Gruntz, Cuno Pfister, and Clemens Szyperski for a number of fruitful discussions. The referees' helpful comments are also gratefully acknowledged.

References

1. Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Proceedings of OOPSLA '97*, pages 49–65. ACM Press, 1997.
2. Kim B. Bruce, Robert van Gent, and Angela Schuett. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proceedings of ECOOP '95*, pages 27–51. LNCS 952, Springer Verlag, 1995.
3. Martin Büchi and Wolfgang Weck. Java needs compound types. Technical Report 182, Turku Centre for Computer Science, 1998. http://www.tucs.fi/publications/techreports/TR182.html.
4. Luca Cardelli. Type systems. In *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997. http://www.luca.demon.co.uk/Papers.html.
5. Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordanand Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Research Report 52, Systems Research Center, Digital Equipment Corporation, Palo Alto, November 1989. http://www.research.digital.com/src/m3defn/html/.
6. David Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.
7. Netscape Communications. Netscape Plug-Ins, 1998. http://developer.netscape.com/docs/manuals/communicator/ plugin/index.htm.
8. William Cook. A proposal for making Eiffel type-safe. In *Proceedings of ECOOP '89*, pages 57–70. Cambridge University Press, 1989.
9. M. Coppo and M. Dezani-Ciancaglini. A new type assignment for λ-terms. *Archiv. Math. Logik*, 19:139–156, 1978.
10. Brad Cox. Planning the software industrial revolution. *Software Technologies of the 90's special issue of IEEE Software magazine*, November 1990.
11. B. Gomes, D. Stoutamire, B. Weisssman, and H. Klawitter. Sather 1.1 : Language essentials, 1998. http://www.icsi.berkeley.edu/~sather/Documentation/LanguageDescription/contents.html.
12. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.
13. Object Management Group. The common object request broker: Architecture and specification, 1997. Revision 2.0, formal document 97-02-25, http://www.omg.org.
14. Rosziati Ibrahim and Clemens Szyperski. The COMEL language. Technical Report FIT-TR-97-06, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, 1997. http://www.fit.qut.edu.au/TR/techreports/FIT-TR-97-06.ps.Z.
15. Ralph Keller and Urs Hölzle. Binary component adaptation. In *Proceedings of ECOOP '98*. LNCS, Springer Verlag, 1998. http://www.cs.ucsb.edu/oocsb/papers/ecoop98.html.

16. Konstantin Läufer, Gerald Baumgartner, and Vincent F. Russo. Safe structural conformance for Java. Technical Report CSD-TR-96-077, Department of Computer Science, Purdue University, 1996.
17. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.
18. Lingsoft. Orthografix: Finnish proofing tools for Microsoft Word, 1998. http://www.lingsoft.fi/.
19. Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
20. Boris Magnusson. Code reuse considered harmful. *Journal of Object-Oriented Programming*, 4(3):8, November 1991.
21. Michael Mattsson and Jan Bosch. Framework composition: Problems, causes and solutions. In *Proceedings of TOOLS USA 97*, 1997.
22. McIllroy. Mass-produced software components. In Peter Naur, Brian Randell, and J. N. Buxton, editors, *Software engineering: concepts and techniques: proceedings of the NATO conferences. The Conference on Software Engineering held in Garmisch, Germany, 7th to 11th October 1968*. Petrocelli/Charter, 1976.
23. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
24. Anna Mikhajlova and Emil Sekerinski. Class refinement and interface refinement in object-oriented programs. In *Proceedings of FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, pages 82–101. LNCS 1313, Springer Verlag, 1997.
25. NeXT Software, Inc. *Object-Oriented Programming and the Objective-C Language*. Addison-Wesley, 1993. http://developer.apple.com/techpubs/rhapsody/ObjectiveC/.
26. Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170. ACM Press, 1998.
27. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828, Springer Verlag, 1994. See also http://www.cl.cam.ac.uk/Research/HVG/isabelle.html.
28. Roly Perera and Peter Bertelsen. The unofficial Java spec report, 1997. http://www.ergnosis.com/jsr/.
29. Cuno Pfister. Component software: A case study using BlackBox components (online tutorial of the BlackBox Component Builder), 1997. http://www.oberon.ch.
30. Benjamin C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, April 1997.
31. John C. Reynolds. Design of the programming language Forsythe. In *Algol-like Languages*, volume 1, pages 173–234. Birkhäuser, 1997. Also available as CMU-CS-96-146, ftp://reports.adm.cs.cmu.edu/usr/anon/1996/CMU-CS-96-146.ps.gz.
32. Dale Rogerson. *Inside COM*. Microsoft Press, 1996. See also http://www.microsoft.com/com/.
33. P. Sallé. Une extension de la théory des types en λ-calcul. In *Proceedings of Automata, Languages and Programming*, pages 398–410. LNCS 61, Springer Verlag, 1978.
34. Sun Microsystems, Inc. Java Beans, 1997. http://splash.javasoft.com/beans/.
35. Clemens Szyperski. *Component Software : Beyond Object-oriented Programming*. Addison-Wesley, 1998.
36. David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*. LNCS, Springer Verlag, 1998, to appear.

Paper V

Generic Wrapping

Martin Büchi and Wolfgang Weck

Technical Report 317, Turku Centre for Computer Science, April, 2000. Shortened version: Generic Wrappers. In Proceedings of ECOOP 2000, *Lecture Notes in Computer Science*. Springer Verlag, June 2000.

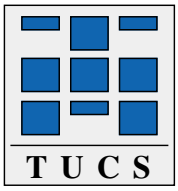
Generic Wrapping

Martin Büchi

Turku Centre for Computer Science
Lemminkäisenkatu 14A, FIN-20520 Turku
Martin.Buechi@abo.fi, http://www.abo.fi/~Martin.Buechi/

Wolfgang Weck

Oberon microsystems Inc.
Technoparkstrasse 1, CH-8005 Zürich
weck@oberon.ch, http://www.abo.fi/~Wolfgang.Weck/



Turku Centre for Computer Science
TUCS Technical Report No 317
April 2000

ISBN 952-12-0569-5
ISSN 1239-1891

Abstract

Component software means reuse and separate marketing of pre-manufactured binary components. This requires components from different vendors to be composed very late, possibly by end users at run time as in compound-document frameworks.

To this aim, we propose generic wrappers, a new language construct for strongly typed class-based languages. With generic wrappers, objects can be aggregated at run time. The aggregate belongs to a subtype of the actual type of the wrapped object. A lower bound for the type of the wrapped object is fixed at compile time. Generic wrappers are type safe and support modular reasoning.

This feature combination is required for true component software but is not achieved by known wrapping and combination techniques, such as the wrapper pattern or mix-ins.

We analyze the design space for generic wrappers, e.g. overriding, forwarding vs. delegation, and snappy binding of the wrapped object. As a proof of concept, we add generic wrappers to Java and report on a mechanized type soundness proof of the latter.

Keywords: Component software, late composition, type systems, language design, generic wrappers, mix-ins, Java

TUCS Research Group
Programming Methodology Research Group

1 Introduction

Component software enables the development of different parts of large software systems by separate teams, the replacement of individual software parts that evolve at different speeds without changing or reanalyzing other parts, and the marketing of independently developed building blocks. Components are binary units of independent production, acquisition, and deployment [53].

Component technology aims for late composition, possibly by the end user. Compound documents, e.g. a Word document with an embedded Excel spreadsheet and a Quicktime movie, as well as Web browser plug-ins and applets are examples of this. Late composition is a major difference between modern components and traditional subroutine libraries, such as Fortran numerical packages, which are statically linked by the developer.

Flexible late composition is one goal, prevention of unsafe compositions, such as adding scroll bars to a file descriptor, leading to 'method not understood' errors and possible system malfunction, is the other. Type systems can help to prevent this kind of run-time errors by prohibiting unsafe compositions. However, static type systems in class-based languages like Java [21], C++ [48], and Eiffel [33] tend to promote inflexible composition mechanisms, such as inheritance, which is fixed at compile time for a whole class.

Untyped prototype-based languages such as Self [54] are more flexible. Here, inheritance relationships can be decided at run time on a per-object base. The price of this flexibility is the lack of certain compile-time and as-early-as-possible run-time error detection: Assignments that may later on cause 'method not understood' errors don't cause any errors at compile time or at the time of their execution. Rather, the errors occur much later when the method is called. Flagging an error at compile time is preferable because it happens in presence of the programmer. Run-time errors, on the other hand, might occur at the clients' sites. Even in this case, trapping as close as possible to the place, where things started to go wrong, greatly facilitates debugging. Furthermore, component-wise (modular) reasoning, another requirement for independently developed components [53], is practically impossible in very flexible prototype-based languages.

In this paper we present an inflexibility problem in class-based languages and propose a new solution that partly borrows from prototype-based languages yet retains the possibility for maximal static and as-early-as-possible run-time error detection and modular reasoning.

Late composition is most pressing for items defined by different components, which may themselves be combined by an independent assembler or even by the user at run-time. Component standards such as Microsoft's COM [45], JavaBeans [50], and CORBA Components [40] are on the binary level. Components can be created in any language for which a mapping to the binary standards exists. However, binary standards are most easily programmed to in languages that support the same composition mechanisms. Furthermore, only direct language-level support can provide the desired machine checkable safety using types. Hence, composition

mechanisms in programming languages are relevant, even though components are binary units.

The mechanism suggested in this paper is partly inspired by COM's aggregation, but it doesn't yet have an exact equivalent in any of the aforementioned binary component standards.

Overview Section 2 illustrates with examples a problem of existing composition mechanisms and defines the requirements for a better solution. In Sect. 3, we show why existing technology does not sufficiently address these requirements. We introduce generic wrappers as a solution to the aforementioned problems in Sect. 4. Next, we discuss the design space for generic wrappers in Sect. 5 and the interplay with other type mechanisms in Sect. 6. As a proof of concept we add generic wrapping to Java in Sect. 7 and report on a mechanized type soundness proof of the extended language in Sect. 8. Section 9 introduces reflective mix-ins as an alternative to generic wrappers. Finally, Sect. 10 points to related work and Sect. 11 draws the conclusions.

2 The Problem

In this section, we describe some applications that cannot be satisfactorily realized with existing composition mechanisms. We also introduce some terminology, and distill a set of requirements.

2.1 Examples

We consider a problem in the realm of compound documents and then show that the same difficulties arise in many other domains.

Embedded views in compound documents for on-screen viewing, such as an Excel spreadsheet in a Word document, may be so large that they require their own scroll bars. Likewise, the user may want to add borders or identification tags to embedded views. It is even possible, that a user wants several such decorators added to the same embedded view.

There may exist different scroll bars from different vendors, which don't know all the other decorator or embedded view vendors. Decorators are typical examples of third-party components that users want to select to meet their specific needs. One user may want proportional scroll bars, another may like blinking borders to draw the boss' attention to the excellent sales figures, and still another may require immutable 128-bit identification tags.

In a compound-document framework similar to Java Swing or Microsoft OLE, let IView be the interface implemented by all classes whose instances can be displayed on screen and inserted into containers. Typical examples of classes implementing IView are TextView, GraphicsView, SpreadsheetView, and ButtonView.

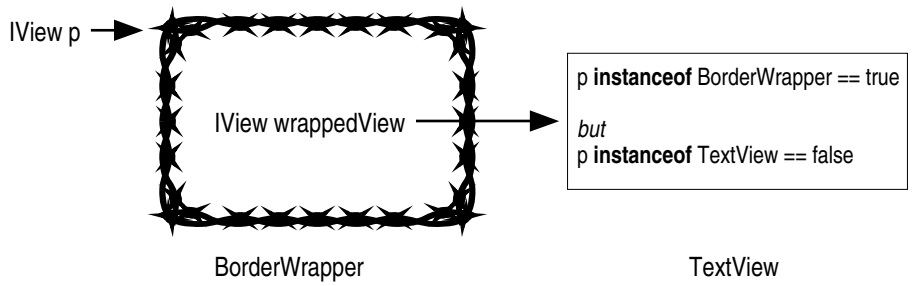


Figure 1: The wrapper is not fully transparent to clients of the embedded view

One way to implement decorators is with wrappers [20, Decorator Pattern]. A border wrapper is itself a view, that is it implements the IView interface. Hence it can itself be inserted into a compound document container. Furthermore the wrapper contains a reference of type IView to a wrapped view, which in a specific instance may be a TextView. The wrapper forwards most requests to the wrapped view, possibly after performing additional operations such as drawing the border.

Unfortunately, this approach has a serious disadvantage. If we wrap a border around a TextView, then the aggregate is only a BorderWrapper, but not a TextView with all of the latter's methods (Fig. 1). Hence, a spell check operation on all embedded text views in a document will not recognize a bordered TextView as containing text, unless it knows how to search inside wrappers from different manufacturers.

A standard interface, like IViewWrapper to be implemented by all view wrappers could ease the problem of searching inside different wrappers:

```
interface IViewWrapper {
  IView getWrappee();
}
```

However, instead of a simple type test, the spell checker would have to loop through all the wrappers:

```
IView q=p;
while(!(q instanceof TextView) && q instanceof IViewWrapper) {
  q=((IViewWrapper)q).getWrappee();
}
if(q instanceof TextView) {...;}
```

This solution is cumbersome for several reasons: First, it requires 5 lines of code instead of a simple type test. Second, it only works if there is a unique standard for wrappers, such as IViewWrapper. Third, it doesn't let the wrapper maintain invariants ranging over both itself and the wrapped object because clients have direct access to the latter.

To work with any type of object, this approach would require a wrapper interface to be defined for any reference type, instance of which might possibly be wrapped. Still the spell checker should be able to locate a wrapped TextView with the above code, whether the wrapper implements IViewWrapper or ITextViewWrapper. Hence, ITextViewWrapper must be a subtype of IViewWrapper, which is only the case in languages that allow covariant specialization of method return types (i.e. of getWrappee) in subclasses. Using a single wrapper interface that defines the return type of getWrappee to be Object would result in a loss of static type information. Parametric polymorphism with covariant subtyping and run-time type information solves this problem, but doesn't address the above three shortcomings.

Support for certain common kinds of wrappers may also be built into the wrapped objects. For example, JComponent, the correspondence to our IView in Java Swing, supports borders as insets. However, identification tags and other kinds of wrapper that were not previewed by the Swing designers are left out.

As a second example, let us consider a forms container that requires all its embedded views to implement the interface IControl. Assume that ButtonView implements IControl and that BorderWrapper doesn't. Hence, a bordered ButtonView cannot be inserted into a forms container: The type system rightfully prevents us from passing a BorderWrapper wrapping a ButtonView as the first parameter to the method insert(IControl c, Point pos). Otherwise a 'method not understood' error could occur when the container tries to call one of the methods declared in IControl. Passing just the wrapped ButtonView as parameter to insert is not a solution, because we would loose the border. The only workaround is to change the type of the first parameter of insert to IView and test that the actual parameter implements IControl or wraps an object that does so. Furthermore, we then either have to store two references per embedded view, one to the outermost wrapper and one to the object implementing IControl, or have the container loop through all the wrappers each time it wants to call a wrapped view's method declared in IControl.

Examples of wrappers in different applications are abundant. Documented cases include window decorators [20], the Microsoft AFC wrapper for AWT components [14], the view wrapper ComponentView for inserting AWT components into Java Swing texts [49], a physical access control system that adds surveillance with wrappers [22], a wrapper that adds additional relations [1], and the stream decorators in the Java library [49]. Many of these applications could be generalized and additional problems could be tackled with wrappers if the problems described above would be solved.

2.2 Terminology

We use the following terminology: A wrapped object is called a *wrappee*. A wrapper and a wrappee together are referred to as an *aggregate*. The declared type of a variable is referred to as *static* (compile-time) *type*. The type of the actually referenced object is called the variable's *dynamic* (actual, run-time) *type*. Likewise, we distinguish between the *static* (declared, compile-time) and the *dynamic* (actual,

run-time) *wrappee type*. For example, for an instance of BorderWrapper, declared to wrap an IView, and actually wrapping a TextView, the static wrappee type is IView and the dynamic wrappee type is TextView.

In discussions, we use the notation C.m to refer to the implementation of instance method m in class C. The subtype relation is taken to be reflexive; e.g., TextView is a subtype of itself.

Except where otherwise stated, the discussion in the first 6 sections applies to most strongly-typed class-based languages such as Java, Eiffel, and C++. For simplicity, we use Java terminology throughout the paper. A Java interface corresponds to a fully abstract class in Eiffel and C++.

2.3 Requirements

From the above examples we can distill a number of requirements for a wrapping mechanism. Numbers in parentheses refer to the summary of requirements in Fig. 2.

The user wants to select which border to wrap around which view. At compile time, the implementor of BorderWrapper doesn't know whether an instance of her class will wrap a TextView, a GraphicsView, or any other view that might even be only implemented in the future. Thus, the actual type and instance of the wrappee must be decidable at run time (1). Furthermore, wrappers must be applicable to any subtype of the static wrappee type (2). In this paper we only consider wrapping of specific instances (selective wrapping), but not adaption of all instances of a given class (global wrapping).

An aggregate of a BorderWrapper wrapping a ButtonView should be insertable into a controls container, even though only the wrappee implements the required interface IControl. Therefore, an aggregate should be an element of the wrapper and the actual wrappee type (3). This also implies that all methods of the wrappee can be called by clients and that they can make these calls directly on a reference to the wrapper.

Upon calling paint on an aggregate of a BorderWrapper and a TextView, the border's paint method should be executed. The latter first draws the border and then calls the paint method of the wrapped view with an adapted graphics context. Thus, wrappers must be able to override methods of the wrappee (4).

If clients can have direct references to the wrappee, they can call overridden methods. For example, a client could call the paint method of the embedded view with the graphics context (dimensions) of the whole aggregate. Thence, a wrapper should be able to control whether clients can directly access the wrappee (5).

A wrapper may depend on the wrappee's state being in a certain relationship to its own, as expressed by an invariant ranging over both state spaces. By overriding methods of the wrappee that could be used to invalidate this invariant and not granting clients direct access to the wrappee, this invariant can be partly protected. However, the actual wrappee type may always provide additional methods that

1. *Run-time applicability.* The actual type and instance of the wrappee must be decidable at run time.

2. *Genericity.* Wrappers must be applicable to any subtype of the static wrappee type.

3. *Transparency.* An aggregate should be an element of the wrapper and the actual wrappee type.

4. *Overriding.* Wrappers must be able to override methods of the wrappee.

5. *Shielding.* A wrapper should be able to control whether clients can directly access the wrappee.

6. *Safety.* The type system should prevent as many run-time errors as possible statically and signal errors as early as possible at run time.

7. *Modular reasoning.* Modular reasoning should be possible in the presence of wrapping.

Figure 2: Requirements for a Wrapping Mechanism

may be used to invalidate the invariant. Requirement 3 states that these methods are accessible to clients.

Early detection of errors and the possibility for modular reasoning have already been identified as general requirements for component-oriented programming. We state them here as explicit requirements (6 and 7) for the purpose of assessing composition mechanisms.

We may want to put both scroll bars and a border around a text view. Hence, multiple wrapping should be supported. We refrain, however, from explicitly listing this as one of our requirements, because it is satisfied by all surveyed mechanisms.

Finally, it is desirable that classes are not required to follow any coding standards for their instances to be wrappable. Otherwise, instances of classes programmed to different standards and of legacy classes are left out. Since certain coding standards can be established, as shown by JavaBeans, and since certain automatic rewriting —even of binary code— is possible, we consider this as a nice-to-have feature, but do not make it a formal requirement.

3 Why Existing Technology Is Insufficient

In this section we show why existing technology fails to address the above requirements.

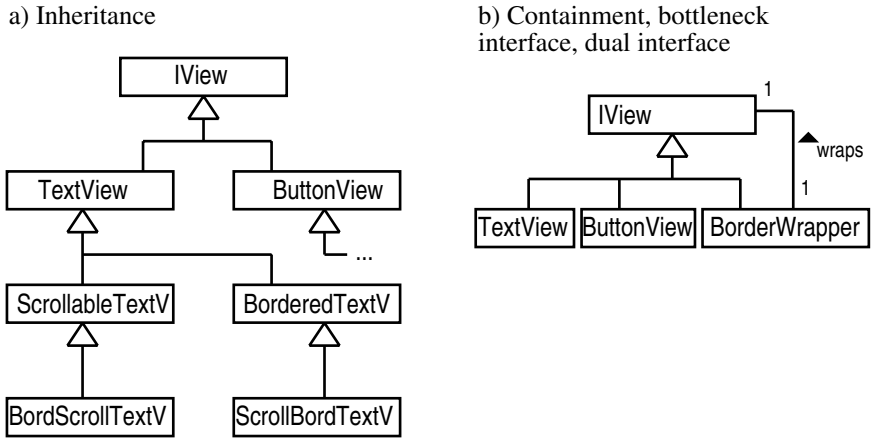


Figure 3: Solution Attempts in Class-Based Languages

3.1 Single-object solutions

Inheritance Feature combination by multiple inheritance produces specialized combination classes, such as BorderedTextView and BorderedGraphicsView. Thus, it combines the functionality of the wrapper and the wrappee into a single object. However, combinations can only be made at compile time by a vendor having access to both the border and the view. Run-time feature composition in interface builders or in compound documents is impossible with inheritance. Hence, this approach fails requirement (1). The modular reasoning requirement (7) is not fully satisfied because of the close coupling between super- and subclass, leading to the semantic fragile base class problem [36]. Furthermore, inheritance suffers from the combinatorial explosion problem, as illustrated by Fig. 3 a.

Mix-ins Parametric/bounded polymorphism, where the type parameter can serve as a supertype of the parameterized type, gives a special form of inheritance. Multiple combinations of wrapper and wrappee types can be created without textual code duplication. With this kind of *mix-ins*[1] [1] we could define the generic border type

class ParBorderedView<Wrappee **implements** IView> **extends** Wrappee {...}

and could derive the class ParBorderedView<TextView> of bordered text views and the class ParBorderedView<ButtonView> of bordered button views. However, also with mix-ins all combinations must be made at compile time. Hence, they fail like normal inheritance the requirement of run-time applicability (1).

[1] Support for mix-ins is rare. Examples include C++ templates, which delay most checking to the derivation of classes, Jigsaw [5], and three extension proposals for Java [1, 18, 3].

3.2 Two-object solutions

Containment The containment approach, also known as the decorator pattern [20], has already been sketched along with the presentation of the example in Sect. 2.1. It is illustrated by Fig. 3 b. The wrapper itself subtypes the static wrappee type and contains a field with a reference to the wrappee. As stated above, this approach fails the transparency requirement (3). Clients can only use the extended functionality of the wrappee by directly accessing it. Thus, the implementor has to choose between the shielding requirement (5) and making the full functionality of the wrappee available to clients.

An additional problem surfaces if the static wrappee type is a class with public fields. In this case, we end up with two unsynchronized copies of these fields in the wrapper and the wrappee.

Specialized wrappers Instead of creating a single BorderWrapper, we could define specialized border wrappers with matching static wrappee types for any kind of view such as TextViewBorder and GraphicsViewBorder. However, this solution attempt fails the run-time requirement (1) for the type: If the border vendor is not aware of SpreadsheetView, there will not be a matching border. Even in a closed system this approach suffers from a combinatorial explosion of classes like the inheritance approach.

Bottleneck interface In the bottleneck approach, the wrapper implements the declared wrappee type and holds a private reference to the wrappee (Fig. 3 b). The difference to the containment approach is that the wrappee only contains a single public method, the message handler, with a parameter containing the instructions what should actually be done. The wrapper also has a message handler method. The latter forwards any message that it doesn't understand itself to the wrappee. This approach does not make good use of the static type system. Fewer errors can be detected at compile time. Run-time type tests cannot be used to determine which messages are understood. Furthermore, callers have to be prepared to handle pseudo method-not-understood return values from the message handler. Thus, errors are not restricted to type casts. In summary, this approach makes the full functionality of the wrappee available to clients, but fails the transparency (3) as well as the safety requirement (6). It requires adherence to special coding standards, but bottleneck interfaces could be generated automatically.

Dual interfaces The containment and the bottleneck interface approach can be combined to get dual interfaces. Wrappees have in addition to their normal public methods a message handler through which all normal methods can be called (Fig. 3 b). Hence, methods of the static wrappee type can be called directly in a type-checked manner and additional functionality of the dynamic wrappee type through the message handler. Dual interfaces fare slightly better than bottleneck interfaces with respect to safety, but otherwise have the same drawbacks.

Technology \ Requirement	Run-time applicability (1)	Genericity (2)	Transparency (3)	Overriding (4)	Shielding (5)	Safety (6)	Modular reasoning (7)
Inheritance		(b)	✔	✔	n/a	✔	(c)
Parameterized mix-ins		(b)	✔	✔	n/a	✔	(c)
Containment	✔	(b)		(d)	(d)	(e)	✔
Specialized wrappers[(a)]	(f)	(b)	✔	✔	✔	✔	✔
Bottleneck interface	✔	✔		✔	✔		
Dual interface	✔	(b)		✔	✔	(e)	
Delegation in prototype-based languages	✔	✔	n/a	✔	✔		

(a) If only used with specific type, otherwise like containment.
(b) Yes, but with exceptions due to signature clashes.
(c) Limited due to tight coupling.
(d) Either full functionality availability or overriding and shielding.
(e) Type safety for static wrappee type.
(f) Type determined at compile time.

Figure 4: Properties of Existing Technologies

Delegation in prototype-based languages Prototype-based languages, such as Self [54], use a parent object to which the receiving object delegates messages that it does not understand itself. A bordered text view could be implemented by a border object with a text view parent. Due to the lack of (static) typing and because of the possibility to reassign the parent object, prototype-based languages fail the requirements of safety (6) and modular reasoning (7) [20].

3.3 Summary

We conclude that none of the existing technologies gives a satisfactory solution to the problem at hand. Figure 4 summarizes the results. Further language specific and binary level solution approaches are described in Sect. 10.

4 Generic Wrappers

To solve the problem stated in Sect. 2, we introduce generic wrappers. Generic wrappers are classes that are declared to wrap instances of a given reference type (class, interface) or of a subtype thereof. Like an extends clause to specify a superclass, we use a wraps clause to state the static wrappee type. This also declares the wrapper class to be a subtype of the static wrappee type. For example, the declaration

class LabelWrapper **wraps** IView {...}

states that each instance of the class LabelWrapper wraps an instance of a class that implements IView. The declaration makes class LabelWrapper a subtype of IView. Thus, instances of LabelWrapper can be assigned to variables of type IView

and LabelWrapper has all public members (methods, fields) of IView. Forwarding/delegation of calls to the methods of IView is implicit, that is there is no need to write explicit stubs.

To assure that this subtyping relationship always holds (and thereby that forwarding of calls never fails) must instances of LabelWrapper always wrap an instance of a subtype of IView —already during the execution of constructors. Hence, the wrappee must be passed as a special argument (in our syntax delimited by <>) to class instance creation expressions:

```
TextView t = ...; IView v = new LabelWrapper<t>(...);
```

The compiler checks that the declared type of variable t is a subtype of the static wrappee type. The wrapper class instance creation expression throws an exception if the value of t is null or if t were an expression and its evaluation throws an exception. In both cases, no wrapper object is created and the value of v remains unchanged.

The particularity of generic wrappers is that their instances are not only of the static, but also of the actual wrappee type. For example, a LabelWrapper wrapping a TextView is also of the latter type and not just of type IView. Hence, such an aggregate can be assigned to a variable of type TextView and the latter's methods can be called on it. In the following program fragment, which is based on the definition of LabelWrapper above, the type test returns true and the cast succeeds:

```
IView v = new LabelWrapper<new TextView()>(...);
TextView t2; if(v instanceof TextView) {t2=(TextView)v;}
```

The adjective 'generic' in generic wrapper stands for the reuse of parameterizable abstractions, which we have added to the plain wrapper pattern.

Methods declared in the wrapper override those in the wrappee analogously to overriding in subclasses.

In constructors and instance methods of generic wrapper classes, the keyword wrappee references the wrappee. It can be treated like an implicitly declared and initialized final instance field with some restrictions on use, as discussed below. Hence, wrappers can call overridden methods of the wrappee using the keyword wrappee corresponding to super for overridden methods of superclasses. For example, the paint method of BorderWrapper might look as follows:

```
public void paint(Graphics g) {
  ...; // paint border
  wrappee.paint(g1); // paint wrapped view with adapted graphics context
}
```

A wrapper that is aware of certain subtypes of the static wrappee type, can also use the keyword wrappee in type tests. For example, a wrapper that displays the length of the text in the wrapped view, if the latter is a TextView, might contain the following code fragment:

```
if(wrappee instanceof TextView) {int len=((TextView)wrappee).textLength(); ...;}
```

Preliminary evaluation Although this basic definition still leaves many aspect open, we can evaluate which requirements (Fig. 2) it fulfills independently of how the details are fixed. The actual type and instance of the wrappee can be decided upon at run time. Hence, requirement (1) is satisfied.

Wrappers are applicable to a all of the static wrappee type's subtypes, for which no unsound overriding would occur. An example of the latter might be that a method with signature m() and return type void would be overridden by one with the same signature, but a different return type, as discussed below. Thus, the genericity requirement (2) is mostly fulfilled.

As defined above, instances of generic wrappers are members of the actual wrappee type. Therefore, the transparency requirement (3) is satisfied. Note that none of the surveyed existing mechanisms satisfied both the run-time applicability and the transparency requirement (Fig. 4).

The fulfillment of the shielding (5) and modular reasoning (7) requirements cannot be judged without fixing more details.

The compiler ensures that an aggregate is always of the static wrappee type and, thereby, that all calls to methods of the static wrappee type will succeed. Run-time tests can be used to check whether the aggregate is of a certain type. Only insufficiently guarded casts may fail. Calls to methods of the actual wrappee type always find a matching method. Hence, the type system fulfills the safety requirement (6) by preventing as many run-time errors as possible statically and signaling errors as early as possible at run time.

5 Design Space for Generic Wrappers

The basic definition of generic wrappers in the previous section leaves many aspects open. In this section, we investigate the design space for generic wrappers.

The time of binding has a major influence on the design space of generic wrappers as compared to inheritance. With inheritance, the superclass is bound at compile time. With generic wrappers the actual type and instance of the wrappee first become known at wrap time, that is, run time. Later binding brings flexibility, but means that certain compatibility checks asserting type soundness and, thereby, the success of all method lookups have to be delayed (Fig. 5). A notable feature of generic wrappers is that an existing wrapper object can be wrapped again. Thus, it remains always possible to add new functionality to an aggregate.

Dynamic linking partly blurs this distinction. The name of the superclass is fixed at compile time, but the actual version and, therefore, the members and their semantics are not known until load time. For example in Java, the loading of a class may be delayed until an instance thereof is created. In this case, the compatibility with the used superclass is checked as late as the compatibility between a wrapper and the actual wrappee type. In conclusion, dynamic linking postpones compatibility checking to run time without fully exploiting the flexibility thereof.

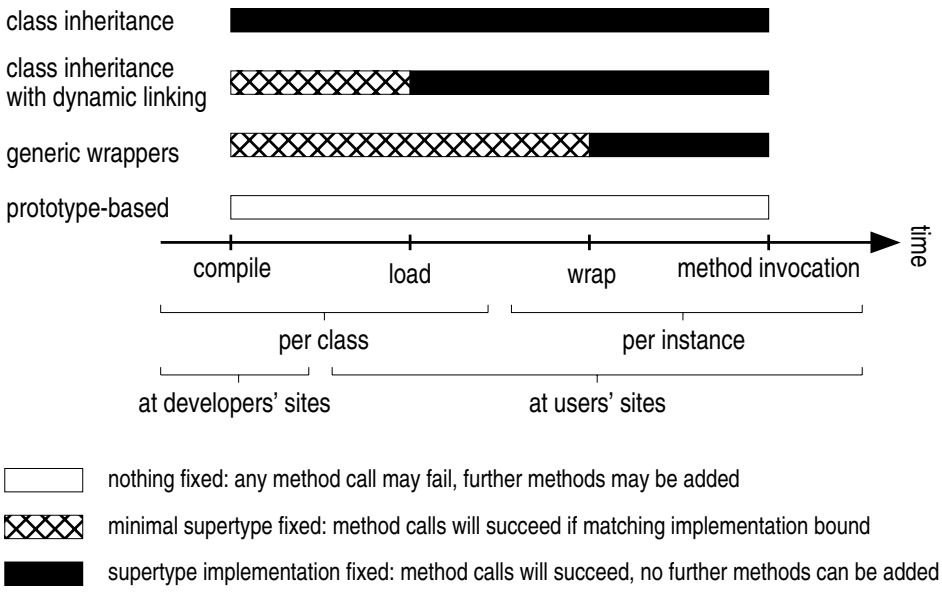


Figure 5: What Is Asserted to Hold from Where on?

5.1 Overriding of instance methods

Overriding of instance methods in subclasses is governed by certain rules to guarantee both type and semantic soundness. The same rules extend to overriding of methods of the wrappee by methods of the wrapper. For example to guarantee type soundness in Java, the overridden method must not be final, the return type of the overriding method must be the same as that of the overridden method, the overriding method must be at least as accessible, the overriding method may not allow additional types of exceptions to be thrown, and an instance method may not override a class method. To also guarantee semantic soundness, the overriding method must be a behavioral refinement of the overridden method [32].

Although the actual type of the wrappee isn't known until wrap time, we can perform certain checks at compile time. We can check that overriding of methods of the static wrappee type by methods of the wrapper respect the above rules. Any violation of the type rules would necessarily also lead to a violation in combination with any actual wrappee type, i.e., a subtype of the static wrappee type.

Because the actual wrappee may have more methods than the static wrappee type, overriding conflicts may nevertheless occur at wrap time, i.e., when the combination of the wrapper and the wrappee first becomes visible. In Fig. 6, three overriding conflicts occur when wrapping an instance of A in an AWrapper. The methods A.m an A.o would be overridden by semantically incompatible ones and AWrapper.n cannot override A.n because they have different return types.

Below we discuss two approaches to this problem. The first checks type soundness at wrap time and throws an exception if wrapping would be type unsound. To

```
interface IA {
  int m(); // return 0 or 1
}

class AWrapper wraps IA {
  public int m() {return 0;};
  public void n() {...};
  public int o() {return 0;};
}
```

```
class A implements IA {
  public int m() {return 1};
  public int n() {...};
  public int o() {return 1};
}

IA a=new A();
AWrapper w=new AWrapper<a>();
```

Figure 6: Overriding Example

decrease the probability of unsound overriding, we suggest a number of coding conventions. The second approach avoids wrap-time type problems by relying on a different form of method lookup and subsumption. We conclude with a short refutation of static approaches.

In this section we assume that there are no final classes and no method header specialization in subtypes (overriding non-final with final methods, overriding with restricted exception throws clauses and higher accessibility, as well as covariant return type and contravariant parameter type specialization) in our language. The interaction of final classes and method header specialization with generic wrappers is discussed in Sect. 6.2.

5.1.1 Wrap-time tests and coding conventions

At wrap time, we can automatically check whether overriding of methods of the actual wrappee by the wrapper is type sound. If this is the case, we can create the wrapper instance. Otherwise, we throw an exception. Wrap-time tests require enough information in the binary code. Java byte code, for example, satisfies this requirement.

Wrap-time exceptions are undesirable, yet they are preferable over unsuccessful method lookup as in prototype based languages like Self. First, if components are combined by an assembler, she can much more easily check all combinations than all method calls on all combinations. Second, if an error occurs, detecting it as early as possible facilitates debugging, as expressed by requirement (6).

To reduce the probability of wrap-time conflicts, we could use laxer rules for wrap-time overriding. For example, Java's binary compatibility prescribes laxer rules for the load-time compatibility checks between a class and its used superclass. In analogy, we could, e.g., allow overriding of a method of the wrappee by a method of the wrapper with an incompatible exception throws clause. Binary compatibility is a last resort for coping with changes to a superclass, fixed at compile time. On the other hand, generic wrappers promote the use of subtypes of the static wrappee type. Furthermore, laxer typing rules threaten semantic soundness, which

must be the ultimate goal. Hence, we believe that the strict rules should be used for generic wrappers at wrap time also.

We suggest to adhere to the following four coding conventions, which can greatly reduce the possibility of both type and semantic conflicts at wrap time:

(a) Classes only define (non private) methods declared in implemented interfaces.

(b) No two interfaces, not related by extension, declare methods with the same signature.

(c) Interfaces have semantic specifications and methods in classes are semantic refinements of their correspondences in the implemented interfaces.

(d) Method calls are only made on variables of interface, but not class types.[2]

We analyze the conventions for method o of Fig. 6. Convention (a) implies that both AWrapper and A implement interfaces declaring a method o. Furthermore, (b) dictates that this must be the same interface, say IO. The idea of class refinement [37], and the related notion of behavioral subtyping [2, 32], is that interfaces have semantic specifications and that methods in subtypes are behavioral refinements of the corresponding methods in their supertype. Assuming that both AWrapper.o and A.o are refinements of IO.o, we can deduce that both 0 and 1 are correct return values. Finally, condition (d) implies that a call x.o() may only be written for x of static type IO. In this case, the value 0 returned by the overriding method AWrapper.m meets our expectations.

If (a) or (b) is not adhered to, then a type conflict may occur as illustrated by method n of Fig. 6. If (c) is not adhered to, it could be that IO.o specifies the return value to be 1, which would not hold in the above case. Finally, if (d) is not respected, we could make a call x.o on a variable of type A. If x contained a reference to an AWrapper wrapping an A, we would get a return value of 0 although we expected 1.

Conventions (a) and (d) could easily be enforced by a programming language. Instead of (b) a language can require qualified notation for member access instead of merging namespaces of interfaces. Convention (c) requires semantic proofs and is, therefore, more difficult to check. These conventions also avoid semantic problems in the overriding in subclasses. Hence, they are implicitly advocated as good style for object-oriented programming [20, 53, 15] and correspond to Microsoft COM's rules/guidelines for the binary level.

In conclusion, wrap-time checking allows us to avoid type unsound overriding. Furthermore, adherence to some also otherwise beneficial coding conventions can greatly reduce the possibility of type or semantic unsound overriding.

[2] Self calls, which are of course also allowed, are discussed in Sect. 5.3.

5.1.2 An alternative form of method lookup

An alternative is to have the wrapper only override methods already present in the static wrappee type. In Fig. 6, this would mean that only A.m would be overridden by AWrapper.m.

Instead of overriding additional methods of the actual wrappee, in the example A.n and A.o, we allow an aggregate to contain multiple methods with the same signature and base the dispatch on run-time context information. In the simplest case, the dispatch is based on the static type of the receiver:

```
AWrapper w=new AWrapper<new A()>();
int i; i=w.o(); // executes AWrapper.o, i=0
A a=(A)w; i=a.o(); // executes A.o, i=1
```

In more general cases, the dispatch is not only based on static, but on run-time context information, i.e., an object's history of subsumptions. To illustrate this, assume that method o is declared in interface IO and that both AWrapper and A implement IO. In the following code fragment, added to the above, the static type of the receiver is in both cases IO, but different implementations are executed:

```
IO x;
x=w; i=x.o(); // executes AWrapper.o, i=0
x=a; i=x.o(); // executes A.o, i=1
```

The problem is that there are two occurrences of IO in the aggregate. Thus, we have to choose one for subsumption.[3] Multiple non-virtual inheritance in C++ has a similar semantics.

In languages that do not support final classes or method header specialization (Sect. 6.2), this form of method lookup avoids wrap-time exceptions. However, to also achieve semantic soundness, we still need to adhere to the above four coding conventions. Otherwise, we could execute a.m() in the fragment above and be surprised that we don't get 1 as result. The soundness problems caused by specialization could only be avoided by fully giving up overriding.

Method lookup and subsumption are more complex with this approach. Furthermore, adding this to a single-inheritance language with 'normal' method lookup and subsumption for inheritance, we end up with two different forms of method lookup and subsumption. The technicalities of this approach for compile-time composition of mix-ins can be found in [18].

5.1.3 Refutation of static approaches

Here we briefly discuss why some approaches that avoid possible wrap-time conflicts at compile time and that are based on normal overriding have serious deficiencies.

[3] In our case, we have already subsumed the aggregate to be of type AWrapper, respectively A. A true choice would be needed in the first line if w were of a subtype of both AWrapper and A, e.g., the compound type [AWrapper, A] [7].

Allowing the wrapper to only override methods of the static wrappee type, but not add additional methods would avoid the problem of unsound overriding of additional methods in the actual wrappee, e.g. A.n. However, not allowing additional methods in the wrapper would be a severe restriction, which would greatly reduce the usefulness of generic wrappers. Furthermore, this approach would fail in languages that support final classes or method header specialization (Sect. 6.2).

Negative type information [44, 10, 23] could express that subtypes of IA must not have a method, like n, that might be overridden in an unsound way. However, this would also mean that an aggregate of an AWrapper and a subtype of IA would not be of a subtype of IA and could, therefore, not be referenced by a variable of type IA. Furthermore, negative type information cannot be expressed in type systems of current languages.

Requiring the exact type of the wrappee to be known at compile time, a third approach, would contradict the requirement of run-time applicability (1).

5.2 Hiding of fields and class methods

In many languages, fields and class methods are hidden rather than overridden in subtypes. Hiding of fields, if permitted, is not problematic because the hiding field may have a different type than the hidden field. The static wrappee type is used to access hidden fields in the actual wrappee. Hiding of class methods is usually governed by similar requirements as overriding of instance methods. Thus, the same two options apply.

5.3 Forwarding vs. delegation

The difference between forwarding (also called redirection and consultation) and delegation is the binding of the self parameter in the wrappee when called through the wrapper. With delegation, the self parameter is bound to the wrapper, with forwarding it is bound to the wrappee. Figure 7 illustrates the difference with a client calling method m of the wrappee on a reference to the wrapper.

Forwarding is a form of automatic message resending; delegation is a form of inheritance with binding of the parent (superclass) at run time, rather than at

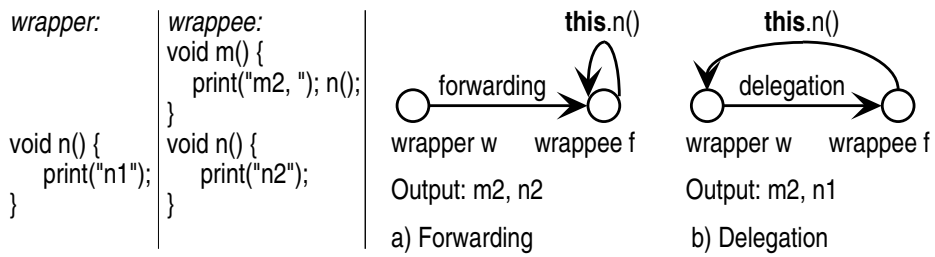


Figure 7: Forwarding vs. Delegation

compile/link time as with 'normal' inheritance [31]. Delegation vs. forwarding, the binding time of the parent, and the support for type transparency are almost orthogonal design dimensions.

In all cases, super calls in the wrappee invoke methods of its superclass and not the wrapper's superclass. During these calls, this is bound to the wrappee with forwarding and to the wrapper with delegation.

The advantage of delegation over forwarding is that the wrapper can better modify and customize the behavior of the wrapped object. The advantage of forwarding is that it eases modular reasoning: As illustrated, delegation can lead to *up-calls* from the wrappee to the wrapper. With forwarding, control stays within the wrappee once a call has been forwarded to it. The wrapper cannot interfere with the flow of control inside the wrappee [53]. Thus, forwarding gives a looser coupling not suffering from the semantic fragile base class problem [36]. This is especially important for component software because the wrapper and the wrappee may be developed independently and composed at run time. Furthermore, forwarding does –unlike delegation– not break encapsulation [46].

5.4 Replacing a wrappee

The wrappee of a given wrapper could be replaced by another object, the type of which is a subtype of the old dynamic wrappee type. It is not sufficient that the new wrappee is a subtype of the static wrappee type: A BorderWrapper wrapping a TextView can be referenced by a variable of static type TextView. Replacing the wrappee by a ButtonView would violate type soundness.

Let ExtTextView be a subclass of TextView. Then, a border wrapper wrapping an ExtTextView should by subsumption be treatable like a border wrapper wrapping a TextView. If we allow a wrappee to be replaced, this is no longer the case. Replacing the ExtTextView in the first aggregate by a TextView is unsound whereas replacing the TextView in the second aggregate by another TextView is sound.

Although cyclic wrapping is type sound in combination with certain features, it is for semantic reasons mostly undesirable. Cyclic wrapping is prevented by the construction process, because the wrappee must be passed as an argument to the wrapper instance creation expression (Sect. 4). If we don't want cyclic wrapping, we also have to prevent it in the replacement of a wrappee.

For semantic reasons, we think that the wrappee should not be exchangeable. By fixing the wrappee for the lifespan of the wrapper, the system becomes more static and, therefore, simpler to analyze and reason about.

5.5 Direct client references to the wrappee

There are both advantages and disadvantages to allowing clients to hold direct references to the wrappee and being able to invoke the latter's methods —bypassing a possible overriding by the wrapper. On the positive side, this may give clients the possibility to invoke methods that are 'accidentally' overridden. For example, both

BorderWrapper and TextView may define instance methods setColor with the same parameters. Without direct access to the wrappee, clients may not be able to change the text color. With direct access to the wrappee, this is possible. However, only clients that are aware that they reference a wrapped TextView rather than a bare one can do so. With the alternative method lookup (Sect. 5.1.2), TextView.setColor can also be accessed through a cast, unless the method is already declared in the static wrappee type IView.

The disadvantage of direct client references to the wrapper is that it gives an additional way for clients to invalidate invariants ranging over both the wrapper and the wrappee. Furthermore, we end up with different reference values to the same aggregate; thus, loosing the unique identity and the possibility of direct reference comparison.

On a middle ground, we could allow clients to access overridden or hidden members of the wrappee using a special qualified syntax, e.g. w.wrappee.getSize(), but not allow direct references. That is, x=w.wrappee would not be legal. This approach restricts direct client access to few in the source code well visible places and solves the problem of different reference values to an aggregate. To safely access wrap-time overridden members, we would additionally need run-time type tests of the wrappee by clients and casts in the qualified access. For example, to invoke the method setColor, not present in IView, of a wrapped TextView, we would write:

```
if(w.wrappee instanceof TextView) {((TextView)w.wrappee).setColor(c);}
```

If we in principle permit direct client references to the wrappee, we can still let the developer of a wrapper decide for each wrapper class or instance, whether to actually allow such references.

Analogies to overriding in inheritance shows that all of the above options are used in some languages: Java does not allow clients to access overridden methods. In Self, clients can have direct references to parent objects. Finally, overridden methods can be invoked by clients using qualified accessors in C++.

The transparency of generic wrappers reduces the need for direct client references. In the containment approach (Sect. 3.2) all functionality that the dynamic wrappee type provides beyond the static wrappee type can only be made accessible by giving clients direct access to the wrappee. With generic wrappers, on the other hand, the full functionality of the dynamic wrappee type is available through the wrapper —except for accidentally overridden methods.

Whether we allow the wrapper to hand out direct references or not, we have the problems of existing references to the wrappee and of the wrappee handing out self references. Even if we in principle permit direct references, we may want to restrict them to clients that explicitly ask for them and are aware of the dangers.

5.5.1 Redirection of existing references

The problem of existing references vanishes if there aren't any. In analogy to aggregation in Microsoft's COM (Sect. 10), we could require the wrappee to be created along with the wrapper and not allow the wrappee's constructor to pass out self references. The latter condition can, however, only be checked with a semantic proof that can in general not be performed automatically. Furthermore, experience with COM showed that this approach is often too restrictive [42].

The second best case is a single reference to the object to be wrapped. In type systems with aliasing control [25, 12] that can guarantee uniqueness of references we could restrict wrapping to unique references. This single existing reference to the wrappee, which is used as argument in the wrapper construction, could then either be redirected to the wrapper or be set to null. The restriction to unique references may severely limit the applicability of wrappers. Furthermore, aliasing control is not common.

For mainstream languages we see the following options:

1. Keep the references to the wrapped object unchanged. This is only an option if we allow direct references to the wrappee. Unfortunately, clients won't recognize if an object they refer to has been wrapped. Hence, they might unknowingly invoke overridden methods of the wrappee and, thereby, cause the aforementioned semantic problems.

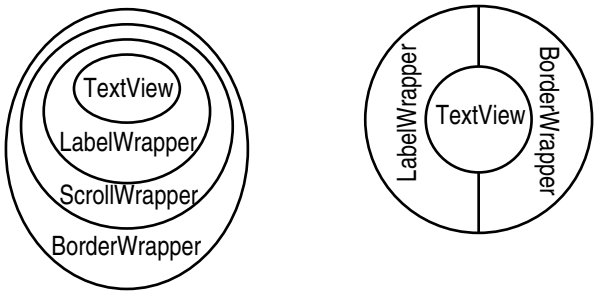
2. Set all existing references to the wrappee to null. Although this approach is type sound, it is clearly unacceptable.

3. Update all existing references to point to the wrapper. Thanks to the transparency of generic wrappers this is sound. Since the type of a reference can only be increased by wrapping, assumptions —gained using run-time type test— that a reference is at least of a certain type are not falsified. On the other hand, negative type assumptions may be invalidated.

5.5.2 Handing out of self references

Wrappees may pass out self references, e.g. for event listener registration. If we don't want direct client references to the wrappee or only allow the wrapper to hand them out, we need to address this issue. The draconian solution is to disallow the use of this in the wrappee except for member access. This is, however, very restrictive and excludes instances of legacy classes not adhering to this restriction from being used als wrappees.

Alternatively, we may define this in the wrappee to reference the wrapper except when used for member access. In combination with forwarding or with delegation and direct client calls of wrappee methods, we get a little semantic curiosity: For variable x of the wrappee's type, this.m() invokes the wrappee's implementation of m() and x=this; x.m() calls the wrapper's overriding implementation, if the latter exists.



a) Conjunctive wrapping b) Disjunctive wrapping

Figure 8: Conjunctive and Disjunctive Wrapping

5.6 Multiple wrapping

There are two forms of multiple wrapping, *conjunctive* and *disjunctive* wrapping (Fig. 8). Conjunctive (also called additive or recursive) wrapping applies multiple wrappers around each other. For example, we might wrap a TextView in a ScrollWrapper and the latter with a BorderWrapper. Cyclic wrapping has been discussed in Sect. 5.4. Infinite wrapping chains are not a problem in practice. They could only occur in infinite executions on computers with infinite amounts of memory.

Disjunctive wrapping presents the same wrappee with different wrappers. It has analogous drawbacks as direct client references to the wrappee, namely the possibility of invalidating invariants ranging over the wrappee and one of its other wrappers. The application of disjunctive wrappers is tricky: Automatic redirection to the outermost wrapper (Sect. 5.5) doesn't work, because there is no single outermost wrapper. Furthermore, disjunctive wrapping is only possible if we allow direct client references to the wrappee, provide a special statement for the simultaneous application of multiple wrappers, or let an existing wrapper wrap its wrappee reference without updating it. With type transparency, disjunctive wrapping can in most cases be replaced by conjunctive wrapping because the full dynamic wrappee type is visible through all wrappers.

If we allow direct client references to the wrappee but not disjunctive wrapping, we have to define what happens if a client wraps an object that is already wrapped. The options are disallowing it and throwing an exception if tried, putting the new wrapper between the wrappee and the old wrapper, and applying the new wrapper around the old wrappee. Thanks to the transparency of generic wrappers, all options are type sound.

5.7 Concealment

In certain cases, a wrapper may want to conceal part of the wrappee from clients. For example, a ConfidentialWrapper and its wrappee should not be serializable for confidentiality reasons. Thus, the wrapper wants to conceal interface Serializable

from clients in case the wrappee implements it. For this case, a conceals clause may be useful in combination with wraps:

class ConfidentialWrapper **wraps** IView **conceals** Serializable {...}

With this definition, no instance of a ConfidentialWrapper aggregate will ever be an element of Serializable. That is, for a variable x referencing such an aggregate, x instanceof Serializable will be false.

Alternatively, a wrapper could be transparent for explicitly listed types only:

class SpecialWrapper **wraps** IView **hoists** IText, IGraphics {...}

When such a SpecialWrapper wraps an instance of a class implementing IText then the functionality declared in IText can be accessed through the wrapper. On the other hand, if the same class also implements another interface, say IContainer, the latter's functionality cannot be accessed through the wrapper and the wrapper cannot be assigned to a variable of static type IContainer. Although transparency is restricted, this approach differs from the containment approach (Sect. 3.2) in that the type of the aggregate depends on the actual type of the wrappee.

Concealment may be practical for special cases, but it causes type soundness problems because the aggregate is not a subtype of the wrappee. Existing references to the wrappee cannot be redirected to the wrapper, if the latter conceals (part of) the static type of the variable containing the reference. Concealment also causes similar problems in combination with solutions 2 and 3 of applying a wrapper to an already wrapped object (Sect. 5.6). Furthermore, with delegation self calls of the wrappee to methods that are concealed by the wrapper fail. For this, a workaround would be to conceal types only from clients, but not from the aggregate itself.

These problems may, but do not necessarily occur in a given system that uses concealment. In analogy to Eiffel allowing subclasses to conceal[4] inherited members, we could allow concealment of types. This would, however, require system validity checks of complete systems.

5.8 Multiple wrappees

So far, we have assumed that a given wrapper instance wraps exactly one object. This could be generalized to a fixed or arbitrary number of objects, thereby providing a single view of a subsystem implemented by multiple objects corresponding to the facade pattern [20]. Similar to multiple code inheritance, this works well unless different wrappees implement methods with the same signature and the wrapper does not override them. In this case, message lookup needs to be redefined. Similar to wrap-time overriding (Sect. 5.1), such conflicts caused by type transparency may not be visible at compile time.

[4]This is called 'hiding' in Eiffel. We don't use this term here to avoid confusion with Java style hiding of class methods and fields (Sect. 5.1).

There are four partly combinable approaches for augmenting the definition of method lookup. The first two are similar to possible solutions for type-sound wrap-time overriding.

1. We can disallow ambiguous aggregates by checking for static conflicts at compile time and throwing an exception at run time when trying to wrap objects that would result in an ambiguity. This approach, however, fails the genericity requirement (2).

2. We can leave the problem unresolved until an ambiguous method is called and only then throw an exception. Self uses this approach in presence of multiple parents. This fails the requirement of as-early-as-possible error detection (6).

3. Message lookup proceeds according to a certain strategy (depth first, breadth first) and a certain order (declaration order, alphabetic order of wrappee type names, etc.). The first matching method is chosen. Any such strategy and order would be rather arbitrary, as illustrated by the criticism of CLOS using syntactic order to choose the correct multi-method [11].

4. The wrapper explicitly defines a lookup strategy and order for conflict resolution. This solution is rather complex and may still not have the desired effect.

6 Interaction With Other Typing Mechanisms

In this section we discuss the interaction of generic wrappers with other common typing mechanisms.

6.1 Subclassing

Here we investigate whether and how generic wrappers can substitute inheritance and how the two may be combined.

6.1.1 Generic wrappers as a substitute for inheritance

If we choose delegation (Sect. 5.3) for generic wrappers, then they can be used to simulate class-based inheritance as follows:

```
class D extends C {...};
D d=new D();
```

Inheritance

```
class D wraps C {...};
D d=new D<new C()>();
```

Simulation with generic wrappers

The main difference is that at compile time we only know the lower bound of the wrappee type for generic wrappers, whereas with inheritance we know the exact superclass. This can be interpreted as flexibility or as lack of knowledge.

If, on the other hand, we use forwarding instead of delegation for generic wrappers, then we cannot modify the semantics of self calls in methods of the supertype. Thus, such generic wrappers cannot be used to simulate inheritance. Considering the advantages of the looser coupling, generic wrappers might still be used in a language as a replacement for inheritance.

6.1.2 Subclassing of wrapper classes

In most mainstream languages like Java, Eiffel, and C++ subclassing implies subtyping. For this principle to extend to wrappers, a subclass of a wrapper class C has to be declared to wrap the same type X as C or a subtype of X. Covariant specialization of the static wrappee type is possible unless the wrappee can be replaced (Sect. 5.4) using a method like setWrappee(StaticWrappeeType w), where the static wrappee type occurs in a contravariant position.

A seeming alternative to a subclass D of a wrapper class C being itself a wrapper would be that D implements the static wrappee type of C. For example, if C wraps X then D implements X would suffice. However, methods of C may contain accesses to wrappee members using the keyword wrappee. These accesses would be undefined in D because instances of D do not have a wrappee. It would be type sound, but semantically undesirable, to let wrappee be a self reference in such cases. Hence, implementing a superclass' static wrappee type instead of wrapping an object of that type is not an alternative.

Forcing subclasses of wrapper classes to be wrappers themselves implies a restriction on the legal static wrappee types. Let C be a wrapper class with static wrappee type X. Then X must not be equal to C or be a subclass of C. If X is itself a wrapper class, it must not —directly or transitively— be declared to wrap C or a a subclass of C. Only infinite and circular chains, which we both forbid, could be elements of a class C not adhering to these rules. It is, however, for example legal for a BorderWrapper to have the static wrappee type IView (or even some other wrapper type) and have an instance of BorderWrapper wrap another BorderWrapper.

6.2 Method Header Specialization and Final Classes

Some languages allow overriding methods to have more specialized headers. For example, Java allows a non-final method to be overridden by a final one and allows the overriding method to have a more restricted exception throws clause and a higher accessibility. Other languages also allow covariant return type and contravariant parameter type specialization.

This creates problems with overriding by wrappers, even if the overriding is statically visible. Wrapping an instance of B with a BWrapper in Fig. 9, would

```
interface IB {
  void p() throws SomeException;
}

class BWrapper implements IB wraps IB {
  public void p() throws SomeException {...};
}

class B implements IB {
  public final void p() {...};
}

IB b=new B();
BWrapper w=new BWrapper<b>(); // illegal wrapping caught by exception
((B)w).p() // final method would be overridden and exception might be thrown
```

Figure 9: Method Header Specialization Example

override the final method B.p with an empty throws clause by BWrapper.p, which may throw SomeException.

To prevent such unsound overriding, we have to use wrap-time exceptions. Thus, in languages with method header specialization, even the alternative form of method lookup (Sect. 5.1.2) cannot fully avoid the problem of wrap-time exceptions. Method header specialization is the type correspondent of semantic refinement discussed in Sect. 5.1.1.

Final classes pose a similar problem. They should not be subtyped. Thus, it is a compile-time error to declare a wrapper with a static wrappee type that is a final class type. At run time, an exception is thrown if an attempt is made to wrap an instance of a final class.

6.3 Overloading resolution

Many languages support overloading of method names, that is classes containing multiple methods with the same name, but different numbers or types of parameters. For every call, the signature of the method to be invoked is determined at compile time. Compile-time selection of the invoked method's signature means that a better fitting signature of the run-time type, e.g. a subclass, is ignored. The same principle applies to generic wrappers: There is no need for a costly search of a possibly better fitting signature in the actual wrappee during method invocation.

6.4 Parametric types

Parametric types and methods, like C++ templates and generic Eiffel classes, allow compile-time reuse of generic classes and interfaces by providing type parameters and, thereby, creating generically derived classes. Java doesn't support parametric

types. We use here the C++ like syntax with bounds common to most proposals (e.g. [1]) for adding F-bounded polymorphism to Java.

Generic wrappers and parametric types can be combined without problems. Instances of generically derived classes don't distinguish themselves from instances of normal classes. Hence, they can be normally wrapped.

Generic wrappers can be parameterized. As in combination with inheritance, the type parameter might be implicitly limited by soundness constraints for overriding and hiding. Let classes C and D be defined as follows:

```
class C {
  void m(String s) {...}
}

class D<T> wraps C {
  int m(T s) {...}
}
```

Using String for the parameter T, i.e. D<String>, we would get two methods m(String s). Hence, such a derivation has to be forbidden at compile time. An analogous problem occurs in combination of parametric types with inheritance rather than generic wrapping, as illustrated by replacing wraps by extends in the declaration of D above.

By allowing the type parameter in the wraps clause we can make use of possible additional compile time knowledge about the wrappee. Compare the classes E1 and E2, where M stands for some member declarations:

```
class E1 wraps I {M}
class E2<T implements I> wraps T {M}
```

Any legal wrappee of an instance of a derived class of E2 is also a legal wrappee of an instance of E1. However, derived classes of E2, such as E2<C> (assuming that C implements I), can be used to give more static type information. Similar static type information can be provided by a subclass of E1 with covariantly specialized static wrappee type C or with compound types [7], e.g. [E1, C].

Generic wrappers can also be used as bounds in generic classes and as actual parameters in generic derivations.

6.5 Compound types

Compound types [7] let us express directly that the type of a parameter must subtype a set of named reference types, thereby optimally supporting flexible behavioral typing.[5] For example, the compound-typed variable [ILabel, IText] v may reference an instance of a class that implements both ILabel and IText, or the value of v may be null.

[5] Cecil's [11] greatest lower bound types, written ILabel & IText, and Objective-C's [38] multiple protocols, written <ILabel, IText>*, are similar to compound types in Java.

```
signText(Key privateKey, [ILabel, IText] idText) {...}

class LabelWrapper implements ILabel wraps IText {...}

Text t; ...;
signText(k, ([ILabel, IText]) new LabelWrapper<t>)
```

Figure 10: Summary of example definitions

Compound types let us specify that a parameter must be a text with a label, a text with a border, or even a text with both a label and a border. For example, the method signText(Key privateKey, [ILabel, IText] idText) could set the label to the signature of the text calculated with privateKey (Fig. 10). Let LabelWrapper implement ILabel and wrap IView and let TextView implement IText. An instance of LabelWrapper wrapping a TextView could be passed as second argument to method signText. Without type transparency, this would not be possible.

If the source code of signText were under our control, we could declare the parameter idText to have type ILabel and then in the body of signText get the wrappee and test its type. This approach has, however, several drawbacks: First, the typing of the parameters conveys less precise information. Users don't immediately see what parameters are legal. Second, instances of non-wrapper classes that only implement ILabel are legal parameters. Thus, errors that could be caught at compile time are not. Third, the implementation has to differentiate between parameters that implement the two interfaces in one or two objects. In conclusion, compound types allow for more precise typing, but without type transparency certain aggregates would not be legal parameters.

Constituent types of a compound type being implemented by different objects may, however, lead to undesirable semantic effects. Assume, for example, that TextView also implements ILabel and while the text is modified constantly updates the signature that it itself stores. In this case, modifying the text contents of a TextView wrapped by a LabelWrapper and then reading the label, may unexpectedly returns a wrong signature, namely that stored in the wrapper and not that of the wrappee. The same problem exists, of course, in the decorator pattern (Sect. 3.2).

7 Generic Wrappers in Java

As a proof of concept, we add generic wrapping to Java. We present generic wrappers as a strict extension, that is existing Java programs need not be changed and instances of existing classes can be wrapped.

We select a consistent set of features from the aforementioned design choices and give a definition of generic wrappers in Java. We base our choices on the motivating examples and the above discussions, without repeating the latter. Next, we discuss selected integration issues with the Java library. Finally, we show how the defined mechanism solves the motivating problem.

In the next section, we report on a mechanized type soundness proof for the presented solution. A discussion of efficient implementation strategies is beyond the scope of this paper.

7.1 Feature selection and language integration

Both compile-time and wrap-time overriding and hiding are governed by the same rules as (compile-time) overriding and hiding in subclasses. Furthermore, we don't allow instances of final classes to be wrapped. Violations of these rules by the wrapper/static wrappee pair are flagged at compile time; violations by the wrapper/actual wrappee pair cause exceptions at the time of wrapping.

To get loose coupling between the wrapper and the wrappee and to facilitate semantic reasoning we chose forwarding over delegation and fix the wrappee for the lifespan of the wrapper. All existing references to the wrappee are redirected to the wrapper upon wrapping. We define this in the wrappee to refer to the wrapper except when used for member access. For example, this.x is the field x of the wrappee, but s.register(this) passes a reference to the wrapper as parameter. This approach guarantees a unique identity of the aggregate from the clients' point of view.

In a tribute to flexibility, we allow clients to explicitly attain direct references to the wrappee. Still, we hope this feature proves to be superfluous. The implementor of the wrapper class determines whether clients can get direct references to the wrappee by putting an access modifier (private, protected, public) between the keyword wraps and the static wrappee type, e.g:

```
class LabelWrapper3 wraps public IView {...}
```

The access modifier of the wrappee in a subclass must provide at least as much access as that in the superclass.

The keyword wrappee can be treated like the name of a final instance field of the wrapper class with the used modifier, e.g public in the above example. For example, let x be a variable of type LabelWrapper3. Then clients can access the wrappee as x.wrappee. To navigate back from a wrappee to its outermost wrapper, the method:

```
public final Object getWrapper() {return this;}
```

is added to the class Object. With the above definitions, this method returns a reference to the wrapper if the receiver object is wrapped and otherwise to the receiver itself.

We allow only conjunctive, but not disjunctive wrapping. Wrapping an already wrapped object corresponds to wrapping its outermost wrapper. Because it is not sound in combination with the above features, we don't allow concealment. Every wrapper has exactly one wrappee. (The wrappee may of course itself be a wrapper.)

Although believed not to cause any problems, we do not allow array objects to be wrapped, as would be possible for wrappers, the static wrappee type of which is an array type, Object, Cloneable, or Serializable. The latter are the only interfaces implemented by arrays.

Grammar The grammar for class declarations and class instance creation expressions is augmented as follows [21]:

ClassDeclaration:	Modifiers$_{opt}$ **class** Identifier Super$_{opt}$ Interfaces$_{opt}$ Wrapper$_{opt}$ ClassBody
Wrapper:	**wraps** AccessModifier$_{opt}$ ReferenceType
AccessModifier:	one of **public protected private**
ClassInstanceCreationExpression:	**new** ClassType Wrappee$_{opt}$ (ArgumentList$_{opt}$)
Wrappee:	< Expression >

Additionally, wrappee is added as an alternative to the PrimaryNoNewArray production.

Synchronized methods To simplify synchronization between threads, acquiring a lock on a wrapper instance also locks the wrappee, possibly recursively in case of conjunctive wrapping.

7.2 Library integration

The library being an integral part of Java —the description of three packages is even part of the Java language specification— we discuss how selected features interplay with generic wrappers. Generic wrappers integrate in a straightforward way with most libraries, often providing new possibilities.

Serialization For instances of a Java class to be serializable, the class must implement the empty interface Serializable. The serialization of wrappers is problematic in case the wrapper implements Serializable, but the wrappee doesn't. Type soundness forbids us to not externalize the wrappee and set the wrappee reference to null upon deserialization. We see the following options:

1. Disallow serializable wrappers to wrap instances of non-serializable classes.
   (a) Using compound types (Sect. 6.5), this can be done statically without otherwise restricting applicability: A wrapper class that implements Serializable must require the wrappee to do the same, e.g., a label wrapper implementing Serializable would have to be declared as LabelWrapper wraps [IView, Serializable].
   (b) Expressing this requirement without compound types adds unnecessary restrictions, if the desired wrappee type, e.g. IView, is not a subtype of Serializable. In this case we have to declare a subinterface of IView

and Serializable. However, due to by-name equivalence of types this unnecessarily excludes classes implementing the two directly [7].

(c) Instead of static checks we could resort to throwing an exception when trying to wrap an instance of a non-serializable class by a serializable wrapper as we do for unsound overriding (Sect. 5.1).

2. Treat the wrappee like an object referenced by a field of the wrapper and throw a NotSerializableException when trying to serialize the aggregate. This ruins the clients' perception of the aggregate being a single object.

3. If the wrappee has a no-argument constructor, only serialize the wrapper and create a new wrappee upon deserialization. Otherwise, use one of the above options. This is analogous to a subclass of a non-serializable superclass.

4. Ignore security and other concerns of the implementor of the wrappee and serialize the latter nonetheless.

With compound types, we choose the first option because it solves the problem at compile time without introducing any unnecessary restrictions. Otherwise we'd use the second option.

A dual problem occurs if the wrappee implements Serializable, but the wrapper doesn't. With concealment, we could conceal the interface from clients. Without, we get almost dual options. However, due to the lack of static negative type information there is no correspondence options 1 (a) and 1 (b), except the very restrictive requirement that all wrapper classes must implement Serializable. We, therefore, choose the second option of throwing a NotSerializableException when trying to serialize a wrapper that is not serializable.

Cloning The general contract of clone is that it creates and returns a copy of the receiver object [21]. Because we don't allow disjunctive wrapping, clone of a wrapper has to either create a deep copy or throw a CloneNotSupportedException.

The method clone is defined in Object, classes implement the empty interface Cloneable to indicate that they actually support cloning. If a wrapper implements the interface, but the dynamic wrappee type doesn't, we have a similar problem as for serialization. Analogously we have the options of disallowing a wrapper that implements Cloneable to wrap an instance of a class that doesn't, throw a CloneNotSupportedException, or create a new wrappee with the no-argument constructor if present and accessible. We opt again for the first choice.

The dual problem of the wrappee, but not the wrapper, implementing Cloneable is again solved with the second option. The implementation of clone, that the wrapper inherits from Object and which overrides the implementation in the wrappee, throws a CloneNotSupportedException.

7.3 Assessment

Our mechanism fulfills all requirements (Fig. 2) except for genericity (2). The latter fails in cases where overriding or hiding would not be sound. We consider this acceptable because exceptions are already thrown at the time of wrapping — and not at the time of member access— and because creation of new instances can also fail for other reasons with an exception in existing Java.

Clearly, the motivating problems (Sect. 2.1) can be solved with the presented generic wrappers for Java: Let BorderWrapper be declared as follows:

```
class BorderWrapper wraps IView {...;}
```

If such a border wraps a TextView, the aggregate is of type TextView and is, therefore, recognized as such by the spell check procedure of all embedded views. Likewise, if such a border wraps a ButtonView, the aggregate is of type IControl implemented by ButtonView. Hence, it can be inserted into a forms container. The developers of TextView, ButtonView, the spell check operation, and the forms container don't have to do any special coding for this to work.

8 Type Soundness

In this section, we report on a mechanically verified formal proof of type soundness of Java extended with generic wrappers. Type soundness intuitively means that all values produced during any program execution respect their static types. An immediate corollary of type soundness is that method calls always execute a suitable method, that is, there are no 'method not understood' errors at run time. Type soundness is not a trivial property, especially for polymorphic languages [6, 9]. It came to prominence with the discovery of its failure in Eiffel [13, 34]. Static typing loses much of its raison d'être if type soundness does not hold.

Our proof of type soundness for generic wrappers is based on the work of von Oheimb and Nipkow [41], a much extended version of [39]. They have formalized a large subset of Java and mechanically proved type soundness with the theorem prover Isabelle/HOL [43].

For this paper, we have added generic wrappers to this formalization. For simplicity, we have extended the formalization of the existing Java type system, rather than our previous extension with compound types [7]. Finally, we adapted the proofs and ran them through Isabelle/HOL.[6]

8.1 Definitions

Here, we present the widening and casting relations, which are interesting in their own rights. Since all type judgments involving arrays are unchanged, they are

[6] At http://www.abo.fi/~mbuechi/publications/GenericWrapping.html the Isabelle theories are available.

omitted in this presentation. A full report of all the mechanical details is beyond the scope of this paper.

The Java language specification introduces identity and irreflexive widening conversions separately. The Java language specification [21] uses the term 'widening' for its form of subtyping. Since identity conversions are possible in all conversion contexts permitting widening, the two are merged in the formalization. The expression $\Gamma \vdash S \preceq T$ says that in program environment Γ objects of type S can be transformed to type T by identity or widening conversion. In particular, expressions of type S can be assigned to variables of type T and expressions of type S can be passed for formal parameters of type T.

We use the following naming conventions:

C, D	classes	A	list of classes
I, J	interfaces	S, T	arbitrary types
R	reference type	Γ	program, environment

The judgment $\Gamma \vdash C \prec_{\mathsf{C}} D$ expresses that class C is a subclass of class D, $\Gamma \vdash C \rightsquigarrow I$ that class C implements interface I, and $\Gamma \vdash I \prec_{\mathsf{I}} J$ that I is a subinterface of J (Fig. 11). Furthermore, $\mathsf{is\_type}\ \Gamma\ T$ expresses that T is a legal type in Γ, $\mathsf{RefT}\ R$ denotes reference type R, and NT stands for the null type.

$\mathsf{Class}\ C$ stands for the class type C and $\mathsf{Iface}\ I$ for the interface type I. Furthermore, the discriminators $\mathsf{is\_class}\ \Gamma\ C$ and $\mathsf{is\_iface}\ \Gamma\ I$ are used.

In our formalization we now have two kinds of classes: normal (non-wrapper) classes and wrapper classes. The discriminator $\mathsf{is\_wrapper}\ \Gamma\ C$ is true if C is a wrapper class and false otherwise. $\mathsf{WrappeeOf}\ \Gamma\ C$ denotes the static wrappee type of class C in program environment Γ.

At run time, instances of wrapper classes are of aggregate types. Aggregate types are finite lists of at least two class types. An instance of the wrapper class C wrapping an instance of the wrapper class D that itself wraps an instance of the (non-wrapper) class E belongs to type $\mathsf{Aggregate}\ [C, D, E]$.

The discriminator $\mathsf{is\_aggregate}\ \Gamma\ A$ is true if A is a list of class names, all but the last element of A denote wrapper classes in Γ, the last element denotes a non-wrapper class, for each $i \in 0..\mathsf{length}\ A - 2$ there exists a $j_i > i$ such that the j_ith element extends, implements, or is equal to the static wrappee type of the ith element, and there are no clashes between method signatures of the elements of A.

$\Gamma \vdash S \preceq T$	S widens to ('is subtype of') T in Γ
$\Gamma \vdash C \prec_{\mathsf{C}} D$	C is a subclass of D in Γ
$\Gamma \vdash C \rightsquigarrow I$	C implements I in Γ
$\Gamma \vdash I \prec_{\mathsf{I}} J$	I is a subinterface of J in Γ
$\Gamma \vdash S \preceq_{?} T$	cast from S to T permissible at compile time in Γ

Figure 11: Summary of notation

Since there are no variables of aggregate type and because we do not allow the dynamic reassignment of wrappees, we only need widening rules with aggregates on the left-hand side of the conclusion judgment.

The following six typing judgments apply unchanged also to wrapper classes:

$$\frac{\mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash T \preceq T} \qquad \frac{\mathsf{is\_type}\ \Gamma\ (\mathsf{RefT}\ R)}{\Gamma \vdash \mathsf{NT} \preceq \mathsf{RefT}\ R}$$

$$\frac{\Gamma \vdash I \prec_{\mathsf{i}} J}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Iface}\ J} \qquad \frac{\mathsf{is\_iface}\ \Gamma\ I; \mathsf{is\_class}\ \Gamma\ \mathsf{Object}}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Class}\ \mathsf{Object}}$$

$$\frac{\Gamma \vdash C \prec_{\mathsf{C}} D}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D} \qquad \frac{\Gamma \vdash C \rightsquigarrow J}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Iface}\ J}$$

The following widening rules involving wrapper classes are used at compile time:

$$\frac{\mathsf{is\_wrapper}\ \Gamma\ C;\ \Gamma \vdash \mathsf{WrappeeOf}\ \Gamma\ C \preceq \mathsf{Class}\ D}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D}$$

$$\frac{\mathsf{is\_wrapper}\ \Gamma\ C;\ \Gamma \vdash \mathsf{WrappeeOf}\ \Gamma\ C \preceq \mathsf{Iface}\ J}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Iface}\ J}$$

The following widening rules involving aggregates are used at run time (set converts a list into a set):

$$\frac{\mathsf{is\_aggregate}\ \Gamma\ A;\ \exists C \in \mathsf{set}\ A.\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D}{\Gamma \vdash \mathsf{Aggregate}\ A \preceq \mathsf{Class}\ D}$$

$$\frac{\mathsf{is\_aggregate}\ \Gamma\ A;\ \exists C \in \mathsf{set}\ A.\Gamma \vdash C \rightsquigarrow J}{\Gamma \vdash \mathsf{Aggregate}\ A \preceq \mathsf{Iface}\ J}$$

The casting relation $\Gamma \vdash S \preceq_? T$ states, that a cast from type S to type T is permissible at compile time, that is, the type cast '(T)e', where e is of type S, might succeed at run time. This is interesting because if it can be proven to always fail, the compiler can already flag an error.

If $\Gamma \vdash S \preceq T$ holds, the cast can be proven to always succeed. Otherwise, a run-time validity test must be performed to check whether $\Gamma \vdash R \preceq T$ holds for the run-time type R of the cast operand. The following general casting conversions are applicable to wrapper classes as well:

$$\frac{\Gamma \vdash S \preceq T}{\Gamma \vdash S \preceq_? T} \qquad \frac{\mathsf{is\_class}\ \Gamma\ C;\ \mathsf{is\_iface}\ \Gamma\ J}{\Gamma \vdash \mathsf{Class}\ C \preceq_? \mathsf{Iface}\ J} \qquad \frac{\mathsf{is\_iface}\ \Gamma\ I;\ \mathsf{is\_class}\ \Gamma\ D}{\Gamma \vdash \mathsf{Iface}\ I \preceq_? \mathsf{Class}\ D}$$

The following two casting rules have weaker conditions in the presence of generic wrappers:

$$\frac{\mathsf{is\_class}\ \Gamma\ C;\ \mathsf{is\_class}\ \Gamma\ D}{\Gamma \vdash \mathsf{Class}\ C \preceq_? \mathsf{Class}\ D} \qquad \frac{\mathsf{is\_iface}\ \Gamma\ I;\ \mathsf{is\_iface}\ \Gamma\ J}{\Gamma \vdash \mathsf{Iface}\ I \preceq_? \mathsf{Iface}\ J}$$

8.2 Theorems and conclusions

With the above definitions we proved that evaluation and execution are type sound and that method lookup always succeeds. These theorems on the extended type system correspond to the ones proved by von Oheimb and Nipkow for Java without generic wrappers. The first two theorems are syntactically equivalent to the ones of von Oheimb and Nipkow. Semantically they are, however, different because the types include generic wrappers. The method lookup theorem is both syntactically and semantically different.

The currently by von Oheimb and Nipkow formalized subset of Java, on which we build, still does not capture all features. Of them final classes, modifiers (currently only static), interface fields, and methods of the class Object would be relevant for generic wrappers. In particular, final classes would allow us to slightly strengthen some of the premises in the above casting rules.

The main advantages of a mechanized over a paper-and-pencil proof are additional confidence and better support for extensions. We would like to stress the second aspect. Not only did the formalization result in a soundness proof, but the proof tool also reminded us of what all needed to be defined about generic wrappers before the desired properties could be established. Most proof scripts worked without modifications. The fact that all theorems were reproved mechanically for the extended language definition conveys more confidence than the typical adaptation of a paper-and-pencil proof with 'this-should-still-hold' handwaving.

9 Reflective Mix-Ins

Mix-ins with capabilities to create new derived classes at run time provide a single-object alternative to generic wrappers. To our knowledge, gbeta (Sect. 10) is the only typed language that supports the derivation of mix-ins from generic classes at run time. However, since gbeta differs greatly from most other object-oriented languages such as Java, Eiffel, and C++, a transfer of this mechanism to other languages is not straightforward. Hence, this section should be understood as an alternative proposal, not as a comparison with existing languages.

9.1 The proposed mechanism

To illustrate the mechanism, consider again the parameterized class ParBordered-View:

class ParBorderedView<Wrappee **implements** IView> **extends** Wrappee {...}

Usually, derivations such as ParBorderedView<TextView> can only be made at compile time. Hence, we concluded in Sect. 3.1 that mix-ins do not satisfy the requirement of run-time applicability (1) as, for instance, required to implement compound documents.

We could, however, allow derivations to be made at run-time. We outline such a proposal for Java based on reflection. In Java the static method forName of Class gets the class object of a class. Assume that this also works for parameterized classes. Thus a reference to the class object of ParBorderedView can be assigned to the variable pc as follows:

```
Class pc=Class.forName("ParBorderedView");
```

From this, we can get the class object of a derived class with the postulated method derivation. The type argument of the latter is a class object.

```
Class dc=pc.derivation(Class.forName("TextView"));
```

If needed, this creates a new derived class. Actual run-time applicability comes from the fact that we can replace the string constant "TextView" by a variable. Finally, we can create instances of the derived class using the reflection method newInstance:

```
Object o=dc.newInstance();
```

As shown, this can be used to create arbitrary bordered views at run time. Let us assess this solution with respect to the requirements (Fig. 2) and in comparison with generic wrappers.

9.2 Comparison with generic wrappers

Reflective mix-ins have roughly the same properties as generic wrappers that require the wrappee to be created together with the wrapper. Consistency with mix-ins derived at compile time further restricts the design space of reflective mix-ins as compared to generic wrappers.

9.2.1 Combination of wrapper and wrappee into a single object

Reflective mix-ins combine the wrapper and the wrappee into a single object. This has several consequences. First, reflective mix-ins cannot be used to wrap existing objects.

Second, the replacement of a wrappee (Sect. 5.4) and direct client references to the wrappee (Sect. 5.5) are not applicable. On the negative side, the latter implies that accidentally overridden methods cannot be called by clients. On the positive side, the problems of redirecting existing references and of handing out self references don't exist. However, these problems don't occur with generic wrappers either, if we force the wrappee to be created along with the wrapper.

Third, the combination of the wrapper and the wrappee into a single object means that only conjunctive, but not disjunctive wrapping is possible (Sect. 5.6).

9.2.2 Various differences and similarities

The combination of the super- and subclass becomes first visible when the corresponding derived class object is generated with the method derivation. Hence, possibly unsound combinations due to overriding and hiding conflicts can only be caught at run time by exceptions as for generic wrappers (Sect. 5.1).

Consistency with mix-ins derived at compile time dictates that we use delegation rather than forwarding. Thus, reflective mix-ins suffer from the semantic fragile base problem. The latter is aggravated by the fact that the base class is not statically known and the combination cannot be analyzed at compile time.

The static type safety of reflective mix-ins and generic wrappers is roughly equivalent. However, the return type of newInstance and all other reflective methods for creating instances from a class object being Object, we always need an initial cast with reflective mix-ins. Furthermore, the parameters of constructors cannot be statically type checked.

An advantage of reflective mix-ins is that they allow the type parameter to be used in other places than just in the extends clause. The usage must, however, be restricted to covariant or even private occurrences to maintain subtyping.

9.2.3 Overloading resolution

Reflective mix-ins cause two kinds of overloading resolution problems that do not occur with generic wrappers. These problems arise from the following two design principles: A generically derived class that is derived at compile time has the same semantics as a plain class with the same members (copy semantics). The semantics of a derived class is independent of the time of derivation (compile or run time). Thus, the copy principles also applies to classes that are derived at run time using reflection. Based on this, we can illustrate the two problems.

```
class C {}
class D extends C {
  void m(Integer x) {...}
}
class X {
  static Object n(Object x) {...}
  static int n(D x) {...}
}
class W<A extends C> extends A {
  void m(String x) {...}
  void o() {
    Object y=X.n(this); // different resolution for W<D> causes type error
    m(null); // resolution ambiguous for W<D>
  }
}
```

Figure 12: Overloading Resolution Problems of Reflective Mix-Ins

First, the most specific method may depend upon the derivation parameter. In Fig. 12, the most specific method for the call X.n(this) is the one with return type Object in the derived class W<C>. However, in the derived class W<D> the most specific method is the one with return type int. In this case the assignment to y is ill-typed.

Second, in languages, such as Java, without a total order between methods for overloading resolution, calls may be ambiguous in certain derived classes. The call m(null) is unambiguous in W<C>, but it is ambiguous in the derivation W<D>.

Like overriding conflicts, changes in overloading resolution and overloading ambiguities can only be caught at the time of derivation by raising an exception. These problems do not exist for generic wrappers (Sect. 6.3). Because combinations with actual wrappee types are only made at run time, there is no need to follow the copy semantics by analogy to a static case. Thus, overloading can be resolved based on the static wrappee type. Hence, generic wrappers fare slightly better than reflective mix-ins with respect to the genericity requirement (2). A more detailed discussion of overloading resolution for static mix-ins can be found in [3].

10 Related Work

Section 3 already provides an overview of some related mechanisms. With the exception of delegation, where a final comparison with our mechanism is deemed interesting, these technologies are not discussed again here.

10.1 Language mechanisms

Delegation in prototype-based languages What do we gain with generic wrappers over delegation in prototype-based languages?

First, the static wrappee type and calls to it can be statically type checked. Some prototype-based languages, such as Cecil [11], also have (optional) static type systems. However, these languages require the exact type or even the concrete instance of the parent object to be known at compile time. The same approach is taken by prototype-based object calculi, e.g. [17]. Thus, they fail the requirement of run-time applicability (1).

Second, with generic wrappers the dynamic wrappee type can be checked with run-time type tests.

Third, type casts are the only points of failure; method lookup always succeeds. This greatly simplifies debugging by indicating errors closer to where they occur.

Fourth, generic wrappers are targeted at mainstream class-based languages.

For our exemplary generic wrappers in Java, we have chosen a set of distinguishing features that facilitate modular reasoning. First we use forwarding rather than delegation. Second the wrappee is assigned snappily differentiating it from reassignable parent fields. Third, we disallow disjunctive wrapping. The latter is no

problem because we get sharing of behavior from classes whereas prototype-based languages have to use shared parents for this.

Lava Kniesel [30] has implemented an extension of Java with wrappers. The main difference to our generic wrappers is that in his proposal the aggregate is not a subtype of the actual, but only of the static wrappee type. Thus his proposal fails the transparency requirement (3) and is more limited in its applicability. Lava's wrappers are a form of the decorator pattern (Sect. 3.2) with automatically generated forwarding stubs and multiple wrappees combined with delegation. Wrappees can be reassigned, thereby, complicating semantic reasoning. The proposal is not type sound because the wrappees are assigned within the constructor. Independent extensibility, the focus of our proposal, is not well supported.

Delegation for software and subject composition Harrison et al. [24] discuss options for different bindings of this in the decorator and facade patterns. They show how to implement delegation using either stored or passed pointers in class-based languages. Furthermore, they propose a declarative approach, to be used by component assemblers, permitting the binding of this to be customized on a per-method base. Their solution does not address the shortcomings of the decorator pattern with respect to our requirements. Namely, it does not provide for transparency (3).

gbeta gbeta [16], a generalized version of BETA, supports two forms of dynamic (parent fixed at run time) inheritance through multiple inheritance. Dynamic object specialization is a dynamic modification of the structure of an existing object, preserving object identity. For example, the statement somePtn##->anObject## enhances the structure of anObject with the pattern somePtn. Furthermore, gbeta allows a form of reflective mix-ins through non-constant virtual types as superpatterns.

Because gbeta uses submethoding with INNER rather than overriding, it is not obvious how the mechanisms of gbeta could be transferred to more 'standard' object-oriented languages.

Dynamic mix-ins Steyaert et al. [47] propose dynamic inheritance through mix-ins. The catch is that each object must contain a specification of all its potential enhancements. This renders their proposal inapplicable for mutually unaware component vendors. Mezini [35] also presents a sophisticated, but complex approach to object evolution without name collisions. However, her work is untyped.

Cecil In addition to the combination of prototypes with an (optional) static type system, Cecil [11] has two more features worth a comparison: predicate objects and multi-methods. Predicate objects are Cecil's more restricted alternative to dynamic inheritance. An object o that inherits from all parents of a predicate object

p automatically also inherits from p if the state of o satisfies the predicate of p. Predicate objects permit important states of objects to be explicitly identified and named. However, with respect to the problem at hand, they are mere syntactic sugar for if or case statements in the methods of the parent objects.

Multi-methods are, ignoring modularization, just elegant syntactic sugar for an explicit coding of a Cartesian product [52]. Since multi-methods can —with certain restrictions to guarantee a best fit [11]— be defined outside the classes of their receivers, they can be used to modify a component without changing the latter's source code [26]. However, they do not address the problems of independent extensibility and run-time applicability. Furthermore, they cannot be used to selectively change the behavior of certain instances only.

Lag*oo*na Lag*oo*na [19] is a single dispatch language that separates messages from reference types. Any message (without a return type) can be sent to any object. For messages without return types, object types can provide a default method with programmable forwarding. Thus, wrappers could simply forward messages that they don't understand to their wrappees. However, only additional methods with return type void can be called. The wrapper is not a subtype of the actual wrappee and type test cannot be used directly to test whether a message will be understood. Hence, Lag*oo*na does not satisfy the transparency requirement (3). With respect to the problem at hand, forwarding in Lag*oo*na is just a syntactically sugared version of bottleneck interfaces.

Fewer errors are caught by the type system because any message can be sent to any object and semantic reasoning is difficult due to the programmable resending.

Objective C Categories in Objective C [38] allow classes to be extended with a new set of methods/protocols independently of the original class definition. This compile-time mechanism corresponds to creating a subclass and globally replacing all occurrences of the superclass by the subclass. Categories modify whole classes, rather than individual objects. Categories do not fulfill the requirements of run-time applicability (1) and genericity (2).

Binary Component Adaption (BCA) BCA [28] provides for similar adaption of Java binaries as categories for Objective-C binaries. Thus, BCA does not solve the problem at hand either.

Aspect-oriented programming Aspects [29] are a new category of programming construct that 'cross-cut' the modularity of traditional programming constructs. So an aspect can localize, in one place, code that deeply affects the implementation of multiple classes or methods. Aspects modify classes at compile time. Hence, they do not address the problems of run-time composition of objects created by different components from different vendors.

Mix-in calculus Bono et al. have developed a formal calculus of classes and mix-ins [4]. Method declarations in mix-ins are explicitly marked as overriding an existing method or introducing a new method. The lower type bound (static wrappee type) is computed from the signature of a mix-in. Redefined methods give positive type information and new methods negative type information. Subtyping is determined by the types' structures. Negative type information is used to avoid mix-in-application-time exceptions.

We believe that name equivalence for types in combination with our coding conventions (Sect. 5.1.1) is better suited to avoid accidental overriding. First, with structural subtyping a method marked as redefining may override an unrelated method that happens to have the same signature. Our solution avoids this. Second, a method m marked as new cannot override a method m from the actual base class even if the two were meant to correspond (as in our system expressed by the fact that the wrapper and the wrappee implement an interface IM declaring m). In our approach, overriding is possible in this case.

The addition of new/redefined method attributes to our generic wrappers would not be very useful. Positive type information can be expressed by the explicitly named static wrappee type. Declaring a method m in the wrapper to be new if the static wrappee type contains a method with the same signature is pointless because it leads to a compile-time error. This leaves us with the possibility to mark a method n as new if the static wrappee type does not contain a method with this signature. For this to be useful, the type system would have to support negative type information. As discussed in Sect. 5.1.3, this is rare and causes other problems.

10.2 Binary component standards

As stated in Sect. 1, wrapping of objects created by different components requires binary standards. Thus, we survey below the most common component standards.

However, even with wrapping on the binary level, direct language-level support has many advantages. First, it makes it simpler and less error-prone to write components for binary wrapping mechanisms, because component instances can be referenced by normal, tightly typed variables and method calls can be type checked. Second, the full power of type systems for early error detection can only be used with programming language support.

Microsoft COM COM is a language-independent binary component standard. It provides two forms of object composition for reuse: containment and aggregation [45]. With containment, the wrapper (outer) holds a reference to the wrappee (inner) and must provide explicit forwarding stubs. Thus, COM containment shares most properties with its language-level sibling (Sect. 3.2).

A COM object implements a set of interfaces. Clients have only references to these interfaces. Each interface has a different address. Thus, if a client has a reference to one interface of a given object, it cannot directly access functionality provided by the same object through a different interface. Instead, the client has

to call the method QueryInterface, the first method of any interface, with the name of the desired interface as parameter. COM aggregation makes use of this indirection. When the wrapper is asked for an interface that it does not implement itself, it forwards the QueryInterface call to the wrappee. Alternatively, it may explicitly conceal interfaces of the wrappee by answering request negatively itself instead of forwarding them. The wrappee holds a back pointer to the wrapper. When the wrappee's QueryInterface is called directly by a client, the wrappee forwards the call to the wrapper. In summary, unless the wrapper conceals part of the wrappee, aggregation satisfies the transparency requirement (3). On the negative side, aggregation only works with specially coded classes as wrappees. A programming language could enforce the rules for aggregation so that all components written in this language would be aggregable. Aggregation requires the inner object to be created along with the outer object to guarantee that only the wrapper holds a reference to the wrappee.

Ibrahim and Szyperski [27] have formalized parts of COM, including containment, aggregation, and QueryInterface. The latter is replaced by typecase statements in their exemplary language COMEL. Aiming for a truthful formalization, COMEL has the same properties as COM on the binary level.

JavaBeans In its current version, JavaBeans does not support wrappers. A very rudimentary draft proposal for an object aggregation/delegation model [8] was scrapped after public criticism. In this conventions-based approach, the wrapper (delegator) was to hold references to a number of wrappees (delegatees), but not to implement the static wrappee type. Every wrapper was supposed to implement the interface Aggregate:

```
public interface Aggregate {
  Object getInstanceOf(Class delegateInterface);
  boolean isInstanceOf(Class delegateInterface);
}
```

Instances of Class represent classes and interfaces in a running Java application. Thus, delegates could have been retrieved by naming the desired interface or class. Any object could have been wrapped, but only so-called 'cognizant' delegates would have contained a back pointer allowing the discovery of the delegator from the delegate.

Enterprise JavaBeans and CORBA Components Enterprise JavaBeans [51] and CORBA Components [40] are enterprise component standards. Their focus is on containers providing such functionality as transactions, security, events, and persistence. However, they do not provide any special support for wrappers. Like their COM siblings, CORBA components can implement multiple interfaces (facets), but only navigation within components is provided by provide_name, the rough equivalent to COM's QueryInterface.

11 Conclusions

Late composition of software components from different vendors is the essence of component software, enabling component markets and flexible reuse. One form of late composition is the combination of features implemented by different vendors into object-aggregates that appear as single objects to their clients. Our analysis shows, that existing technologies fail to fully unlock this power.

To remedy the problem, we have proposed generic wrappers, a typed form of dynamic inheritance. We have analyzed the design space with respect to both type soundness and semantic intuition, desirability, and consistency with existing mechanisms, such as subclassing. One option is forwarding instead of delegation to loosen the coupling and, thereby, avoid the semantic fragile base class problem. Another option is the snappy assignment of the wrappee to facilitate modular semantic reasoning.

As a proof of concept, we have chosen a consistent set of desirable features for a concrete mechanism, which we added to Java. We have given a mechanized proof of type soundness for the extended language. Additionally, the formalization provides an operational semantics for Java extended with generic wrappers.

Acknowledgments David von Oheimb and Tobias Nipkow provided us with their formalization of Java and helped us with our extensions. We would like to thank Ralph Back, Dominik Gruntz, Cuno Pfister, and Clemens Szyperski for a number of fruitful discussions and comments.

References

[1] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Proceedings of OOPSLA '97*, pages 49–65. ACM Press, 1997.

[2] Pierre America. Designing an object-oriented programming language with behavioral subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop*, pages 60–90. LNCS 489, Springer Verlag, 1991.

[3] D. Ancona, G. Lagorio, and E. Zucca. Jam: A smooth extension of Java with mixins. Technical report, DISI, University of Genova, 1999.

[4] Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *Proceedings of ECOOP '99*, pages 43–66. LNCS 1628, Springer Verlag, 1999.

[5] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.

[6] Kim B. Bruce, Robert van Gent, and Angela Schuett. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proceedings of ECOOP '95*, pages 27–51. LNCS 952, Springer Verlag, 1995.

[7] Martin Büchi and Wolfgang Weck. Compound types for Java. In *Proceedings of OOPSLA '98*, pages 362–373. ACM Press, 1998. http://www.abo.fi/~mbuechi/publications/OOPSLA98.html.

[8] Laurence Cable and Graham Hamilton. *A Draft Proposal for a Object Aggregation/Delegation Model for Java and JavaBeans (Version 0.5)*. Sun Microsystems, April 1997.

[9] Luca Cardelli. Type systems. In *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997.

[10] Luca Cardelli and John C. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1(1):3–48, 1991.

[11] Craig Chambers. The Cecil language: Specification & rationale (version 2.1). Technical report, University of Washington, March 1997.

[12] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of OOPSLA '98*, pages 48–64. ACM Press, 1998.

[13] William Cook. A proposal for making Eiffel type-safe. In *Proceedings of ECOOP '89*, pages 57–70. Cambridge University Press, 1989.

[14] Stephen R. Davis. *AFC Programmer's Guide*. Microsoft Press, 1998. See also http://www.microsoft.com/java/afc/.

[15] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison Wesley, 1998. http://www.catalysis.org.

[16] Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.

[17] Kathleen Fisher and John C. Mitchell. Notes on typed object-oriented programming. In *Proceeding of Theoretical Aspects of Computer Software*, pages 844–885. LNCS 789, Springer Verlag, 1994.

[18] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 171–183. ACM Press, 1998.

[19] Michael Franz. The programming language Lag*oo*na: A fresh look at object-orientation. *Software – Concepts and Tools*, 18(1):14–26, March 1997.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[21] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[22] Mark Grand. *Patterns in Java*, volume 1. John Wiley & Sons, 1998.

[23] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Proc. 18th ACM Symp. Principles of Programming Languages*, pages 131–142. ACM Press, 1991.

[24] William Harrison, Harold Ossher, and Peri Tarr. Using delegation for software and subject composition. Technical Report RC-20946 (92722), IBM Research Division, T.J. Watson Research Center, August 1997.

[25] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of OOPSLA '91*, pages 271–285. ACM Press, 1991.

[26] Urs Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings of ECOOP '93*, pages 36–56. LNCS 707, Springer Verlag, 1993.

[27] Rosziati Ibrahim and Clemens Szyperski. Can the component object model (COM) be formalized? – The formal semantics of the COMEL language. In *Proceedings of IRW/FMP'98*. Technical Report TR-CS-98-09, The Australian National University, September 1998.

[28] Ralph Keller and Urs Hölzle. Binary component adaptation. In *Proceedings of ECOOP '98*, pages 307–329. LNCS 1445, Springer Verlag, 1998.

[29] Gregor Kiczales et al. Aspect-oriented programming. In *Proceedings of ECOOP '97*, pages 220–242. LNCS 1241, Springer Verlag, 1997.

[30] Günter Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings of ECOOP '99*. LNCS 1628, Springer Verlag, 1999.

[31] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of OOPSLA '86*, pages 214–223. ACM Press, 1986.

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

[33] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, second edition, 1992.

[34] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[35] Mira Mezini. Dynamic object evolution without name collisions. In *Proceedings of ECOOP '97*, pages 190–219. LNCS 1241, Springer Verlag, 1997.

[36] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *Proceedings of ECOOP '98*, pages 355–374. LNCS 1445, Springer Verlag, 1998.

[37] Anna Mikhajlova and Emil Sekerinski. Class refinement and interface refinement in object-oriented programs. In *Proceedings of FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, pages 82–101. LNCS 1313, Springer Verlag, 1997.

[38] NeXT Software, Inc. *Object-Oriented Programming and the Objective-C Language*. Addison-Wesley, 1993.

[39] Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170. ACM Press, 1998.

[40] Object Management Group. CORBA components, 1999. Revision February 15, 1999, formal document orbos/99-02-01, http://www.omg.org.

[41] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, pages 119–156. LNCS 1523, Springer Verlag, 1999.

[42] Geoff Outhred and John Potter. Extending COM's aggregation model. In *Component-Oriented Software Engineering Workshop (in conjunction with the Australian Software Engineering Conference)*, 1998.

[43] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828, Springer Verlag, 1994. http://www.cl.cam.ac.uk/Research/HVG/Isabelle/.

[44] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proc. 16th ACM Symp. Principles of Programming Languages*, pages 242–249. ACM Press, 1989.

[45] Dale Rogerson. *Inside COM*. Microsoft Press, 1996.

[46] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of OOPSLA '86*, pages 38–45. ACM Press, 1986.

[47] Patrick Steyaert and Wolfgang De Meuter. A marriage of class- and object-based inheritance without unwanted children. In *Proceedings of ECOOP '95*, pages 127–144. LNCS 952, Springer Verlag, 1995.

[48] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.

[49] Sun microsystems. Java platform, 1998. http://java.sun.com.

[50] Sun Microsystems, Inc. Java Beans, 1997. http://java.sun.com/beans/.

[51] Sun Microsystems, Inc. Enterprise JavaBeans, 1999. http://java.sun.com/products/ejb/.

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

[53] Clemens A. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[54] D. Ungar and R.B. Smith. Self: The power of simplicity. In *Proceedings of OOPSLA '87*, pages 227–241. ACM Press, 1987. Revised version in *Lisp and Symbolic Computation*, 4(3), 187–205, 1991.

Paper VI

Action-Based Concurrency and Synchronization for Objects

Ralph Back, Martin Büchi, and Emil Sekerinski

Originally published in: M. Bertran and T. Reus, editors, *Proceedings of the Fourth AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software (ARTS)*, volume 1231 of *Lecture Notes in Computer Science*, pages 248–262. Springer Verlag, May 1997.

Action-Based Concurrency and Synchronization for Objects

Ralph Back, Martin Büchi, Emil Sekerinski

Åbo Akademi University, Department of Computer Science
Lemminkäisenkatu 14A, 20520 Turku, Finland
{backrj, mbuechi, esekerin}@abo.fi

Abstract. We extend the Action-Oberon language for executing action systems with type-bound actions. Type-bound actions combine the concepts of type-bound procedures (methods) and actions, bringing object orientation to action systems. Type-bound actions are created at runtime along with the objects of their bound types. They permit the encapsulation of data and code in objects. Allowing an action to have more than one participant gives us a mechanism for expressing n-ary communication between objects. By showing how type-bound actions can logically be reduced to plain actions, we give our extension a firm foundation in the Refinement Calculus.

1 Introduction

Action-Oberon extends Oberon-2 [22] with actions for modeling parallel and distributed computations. The extension is based on the theory of action systems [6] and was proposed by Back and Sere [8] and implemented by Hedman [14]. An action system is a parallel or distributed program where parallel activity is described in terms of guarded actions. Enabled actions are executed atomically in a nondeterministic order to model parallelism. Atomicity of actions guarantees that a parallel execution of an action system gives the same results as a sequential nondeterministic execution in Action-Oberon (serializability).

Action-Oberon supports only plain actions, which may optionally be replicated over a constant range of integers. Plain actions describe updates to the variables visible in the module in which they are declared. The new type-bound actions combine the principles of type-bound procedures (methods) and actions. Bound to one or more types, they are created dynamically whenever an object of a bound type is created. They describe updates to the objects to which they are bound, as well as to the variables visible in their declaration module. Järvinen and Kurki-Suonio first proposed the marriage of object-oriented concepts and action systems in the DisCo language [16]. Their basic idea is the same as ours, but the actual definitions differ greatly due to the form of object orientation, the base language, the underlying logic, and the interpretation.

We have built an environment, in form of an Action-Oberon to Oberon-2 compiler and an associated runtime/simulation system under Oberon/F [21], which allows extended Action-Oberon programs to be executed. The environment helps to debug specifications and isolate critical properties worth formal proofs. Our environment is a play

ground for action systems and not an attempt to add concurrency to the Oberon language and/or system.

Section 2 presents the Action-Oberon base language, Sect. 3 explains the type-bound actions, Sect. 4 elaborates on the deactivation of type-bound actions and the deallocation of objects, Sect. 5 discusses inheritance of type-bound actions, Sect. 6 provides a foundation for type-bound action in the Refinement Calculus, Sect. 7 points to related work, and Sect. 8 draws the conclusions.

2 Action-Oberon Base Language

Oberon-2 [22] is the successor of Pascal and Modula-2. Modula-2 adds modularization to Pascal. Oberon-2 extends Modula-2 with object-oriented concepts in form of type extension on record types (subtyping/inheritance) as well as type-bound procedures (methods). Oberon-2 has been chosen as a base language because of its simplicity and its similarity to previously used ad-hoc notations of action systems.

Action-Oberon [8] adds actions and guarded procedures to Oberon-2. Action systems are represented by Oberon modules. All actions are executed repeatedly in a loop until all actions are disabled. Selection of enabled actions is nondeterministic and is not bound to a fairness pledge. The nondeterminism is demonic, in the sense that there is no way of influencing which action is chosen. The simulation environment provides, however, the possibility to install one's own scheduler or to manually select actions. The module body contains the initialization. Parallel composition of action systems corresponds to loading several modules into memory at once. Actions from all loaded modules are executed in one big loop; that is, they may be interleaved in any order. The

```
MODULE OneFish;

  CONST
    height=10;
    width=20;

  VAR
    x, y: INTEGER;
    right, up: BOOLEAN;

  ACTION MoveRight
    WHEN right & (x#width);
  BEGIN INC(x)
  END MoveRight;

  ACTION MoveLeft
    WHEN ~right & (x#0);
  BEGIN DEC(x)
  END MoveLeft;

  ACTION Bounce Right
    WHEN right & (x=width);
  BEGIN right:=FALSE
  END BounceRight;

  ACTION BounceLeft
    WHEN ~right & (x=0);
  BEGIN right:=TRUE
  END BounceLeft;

  ACTION MoveUp (* code *)
  ACTION Move Down (* code *)
  ACTION BounceUp (* code *)
  ACTION BounceDown (* code *)

BEGIN
  x:=0; y:=0; right:=TRUE; up:=TRUE
END OneFish.
```

Fig. 1. Screen saver OneFish

combined action system can only terminate when none of the loaded modules contains an enabled action.

Actions are declared like procedures without parameters. The guard of an action is given as a (side-effect free) boolean expression. Omitting the guard corresponds to an always enabled guard.

Throughout the paper we use the example of a fish screen saver. In our first version OneFish (Fig. 1), a single fish swims around the screen. The fish's current position is given by cartesian coordinates x (horizontal axis) and y (vertical axis). The fish is either moving right (right = TRUE) or left and either up (up = TRUE) or down. When it reaches a border it changes direction. Note that the lack of a fairness assumption means that the fish might only move along one axis, although the guard for moving along the other axis is infinitely often true.

Our screen saver is an example of an action system which never terminates. Hence, our interest does not lie in its input/output behavior, but in its possible traces (sequences of states).

Actions may optionally be replicated over one or more constant ranges of integers, generating a number of similar actions. We use this mechanism to add more fishes to our screen saver in the next version ManyFishes (Fig. 2). The action declaration ACTION MoveRight(i: 0..many-1) generates an action for each i between 0 and many-1. The replicator i can be used like a constant in the guard and body of the action.

Like actions, procedures may be protected by an optional guard [7]. If the evaluation of the guard of an action or the execution of its body would lead to a call of a disabled

```
MODULE ManyFishes;

  CONST
    many=5;
    height=10;
    width=20;

  VAR
    x, y: ARRAY many OF INTEGER;
    right, up: ARRAY many OF BOOLEAN;
    k: INTEGER;

  ACTION MoveRight(i: 0..many-1)
    WHEN right[i] & (x[i]#width);
  BEGIN INC(x[i])
  END MoveRight;

  ACTION MoveLeft(i: 0..many-1)
    WHEN ~right[i] & (x[i]#0);
  BEGIN DEC(x[i])
  END MoveLeft;

    ACTION Bounce Right(i: 0..many-1) (* code *)
    ACTION BounceLeft(i: 0..many-1) (* code *)
    ACTION MoveUp(i: 0..many-1) (* code *)
    ACTION Move Down(i: 0..many-1) (* code *)
    ACTION BounceUp(i: 0..many-1) (* code *)
    ACTION BounceDown(i: 0..many-1) (* code *)

  BEGIN
    FOR k:=0 TO many-1 DO
      x[k]:=k; y[k]:=k;
      right[k]:=TRUE; up[k]:=TRUE
    END
  END ManyFishes.
```

Fig. 2. Screen saver ManyFishes

procedure, the action is considered to be disabled. Note that no waiting for the guard to become true takes place, as is common with monitors or semaphores.

3 Type-Bound Actions

Being useful in certain cases, replication is awkward at best when we have to replicate several actions over the same range, as in Fig. 2. It provides no encapsulation of data and code within a single entity, (pseudo-) dynamic creation of new entities is cumbersome and error-prone – even if we added dynamically extendible arrays and variable replication ranges.

Thus, borrowing from the concept of object orientation, we add type-bound actions to Action-Oberon. The declaration ACTION (f: Fish) MoveRight leads to the dynamic creation of an action for each object of type Fish that we create. The bound variable f is called participant and may be used like a variable in the action. It corresponds to the receiver (self) of a type-bound procedure. Figure 3 gives our screen saver using type-bound actions.

Suppose we want to program some special behavior if two fishes meet. We can do this with ACTION (f1, f2: Fish) Meet (Fig. 4). We allow an action to have several

```
MODULE OOFishes;

  CONST
    height=10;
    width=20;
    many=5;

  TYPE
    Fish=POINTER TO FishDesc;
    FishDesc=RECORD
          x, y: INTEGER;
          right, up: BOOLEAN
    END;

  VAR
    fi: Fish;
    k: INTEGER;

  ACTION (f: Fish) MoveRight
    WHEN f.right & (f.x#width);
  BEGIN INC(f.x)
  END MoveRight;

  ACTION (f: Fish) MoveLeft
    WHEN ~f.right & (f.x#0);
  BEGIN DEC(f.x)
  END MoveLeft;

    ACTION (f: Fish) Bounce Right (* code *)
    ACTION (f: Fish) BounceLeft (* code *)
    ACTION (f: Fish) MoveUp (* code *)
    ACTION (f: Fish) Move Down (* code *)
    ACTION (f: Fish) BounceUp (* code *)
    ACTION (f: Fish) BounceDown (* code *)

    PROCEDURE CreateFish(VAR nf: Fish;
          x, y: INTEGER; right, up: BOOLEAN);
    BEGIN
      NEW(nf); nf.x:=x; nf.y:=y;
      nf.right:=right; nf.up:=up
    END CreateFish;

  BEGIN
    FOR k:=0 TO many-1 DO
      CreateFish(fi, k, k, TRUE, TRUE)
    END
  END OOFishes.
```

Fig. 3. Screen saver OOFishes

```
ACTION (f1, f2: Fish) Meet
  WHEN (f1.x=f2.x) & (f1.y=f2.y) & (f1#f2);
  VAR baby: Fish;
BEGIN
  (* do something: i.e.
     - change direction
     - create new fish
     - remove one of the fishes *)
END Meet;
```

Fig. 4. Type-bound action Meet

participants, i.e. f1 and f2, of various types. An instance of Meet will be created at runtime for each tuple of fishes, including double instantiations of the same fish. Hence, we have to explicitly strengthen the guard of Meet if we do not desire fishes to meet themselves (no aliasing). Actions with n participants lend themselves to symmetrically express n-ary communication, which is difficult in most other formalisms for $n > 2$.

Action names are treated as global identifiers of their modules. The complete EBNF for actions is given in Fig. 5.

```
Action       = ACTION [Participants] IdentDef [Replicators] [Guard] ";"
               DeclSeq [BEGIN StatementSeq] END identifier.
Participants = "(" VarDecl {";" VarDecl} ")".
Replicators  = "(" Repl {";" Repl} ")".
Repl         = identifier ":" ConstExpr ".." ConstExpr.
Guard        = WHEN Expr.
```

Fig. 5. EBNF of extended action declaration

If we add action Meet to OOFishes, it is not guaranteed that Meet will be executed whenever two fishes are at the same coordinates because the fishes' move actions are also enabled; their guards would have to be strengthened if desired.

We could imagine several behaviors if two fishes meet. We could for example change the direction of one fish or we could have them produce a baby fish by invoking NEW. Without object-orientation, but only plain replication, we would have to extend our data arrays and ranges separately to get the same effect.

4 Deactivation and Deallocation

Consider the case where we would want one fish to eat the other. How do we remove the dead fish from our system, that is how do we prevent it from participating in actions and how do we recycle its allocated memory? In Oberon-2, objects may be garbage collected if they are no longer referenced from one of the loaded modules. Having introduced type-bound actions, we cannot simply adopt this condition. Consider the case where we remove the last reference to an object. Should this object still be able to have one of its type-bound actions executed until it is garbage collected? If so, this

action could again set a pointer to the object and, herewith, revive it. On the other hand, if the object looses its eligibility to participate in actions with the removal of the last reference, we unnecessarily restrict the independence of our active objects and – in an extendible system where we often don't know the number of references to an object – loose control over the duration of an object's active life cycle. We can prevent an object from being collected by keeping a reference to it, but we cannot enforce an object to be disabled. Given the undesirable properties of the 'naturally' extended conditions for garbage collection, we enumerate the possible solutions which preserve pointer safety (no dangling pointers) and summarize their properties in Fig. 6:

1. An object may be collected after the last reference to it vanishes. Until then, it is eligible to participate in actions (as above).
2. An object may be collected after the last reference to it vanishes. An unreachable object cannot have one of its bound actions executed (as above).
3. An object may only be garbage collected if it is no longer referenced and none of its bound actions can ever be enabled again. Clearly, the second condition can in practice not be verified; hence, no automatic garbage collection can be implemented.
4. An object o is deallocated with a special command KILL(o). The precondition of KILL(o) is that o is the only reference to the object. As the declaration of type-bound actions is not restricted to their participants' declaration modules (see below), we stand the danger in an extendible system of prematurely killing an object. Additionally, an unreferenced object, which will never again have one of its bound actions enabled, cannot be deallocated and, therefore, creates a memory leak.
5. The eligibility of an object o to participate in actions is removed with a special command DEACTIVATE(o). Meanwhile, all references are kept. An object can be

Property	1	2	3	4	5
pointer-safety	yes	yes	yes	yes	yes
recycling of memory feasible	yes	yes	no	yes	yes
duality of constructor and destructor	yes	yes	yes	yes	no
manual disabling of actions without explicit flag	no	no	no	yes	yes
revival impossible	no	yes	yes	yes	yes
active lifespan independent of references	no	no	yes	yes	yes
execution model without reference count	no	no	yes	yes	yes
safe deallocation	yes	yes	yes	no	yes
safe disabling of actions	no[1]	no[1]	yes	no[2]	no[2]
avoids memory leaks	yes	yes	yes	no	no

[1]Due to dependency on references.
[2]Due to explicit termination with KILL, respectively DEACTIVATE.

Fig. 6. Properties of different deallocation schemes for objects with type-bound actions

garbage collected, if it has been deactivated and it is no longer referenced. We can interpret this as a special case of situation 3 where each object has a flag alive which is initially true, added as an implicit conjunct to each action guard, and can only be set to false by invoking DEACTIVATE. Creation and deactivation are not duals, as the latter only revokes an object's active behavior. As with solution 4, we have the problem of memory leaks.

We can model any of the above choices in the Refinement Calculus (Sect. 6). However, the computation model is simpler if an object's eligibility to participate in actions does not depend on it being referenced and the model must not include a reference count.

Going back to our consumed fish example, solutions 4 and 5 let us solve the problem without introducing a liveness flag and the corresponding guards in all actions. To keep the theory simple, make recycling of memory feasible, avoid cluttering of code by explicit flags, and prevent the introduction of aborts, we choose solution 5. If o has already been deactivated DEACTIVATE(o) is skip; DEACTIVATE(NIL) is abort. So far, the loss of duality between creation and destruction and the premature disabling of actions have not caused any problems in our examples.

The existence of both modules and classes (types and associated type-bound procedures/actions) in Action-Oberon provides for more compositionality. Modules are compile-time abstractions which provide for scoping and may contain several classes, the latter providing for extensibility and being a run-time abstraction that defines the structucture and behavior of objects. This separation of concerns allows objects to be bundled to components [23,21]. In Action-Oberon this gives us more compositionality on the module level by restricting the outside visibility of attributes and methods and still allows for privileged access between more closely related classes.

Unlike type-bound procedures, type-bound actions may be declared in any module where the participant types are visible, with access to the fields (instance variables) according to the Oberon-2 export/import visibility rules. This is needed for defining actions with participants stemming from different modules.

5 Inheritance of Type-Bound Actions

We can add some variety to our aquarium by defining special kinds of fishes. If we create a type Shark as subtype of Fish (Fig. 7), sharks have all actions of normal fishes bound to them plus possibly additional ones, i.e. ShowTeeth.

We might also want to override (redefine, extend) some actions for sharks, i.e. have sharks become hungrier whenever they move and eat another fish they meet when they are hungry enough. We could create an action ACTION (s: Shark; f: Fish) Meet, if we permitted overriding. This would immediately raise two problems. Consider a fish φ and a shark σ. Should we now have two actions Meet, the original one for (φ,σ) and the redefined for the reversed tuple (σ,φ) (Fig. 8 a)?

Secondly, this would require multiple dispatch, as actions can have several participants. Assume that we also define a subtype Piranha of Fish and override the meet action for ACTION (f: Fish; p: Piranha) Meet. Which action body would we choose for

```
TYPE
  Shark=POINTER TO SharkDesc;
  SharkDesc=RECORD (Fish)
    hunger: INTEGER
  END;

  ACTION (s: Shark) ShowTeeth;
  BEGIN (* show teeth *)
  END ShowTeeth;
```

Fig. 7. Subtype Shark

a tuple of a shark and a piranha (Fig. 8 b)? Requiring each combination of (normal) fishes, sharks and piranhas (Fig. 8 c) to be defined is not practical in a modular system where the different subtypes can be defined in different modules and where modules are not statically linked. The solution of Chambers and Leavens for multiple dispatch of methods [10], which requires a designated topmost module and flags errors of other modules when compiling this module, is against the spirit of open systems and independent extension [24]. While the first problem could be partly solved by introducing a special notation for symmetrical participants, the second one has no solution which is orthogonal to separate extension. Hence, we do not permit overriding of actions.

We could allow overriding for actions with only one participant as this does not require multiple dispatch and, therefore, does not create the problems described here. For simplicity's and orthogonality's sake we do not. Instead, we simulate overriding of type-bound actions by using overriding of type-bound procedures. Figure 9 shows how we override MoveRight by replacing the body with a single call to a type-bound procedure. In the implementation of MoveRight for Shark, s.MoveRight^ is a super-call to the overridden type-bound procedure which in our case is a call to MoveRight for Fish. Instead of replacing the complete guard with a call to a type-bound procedure, we choose in this example to explicitly state the conjuncts common to all extensions. This form of overriding requires explicit provisions to be made, i.e. introducing the

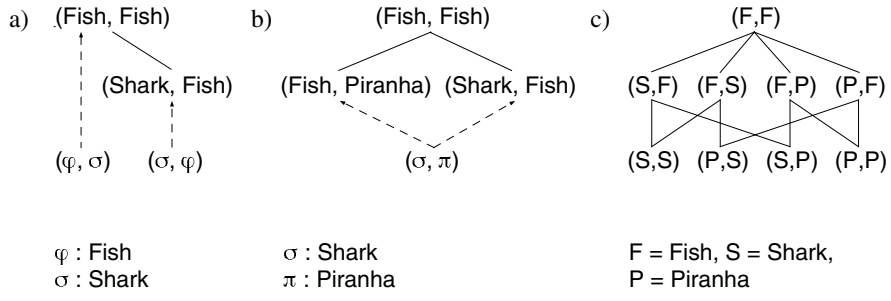


Fig. 8. Problems of overriding actions with more than one participant

```
PROCEDURE (f: Fish) MoveRight;
BEGIN INC(f.x)
END MoveRight;

PROCEDURE (s: Shark) MoveRight;
BEGIN s.MoveRight^; INC(s.hunger)
END MoveRight;

PROCEDURE (f: Fish) WantToMove(): BOOLEAN;
BEGIN RETURN TRUE
END WantToMove;

PROCEDURE (s: Shark) WantToMove(): BOOLEAN;
BEGIN RETURN s.hunger<10
END WantToMove;

ACTION (me: Fish) MoveRight
  WHEN me.right & (me.x#width) & me.WantToMove;
BEGIN me.MoveRight
END MoveRight;
```

Fig. 9. Simulating overriding for actions with only one participant

constant function WantToMove which we override for our lazy sharks which don't move if they are too hungry. However, there is also the argument that unless the designer has arranged for it, reuse and overriding never work in practice anyhow [17]. Overriding of type-bound procedures and dynamic type tests give all that is needed.

Another approach would be not to inherit actions, i.e. a Shark would not automatically have all actions defined for Fish. This would be orthogonal to polymorphism as actions are not called explicitly; it would not create the danger of invoking undefined actions as the method deletion mechanism of Smalltalk does. Because inheritance of type-bound actions has proved to be desirable in practice, we have adopted it in Action-Oberon. E.g. without inheritance of type-bound actions, we would have to explicitly give all the move actions for sharks.

6 A Semantics for Type-Bound Actions

In this section we give a formal semantics to type-bound actions by reducing them to plain actions. We have four levels to express an action system: Action-Oberon with type-bound actions, Action-Oberon without type-bound actions, action systems, and the Refinement Calculus. The translation from Action-Oberon without type-bound actions to action systems is given by Back and Sere [8], the translation from action systems to the Refinement Calculus and the mathematical treatment of action systems is due to Back, Kurki-Suonio and von Wright [6,3,4,9]. Figure 10 shows these three levels for a sample program. A plain action ACTION A WHEN G; BEGIN S END A; translates to the guarded statement $G \rightarrow S$ which is only enabled if G holds. An Action-Oberon module with several actions translates to a do loop with a demonic choice between the actions. On the Refinement Calculus level, a predicate in square brackets denotes a

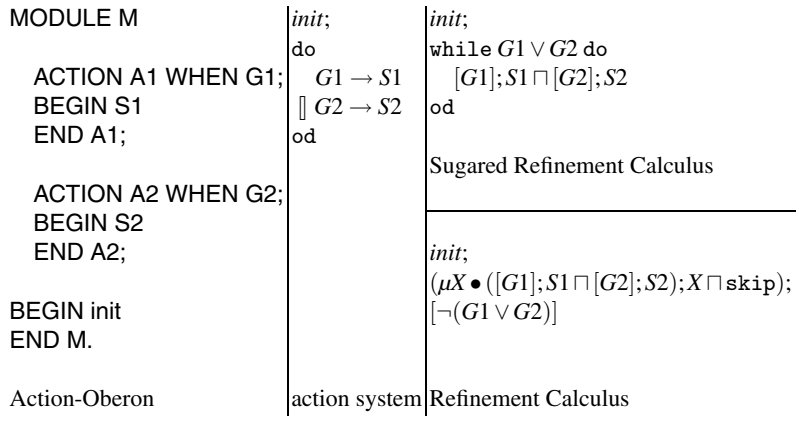


Fig. 10. Translation of an Action-Oberon module

guard, which is equivalent to `skip` if the predicate holds and `magic` otherwise. The meet ($\sqcap$) denotes the demonic choice, and μ stands for the least fixpoint. The refinement calculus level only applies to the input/output, but not to the trace semantics.

A type-bound action is defined as follows: There is only one instance of each type-bound action. The guard implicitly stands for 'there exists a tuple that satisfies the stated guard' and the first statement demonically (nondeterministically) chooses one such tuple. We first use this intuition to sketch a simulation of type-bound actions in Action-Oberon without type-bound actions. Thereafter, we also provide a direct translation to action systems by formalizing this idea.

We keep a data structure of all types and their inheritance relation as well as a set of all active objects of each type. We turn each type-bound action into a plain action of the same name by replacing the guard by a traversal function which returns true, if it finds a tuple of possible participants from the respective sets of active participants. We add an additional first statement, which demonically chooses one possible tuple and assigns it to the participant variables. This is, up to optimizations, how the current implementation works.

Alternatively, we can map type-bound actions directly to their action system equivalent. Using the Refinement Calculus typed higher-order logic, we denote the record types by $R_1,\ldots,R_n$ and the corresponding pointer types by $P_1,\ldots,P_n$. Records are represented as tuples and type extension ([width] subtyping/inheritance) corresponds to tuple extension. We model the heap as a partial function (functional relation) from pointers to records and use a boolean flag for each record to indicate whether an object is active:

$$heap : P_1 + \ldots + P_n + \mathtt{NIL} \nrightarrow (R_1 + \ldots + R_n) \times Bool$$

Initially, *heap* is empty. To manipulate the sum types (disjoint union), we define families of injection functions $\mathtt{in_i} : P_i \rightarrow P_1 + \ldots + P_n + \mathtt{NIL}$ (also for records), projection functions $\mathtt{out_i} : P_1 + \ldots + P_n + \mathtt{NIL} \rightarrow P_i$, and corresponding discriminator functions

$\mathtt{is_i} : R_1 + \ldots + R_n \to \mathit{Bool}$. NIL is the one-element type $\{\mathtt{nil}\}$ for modeling NIL pointers, for which we get the following invariant: $\mathtt{in_{NIL}}\ \mathtt{nil} \notin \mathtt{dom}\ \mathit{heap}$. We also define $\mathtt{is_i^*}$ to be the transitive closure of $\mathtt{is_i}$ with respect to the direct subtype/inheritance relation ($R_j <: R_i \stackrel{\text{def}}{=} R_j$=RECORD ($R_i$) ... END):

$$\mathtt{is_i^*}\ r \stackrel{\text{def}}{=} (\mathtt{is_i}\ r) \vee (\exists j.R_j <: R_i \wedge \mathtt{is_j^*}\ r)$$

In fact, $\mathtt{is_i^*}$ corresponds to Oberon's IS statement: r IS $R_i \equiv \mathtt{is_i^*}\ r$. Furthermore, we define dom to return the domain of a partial function. We define fst and snd as the first and second projections of a tuple, and $\mathtt{prj_k}$ as the kth projection.

We express NEW as a family of predicate transformers. For pointer p of Action-Oberon type P_i we get:

$$\begin{aligned} \mathit{NEW}_i(p) \stackrel{\text{def}}{=} & (\sqcap x : P_i | \mathtt{in_i}\ x \notin \mathtt{dom}\ \mathit{heap} \bullet p := \mathtt{in_i}\ x); \\ & \mathit{heap} := \mathit{heap} \cup \{p \mapsto (\mathtt{in_i}\ 0_{R_i}, T)\} \end{aligned}$$

The only change to NEW with respect to Oberon-2 is the initialization of the boolean flag indicating that the object is active: First we demonically – indicated by the meet – choose a free location in the heap and assign it to p and then augment the partial heap function by the mapping from p to the 0-record with unspecified values of the referenced type. To exhibit the fact that DEACTIVATE only changes the activity flag of the referenced record, we give it in terms of a relational override $\lhd$ defined for relations r, s:

$$r \lhd s \stackrel{\text{def}}{=} \{(x \mapsto y) \in r | x \notin \mathtt{dom}\ s\} \cup s$$

$$\begin{aligned} \mathit{DEACTIVATE}(p) \stackrel{\text{def}}{=} & \{p \in \mathtt{dom}\ \mathit{heap}\}; \\ & \mathit{heap} := \mathit{heap} \lhd \{p \mapsto (\mathtt{fst}\ (\mathit{heap}\ p), F)\} \end{aligned}$$

DEACTIVATE asserts – indicated by the braces – that p is a valid pointer and then sets the activity flag of the referenced record to false. We can now give the translation for a type-bound action:

$$\begin{aligned} & \text{ACTION (p: } P_i\text{) A WHEN G; BEGIN S END A;} \\ \stackrel{\text{def}}{=} & (\sqcap q \in \mathtt{dom}\ \mathit{heap} | \mathtt{is_i^*}\ (\mathtt{fst}\ (\mathit{heap}\ q)) \wedge \mathtt{snd}\ (\mathit{heap}\ q) \bullet \\ & \mathtt{begin\ var}\ p : P_1 + \ldots + P_n + \mathtt{NIL}; p := q; G \to S\ \mathtt{end}) \end{aligned}$$

The quantified variable q is a logical variable, whereas p, which is only visible within the block bracketed by begin and end, is a program variable which may also appear on the left hand side of assignments. The action is enabled if there is at least one possible participant for which G holds. More formally, the guard and the body of a predicate transformer S are defined as $gA = \neg \mathtt{wp}(A, \mathit{false})$ and $sA = \{gA\}; A$. This allows us to view a type-bound action as a guarded statement. The generalization to actions with more than one participant is straightforward.

A statement containing a record field access is the sum of predicate transformers over all record, respectively pointer types. Let for example p be a pointer to a record

whose kth field is x: INTEGER and let h: INTEGER. The assignment h:=p.x in Action-Oberon then corresponds to the following predicate transformer:

$$
\begin{array}{l}
\quad \mathsf{h{:=}p.x} \\
\stackrel{\text{def}}{=} \{p \in \mathtt{dom}\, \mathit{heap}\}; \\
\quad (h := \mathtt{prj_k}\,(\mathtt{out_1}\,(\mathtt{fst}\,(\mathit{heap}\,p)))) + \\
\qquad \ldots + \\
\qquad (h := \mathtt{prj_k}\,(\mathtt{out_n}\,(\mathtt{fst}\,(\mathit{heap}\,p)))) + \\
\qquad \mathtt{abort}
\end{array}
$$

The properties of the summation operator for predicated transformers were explored by Back and Butler [5], based on Nauman's [20] and Martin's [19] category theoretical considerations. Late binding of type-bound procedures is modeled analogously.

Hence, we can use type-bound actions in our extended Action-Oberon programs and use their unsugared form for reasoning in the Refinement Calculus or use the above correspondence to give a sugared form of the relevant inference rules.

Interestingly, there are two other 'natural' explanations for type-bound actions which give identical semantics:

1. Whenever an object is created, the set of actions is augmented by the corresponding bound actions. Dually, when an object's eligibility to participate in actions is revoked, the associated actions are removed. This model is possible as the action system formalism does not require the set of actions to be constant or finite.
2. We start with an infinite number of objects in a special not-yet-created state and an infinite number of corresponding actions the guards of which test that all participant are in the created state. Creating an object corresponds to changing its state to created.

7 Related Work

Our work on Action-Oberon was inspired by the original Action-Oberon and DisCo. Other related work includes Unity, its successor Seuss, IP, and a number of frameworks for active objects.

DisCo [16] first introduced type-bound actions. DisCo's concept of 'inheritance' corresponds to having a field of the inherited type in our terminology; hence, there is no overriding. DisCo does not have any type-bound procedures, making it lack any form of dynamic binding. The language contains no loops or recursion. Guarded procedures do not exist.

In Unity [11] 'actions' are restricted to (quantified) multiple deterministic assignments, and the set of actions must be finite and constant; on the other hand, it has fairness and progress properties. We are not aware of anything like type-bound actions in Unity. Due to the lack of nondeterminism in assignments and the restriction to a finite constant set of 'actions' none of our explanations for type-bound actions could be applied in Unity.

Seuss [12] gives the notion of boxes which correspond to our object types and clones which are instantiations thereof. Boxes have local variables (non-exported instance variables), actions (type-bound actions), and procedures (type-bound procedures) which

may also be partial (guarded). The set of clones is static. By also requiring all actions to terminate, Seuss can provide fairness. As all actions reside within one clone, there is no possibility to create an action with more than one participant. As in our model, an action calling a disabled procedure fails. However, the disabled procedure can still change the state of the callee, by executing the code associated to a 'negative alternative'.

In IP [13], processes are the main structuring elements rather than implementation details arising from the target machine's architecture. Multiparty interactions provide for communication, synchronization, and agreement. Processes can only access non-local variables within interactions. Interprocess communication abstraction is realised in form of teams which facilitate dynamic process creation. Teams are often used analogously to type-bound actions; roles in teams correspond to participants. Conflict propagation in coordinated enrolement causes lookahead computation similar to guarded procedures.

Formalisms and languages for active objects are characterized by different objects executing in parallel. New objects can be created dynamically. Objects communicate by message passing, which is the only way to have an object do something. Generally, objects do not contain any actions. Triggered procedures in object-oriented databases are a notable exception to this rule; however, the triggered (guarded) procedures are usually executed as part of the transaction setting off the trigger. Due to the lack of actions, the condition for garbage collection is simple reachability, respectively knowledge of an object's mail address. *N*-ary communication between objects can generally not be expressed in a symmetric fashion. Hewitt's actor model [15,1] was the first formal model of active objects. More recently, CCS and the π-calculus have been used to give a semantics to members of the POOL family [2,18,25].

8 Conclusions

We have extended the Action-Oberon language with type-bound actions encapsulating both data and actions in objects by combining the principles of object-orientation and action systems. Actions with n participants provide a symmetrical mechanism to express n-ary communication between objects. Of the solutions for disabling an object as participant of any action and recycling its allocated memory, we found the explicit DEACTIVATE command to have the most desirable properties. Due to the conflict between multiple dispatch and independent extensibility, overriding of type-bound actions is prohibited. Overriding of type-bound procedures and dynamic type tests provide for selective overriding. By reducing type-bound actions to plain actions they are given a formal semantics in the Refinement Calculus framework which allows for concise mathematical reasoning.

We are interested in increasing our collection of examples manifesting the usefulness of type-bound actions. Simulation of mechanical systems is a very promising application. We are also interested in adding fairness and/or priorities to Action-Oberon. Additionally, we are trying to show the value of object-encapsulation for atomicity refinement of actions and for synchrony-loosening refinements.

We would like to thank Wolfgang Weck, Jim Grundy, Kaisa Sere, and Philipp Heuberger for a number of clarifying and fruitful discussions on the topic of this paper.

References

1. Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
2. Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.
3. R. Back. Refinement calculus, part II: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness. Proceedings*. LNCS 430, Springer Verlag, 1990.
4. R. Back. Refinement of parallel and reactive programs. In M. Broy, editor, *Program Design Calculi*, NATO ASI Series, pages 73–92. Springer-Verlag, 1993.
5. R. Back and M. Butler. Exploring summation and product operators in the refinement calculus. Technical Report on Computer Science & Mathematics, Ser. A. No 152, Åbo Akademi, 1994.
6. R. Back and R. Kurki-Suonio. Distributed co-operation with action systems. *ACM Transactions on Programming Languages and Systems 10:513–554*, 1988.
7. R. Back and K. Sere. Action systems with synchronous communication. In *IFIP TC 2 Working Conference on Programming Concepts, Methods and Calculi (PROCOMET '94)*, pages 107–126. Elsevier, 1994.
8. R. Back and K. Sere. From action systems to modular systems. In *Proceeding of Formal Methods Europe '94*. LNCS 873, Springer Verlag, 1994.
9. R. Back and J. von Wright. Trace refinement of action systems. In *CONCUR 94*, pages 367–384. LNCS 836, Springer Verlag, 1994.
10. Craig Chambers and Gary T. Leavens. Type checking and modules for multi-methods. Technical Report #95-19, Iowa State University, August 1995.
11. K. M. Chandy and J. Misra. *Parallel Program Design – A Foundation*. Addison Wesley, 1988.
12. K.M. Chandy. A discipline of multiprogramming. Available from the PSP group's ftp site ftp://ftp.cs.utexas.edu/pub/psp/seuss/discipline.ps.Z, June 1996.
13. N. Francez and I. Forman. *Interacting Processes: A Multiparty Approach to Coordinated Distributed Programming*. ACM Press, 1996.
14. Eric J. Hedman. Action-Oberon. Master's thesis, Åbo Akademi University, 1995.
15. Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3), 1977.
16. H.-M. Järvinen and R. Kurki-Suonio. DisCo specification language: Marriage of action and objects. In *Proceedings of 11th International Conference on Distributed Computing Systems*, pages 142–151, Arlington, Texas, 1991. IEEE Computer Society Press.
17. R.E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June 1:2 1988.
18. Cliff B. Jones. A π-calculus semantics for an object-based design notation. In *Proceedings of CONCUR 93*, pages 158–172. LNCS 715, Springer Verlag, 1993.
19. C.E. Martin. *Preordered Categories and Predicate Transformers*. PhD thesis, Programming Research Group, Oxford University, 1991.
20. D.A. Naumann. *Two-Categories and Program Structure: Data Types, Refinement Calculi, and Predicate Transformers*. PhD thesis, University of Texas at Austin, 1992.
21. Oberon microsystems, Inc. *Oberon/F*. http://www.oberon.ch, 1995.
22. P. Mössenböck and N. Wirth. The programming language Oberon-2. *Structured Programming 12:179–195*, 1991.
23. Clemens A. Szyperski. Import is not inheritance – Why we need both: Modules and classes. In *Proceedings of ECOOP 92*, pages 19–32. LNCS 615, Springer Verlag, 1992.

24. Clemens A. Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings of the 19th Austalasian Computer Science Conference, Melbourne*, 1996.
25. D.J. Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, 1995.

Paper VII

Refining Concurrent Objects

Martin Büchi and Emil Sekerinski

Conditionally accepted to Fundamenta Informaticae with request for changes.

Refining Concurrent Objects

Martin Büchi[1] and Emil Sekerinski[2]

[1] Turku Centre for Computer Science,
Lemminkäisenkatu 14A, 20520 Turku, Finland;
Martin.Buechi@abo.fi
[2] McMaster University,
1280 Main Street West, Hamilton, Ontario, Canada, L8S4K1;
emil@mcmaster.ca

Abstract. We study the notion of class refinement in a concurrent object-oriented setting. Our model is based on a combination of action systems and classes. An action system describe the behavior of a concurrent, distributed, or interactive system in terms of the atomic actions that can take place during the execution of the system. Classes serve as templates for creating objects. To express concurrency with objects, we add actions to classes.
We define class refinement based on trace refinement of action systems. Additionally, we give a simulation-based proof rule. We show that the easier to apply simulation rule implies the trace-based definition of class refinement.
Class refinement embraces algorithmic refinement, data refinement, and atomicity refinement. Atomicity refinement allows us to split large atomic actions into several smaller ones. Thereby, it paves the way for more parallelism. We investigate the special case of atomicity refinement by early returns in methods.

1 Introduction

For the development of larger programs, a recommended practice is to separate a concise but precise specification of what the program should do from a possibly involved and detailed implementation. We view the specification as an abstract program P and the implementation as a concrete program Q. The task of ensuring that the implementation satisfies the specification is eased by introducing intermediate programs such that each program is a *refinement* of the previous one, formally expressed as:

$$P = P_0 \sqsubseteq P_1 \sqsubseteq P_2 \sqsubseteq \ldots \sqsubseteq P_n = Q$$

In *algorithmic refinement* steps abstract (or more abstract) statements are replaced by concrete (or more concrete) statements whereas in *data refinement* steps abstract (or more abstract) data structures are replaced by concrete (or more concrete) data structures. For the development of concurrent programs, in *atomicity refinement* steps sequential (or less concurrent) parts are replaced by by concurrent (or more concurrent) ones.

These general principles are applied here to classes. For example, a file can be specified as an object of a class whose state is a sequence and a current position and whose read and write operations access the sequence at the current position. A typical implementation of this class would use a cache for storage and would process write operations

in the background, hence changing the state space and introducing concurrency. In any case, the illusion to the user of the write operation is maintained that the operation is executed *atomically*. In this example, concurrency is introduced in the implementation for allowing a better utilization of resources, which is an aspect we are interested in without formalizing it.

In this paper we propose a formal model for objects with attributes and methods, with self- and super-calls in methods, classes with inheritance, and action-based concurrency. Objects have actions which, as long as they are enabled, may execute and change the object's state while other parts of the program are in progress. As in class-based programming languages, classes serve as templates for creating objects and inheritance is understood as a mechanism for modifying classes.

The notion of class refinement expresses that an object of the refining class behaves as an object of the refined class. Class refinement between two classes is defined in terms of the observable traces of programs with instances of those classes. We give a simulation condition for establishing class refinement by using a relation between the attributes of those classes. As the main result, we prove that simulation by relation implies class refinement in a setting with dynamic object structures.

The proposed class refinement extends class refinement as defined for sequential objects [27, 26] by adding actions to classes. Class refinement has also been studied under the name behavioral subtyping in less formal settings guaranteeing only partial correctness by America [2] and by Liskov and Wing [24]. Different models for classes and objects have been proposed [1]. We extend the model of classes as self-referential structures with a delayed taking of the fixed point of [31, 16].

The action system model for parallel, distributed, and reactive systems was proposed by Back and Kurki-Suonio [7, 8]. The same basic approach has later been used in other models for distributed computing, notably UNITY [14] and TLA [21].

An action system describes the behavior of a concurrent system in terms of the atomic actions that can take place during the execution of the system. Action systems allow a succinct description of the overall behavior of a system. Furthermore, action-based approaches do not force us to fix the flow of control where doing so is unnecessary for an abstract specification (see e.g. [14]). Action systems can be used to express various forms of communication, e.g. shared variable, rendez-vous, and bounded channels, as well as different interaction mechanisms, e.g. semaphores, critical regions, and 4-phase handshake [8, 14].

Back and Sere [9] have added procedures to action systems. They, as well as Sere and Waldén [30] and Bonsangue et al [13], have also studied input/output refinement of action systems with methods, which is similar to our classes after self- and super-references have been resolved. Using trace refinement, we extend those results to reactive behavior and handle non-terminating systems.

The action system model has been extended with different notions of objects. Järvinen and Kurki-Suonio [18] used aggregation rather than inheritance and overriding, based their semantics on TLA, and concentrated on superposition refinement. Back et al [6] concentrated on the design of a language. Bonsangue et al [13] developed a less formal model with an action-system-per-object semantics. Seuss [28] also combines

objects with action-based concurrency. The catch in Seuss is that the set of objects (called clones) is static.

Atomicity refinement has first been proposed by Lipton [23]. Back studied input/output behavior preserving atomicity refinement in action systems [4, 5]. Sere and Waldén [30] and Bonsangue et al [13] have extended this to procedures and methods, still refining only input/output behavior. Lamport and Schneider [22] and Cohen and Lamport [15] have studied atomicity refinement in TLA considering liveness properties beyond termination. De Bakker and de Vink [17] give an overview of atomicity refinement in process algebras and Petri nets. The idea of an early return, or release, statement has been proposed by Jones [19, 20] in a framework with explicit constructs for parallelism.

Our calculus for concurrent objects is meant to provide a design notation for programs to be implemented in concurrent object-oriented languages, such as POOL, Modula-3, and Java. Programs can be expressed more abstractly than in those languages. The synchronization and communication mechanisms of these programming languages can be formally introduced in refinements.

Outline. In Section 2 we review the fundamentals of statements and action systems. Section 3 introduces classes with attributes, methods, and actions as well as local object creation, inheritance, and self- and super-references in methods and actions. Section 4 defines class refinement in terms of the externally observable behavior, gives a condition for class simulation using a relation, and proves that class simulation implies class refinement for a system with a single object of a given class. Section 5 introduces dynamic object structures and extends the discussion of class refinement and class simulation to that setting. In Section 6 we study early returns as a special case of atomicity refinement. Finally, Section 7 draws the conclusions.

2 Statements and Action Systems

The refinement calculus, which provides the foundation for our work, is due to Back, Morgan, and von Wright [3, 29, 11]. We review the fundamentals of statements defined by predicate transformers following [11] and of action systems following [10].

2.1 Statements

State predicates of type $\mathcal{P}\Sigma$ are functions from elements of type Σ to *Bool*. Relations of type $\Delta \leftrightarrow \Omega$ are functions from Δ to (state) predicates over Ω. Predicate transformers of type $\Delta \mapsto \Omega$ are functions from predicates over Ω (the postconditions) to predicates over Δ (the preconditions):

$$\begin{array}{lcl} \mathcal{P}\Sigma & \widehat{=} & \Sigma \to \mathit{Bool} \\ \Delta \leftrightarrow \Omega & \widehat{=} & \Delta \to \mathcal{P}\Omega \\ \Delta \mapsto \Omega & \widehat{=} & \mathcal{P}\Omega \to \mathcal{P}\Delta \end{array}$$

On predicates, conjunction $\wedge$, disjunction $\vee$, implication $\Rightarrow$, and negation $\neg$ are defined by the pointwise extension of the corresponding operations on *Bool*. The entailment ordering $\leq$ is defined by universal implication. The predicates *true* and *false* represent the

universally true, respectively false predicates. On relations, we use union $\cup$, intersection $\cap$, relational composition $\circ$, and the relational image $R\ [p]$ of a predicate p, defined by $R\ [p]\ y\ \widehat{=}\ (\exists x \bullet R\ x\ y \wedge p\ x)$. The identity relation is denoted by Id.

Statements are defined by predicate transformers because only their input/output behavior is of interest. Thus, for statement S and predicate q we have $S\ q = wp(S,q)$, where wp is in Dijkstra's notation the weakest precondition of statement S to establish postcondition q. More precisely, we identify program statements with monotonic predicate transformers, i.e. predicate transformers S for which $p \leq q \Rightarrow S\ p \leq S\ q$.

The sequential composition of predicate transformers S and T is defined by their functional composition:

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

The identity on predicate transformers is denoted by $skip$. The guard $[p]$ skips if p holds and "miraculously" establishes any postcondition if p does not hold. The guard $[false]$ is called $magic$. The assertion $\{p\}$ skips if p holds and establishes no postcondition if p does not hold (the system crashes). The (never holding) assertion $\{false\}$ is called $abort$:

$$\begin{array}{lcllcl} skip\ q & \widehat{=} & q & [p]\ q & \widehat{=} & p \Rightarrow q \\ magic\ q & \widehat{=} & true & \{p\}\ q & \widehat{=} & p \wedge q \\ abort\ q & \widehat{=} & false & & & \end{array}$$

The demonic (nondeterministic) choice $\sqcap$ establishes a postcondition only if both alternatives do. The angelic choice $\sqcup$ establishes a certain postcondition if at least one alternative does. The relational updates $[R]$ and $\{R\}$ both update the state according to relation R. If several final states are possible, then $[R]$ chooses one demonically and $\{R\}$ chooses one angelically. If R is of type $\Delta \leftrightarrow \Omega$, then $[R]$ and $\{R\}$ are of type $\Delta \mapsto \Omega$:

$$\begin{array}{lcllcl} (S \sqcap T)\ q & \widehat{=} & (S\ q) \wedge (T\ q) & [R]\ q\ \delta & \widehat{=} & (\forall \omega \bullet R\ \delta\ \omega \Rightarrow q\ \omega) \\ (S \sqcup T)\ q & \widehat{=} & (S\ q) \vee (T\ q) & \{R\}\ q\ \delta & \widehat{=} & (\exists \omega \bullet R\ \delta\ \omega \wedge q\ \omega) \end{array}$$

We generalize the binary demonic choice to the choice among a fixed set of statements:

$$(\sqcap i \in I \bullet S)\ q\ \widehat{=}\ (\forall i \in I \bullet S\ q)$$

As a variant, we allow the choice to be restricted by a state predicate:

$$(\sqcap i \mid p \bullet S)\ \widehat{=}\ (\sqcap i \bullet [p]\ ;\ S)$$

All of the above constructs are monotonic. The universally and the positively conjunctive predicate transformers are two important subsets of the monotonic predicate transformers. Let q_i for some index set I and $i \in I$ form a set of predicates. If

$$S(\forall i \in I \bullet q_i) = (\forall i \in I \bullet S\ q_i)$$

holds for any index set I, then S is universally conjunctive. If the condition holds for nonempty sets I, then S is positively conjunctive. Any universally conjunctive predicate transformer is equal to $[R]$ for some relation R. Any positively conjunctive predicate

transformer is equal to $\{p\}\ ;\ [R]$ for some predicate p and some relation R. For example, for any predicate transformers S, T, U we have that

$$(S \sqcap T)\ ;\ U = (S\ ;\ U) \sqcap (T\ ;\ U)$$

but only if U is positively conjunctive we have also that:

$$U\ ;\ (S \sqcap T) = (U\ ;\ S) \sqcap (U\ ;\ T)$$

Other statements can be defined in terms of the above ones, for example the guarded statement $p \rightarrow S \ \widehat{=}\ [p]\ ;\ S$ and the conditional:

$$\textbf{if } p \textbf{ then } S \textbf{ else } T \textbf{ end} \ \widehat{=}\ (p \rightarrow S) \sqcap (\neg p \rightarrow T)$$

The enabledness domain (guard) of a statement S is denoted by $grd\ S$ and its termination domain by $trm\ S$:

$$grd\ S \ \widehat{=}\ \neg S\ false \qquad trm\ S \ \widehat{=}\ S\ true$$

For example, $grd\ (p \rightarrow S) = p \wedge grd\ S$ and $trm\ (\{p\}\ ;\ [R]) = p$.

Refinement. The reflexive and transitive refinement ordering $\sqsubseteq$ is defined by universal entailment:

$$S \sqsubseteq T \ \widehat{=}\ \forall q \bullet S\ q \leq T\ q$$

The loop **do** S **od** executes its body as long as it is enabled. This is defined by taking the least fixed point of the function $F = \lambda X \bullet S\ ;\ X \sqcap [\neg\ grd\ S]$. Sequential composition and nondeterministic choice are monotonic in both operands, so a least fixed point $\mu\ F$ exists and is unique:

$$\textbf{do } S \textbf{ od} \ \widehat{=}\ \mu\, X \bullet S\ ;\ X \sqcap [\neg\ grd\ S]$$

The loop **while** p **do** B is defined as **do** $p \rightarrow B$ **od**, provided that B is always enabled, i.e. $grd\ B = true$.

Data refinement $S \sqsubseteq_R S'$ generalizes (plain) algorithmic refinement by relating the initial and final state spaces of $S : \Sigma \mapsto \Sigma$ and $S' : \Sigma' \mapsto \Sigma'$ with a relation $R : \Sigma \leftrightarrow \Sigma'$:

$$S \sqsubseteq_R S' \ \widehat{=}\ S\ ;\ [R] \sqsubseteq [R]\ ;\ S'$$

Data refinement $S \sqsubseteq_R S'$ can be equivalently defined by $\{R^{-1}\}\ ;\ S \sqsubseteq S\ ;\ \{R^{-1}\}$, where R^{-1} is the relational inverse of R. Algorithmic refinement is a special case of data refinement with the identity relation.

Program Variables. Typically the state space is made up of a number of program variables. Thus the state space is of the form $\Gamma_1 \times \ldots \times \Gamma_n$. States are tuples $(x_1, \ldots, x_n)$. The variable names serve for selecting components of the state. For example, if $x : \Gamma$ and $y : \Delta$ are the only program variables, then the assignment $x := e$ updates x and leaves y unchanged:

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

The nondeterministic assignment $x :\in q$ assigns x an arbitrary element of the set q:

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

The declaration of a local variable $y : \Delta$ with initialization predicate yi extends the state space and sets y to any value for which $yi\ y$ holds. A block construct allows us to temporarily extend the state space with local variables, execute the body of the block on the extended state space, and reduce the state space again:

$$\begin{array}{lcl} \textbf{var}\ y \mid yi \bullet S & \widehat{=} & \textbf{enter}\ y \mid yi\ ;\ S\ ;\ \textbf{exit}\ y \\ \textbf{enter}\ y \mid yi & \widehat{=} & [R] \quad \text{where} \quad R\ x\ (x',y') \equiv x = x' \wedge yi\ y' \\ \textbf{exit}\ y & \widehat{=} & [R] \quad \text{where} \quad R\ (x,y)\ x' \equiv x = x' \end{array}$$

Leaving out the initialization predicate as in $\textbf{var}\ y \bullet S$ means initializing the variable arbitrarily, $\textbf{var}\ y \mid \mathit{true} \bullet S$. Where necessary, we also explicitly indicate the type Δ of the new variable as in $\textbf{var}\ y : \Delta$. Since $\Gamma \times (\Delta \times \Omega)$ is isomorphic to $(\Gamma \times \Delta) \times \Omega$, we can always find functions which transform an expression of one to the other type. Hence we simply write $\Gamma \times \Delta \times \Omega$. For example, if $\Gamma = \Gamma_1 \times \cdots \times \Gamma_n$ then S above would have the type $\Gamma_1 \times \cdots \times \Gamma_n \times \Delta \mapsto \Gamma_1 \times \cdots \times \Gamma_n \times \Delta$. Assuming that variable names select the correct state space component, we can also commute state space components.

Product Statements. For predicates $q_1 : \mathcal{P}\Sigma_1$ and $q_2 : \mathcal{P}\Sigma_2$ the product $q_1 \times q_2$ of type $\mathcal{P}(\Sigma_1 \times \Sigma_2)$ is defined as $(q_1 \times q_2)\ (\sigma_1, \sigma_2) \;\widehat{=}\; q_1\ \sigma_1 \wedge q_2\ \sigma_2$. For predicate transformers $S_1 : \Delta_1 \mapsto \Omega_1$ and $S_2 : \Delta_2 \mapsto \Omega_2$, their product $S_1 \times S_2$ is a predicate transformer of type $\Delta_1 \times \Delta_2 \mapsto \Omega_1 \times \Omega_2$ which corresponds to the simultaneous execution of S_1 and S_2:

$$(S_1 \times S_2)\ q\ (\delta_1, \delta_2) \;\widehat{=}\; \exists q_1, q_2 \mid q_1 \times q_2 \leq q \bullet S_1\ q_1\ \delta_1 \wedge S_2\ q_2\ \delta_2$$

Intuitively, this means that $S_1 \times S_2$ establishes the postcondition $q : \mathcal{P}(\Omega_1 \times \Omega_2)$ from initial state (δ_1, δ_2), if there is a "rectangular" subset $q_1 \times q_2$ of q such that independently S_1 establishes q_1 from δ_1 and S_2 establishes q_2 from δ_2 [12].

Two statements S and T over the same state space are *independent* if they operate on different components of the state space (disjoint variables). This implies that there must exist S' and T' such that $S = S' \times \mathit{skip}$ and $T = \mathit{skip} \times T'$. If R is a relation we say that R is independent of S if $[R]$ and S are independent, or equivalently $\{R\}$ and S are independent. If R and Q are independent of S we have following subcommutativity properties:

$$S\ ;\ [R] \sqsubseteq [R]\ ;\ S \qquad \{Q\}\ ;\ S \sqsubseteq S\ ;\ \{Q\}$$

For simplicity and readability, we usually omit the natural extensions of predicates by *true* and of statements by *skip* when operating on an extended state space.

Procedures. Declaration of a procedure p with value parameters $v : \Delta$, result parameters $r : \Omega$, and body S, written

$$\textbf{procedure}\ p(\ \textbf{val}\ v : \Delta, \textbf{res}\ r : \Omega)\ \textbf{is}\ S$$

defines p to stand for S of type $\Gamma \times \Delta \times \Omega \mapsto \Gamma \times \Delta \times \Omega$, if Γ is the type of the global variables.

A procedure call $p(e,x)$ extends the state space by the value and result parameters, sets the value parameters to e, executes the procedure body, sets the result parameter x, and removes the parameters:

$$p(e,x) \;\widehat{=}\; \textbf{var}\ v,r \bullet v := e\ ;\ p\ ;\ x := r$$

Now suppose that p is a recursive procedure, which is expressed by assuming that S is of the form $s\ p$ for some s. That is, S has a free occurrence of p. The meaning of p is then given by taking the least fixed point of the function s, i.e. the least solution of $\lambda X \bullet X = s\ X$. Statements form a complete lattice with the refinement ordering. Furthermore, we assume that s is defined with p occurring in monotonic positions only. These two conditions guarantee that the least fixed point $\mu\ s$ of s exists and is unique. Hence we can define $p \;\widehat{=}\; \mu\ s$.

A set of mutually recursive procedures is defined by taking the fixed point of statement tuples. For tuples $(s_1,\ldots,s_n)$ and $(s'_1,\ldots,s'_n)$, where s_i and s'_i are statements of the same type, the refinement ordering is defined elementwise:

$$(s_1,\ldots,s_n) \sqsubseteq (s'_1,\ldots,s'_n) \;\widehat{=}\; (s_1 \sqsubseteq s'_1) \wedge \ldots \wedge\ (s_n \sqsubseteq s'_n)$$

Statement tuples also form a complete lattice with the refinement ordering. Let p stand for $(p_1,\ldots,p_n)$, assume $S_1 = s_1\ p,\ldots,S_n = s_n\ p$, and let s stand for $\lambda p \bullet (s_1\ p,\ldots,s_n\ p)$. The set of procedure declarations

$$\textbf{procedure}\ p_1\ \textbf{is}\ S_1,\ldots,\ \textbf{procedure}\ p_n\ \textbf{is}\ S_n$$

defines p to be the least fixed point of s, i.e. $p \;\widehat{=}\; \mu\ s$. Assuming again that all p_i occur only in monotonic positions in all s_j, a least fixed point exists and is unique.

2.2 Action Systems

Statements modeled as predicate transformers can express only atomic computations. In concurrent programs, components of the program interact during the computation. For reactive systems, the possible sequences of observable states rather than the input/output behavior are of interest. Such components can be modeled by action systems. Action systems consist of local variables, an initialization thereof, and a body, which is repeatedly executed as long as it is enabled. Action systems can represent terminating, non-terminating, and aborting computations. Formally an action system is a pair $AS = (ai, A)$ where $ai : \mathcal{P}\Sigma$ is the initializing predicate of the local state. Upon initialization, arbitrary values satisfying ai are chosen for the local variables. The global state space Γ is declared and initialized outside. Action $A : \Gamma \times \Sigma \mapsto \Gamma \times \Sigma$ is a positively conjunctive statement, which acts on the local state of type Σ and global state of type Γ. Because A is positively conjunctive, it can be written as $\{p\}\ ;\ [R]$. The next relation of A relates a state (u,v) in both the enabledness and termination domain to all possible next states (u',v'):

$$nxt\ A\ (u,v)\ (u',v') \;\widehat{=}\; p\ (u,v) \wedge R\ (u,v)\ (u',v')$$

A behavior of AS is a sequence of pairs

$$s = \langle (u_0,v_0),(u_1,v_1),\ldots \rangle$$

where v_0 is the initial value of the local state, such that $ai\ v_0$, and all consecutive elements of the sequence are in the next relation:

$$nxt\ A\ (u_i, v_i)\ (u_{i+1}, v_{i+1})$$

The set $beh\ AS$ is the set of all behaviors. A behavior is terminating if it is finite and for the last element (u_n, v_n) the action A is not enabled, $\neg\ grd\ A(u_n, v_n)$. A behavior is aborting if it is finite and for the last element (u_n, v_n) the action aborts, i.e. (u_n, v_n) is not in the termination domain, $\neg\ trm\ A(u_n, v_n)$. A behavior is non-terminating if it is not of finite length. The set $beh\ AS$ can be thought of as the (disjoint) union of terminating, aborting, and non-terminating behaviors of AS.

We use the following syntax for an action system (ai, A) with local variables a:

$$\textbf{var}\ a \mid ai \bullet\ \textbf{do}\ A\ \textbf{od}$$

Action systems are typically composed of a set of actions $A_1, \ldots, A_n$ operating on different parts of the state space, which we write as:

$$\textbf{var}\ a \mid ai \bullet\ \textbf{do}\ A_1\ [\!]\ \ldots\ [\!]\ A_n\ \textbf{od}$$

In the interleaving model, parallelism of two actions is modeled by taking them in arbitrary, demonically chosen order. Hence the meaning of such an action system is given by taking the nondeterministic choice between all actions:

$$\textbf{var}\ a \mid ai \bullet\ \textbf{do}\ A_1 \sqcap \ldots \sqcap A_n\ \textbf{od}$$

We furthermore consider the case of an indexed set of actions and of set of actions where the possible choice depends on a state predicate:

$$\begin{array}{ll} (\ [\!]\ i \in I \bullet A) & \widehat{=}\ \ (\sqcap i \in I \bullet A) \\ (\ [\!]\ i \mid p \bullet A) & \widehat{=}\ \ (\sqcap i \mid p \bullet A) \end{array}$$

To express various kinds of possibly parallel computations, we use also combinations of these notations, for example as in:

$$\textbf{do}\ (\ [\!]\ i \mid p \bullet A)\ [\!]\ (\ [\!]\ j \mid q \bullet B)\ \textbf{od}$$

Parallel Composition. The parallel composition of action systems $AS = (ai, A)$ and $BS = (bi, B)$ with the same global state space merges the local state spaces (possibly renaming variables to make them mutually distinct) and combines the actions by nondeterministic choice:

$$AS \parallel BS\ \widehat{=}\ (ai \wedge bi, A \sqcap B)$$

This models an arbitrary interleaving of the action of AS and BS without any assumption of fairness. As $grd\ (A \sqcap B) = grd\ A \vee grd\ B$, the combined system terminates only if both A and B are not enabled. As $trm\ (A \sqcap B) = trm\ A \wedge trm\ B$, the combined action system aborts if either A or B aborts. (We omit the explicit state space reordering and the natural extensions by *skip* for A and B to operate on the global state space and their respective local state space in $A \sqcap B$.) Parallel composition is commutative and associative, up to the order of state components.

Given an action system AS, we can make part of its global state space local by $\mathbf{var}\ b \mid bi \bullet AS$, as we do typically for hiding common variables of two action systems composed in parallel. If a and b are disjoint then:

$$\mathbf{var}\ b \mid bi \bullet\ \mathbf{var}\ a \mid ai \bullet\ \mathbf{do}\ A\ \mathbf{od}\ \ \widehat{=}\ \ \mathbf{var}\ a,b \mid ai \wedge bi \bullet\ \mathbf{do}\ A\ \mathbf{od}$$

Trace Refinement. Behaviors contain a local state component, which is not observable from outside. Furthermore, behaviors may contain stuttering steps which are not observable from outside either. A state (u_{i+1}, v_{i+1}) is a stuttering state if $u_i = u_{i+1}$. Traces on the other hand capture only the observable part of behaviors. For a behavior s, its trace $tr\ s$ is obtained by

1. removing all finite sequences of stuttering states from s, and
2. removing the local state component from all states in s.

Behavior s approximates behavior t, written $s \preceq t$, if

- s is aborting and $tr\ s$ is a prefix of $tr\ t$, or
- $tr\ s = tr\ t$.

Trace refinement between action systems AS and BS with the same global state space holds if all behaviors of BS have an approximating behavior of AS:

$$AS \preceq BS\ \ \widehat{=}\ \ \forall t \in beh\ BS \bullet \exists s \in beh\ AS \bullet s \preceq t$$

Since only finite stuttering is removed, an infinite behavior gives rise to an infinite trace and a finite behavior gives rise to a finite trace. Both "concrete stuttering" in BS as well as "abstract stuttering" in AS is allowed.

Simulation. Trace refinement can be shown to hold by simulation. Here we consider forward simulation between $AS = (ai, A)$ and $BS = (bi, B)$ with the same global state space using a relation R. An action $A_\natural$ is a stuttering action if it always terminates and it leaves the global state unchanged:

$$trm\ A_\natural = true \qquad \text{and} \qquad nxt\ A_\natural\ (u,v)\ (u',v') \Rightarrow u = u'$$

Let S^n be the n-fold sequential composition of statement S, defined by $S^0 = skip$ and $S^{n+1} = S\ ;\ S^n$. Let S^* stand for the nondeterministic choice between all n-fold sequential compositions of S, defined by $S^* = (\sqcap\ n \in Nat \bullet S^n)$. Define $AI = \mathbf{enter}\ a \mid ai$ and $BI = \mathbf{enter}\ b \mid bi$. Action system AS is simulated by BS using R, written $AS \preccurlyeq_R BS$, if there are decompositions $A = A_\sharp \sqcap A_\natural$ and $B = B_\sharp \sqcap B_\natural$ such that $A_\natural$ and $B_\natural$ are stuttering actions and:

(a) Initialization: $AI\ ;\ A_\natural^*\ ;\ [R] \sqsubseteq BI\ ;\ B_\natural^*$
(b) Actions: $A_\sharp\ ;\ A_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ B_\sharp\ ;\ B_\natural^*$
(c) Exit Condition: $R[trm\ A \wedge grd\ A] \leq grd\ B$
(d) Internal Convergence: $R[trm\ A \wedge trm\ (\mathbf{do}\ A_\natural\ \mathbf{od})] \leq trm\ (\mathbf{do}\ B_\natural\ \mathbf{od})$

Theorem 1. *Let AS and BS be action systems and R a relation. Then:*

$$AS \preccurlyeq_R BS \Rightarrow AS \preceq BS$$

$p \leq q$	entailment of predicates	Section 2.1
$S \sqsubseteq T$	algorithmic refinement of statements	Section 2.1
$S \sqsubseteq_R T$	data refinement of statements	Section 2.1
$s \preceq t$	approximation of traces	Section 2.2
$AS \preceq BS$	trace refinement of action systems	Section 2.2
$C \preceq^{\circ} D$	class refinement with single object	Section 4.1
$C \preceq^{\uparrow} D$	class refinement with dynamic object structures	Section 5.1
$AS \preccurlyeq_R BS$	simulation of action systems	Section 2.2
$C \preccurlyeq_R^{\circ} D$	simulation of classes with single object	Section 4.2
$C \preccurlyeq_R^{\uparrow} D$	simulation of classes with dynamic object structures	Section 5.1

Fig. 1. Summary of ordering relations

In general, action system refinement is not compositional in the sense that refining one action system would lead to a refinement in an environment with other action systems running in parallel. However, we get compositionality under the additional constraint of *non-interference*. Let $ES = (ei, E)$ be an action system and let R be refinement relation for AS. Action system ES does not interfere with R if

$$trm\ E \wedge r \leq E\ r$$

where $r\ u\ b = R\ (u,a)\ (u,a')$. In other words, r is an invariant of E.

Theorem 2. *Let AS, BS, and ES be action systems, let R be a relation. If ES does not interfere with R then:*

$$AS \preccurlyeq_R BS \Rightarrow AS \parallel ES \preceq BS \parallel ES$$

Figure 1 summarizes the various ordering relations on predicates, statements, traces, action systems, and classes.

3 Objects and Classes

Conventionally, a class is a template that defines a set of attributes and methods. Methods of a class may contain self-references to the method itself and to other methods of the class. Instantiating a class creates a new object with initialized attributes and method bodies as defined by the class. A subclass inherits attributes and methods from its superclass. Furthermore a subclass may add new attributes and overwrite inherited methods. Methods in a subclass may contain super-references to methods in the superclass. Formally, classes are modeled as self-referential recursive structures, where self-references are not resolved at the time the class is declared, but resolving is delayed until objects are created [31].

These principles are extended here: classes define additionally a set of actions, which are inherited in subclasses and may be overwritten. Subclasses may also introduce additional actions. Self-references are possible between both methods and actions. Self-references are resolved at the time when an object is created. Also, both methods and actions may contain super-references to methods and actions in the superclass.

3.1 Classes

Let Σ be the type of the attributes of some class C and let α be a type variable to be instantiated by the type of the global variables and possibly by the type of further attributes of subclasses. Typically, classes have several attributes and programs contain several global variables. Thus, elements of Σ and α are tuples. Attribute and variable names are used for accessing the corresponding components. The set of methods and actions of a class is represented by a tuple with the method and action name accessing the corresponding component. For the types of methods m_i and actions a_j of C we define

$$CM_i = \alpha \times \Sigma \times \Delta_i \times \Omega_i \mapsto \alpha \times \Sigma \times \Delta_i \times \Omega_i \qquad CA = \alpha \times \Sigma \mapsto \alpha \times \Sigma$$

where Δ_i and Ω_i are the types of the value, respectively result parameter of method m_i. Within a class, methods m_i and actions a_j of that class can be referred to by $self.m_i$ and $self.a_j$, respectively. This is formalized by having $self.m_i$ and $self.a_j$ as parameters of all methods and actions, allowing all methods and actions to be referred to by all methods and actions. The usefulness of this generalization becomes clearer when considering inheritance. Let $self$ stand for the tuple of method and action names prefixed by $self$:

$$self = (self.m_1, \ldots, self.m_m, self.a_1, \ldots, self.a_a)$$

Let cm_i be the body of method m_i. Since cm_i may contain calls to other methods and actions of the same object, m_i is a function of $self$:

$$m_i = \lambda self \bullet cm_i$$

Thus, the parameter $self$ may be used inside cm_i. Actions are treated analogously. The collection of all methods and actions of a class can then be expressed as a tuple cs parameterized with $self$,

$$cs = \lambda self \bullet (cm_1, \ldots, cm_m, ca_1, \ldots, ca_a)$$

where $cm_i : CM_i$, $ca_j : CA$, $self.m_i : CM_i$, and $self.a_j : CA$. Note that $self$ is here used to refer to methods and actions, but not to reference attributes (fields) of an object. Attributes are referenced with their unqualified names inside methods and actions.

A class also specifies possible initial values $ci : \mathcal{P}\Sigma$ of its attributes c. Hence a class C takes the form of a tuple:

$$C = (ci, cs)$$

(a) (b)

Fig. 2. Illustration of (a) class C and of (b) taking the fixed point of C. The incoming arrow represents calls to C, the outgoing arrow stands for self-calls of C.

Figure 2(a) illustrates the definition of a class. For defining class C with attributes, methods, and actions as above we use the syntax:

class C
  attr $c \mid ci$,
  meth $m_1($**val** v_1, **res** r_1**)** **is** cm_1,
  $\ldots$,
  meth $m_m($**val** v_m, **res** r_m**)** **is** cm_m,
  action a_1 **is** ca_1,
  $\ldots$,
  action a_a **is** ca_a
end

Objects have all self-calls resolved with methods of the object itself. Self-calls may be mutually recursive, like mutually recursive procedures. Modeling this formally amounts to taking the least fixed point of the function cs (Figure 2(b)). Methods and actions of objects of class C, denoted by $C.m_i$ and $C.a_i$, respectively, are defined by taking the fixed point of the tuple of all methods and actions and then selecting the corresponding method or action:

$$C.m_i \mathrel{\widehat{=}} (\mu\, cs).m_i \qquad C.a_i \mathrel{\widehat{=}} (\mu\, cs).a_i$$

Declaring a variable x to be of class C means declaring it to be of type Σ and initializing it with ci:

$$\textbf{var}\ x : C \bullet S \mathrel{\widehat{=}} \textbf{var}\ x \mid ci \bullet S$$

Such a variable corresponds to a local, stack allocated object in programming languages. Since actions cannot access variables which are local to some statements, concurrency cannot be expressed this way. For this purpose dynamic object structures are introduced later.

A method call $x.m_i$ of object x of class C corresponds to a procedure call with x as a value-result parameter.

$$x.m_i \mathrel{\widehat{=}} \textbf{var}\ c \bullet c := x\ ;\ C.m_i\ ;\ x := c$$

The name of the implicit formal parameter is that of the attributes, namely c. Therefore, c is used to access local data in the body of $C.m_i$. This corresponds to *this* in some programming languages.

Additional value and result parameters are treated as for procedure calls. For convenience, we also use the same notation for selecting an action of an object:

$$x.a_i \;\widehat{=}\; \textbf{var}\ c \bullet c := x\ ;\ C.a_i\ ;\ x := c$$

Example. We illustrate the above definitions with a stylized example. Let class E be defined as follows:

class E
  attr $c \mid c = 0$,
  meth *change* **is** $c :\in NAT$,
  meth *inc* **is** $c := c + 1$,
  action a **is** $true \rightarrow self.change$,
end

If $E = (ei, es)$, then $ei = (c = 0)$ and es is given by:

$$es = \lambda(self.change, self.inc, self.a) \bullet (c :\in NAT, c := c + 1, true \rightarrow self.change)$$

Taking the fixed point of es results in the substitution of the call to *change* by the definition of *change* in E:

$$\mu\, es = (c :\in NAT, c := c + 1, true \rightarrow c :\in NAT)$$

The use of fixed points becomes clear when we consider overriding in inheritance.

3.2 Inheritance

Inheritance is expressed by the application of a modifier to a base class: If D inherits from C, then D is equivalent to L **mod** C, where modifier L corresponds to the extending part of the definition of D. This model of single inheritance is equivalent to dynamic method lookups along the inheritance graph as implemented in most object-oriented languages [16]. We call C the superclass of D and D a subclass of C.

Let C be as above. A modifier L specifies additional attributes, say l of type Λ. We consider only modifiers that redefine all methods of the base class. If a method should remain unchanged, this is expressed by making a supercall to the same method of the base class. A modifier also redefines all actions of the base class and possibly adds new actions.

For defining modifier L with attributes, methods, and actions as above we use the following syntax, where unmentioned methods m_i and actions a_j are defined as $super.m_i$ and $super.a_j$, respectively:

modifier L
  attr $l \mid li$,
  meth m_1(**val** v_1, **res** r_1) **is** lm_1,
  ...,
  meth m_m(**val** v_1, **res** r_m) **is** lm_m,
  action a_1 **is** la_1,
  ...,
  action a_b **is** la_b
end

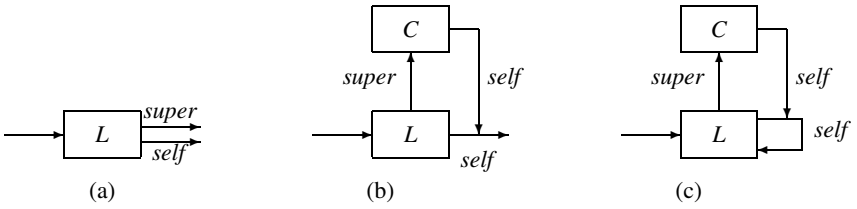


Fig. 3. Illustration of (a) modifier L, of (b) L **mod** C, and of (c) taking the fixed point of L **mod** C

For the types of methods m_i and actions a_j of L we define

$$LM_i = \beta \times \Lambda \times \Sigma \times \Delta_i \times \Omega_i \mapsto \beta \times \Lambda \times \Sigma \times \Delta_i \times \Omega_i$$
$$LA = \beta \times \Lambda \times \Sigma \mapsto \beta \times \Lambda \times \Sigma$$

where β is the type variable for global variables and further attributes in subclasses of D. Thus, we instantiate α of CM_i and CA by $\beta \times \Lambda$. The types of the value and result parameters of method m_i are, exactly as in C, that is Δ_i and Ω_i. Within L, methods m_i and actions a_j of that class can be referred to by $self.m_h$ and $self.a_k$, and those of the superclass C by $super.m_i$ and $super.a_j$, respectively. This is formalized by having $self.m_h, self.a_k, super.m_i$, and $super.a_j$ as parameters of all methods and actions. We let *self* and *super* stand for:

$$self = (self.m_1, \ldots, self.m_m, self.a_1, \ldots, self.a_b)$$
$$super = (super.m_1, \ldots, super.m_m, super.a_1, \ldots, super.a_a)$$

The collection of all methods and actions of modifier L can then be expressed as a tuple *ls* parameterized with both *self* and *super*,

$$ls = \lambda self \bullet \lambda super \bullet (lm_1, \ldots, lm_m, la_1, \ldots, la_b)$$

where $lm_k : LM_k$, $la_h : LA$, $self.m_h : LM_h$, $self.a_k : LA$, $super.m_i : CM_i$, and $super.a_j : CA$. A modifier also specifies initial values $li : \Lambda$ of the new attributes l. Hence a modifier L takes the form of a tuple:

$$L = (li, ls)$$

The modification of C by L binds super-calls in L to C and leaves the self-calls in L and C unresolved for possible further modification (Figure 3(b)):

$$L \textbf{ mod } C \;\widehat{=}\; (li \wedge ci, \lambda self \bullet ls\ self\ (cs\ \overline{self}))$$

Here $\overline{self} = (self.m_1, \ldots, self.m_m, self.a_1, \ldots, self.a_a)$ is identical as *self* in the definition of cs. Self-calls in L **mod** C, including those in methods and action of C, are bound to L when an object is instantiated (Figure 3(c)).

Example. We illustrate inheritance by extending class E of Section 3.1. Modifier F overrides method *change* and adds action b:

modifier F
  meth *change* **is** *super.inc*(),
  action b **is** $c < 10 \rightarrow self.inc()$
end

If $F = (fi, fs)$, then $fi = true$ and fs is given by:

$$fs = \lambda(self.change, self.inc, self.a, self.b) \bullet \lambda(super.change, super.inc, super.a) \bullet (super.inc(), super.inc(), super.a, c < 10 \rightarrow self.inc())$$

The second and third component are the implicit supercalls of not explicitly redefined method *inc* and action a. The application F **mod** E gives the following:

$$\begin{aligned} F \textbf{ mod } E &= (gi, gs) \\ gi &= (x = 0) \\ gs &= \lambda(self.change, self.inc, self.a, self.b) \bullet (c := c+1, c := c+1, \\ &\qquad true \rightarrow self.change(), c < 10 \rightarrow self.inc()) \end{aligned}$$

This illustrates that the super-calls are bound to the definitions in E. On the other hand, the self-calls in both E and F are still unresolved. This makes it possible to add another modifier to F **mod** E. The self-calls are again bound when an instance of F **mod** E is created:

$$\mu\, gs = (c := c+1, c := c+1, true \rightarrow c := c+1, c < 10 \rightarrow c := c+1)$$

4 Class Refinement and Class Simulation

In this section we define class refinement in terms of trace refinement. Also, a simulation condition between classes with a relation is defined and proved to imply class refinement. The reasoning is done with a single object of a class running in isolation; dynamic object creation is considered later.

4.1 Class Refinement

For an object x of class C, let $\mathcal{A}[x]$ be the action system with all its actions. Thus $\mathcal{A}[x]$ specifies how x behaves between external method calls to x:

$$\mathcal{A}[x] = \textbf{do } x.a_1 \;[\!]\; \ldots \;[\!]\; x.a_a \textbf{ od}$$

Let $O[x]$ be an action system observing object x only through method calls: we represent $O[x]$ as the (guarded) choice of either aborting or calling a method of x, where additionally local variables may be updated between method calls. Let $SA, S_1, \ldots, S_m$ be universally conjunctive statements that are independent of the global state, i.e. they access only local variables h:

$$O[x] = \textbf{var } h \mid hi \bullet \textbf{ do } SA \;;\; abort \;[\!]\; S_1 \;;\; x.m_1 \;[\!]\; \ldots \;[\!]\; S_m \;;\; x.m_m \textbf{ od}$$

Let $\mathcal{K}[C]$ be a program operating on an object x of class C such that $\mathcal{K}$ is the full context of x, in the sense that no other program accesses x. We describe $\mathcal{K}[C]$ by an interleaving of method calls to x and of actions of x:

$$\mathcal{K}[C] = \mathbf{var}\ x : C \bullet O[x] \parallel \mathcal{A}[x]$$

Class D is a refinement of class C, written $C \preceq^\circ D$, if using an object of class D instead of C in all possible programs yields a trace refinement of the original program:

$$C \preceq^\circ D \ \widehat{=}\ \forall \mathcal{K} \bullet \mathcal{K}[C] \preceq \mathcal{K}[D]$$

Class refinement between two classes is independent of how the classes are constructed using inheritance. In practice, it is often useful if all subclasses are refinements of their superclass. In typed languages, refinement is most beneficial between a class and its subclasses.

Our theory of refinement applies to classes with inheritance and self- and super-calls as introduced above. Because self- and super-calls in methods and actions are resolved before refinement is considered, there is no textually explicit resolution with fixed points here. Therefore, our treatment of refinement is independent of the model for inheritance and self- and super-calls and is also applicable to models lacking these concepts. In summary, our notion of refinement is targeted at the model of classes introduced in Section 3, but is independent enough to be applicable to other models as well.

4.2 Class Simulation

For proving refinement between classes $C = (ci, cs)$ and $D = (di, ds)$ we use a simulation with a refinement relation R. Define $CI = \mathbf{enter}\ c \mid ci$, $DI = \mathbf{enter}\ d \mid di$, and:

$$CX = C.a_1 \sqcap \ldots \sqcap C.a_a \qquad \text{and} \qquad DX = D.a_1 \sqcap \ldots \sqcap D.a_b$$

Class C is simulated by D using R, written $C \preccurlyeq^\circ_R D$, if there is a decomposition $CX = CX_\sharp \sqcap CX_\natural$ and $DX = DX_\sharp \sqcap DX_\natural$ such that $CX_\natural$ and $DX_\natural$ are stuttering actions and:

(a) Initialization:	$CI\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq DI\ ;\ DX_\natural^*$
(b) Methods:	$C.m_i\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ D.m_i\ ;\ DX_\natural^*$ for all m_i in $m_1, \ldots, m_m$
(c) Actions:	$CX_\sharp\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ DX_\sharp\ ;\ DX_\natural^*$
(d) Method Guards:	$R[\mathit{trm}\ C.m_i \wedge \mathit{trm}\ CX \wedge \mathit{grd}\ C.m_i] \leq \mathit{grd}\ D.m_i \vee \mathit{grd}\ DX$ for all m_i in $m_1, \ldots, m_m$
(e) Exit Condition:	$R[\mathit{trm}\ CX \wedge \mathit{grd}\ CX] \leq \mathit{grd}\ DX$
(f) Internal Convergence:	$R[\mathit{trm}\ CX \wedge \mathit{trm}\ (\mathbf{do}\ CX_\natural\ \mathbf{od})] \leq \mathit{trm}\ (\mathbf{do}\ DX_\natural\ \mathbf{od})$

Theorem 3. *Let C and D be classes and R a relation. Then:*

$$C \preccurlyeq^\circ_R D \Rightarrow C \preceq^\circ D$$

Proof. By the subordinate lemma below and Theorem 1.

Lemma 1. *Let C and D be classes and R a relation. Then:*

$$C \preccurlyeq^\circ_R D \Rightarrow \forall \mathcal{K} \bullet \mathcal{K}[C] \preccurlyeq_R \mathcal{K}[D]$$

Proof. We define:

$$CY = (SA\ ;\ abort) \sqcap (S_1\ ;\ C.m_1) \sqcap \ldots \sqcap (S_m\ ;\ C.m_m)$$
$$DY = (SA\ ;\ abort) \sqcap (S_1\ ;\ D.m_1) \sqcap \ldots \sqcap (S_m\ ;\ D.m_m)$$

We have to show that (a) to (f) above imply $\mathcal{K}[C] \preccurlyeq_R \mathcal{K}[D]$ for any $\mathcal{K}$ as above, which means that for any *hi*, *SA*, and $S_1, \ldots, S_m$:

$$\textbf{var}\ h \mid hi \bullet \textbf{var}\ c \mid ci \bullet \textbf{do}\ CY \ [\!]\ CX\ \textbf{od} \preccurlyeq_R$$
$$\textbf{var}\ h \mid hi \bullet \textbf{var}\ d \mid di \bullet \textbf{do}\ DY \ [\!]\ DX\ \textbf{od}$$

We note that R is independent of h, hence h is not involved in the refinement. According to the definition of action system simulation (Section 2.2) with $AI := CI$, $A_\sharp := CY \sqcap CX_\sharp$, $A_\natural := CX_\natural$, $BI := DI$, $B_\sharp := DY \sqcap DX_\sharp$, and $B_\natural := DX_\natural$ we get four conditions:

(1) Initialization: $CI\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq DI\ ;\ DX_\natural^*$
(2) Actions: $(CY \sqcap CX_\sharp)\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ (DY \sqcap DX_\sharp)\ ;\ DX_\natural^*$
(3) Exit Condition: $R[\mathit{trm}\ (CY \sqcap CX) \wedge \mathit{grd}\ (CY \sqcap CX)] \leq \mathit{grd}\ (DY \sqcap DX)$
(4) Internal Convergence: $R[\mathit{trm}\ (CY \sqcap CX) \wedge \mathit{trm}\ (\textbf{do}\ CX_\natural\ \textbf{od})] \leq \mathit{trm}\ (\textbf{do}\ DX_\natural\ \textbf{od})$

Condition (1) follows immediately from (a). For (2) we calculate, for any *SA* and $S_1, \ldots, S_m$:

$$\begin{array}{ll}
 & (CY \sqcap CX_\sharp)\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ (DY \sqcap DX_\sharp)\ ;\ DX_\natural^* \\
\equiv & \quad \{;\ \text{distributes over}\ \sqcap\} \\
 & (CY\ ;\ CX_\natural^*\ ;\ [R]) \sqcap (CX_\sharp\ ;\ CX_\natural^*\ ;\ [R]) \sqsubseteq ([R]\ ;\ DY\ ;\ DX_\natural^*) \sqcap ([R]\ ;\ DX_\sharp\ ;\ DX_\natural^*) \\
\Leftarrow & \quad \{\text{monotonicity}\} \\
 & (CY\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ DY\ ;\ DX_\natural^*) \wedge (CX_\sharp\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ DX_\sharp\ ;\ DX_\natural^*)
\end{array}$$

The second conjunct follows from (c). We continue with the first conjunct:

$$\begin{array}{ll}
 & CY\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ DY\ ;\ DX_\natural^* \\
\equiv & \quad \{\text{definition of } CY, DY \text{ and } ;\ \text{distributes over}\ \sqcap\} \\
 & (SA\ ;\ abort\ ;\ CX_\natural^*\ ;\ [R]) \sqcap (S_1\ ;\ C.m_1\ ;\ CX_\natural^*\ ;\ [R]) \sqcap \ldots \\
 & \qquad \sqcap (S_m\ ;\ C.m_m\ ;\ CX_\natural^*\ ;\ [R]) \sqsubseteq \\
 & ([R]\ ;\ SA\ ;\ abort\ ;\ DX_\natural^*) \sqcap ([R]\ ;\ S_1\ ;\ D.m_1\ ;\ DX_\natural^*) \sqcap \ldots \\
 & \qquad \sqcap ([R]\ ;\ S_m\ ;\ D.m_m\ ;\ DX_\natural^*) \\
\Leftarrow & \quad \{\text{monotonicity}\} \\
 & (SA\ ;\ abort\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ SA\ ;\ abort\ ;\ DX_\natural^*) \wedge \\
 & (\forall i \in \{1, \ldots, m\} \bullet S_i\ ;\ C.m_i\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ S_i\ ;\ D.m_i\ ;\ DX_\natural^*) \\
\Leftarrow & \quad \{S\ ;\ [R] \sqsubseteq [R]\ ;\ S \text{ for independent } R, S \text{ and } abort\ ;\ S = abort \text{ for any } S\} \\
 & (\forall i \in \{1, \ldots, m\} \bullet S_i\ ;\ C.m_i\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ S_i\ ;\ D.m_i\ ;\ DX_\natural^*) \\
\Leftarrow & \quad \{\text{as } S_i \text{ and } R \text{ are independent}\} \\
 & \forall i \in \{1, \ldots, m\} \bullet S_i\ ;\ C.m_i\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq S_i\ ;\ [R]\ ;\ D.m_i\ ;\ DX_\natural^* \\
\Leftarrow & \quad \{\text{monotonicity}\} \\
 & \forall i \in \{1, \ldots, m\} \bullet C.m_i\ ;\ CX_\natural^*\ ;\ [R] \sqsubseteq [R]\ ;\ D.m_i\ ;\ DX_\natural^*
\end{array}$$

The last line follows from (b). For (3) we calculate, for any SA and $S_1,\ldots,S_m$:

$$
\begin{array}{ll}
& R[\mathit{trm}\ (CY \sqcap CX) \wedge \mathit{grd}\ (CY \sqcap CX)] \leq \mathit{grd}\ (DY \sqcap DX) \\
\equiv & \quad \{\text{as } \mathit{trm}\ (S \sqcap T) = \mathit{trm}\ S \wedge \mathit{trm}\ T \text{ and } \mathit{grd}\ (S \sqcap T) = \mathit{grd}\ S \vee \mathit{grd}\ T\} \\
& R[\mathit{trm}\ CY \wedge \mathit{trm}\ CX \wedge (\mathit{grd}\ CY \vee \mathit{grd}\ CX)] \leq \mathit{grd}\ DY \vee \mathit{grd}\ DX \\
\Leftarrow & \quad \{\text{monotonicity}\} \\
& (R[\mathit{trm}\ CY \wedge \mathit{trm}\ CX \wedge \mathit{grd}\ CY] \leq \mathit{grd}\ DY \vee \mathit{grd}\ DX) \wedge \\
& (R[\mathit{trm}\ CX \wedge \mathit{grd}\ CX] \leq \mathit{grd}\ DX)
\end{array}
$$

The second conjunct follows from (e). We continue with the first conjunct:

$$
\begin{array}{ll}
& R[\mathit{trm}\ CY \wedge \mathit{trm}\ CX \wedge \mathit{grd}\ CY] \leq \mathit{grd}\ DY \vee \mathit{grd}\ DX \\
\Leftarrow & \quad \{\mathit{grd}(S\ ;\ T) \leq \mathit{grd}\ T \text{ if } S \text{ universally conjunctive and } S,T \text{ independent}\} \\
& R[\mathit{trm}\ CY \wedge \mathit{trm}\ CX \wedge \mathit{grd}\ CY] \leq \\
& \quad \mathit{grd}\ DY \vee \mathit{grd}\ (SA\ ;\ DX) \vee \ldots \vee \mathit{grd}\ (S_m\ ;\ DX) \\
\equiv & \quad \{\mathit{grd}\ (S \sqcap T) = \mathit{grd}\ S \vee \mathit{grd}\ T \text{ for any } S,T\} \\
& R[\mathit{trm}\ CY \wedge \mathit{trm}\ CX \wedge \mathit{grd}\ CY] \leq \mathit{grd}\ (DY \sqcap (SA\ ;\ DX) \sqcap \ldots \sqcap (S_m\ ;\ DX)) \\
\equiv & \quad \{R[p] \leq q \equiv p \leq [R]\ q \text{ and } [R](\mathit{grd}\ S) = \mathit{grd}\ (\{R\}\ ;\ S)\ (*)\} \\
& \mathit{trm}\ CY \wedge \mathit{trm}\ CX \wedge \mathit{grd}\ CY \leq \\
& \quad \mathit{grd}\ (\{R\}\ ;\ (DY \sqcap (SA\ ;\ DX) \sqcap \ldots \sqcap (S_m\ ;\ DX))) \\
\equiv & \quad \{;\ \text{distributes over } \sqcap \text{ and } \mathit{abort} \sqcap S = \mathit{abort} \text{ for any } S\} \\
& \mathit{trm}\ CY \wedge \mathit{trm}\ CX \wedge \mathit{grd}\ CY \leq \mathit{grd}\ ((\{R\}\ ;\ SI\ ;\ \mathit{abort}) \sqcap \\
& \quad (\{R\}\ ;\ S_1\ ;\ (D.m_1 \sqcap DX)) \sqcap \ldots \sqcap (\{R\}\ ;\ S_m\ ;\ (D.m_m \sqcap DX))) \\
\Leftarrow & \quad \{\{R\}\ ;\ S \sqsubseteq S\ ;\ \{R\} \text{ if } R,S \text{ independent and } \mathit{grd}\ U \leq \mathit{grd}\ T \text{ if } T \sqsubseteq U\} \\
& \mathit{trm}\ CY \wedge \mathit{trm}\ CX \wedge \mathit{grd}\ CY \leq \mathit{grd}\ ((SA\ ;\ \{R\}\ ;\ \mathit{abort}) \sqcap \\
& \quad (S_1\ ;\ \{R\}\ ;\ (D.m_1 \sqcap DX)) \sqcap \ldots \sqcap (S_m\ ;\ \{R\}\ ;\ (D.m_m \sqcap DX))) \\
\Leftarrow & \quad \{\mathit{trm}\ (S \sqcap T) = \mathit{trm}\ S \wedge \mathit{trm}\ T \text{ and} \\
& \qquad \mathit{grd}\ (S \sqcap T) = \mathit{grd}\ S \vee \mathit{grd}\ T \text{ for any } S,T\} \\
& (\mathit{trm}\ (SA\ ;\ \mathit{abort}) \wedge \mathit{trm}\ CX \wedge \mathit{grd}\ (SI\ ;\ \mathit{abort}) \leq \mathit{grd}\ (SI\ ;\ \{R\}\ ;\ \mathit{abort})) \wedge \\
& (\forall i \in \{1,\ldots,m\} \bullet \mathit{trm}\ (S_i\ ;\ C.m_i) \wedge \mathit{trm}\ CX \wedge \mathit{grd}\ (S_i\ ;\ C.m_i) \leq \\
& \quad \mathit{grd}\ (S_i\ ;\ \{R\}\ ;\ (D.m_i \sqcap DX))) \\
\Leftarrow & \quad \{\{R\}\ ;\ \mathit{abort} = \mathit{abort} \text{ for any } R\} \\
& \forall i \in \{1,\ldots,m\} \bullet \mathit{trm}\ (S_i\ ;\ C.m_i) \wedge \mathit{trm}\ CX \wedge \mathit{grd}\ (S_i\ ;\ C.m_i) \leq \\
& \quad \mathit{grd}\ (S_i\ ;\ \{R\}\ ;\ (D.m_i \sqcap DX)) \\
\Leftarrow & \quad \{\mathit{trm}\ T \leq \mathit{trm}\ (S\ ;\ T) \text{ if } S \text{ universally conjunctive and } S,T \text{ independent}\} \\
& \forall i \in \{1,\ldots,m\} \bullet \mathit{trm}\ (S_i\ ;\ C.m_i) \wedge \mathit{trm}\ (S_i\ ;\ CX) \wedge \mathit{grd}\ (S_i\ ;\ C.m_i) \leq \\
& \quad \mathit{grd}\ (S_i\ ;\ \{R\}\ ;\ (D.m_i \sqcap DX)) \\
\Leftarrow & \quad \{\mathit{trm}\ (S \sqcap T) = \mathit{trm}\ S \wedge \mathit{trm}\ S \text{ for any } S,T \text{ and } ;\ \text{distributes over } \sqcap\} \\
& \forall i \in \{1,\ldots,m\} \bullet \mathit{trm}\ (S_i\ ;\ (C.m_i \sqcap CX)) \wedge \mathit{grd}\ (S_i\ ;\ C.m_i) \leq \\
& \quad \mathit{grd}\ (S_i\ ;\ \{R\}\ ;\ (D.m_i \sqcap DX)) \\
\Leftarrow & \quad \{(\mathit{trm}\ T \wedge \mathit{grd}\ U \leq \mathit{grd}\ V) \Rightarrow \\
& \qquad (\mathit{trm}\ (S\ ;\ T) \wedge \mathit{grd}\ (S\ ;\ U) \leq \mathit{grd}\ (S\ ;\ V)\} \\
& \forall i \in \{1,\ldots,m\} \bullet \mathit{trm}\ (C.m_i \sqcap CX) \wedge \mathit{grd}\ C.m_i \leq \mathit{grd}\ (\{R\}\ ;\ (D.m_i \sqcap DX)) \\
\equiv & \quad \{(*)\ \text{above}\} \\
& \forall i \in \{1,\ldots,m\} \bullet R[\mathit{trm}\ (C.m_i \sqcap CX) \wedge \mathit{grd}\ C.m_i] \leq \mathit{grd}\ (D.m_i \sqcap DX) \\
\equiv & \quad \{\mathit{trm}\ (S \sqcap T) = \mathit{trm}\ S \wedge \mathit{trm}\ T \text{ and} \\
& \qquad \mathit{grd}\ (S \sqcap T) = \mathit{grd}\ S \vee \mathit{grd}\ T \text{ for any } S,T\} \\
& \forall i \in \{1,\ldots,m\} \bullet R[\mathit{trm}\ C.m_i \wedge \mathit{trm}\ CX \wedge \mathit{grd}\ C.m_i] \leq \mathit{grd}\ D.m_i \vee \mathit{grd}\ DX)
\end{array}
$$

The last line follows from (d). Condition (4) follows from (f) by monotonicity. □

A related theorem has first been given for action systems with remote procedures in [9] and in a revised form in [30], which is similar to the corresponding theorem for OO-action systems in [13]. The theorem given here generalizes those in four ways. First, we consider trace refinement and not just input/output refinement. Thus, class refinement also preserves reactive behavior and is meaningful for non-terminating systems. Second, removing abstract stuttering in refinement is explicitly considered. Third, the concrete stuttering action can be more general than a (data-) refinement of *skip*. Fourth, conditions (d) and (e) are weakened by including the termination conditions into the antecedents of the implications.

The case with no explicit abstract stuttering and the concrete stuttering actions being (data-) refinements of *skip* is obtained as a special case. Let C and D be classes and let CI, DI, CX, and DX be defined as above. Assume there exists a decomposition $DX = DX_\sharp \sqcap DX_\natural$ such that $DX_\natural$ is a stuttering action. The conditions for this case are:

(a')	Initialization:	$CI \,;\, [R] \sqsubseteq DI$
(b')	Methods:	$C.m_i \,;\, [R] \sqsubseteq [R] \,;\, D.m_i$ for all m_i in $m_1, \ldots, m_m$
(c')	Main Actions:	$CX \,;\, [R] \sqsubseteq [R] \,;\, DX_\sharp$
(d')	Internal Actions:	$[R] \sqsubseteq [R] \,;\, DX_\natural$
(e')	Method Guards:	$R[\mathit{trm}\ C.m_i \wedge \mathit{trm}\ CX \wedge \mathit{grd}\ C.m_i] \leq \mathit{grd}\ D.m_i \vee \mathit{grd}\ DX$ for all m_i in $m_1, \ldots, m_m$
(f')	Exit Condition:	$R[\mathit{trm}\ CX \wedge \mathit{grd}\ CX] \leq \mathit{grd}\ DX$
(g')	Internal Convergence:	$R[\mathit{trm}\ CX] \leq \mathit{trm}\ (\mathbf{do}\ DX_\natural\ \mathbf{od})$

Condition (d') is equivalent to $\mathit{skip} \sqsubseteq_R DX_\natural$, expressing that the concrete stuttering actions are data refinements of *skip*.

Theorem 4. *Let C and D be classes and R a relation as above. If conditions (a') – (g') hold then* $C \preccurlyeq^\circ_R D$.

Proof. We show that the above conditions (a') – (g') imply the conditions (a) – (f) of class simulation. We set $CX_\sharp := CX$ and $CX_\natural := \mathit{magic}$. Thus we have $CX^0_\natural = \mathit{skip}$, $CX^i_\natural = \mathit{magic}$ for all $i > 0$, and, therefore, $CX^*_\natural = \mathit{skip}$ because $\mathit{skip} \sqcap \mathit{magic} = \mathit{skip}$. With this, (a) follows immediately from (a') and (d').

By reflexivity and transitivity of refinement, we get from condition (d') that $[R] \sqsubseteq [R] \,;\, DX^i_\natural$ for any $i \geq 0$. Since $[R]$ is refined by sequences of any length, it is also refined by their choice, $[R] \sqsubseteq [R] \,;\, DX^*_\natural$. Condition (b) then follows by a transitivity from the following calculation:

$$
\begin{array}{ll}
 & C.m_i \,;\, CX^*_\natural \,;\, [R] \\
\sqsubseteq & \quad \{\text{as } [R] \sqsubseteq [R] \,;\, DX^*_\natural\} \\
 & C.m_i \,;\, [R] \,;\, DX^*_\natural \\
\sqsubseteq & \quad \{\text{condition (b')}\} \\
 & [R] \,;\, D.m_i \,;\, DX^*_\natural
\end{array}
$$

Condition (c) follows analogously using (c'). The remaining conditions (d) to (f) follow directly from (e') to (g'). For (f) we observe that $\mathbf{do}\ CX_\natural\ \mathbf{od} = \mathit{magic}$ and $\mathit{trm}\ \mathit{magic} = \mathit{true}$. □

Corollary 1. *Let C and D be classes and R a relation as above. If conditions (a') – (g') hold then* $C \preceq^{\circ} D$.

As with action system refinement, class refinement is not compositional in the sense that refining the class of an object will not necessarily lead to a system with other objects running in parallel being refined. However, we get compositionality under the additional constraint of non-interference with the environment.

Theorem 5. *Let C and D be classes, ES be an action systems, and R be a relation. If ES does not interfere with R then:*

$$C \preccurlyeq^{\circ}_{R} D \Rightarrow \forall \mathcal{K} \bullet \mathcal{K}[C] \parallel ES \preceq \mathcal{K}[D] \parallel ES$$

Proof. By Lemma 1 and Theorem 2. □

4.3 Example

We use an artificial aquarium as an example. Clearly, the observable sequences of states, denoting the position of the fishes, are the relevant aspect in such a system. A refinement of only the state transformation from initial to final states would be insufficient: A dedicated artificial aquarium has no final state. For its use as a screen saver, input/output refinement would only mean that at the end we are again guaranteed to get the original screen back.

The global variable s : **array** $[0..w-1, 0..h-1]$ **of** *NAT* denotes the state (color) of each quadrant of the screen, with constants $w > 6$ and $h > 6$. The color value 0 stands for background water. The base class *Creature* of all objects in our aquarium is given by:

class *Creature*
  attr $x, y, col \mid 0 \le x < w \wedge 0 \le y < h \wedge col \neq 0$,
  meth *move*(**val** dx, **val** dy) **is**
    $0 \le x+dx < w \wedge 0 \le y+dy < h \rightarrow$
      $skip \sqcap (s[x,y] := 0 \; ; \; x := x+dx \; ; \; y := y+dy \; ; \; s[x,y] := col)$,
  action *newpos* **is**
    $s[x,y] := 0 \; ; \; x :\in \{0..w-1\} \; ; \; y :\in \{0..h-1\} \; ; \; s[x,y] := col$
end

Creatures described by class *Ray* are a refinement with a special form of movement. Rather than jumping wildly around the screen, rays are always at the same vertical position, have a horizontal speed sx, and move at most 3 pixels at once:

class *Ray*
  attr $x, y, col, sx \mid x = 0 \wedge 0 \le y < h \wedge col = 5 \wedge sx = 1$,
  meth *move*(**val** dx, **val** dy) **is**
    $0 \le x+dx < w \wedge -3 \le dx \le 3 \wedge dy = 0 \rightarrow$
      $s[x,y] := 0 \; ; \; x := x+dx \; ; \; s[x,y] := col$,
  action *newpos* **is**
    $0 \le x+sx < w \rightarrow s[x,y] := 0 \; ; \; x := x+sx \; ; \; s[x,y] := col$,
  action *bouncel* **is** $x+sx < 0 \rightarrow sx :\in \{1..3\}$,
  action *bouncer* **is** $w \le x+sx \rightarrow sx :\in \{-3..-1\}$
end

Class *Ray* refines class *Creature* with refinement relation *R*:

$$R\ (s,x,y,col)\ (s',x',y',col',sx') \equiv s = s' \wedge x = x' \wedge 0 \le x < w \wedge y = y' \wedge \\ 0 \le y < h \wedge col = col' \wedge -3 \le sx' \le 3$$

We can use Theorem 4 to prove $Creature \preccurlyeq^{\circ}_{R} Ray$ because we have no explicit abstract stuttering. We set $CX := Creature.newpos$, $DX_{\sharp} := Ray.newpos$, $DX_{\natural} := Ray.bouncel \sqcap Ray.bouncer$, and CI and DI to the respective initialization. Internal convergence (condition (g')) follows by transitivity from the calculation below (assuming that an access to *s* outside the screen aborts):

$$\begin{array}{ll} & R[trm\ CX] \\ = & \quad \{\text{definitions of } trm \text{ and } CX\} \\ & R[\lambda s,x,y,col \bullet 0 \le x < w \wedge 0 \le y < h] \\ = & \quad \{\text{definition of } R\text{, relational image}\} \\ & \lambda s',x',y',col',sx' \bullet 0 \le x' < w \wedge 0 \le y < h \wedge -3 \le sx' \le 3 \\ \le & \quad \{\text{universal implication}\} \\ & \lambda s',x',y',col',sx' \bullet -1 \le x' \le w \vee 0 \le x' + sx' < w \\ = & \quad \{\text{definitions, calculus}\} \\ & trm\ (\mathbf{do}\ DX_{\natural}\ \mathbf{od}) \end{array}$$

The other conditions can also be proved by unfolding the definitions and simple calculus. By Corollary 1 we also get $Creature \preceq^{\circ} Ray$. Hence, replacing a *Creature* by a *Ray* in any context $\mathcal{K}$ produces a trace refinement.

5 Dynamic Object Structures

In this section we introduce dynamic object structures, which allow multiple objects to run concurrently. Furthermore, we extend the discussion of class refinement and class simulation to this setting.

We model the heap as an array and pointers as indices into this array [25]. We first describe the basic ideas using only one class and then generalize it to multiple classes with subtypes.

5.1 Single Class

For a class *C* with attributes of type Σ we declare a program variable *heap* to contain all dynamically created objects:

$$\mathbf{var}\ heap : \mathbf{array}\ NAT\ \mathbf{of}\ \Sigma$$

Pointers to objects of *C* are then simply natural numbers, that is the declaration p : **pointer to** *C* stands for $p : NAT$. We use 0 to denote *nil*, that is the pointer not referencing any object. We use a separate counter *next*, initialized to 1, to generate new pointer values. If *ci* is the initialization of the attributes of *C* and *p* is a pointer, p : **pointer to** *C*, then the creation of a new object is defined by:

$$p := new\ C\ \mathrel{\widehat{=}}\ p := next\ ;\ (\sqcap c \mid ci \bullet heap[p] := c)\ ;\ next := next + 1$$

To handle the way how attributes of objects on the heap are referenced, we have to introduce an indirection for each attribute reference via the receiver (the current object). We denote the receiver by *this* and introduce the shorthand *this.c* for referencing the attribute *c* of the object *heap*[*this*]:

$$this.c = heap[this].c$$

We use this shorthand in both expressions and for assignments in methods. A method call $p.m$ is then defined as (We use the restricted choice rather than the variable notation for *this* because the latter is a logic rather than a program variable.):

$$p.m \;\widehat{=}\; \{p \neq nil\} \;;\; (\sqcap this \mid this = p \bullet C.m)$$

Parameter passing is handled as for procedures. In our formalization, *this* is used to reference the receiver object whereas *self* and *super* are used in classes to reference methods and actions.

Formally, a class C with dynamically created objects is given by $C = (ci, cs)$ as previously, except that *heap* is now necessarily part of the global state and all references in *cs* to attributes go via *heap*. The selection $C.m_i$ and $C.a_i$ are defined as previously and we use the same syntax:

class C
  attr $c \mid ci$,
  meth $m_1($ **val** v_1, **res** $r_1)$ **is** cm_1,
  $\ldots$,
  meth $m_m($ **val** v_m, **res** $r_m)$ **is** cm_m,
  action a_1 **is** ca_1,
  $\ldots$,
  action a_a **is** ca_a
end

With the declaration of class C as above, we associate an action system $\mathcal{A}[C]$ which consists of actions operating on all objects of that class:

$$\mathcal{A}[C] = \textbf{do} \;(\; [\!] \; this \mid 1 \leq this < next \bullet C.a_1 \; [\!] \; \ldots \; [\!] \; C.a_a) \; \textbf{od}$$

This action system is composed in parallel with any other action system using objects of class C.

Example. A class *Creature* with dynamically created objects could be defined by:

class *Creature*
  attr $x, y, col \mid 0 \leq x < w \wedge 0 \leq y < h \wedge col \neq 0$,
  meth *move*(**val** *dx*, **val** *dy*) **is**
    $0 \leq this.x + dx < w \wedge 0 \leq this.y + dy < h \rightarrow$
      $skip \sqcap (s[this.x, this.y] := 0 \;;\; this.x := this.x + dx \;;$
        $this.y := this.y + dy \;;\; s[this.x, this.y] := this.col)$,
  action *newpos* **is**
    $s[this.x, this.y] := 0 \;;\; this.x :\in \{0..w-1\} \;;\; this.y :\in \{0..h-1\} \;;$
    $s[this.x, this.y] := this.col$
end

This declaration stands for:

$$
\begin{array}{l}
\textbf{var } heap : \textbf{array } NAT \textbf{ of } NAT \times NAT \times NAT \\
\textbf{var } next \mid next = 1 \\
\textbf{class } Creature \\
\quad \textbf{meth } move(\textbf{val } dx, \textbf{val } dy) \textbf{ is} \\
\qquad 0 \le heap[this].x + dx < w \wedge 0 \le heap[this].y + dy < h \rightarrow \\
\qquad\quad skip \sqcap (s[heap[this].x, heap[this].y] := 0 ; \\
\qquad\qquad heap[this].x := heap[this].x + dx ; \\
\qquad\qquad heap[this].y := heap[this].y + dy ; \\
\qquad\qquad s[heap[this].x, heap[this].y] := heap[this].col), \\
\quad \textbf{action } newpos \textbf{ is} \\
\qquad s[heap[this].x, heap[this].y] := 0 ; \\
\qquad heap[this].x :\in \{0..w-1\} ; \; heap[this].y :\in \{0..h-1\} ; \\
\qquad s[heap[this].x, heap[this].y] := heap[this].col \\
\textbf{end}
\end{array}
$$

If cr is a pointer to a *Creature* object, cr : **pointer to** *Creature*, then $cr := new\ Creature$ is defined by:

$$
\begin{array}{l}
cr := next ; \\
(\sqcap x, y, col \mid 0 \le x < w \wedge 0 \le y < h \wedge col \neq 0 \bullet heap[cr] := (x, y, col)) ; \\
next := next + 1
\end{array}
$$

A method call $cr.move(2,7)$ stands for:

$$\{cr \neq nil\} \; ; \; (\sqcap this \mid this = cr \bullet Creature.move(2,7))$$

The method selection $Creature.move(2,7)$ stands for:

$$
\begin{array}{l}
\textbf{var } dx, dy \bullet dx, dy := 2, 7 ; \\
0 \le heap[this].x + dx < w \wedge 0 \le heap[this].y + dy < h \rightarrow \\
\quad skip \sqcap (s[heap[this].x, heap[this].y] := 0 ; \; heap[this].x := heap[this].x + dx ; \\
\quad heap[this].y := heap[this].y + dy ; \\
\quad s[heap[this].x, heap[this].y] := heap[this].col)
\end{array}
$$

The action system $\mathcal{A}[Creature]$ associated with *Creature* is:

$$
\begin{array}{l}
\textbf{do} \\
\quad (\; [\!] \; this \mid 1 \le this < next \bullet \\
\qquad s[heap[this].x, heap[this].y] := 0 ; \; heap[this].x :\in \{0..w-1\} ; \\
\qquad heap[this].y :\in \{0..h-1\} ; \; s[heap[this].x, heap[this].y] := heap[this].col) \\
\textbf{od}
\end{array}
$$

5.2 Class Refinement and Class Simulation

We show that with the above definitions the notion of class refinement carries over analogously to dynamic object structures. With the declaration of a class C, we associate an action system $O[C]$, which observes all objects of class C by calling their

methods. We represent $O[C]$ as the (guarded) choice of either aborting or calling a method of x, where additionally local variables may be updated between method calls. Let $SA, S_1, \ldots, S_m, SC$ be universally conjunctive statements that are independent of the global state, i.e. they access only local variables h:

$$
\begin{array}{l}
O[C] = \\
\quad \textbf{var}\ h \mid hi \bullet \\
\quad \textbf{do}\ SA\ ;\ abort \\
\quad [\!]\ (\ [\!]\ this \mid 1 \le this < next \bullet S_1\ ;\ C.m_1\ [\!]\ \ldots\ [\!]\ S_m\ ;\ C.m_m) \\
\quad [\!]\ SC\ ;\ p := new\ C \\
\quad \textbf{od}
\end{array}
$$

Here we assume that p is part of the local variables h. Let $\mathcal{K}[C]$ be a program operating on objects of class C such that $\mathcal{K}$ is the full context of objects of class C, in the sense that no other program accesses the attributes of objects of C or creates new objects of C. We describe $\mathcal{K}[C]$ by an interleaving of method calls to instances of C, creation of new instances of C, and actions of instances of C:

$$\mathcal{K}[C] = \textbf{var}\ heap, next \mid next = 1 \bullet O[C] \parallel \mathcal{A}[C]$$

Class D is a refinement of class C, written $C \preceq^{\uparrow} D$, if using objects of class D instead of C in all possible programs yields a trace refinement of the original program:

$$C \preceq^{\uparrow} D \ \widehat{=}\ \forall \mathcal{K} \bullet \mathcal{K}[C] \preceq \mathcal{K}[D]$$

The conditions for simulation between two classes with dynamically created objects are like those for simulation with a single object, except that all objects on the heap are in the refinement relation. Thus for a refinement relation R between classes $C = (ci, cs)$ and $D = (di, ds)$ we define (g denotes the global variables):

$$
\begin{array}{rl}
\overline{R}\ (g, heap, next)\ (g', heap', next') \ \widehat{=}\ & (\forall q \mid 1 \le q < next \bullet \\
& \quad R\ (g, heap[q])\ (g', heap'[q])) \wedge \\
& next = next'
\end{array}
$$

Furthermore we define $CC = p := new\ C$, $DC = p := new\ D$, and

$$
\begin{array}{l}
CX = (\sqcap this \mid 1 \le this < next \bullet C.a_1 \sqcap \ldots \sqcap C.a_a) \\
DX = (\sqcap this \mid 1 \le this < next \bullet D.a_1 \sqcap \ldots \sqcap D.a_b)
\end{array}
$$

Class C is simulated by D using R, written $C \preccurlyeq^{\uparrow}_{R} D$, if there is a decomposition $CX = CX_\sharp \sqcap CX_\natural$ and $DX = DX_\sharp \sqcap DX_\natural$ such that $CX_\natural$ and $DX_\natural$ are stuttering actions and:

(a) Creation:	$CC\ ;\ CX^*_\natural\ ;\ [\overline{R}] \sqsubseteq [\overline{R}]\ ;\ DC\ ;\ DX^*_\natural$
(b) Methods:	$C.m_i\ ;\ CX^*_\natural\ ;\ [\overline{R}] \sqsubseteq [1 \le this < next]\ ;\ [\overline{R}]\ ;\ D.m_i\ ;\ DX^*_\natural$ for all m_i in $m_1, \ldots, m_m$
(c) Actions:	$CX_\sharp\ ;\ CX^*_\natural\ ;\ [\overline{R}] \sqsubseteq [\overline{R}]\ ;\ DX_\sharp\ ;\ DX^*_\natural$
(d) Method Guards:	$\overline{R}[1 \le this < next \wedge trm\ C.m_i \wedge trm\ CX \wedge grd\ C.m_i] \le grd\ D.m_i \vee grd\ DX$ for all m_i in $m_1, \ldots, m_m$
(e) Exit Condition:	$\overline{R}[trm\ CX \wedge grd\ CX] \le grd\ DX$
(f) Internal Convergence:	$\overline{R}[trm\ CX \wedge trm\ (\textbf{do}\ CX_\natural\ \textbf{od})] \le trm\ (\textbf{do}\ DX_\natural\ \textbf{od})$

Theorem 6. *Let C and D be classes and R a relation. Then:*

$$C \preccurlyeq_R^{\uparrow} D \Rightarrow C \preceq^{\uparrow} D$$

Proof. By the subordinate lemma below and Theorem 1.

Lemma 2. *Let C and D be classes and R a relation. Then:*

$$C \preccurlyeq_R^{\uparrow} D \Rightarrow \forall \mathcal{K} \bullet \mathcal{K}[C] \preccurlyeq_{\overline{R}} \mathcal{K}[D]$$

Proof. We define:

$$\begin{array}{lll} CY & = & (SA \,;\, abort) \sqcap \\ & & (\sqcap this \mid 1 \leq this < next \bullet (S_1 \,;\, C.m_1) \sqcap \ldots \sqcap (S_m \,;\, C.m_m)) \sqcap \\ & & (SC \,;\, CC) \\ DY & = & (SA \,;\, abort) \sqcap \\ & & (\sqcap this \mid 1 \leq this < next \bullet (S_1 \,;\, D.m_1) \sqcap \ldots \sqcap (S_m \,;\, D.m_m)) \sqcap \\ & & (SC \,;\, DC) \\ CI & = & \textbf{enter}\ heap, next \mid next = 1 \\ DI & = & \textbf{enter}\ heap, next \mid next = 1 \end{array}$$

Leaving out the types, we note that *heap* in *CI* is an array of *C* attributes and *heap* in *DI* is an array of *D* attributes. We have to show that (a) to (f) above imply $\mathcal{K}[C] \preccurlyeq_{\overline{R}} \mathcal{K}[D]$ for any $\mathcal{K}$ as above, which means that for any *hi*, *SA*, $S_1, \ldots, S_m$, and *SC*:

$$\begin{array}{l} \textbf{var}\ h \mid hi \bullet \textbf{var}\ heap, next \mid next = 1 \bullet \textbf{do}\ CY \ [\!]\ CX\ \textbf{od} \preccurlyeq_{\overline{R}} \\ \textbf{var}\ h \mid hi \bullet \textbf{var}\ heap, next \mid next = 1 \bullet \textbf{do}\ DY \ [\!]\ DX\ \textbf{od} \end{array}$$

We note that *R* is independent of *h*, hence *h* is not involved in the refinement. According to the definition of action system simulation (Section 2.2) with $AI := CI$, $A_\sharp := CY \sqcap CX_\sharp$, $A_\natural := CX_\natural$, $BI := DI$, $B_\sharp := DY \sqcap DX_\sharp$, $B_\natural := DX_\natural$, and $R := \overline{R}$ we get four conditions:

(1) Initialization: $CI \,;\, CX_\natural^* \,;\, [\overline{R}] \sqsubseteq DI \,;\, DX_\natural^*$

(2) Actions: $(CY \sqcap CX_\sharp) \,;\, CX_\natural^* \,;\, [\overline{R}] \sqsubseteq [\overline{R}] \,;\, (DY \sqcap DX_\sharp) \,;\, DX_\natural^*$

(3) Exit Condition: $\overline{R}[\,trm\,(CY \sqcap CX) \wedge grd\,(CY \sqcap CX)] \leq grd\,(DY \sqcap DX)$

(4) Internal Convergence: $\overline{R}[\,trm\,(CY \sqcap CX) \wedge trm\,(\,\textbf{do}\ CX_\natural\ \textbf{od}\,)] \leq trm\,(\,\textbf{do}\ DX_\natural\ \textbf{od}\,)$

Condition (1) expands to:

$$\textbf{enter}\ heap, next \mid next = 1 \,;\, CX_\natural^* \,;\, [\overline{R}] \sqsubseteq \textbf{enter}\ heap, next \mid next = 1 \,;\, DX_\natural^*$$

First we note that after the initialization of *next* by 1, neither $CX_\natural$ nor $DX_\natural$ is enabled, as $(\sqcap i \mid false \bullet S) = magic$. Therefore, $CX_\natural^* = skip$ and $DX_\natural^* = skip$. The refinement relation $\overline{R}$ quantifies over all objects created on the heap. As *next* is set to 1, there is no such object and the refinement holds vacuously.

For (2) we calculate, for any *SA*, $S_1, \ldots, S_m$, and *SC*:

$$\begin{array}{ll} & (CY \sqcap CX_\sharp) \,;\, CX_\natural^* \,;\, [\overline{R}] \sqsubseteq [\overline{R}] \,;\, (DY \sqcap DX_\sharp) \,;\, DX_\natural^* \\ \equiv & \quad \{\,;\ \text{distributes over}\ \sqcap\} \\ & (CY \,;\, CX_\natural^* \,;\, [\overline{R}]) \sqcap (CX_\sharp \,;\, CX_\natural^* \,;\, [\overline{R}]) \sqsubseteq ([\overline{R}] \,;\, DY \,;\, DX_\natural^*) \sqcap ([\overline{R}] \,;\, DX_\sharp \,;\, DX_\natural^*) \\ \Leftarrow & \quad \{\text{monotonicity}\} \\ & (CY \,;\, CX_\natural^* \,;\, [\overline{R}] \sqsubseteq [\overline{R}] \,;\, DY \,;\, DX_\natural^*) \wedge (CX_\sharp \,;\, CX_\natural^* \,;\, [\overline{R}] \sqsubseteq [\overline{R}] \,;\, DX_\sharp \,;\, DX_\natural^*) \end{array}$$

The second conjunct follows from (c). We continue with the first conjunct:

$$
\begin{array}{ll}
 & CY\ ;\ CX_{\natural}^{*}\ ;\ [\overline{R}] \sqsubseteq [\overline{R}]\ ;\ DY\ ;\ DX_{\natural}^{*} \\
\equiv & \quad \{\text{definitions of } CY \text{ and } DY \text{ and } ;\ \text{distributes over } \sqcap\} \\
 & (SA\ ;\ abort\ ;\ CX_{\natural}^{*}\ ;\ [\overline{R}]) \sqcap \\
 & (\sqcap this \mid 1 \leq this < next \bullet (S_1\ ;\ C.m_1\ ;\ CX_{\natural}^{*}\ ;\ [\overline{R}]) \sqcap \ldots \\
 & \quad \sqcap (S_m\ ;\ C.m_m\ ;\ CX_{\natural}^{*}\ ;\ [\overline{R}])) \sqcap \\
 & (SC\ ;\ CC\ ;\ CX_{\natural}^{*}\ ;\ [\overline{R}]) \sqsubseteq \\
 & ([\overline{R}]\ ;\ SA\ ;\ abort\ ;\ DX_{\natural}^{*}) \sqcap \\
 & (\sqcap this \mid 1 \leq this < next \bullet ([\overline{R}]\ ;\ S_1\ ;\ D.m_1\ ;\ DX_{\natural}^{*}) \sqcap \ldots \\
 & \quad \sqcap ([\overline{R}]\ ;\ S_m\ ;\ D.m_m\ ;\ DX_{\natural}^{*})) \sqcap \\
 & ([\overline{R}]\ ;\ SC\ ;\ DC\ ;\ DX_{\natural}^{*}) \\
\Leftarrow & \quad \{\text{monotonicity}\} \\
 & (SA\ ;\ abort\ ;\ CX_{\natural}^{*}\ ;\ [\overline{R}] \sqsubseteq [\overline{R}]\ ;\ SA\ ;\ abort\ ;\ DX_{\natural}^{*}) \wedge \\
 & ((\sqcap this \mid 1 \leq this < next \bullet (S_1\ ;\ C.m_1\ ;\ CX_{\natural}^{*}\ ;\ [\overline{R}]) \sqcap \ldots \\
 & \quad \sqcap (S_m\ ;\ C.m_m\ ;\ CX_{\natural}^{*}\ ;\ [\overline{R}])) \sqsubseteq \\
 & \quad (\sqcap this \mid 1 \leq this < next \bullet ([\overline{R}]\ ;\ S_1\ ;\ D.m_1\ ;\ DX_{\natural}^{*}) \sqcap \ldots \\
 & \quad \sqcap ([\overline{R}]\ ;\ S_m\ ;\ D.m_m\ ;\ DX_{\natural}^{*}))) \wedge \\
 & (SC\ ;\ CC\ ;\ CX_{\natural}^{*}\ ;\ [\overline{R}] \sqsubseteq [\overline{R}]\ ;\ SC\ ;\ DC\ ;\ DX_{\natural}^{*}) \\
\Leftarrow & \quad \{S\ ;\ [R] \sqsubseteq [R]\ ;\ S \text{ for independent } R,S \text{ and } abort\ ;\ S = abort \text{ for any } S\} \\
 & ((\sqcap this \mid 1 \leq this < next \bullet (S_1\ ;\ C.m_1\ ;\ CX_{\natural}^{*}\ ;\ [\overline{R}]) \sqcap \ldots \\
 & \quad \sqcap (S_m\ ;\ C.m_m\ ;\ CX_{\natural}^{*}\ ;\ [\overline{R}])) \sqsubseteq \\
 & \quad (\sqcap this \mid 1 \leq this < next \bullet ([\overline{R}]\ ;\ S_1\ ;\ D.m_1\ ;\ DX_{\natural}^{*}) \sqcap \ldots \\
 & \quad \sqcap ([\overline{R}]\ ;\ S_m\ ;\ D.m_m\ ;\ DX_{\natural}^{*}))) \wedge \\
 & (SC\ ;\ CC\ ;\ CX_{\natural}^{*}\ ;\ [\overline{R}] \sqsubseteq [\overline{R}]\ ;\ SC\ ;\ DC\ ;\ DX_{\natural}^{*}) \\
\Leftarrow & \quad \{\text{definition of } \sqcap i \mid p \bullet S \text{ and} \\
 & \qquad (\forall i \bullet S \sqsubseteq T) \Rightarrow (\sqcap i \bullet S) \sqsubseteq (\sqcap i \bullet T) \text{ for any } S,T\} \\
 & (\forall this \bullet \forall i \in \{1,\ldots,m\} \bullet \\
 & [1 \leq this < next]\ ;\ S_i\ ;\ C.m_i\ ;\ CX_{\natural}^{*}\ ;\ [R] \sqsubseteq \\
 & \quad [1 \leq this < next]\ ;\ [R]\ ;\ S_i\ ;\ D.m_i\ ;\ DX_{\natural}^{*}) \wedge \\
 & (SC\ ;\ CC\ ;\ CX_{\natural}^{*}\ ;\ [\overline{R}] \sqsubseteq [\overline{R}]\ ;\ SC\ ;\ DC\ ;\ DX_{\natural}^{*}) \\
\Leftarrow & \quad \{S_i \text{ and } R \text{ and } SC \text{ and } R \text{ independent, refinement calculus}\} \\
 & (\forall this \bullet \forall i \in \{1,\ldots,m\} \bullet \\
 & \quad S_i\ ;\ C.m_i\ ;\ CX_{\natural}^{*}\ ;\ [R] \sqsubseteq [1 \leq this < next]\ ;\ S_i\ ;\ [R]\ ;\ D.m_i\ ;\ DX_{\natural}^{*}) \wedge \\
 & (SC\ ;\ CC\ ;\ CX_{\natural}^{*}\ ;\ [\overline{R}] \sqsubseteq SC\ ;\ [\overline{R}]\ ;\ DC\ ;\ DX_{\natural}^{*})
\end{array}
$$

The first conjunct follows from (b) and the second from (a). The proof of (3) is similar to the one of the corresponding condition in Theorem 3 and is left out for brevity. Condition (4) follows from (f) by monotonicity. □

As for the case with a single object, class refinement with dynamic object structures is compositional only under the additional constraint of non-interference with the environment.

Theorem 7. *Let C and D be classes, ES be an action systems, and R be a relation. If ES does not interfere with R then:*

$$C \preccurlyeq^{\uparrow}_{R} D \Rightarrow \forall \mathcal{K} \bullet \mathcal{K}[C] \parallel ES \preceq \mathcal{K}[D] \parallel ES$$

Proof. By Lemma 2 and Theorem 2.

5.3 Multiple Classes and Subtyping

This formalization easily extends to multiple classes with subtyping. We declare for each class C_i with attribute type Σ_i a separate $heap_i$: **array** *NAT* **of** Σ_i. Thus with a class declaration **class** $C_i \ldots$ **end** we associate:

var $heap_i$: **array** *NAT* **of** Σ_i,
var $next_i \mid next_i = 1$

Pointers are extended to tuples with one index indicating the heap and one index indicating the element within the heap. A pointer variable declaration p : **pointer to** C_i stands for $p : NAT \times NAT$. The first component of a pointer p is selected by $p.class$, the second component by $p.ref$. The *nil* value is always represented by $(0,0)$ to make it unique.

Assuming that $C_k, \ldots, C_l$ are all subtypes of C_i (including C_i), object creation, method calls with dynamic dispatch, type tests, and a sample attribute access are defined by:

$$\begin{array}{lcl}
p := new\ C_i & \widehat{=} & p := (i, next_i)\ ;\ (\sqcap c \mid ci_i \bullet heap_i[next_i] := c)\ ; \\
 & & next_i := next_i + 1 \\
p.m & \widehat{=} & \{p \neq nil\}\ ; \\
 & & (\sqcap this \mid this = p \bullet p.class = k \rightarrow C_k.m \sqcap \ldots \sqcap \\
 & & \qquad p.class = l \rightarrow C_l.m) \\
p\ \textbf{instanceof}\ C_i & \widehat{=} & p.class \in \{k, \ldots, l\} \\
x := this.c & \widehat{=} & this.class = k \rightarrow x := heap_k[this.ref].c \sqcap \ldots \sqcap \\
 & & \qquad this.class = l \rightarrow x := heap_l[this.ref].c
\end{array}$$

With each class declaration C_i, we associate an action system $\mathcal{A}[C_i]$ which represents all actions of all objects of that class:

$$\begin{array}{l}
\mathcal{A}[C_i] = \textbf{do} \\
\qquad (\ [\!]\ this \mid this.class = i \wedge 1 \leq this.ref < next_i \bullet C_i.a_1 \sqcap \ldots \sqcap C_i.a_a) \\
\qquad \textbf{od}
\end{array}$$

For a program with classes $C_1, \ldots, C_n$ we take the parallel composition of the action systems for objects of each class. This composition is then to be combined with further action systems containing normal actions and procedures:

$$\mathcal{A}[C_1] \parallel \ldots \parallel \mathcal{A}[C_n] \parallel BS$$

Example. Let class *Creature* be as in Section 5.3 and let *Ray* be an analogously adapted version of class *Ray* in Section 4 for dynamic data structures. Furthermore, we postulate a class *Turtle*, for which $\mathit{Creature} \preccurlyeq^{\uparrow}_{Q} \mathit{Turtle}$ holds for some relation Q. Let G be the following specification of an artificial aquarium, in which new objects may constantly be added and where the most recently created object may be influenced through its *move* method:

$$G = (\,\mathbf{var}\ p : \mathit{Creature} \mid p = \mathit{nil} \bullet \\ \quad \mathbf{do}\ p := \mathit{new}\ \mathit{Creature} \ [\!]\ p \neq \mathit{nil} \to p.\mathit{move}(2,0)\ \mathbf{od}\,) \\ \quad \parallel \mathcal{A}[\mathit{Creature}]$$

By applying Theorem 6 twice and Theorem 7, which both extend to multiple classes, we can show that this specification is trace refined, $G \preceq H$, by implementation H with rays and turtles:

$$H = (\,\mathbf{var}\ p : \mathit{Creature} \mid p = \mathit{nil} \bullet \\ \quad \mathbf{do}\ p := \mathit{new}\ \mathit{Ray} \ [\!]\ p := \mathit{new}\ \mathit{Turtle} \ [\!]\ p \neq \mathit{nil} \to p.\mathit{move}(2,0)\ \mathbf{od}\,) \\ \quad \parallel \mathcal{A}[\mathit{Ray}] \parallel \mathcal{A}[\mathit{Turtle}]$$

6 Early Return

Atomicity refinement is used to increase concurrency by decreasing the granularity of atomic actions. Consider method *rnd* that computes random numbers and for later reference stores them in a time ordered sequence:

meth *rnd*(**res** *y*) **is** $y :\in \mathit{NAT}$; 'store y in sequence'

Using atomicity refinement, we could split up *rnd* so that it returns control to the caller after assigning y and schedules the —if the sequence is kept on secondary storage— time consuming insertion operation for later. Thereby, the execution time of any action a calling *rnd* is reduced. Thus, other actions accessing the same resources as a can be started earlier, thereby increasing concurrency.

We introduce a **release** statement, which facilitates the above type of atomicity refinement. A **release** returns control to the caller of a method and schedules the remainder to be executed later on. If the method containing the **release** statement has

class C
  attr $c \mid ci$,
  meth m **is** S ; **release** ; T,
  meth n **is** U,
  action a **is** V

end

a) Method with **release**

class C
  attr $c, lck \mid ci \wedge lck = 0$,
  meth m **is** $lck = 0 \to S$; $lck := 1$,
  meth n **is** $lck = 0 \to U$,
  action a **is** $lck = 0 \to V$,
  action r **is** $lck = 1 \to T$; $lck := 0$
end

b) Equivalent without **release**

Fig. 4. Definition of **release** as enabling a remainder action

result parameters, they must be assigned before executing **release**. For example, we could rewrite method *rnd* as follows:

meth *rnd*(**res** *y*) **is** $y :\in NAT$; **release** ; 'store y in sequence'

Figure 4 defines **release** as enabling an action r that performs the remainder. The object is locked, that is none of its other methods or actions can be executed, until the remainder action is completed. For simplicity, we do not allow self and reentrant calls and parameter and local variable access in the remainder. These generalizations are made below.

Introducing **release** leads to class refinement:

Theorem 8. *Let C and D be classes which are identical except that method m in C and m in D, referred to as C :: m and D :: m, are defined by:*

meth $C :: m$ **is** S ; T,
meth $D :: m$ **is** S ; **release** ; T

If T does not modify global variables and does not access parameters, then $C \preceq^{\circ} D$ *holds.*

Proof. We apply Theorem 3 with $R\ (u,c)\ (u',c',lck') := u' = u \wedge ((l' = 0 \wedge c' = c) \vee (l' = 1 \wedge nxt\ S\ c\ c'))$, $CI :=$ **enter** $c \mid ci$, $CX_\sharp := V$, $CX_\natural := magic$, $DI :=$ **enter** $c, lck \mid ci \wedge lck = 0$, $DX_\sharp := lck = 0 \rightarrow V$, $DX_\natural := lck = 1 \rightarrow T$; $lck := 0$. The theorem follows by simplifications of the conditions (a) – (f).

Note that Theorem 4 cannot be used for the proof since the remainder action $lck = 1 \rightarrow T$; $lck := 0$ is a concrete stuttering action which is not a (data-) refinement of *skip*.

The **release** statement can be generalized to allow the remainder to access the value parameter and the local variables of the method and also read the result parameter (Figure 5). The values of the parameters and local variables are stored in additional attributes for use by the remainder.

class C
  attr $c := ci$,
  meth m(**val** v, **res** r) **is**
    var $x \bullet S$; **release** ; T,
  meth n(**val** w, **res** s) **is**
    U
  action a **is**
    V
end

a) Method with **release**

class C
  attr $c, lck, m\_v, m\_r, m\_x \mid ci \wedge lck = 0$,
  meth m(**val** v, **res** r) **is**
    $lck = 0 \rightarrow$ **var** $x \bullet S$; $lck, m\_v, m\_r, m\_x := 1, v, r, x$,
  meth n(**val** w, **res** s) **is**
    $lck = 0 \rightarrow U$,
  action a **is**
    $lck = 0 \rightarrow V$,
  action r **is**
    $lck = 1 \rightarrow$ **var** $v, r, x := m\_v, m\_r, m\_x \bullet T$; $lck := 0$
end

b) Equivalent without **release**

Fig. 5. Definition of **release** with remainder accessing parameters and Local Variables

```
class C
  attr c | ci,
  meth m( val v, res r) is
    var x • S ; release ; T,
  meth n( val w, res s) is U,

  action a is V
end
```

a) Method with **release**

```
class C
  attr c, lck, m_v, m_r, m_x | ci ∧ lck = 0
  meth m( val v, res r) is
    p ; var x • S ; lck, m_v, m_r, m_x := 1, v, r, x,
  meth n( val w, res s) is p ; U,
  meth p is
    if lck = 1 then
      var v, r, x := m_v, m_r, m_x • T ; lck := 0
    end ,
  action a is lck = 0 → V,
  action r is lck = 1 → p
end
```

b) Equivalent without **release**

Fig. 6. Definition of **release** supporting multiple calls to an object within an action

Finally, we consider the case where an action contains multiple calls to methods of the same object. If a method of an object that has an outstanding remainder is called then the latter is executed as part of the call. Otherwise, the guard of the methods called after performing a **release** would be false and, therefore, such actions never enabled. Consider action b where o references an object of type C as in Figure 6:

action b **is** (**var** $z : U \bullet o.m(0,z)$; $o.n(0,z)$)

If we simply locked o, that is, defined the implicit guard of n to be $lck = 0$, then b would never be enabled.

We illustrate this with a random number class that stores a sequence of already computed numbers:

```
class C
  attr l := 0, s : array NAT of NAT ,
  meth rnd( res y) is y :∈ NAT ; s[l], l := y, l + 1,
  meth get( val i, res y) is i < l → y := s[i]
end
```

Class C is refined by D, where a **release** is introduced in method *rnd* after the assignment of y. We show directly the expansion according to Figure 6:

```
class D
  attr l := 0, s : array NAT of NAT , lck := 0, rnd_y,
  meth rnd( res y) is p ; y :∈ NAT ; lck, rnd_y := 1, y,
  meth get( val i, res y) is p ; i < l → y := s[i],
  meth p is if lck = 1 then var y := rnd_y • s[l], l, lck := y, l + 1, 0 end ,
  action r is lck = 1 → p
end
```

We have $C \preccurlyeq^{\circ}_{R} D$ for the following R:

$$
\begin{array}{l}
R\ (l,s)\ (l',s',lck',rnd\_y') \equiv lck' \in \{0,1\} \wedge \\
\quad (lck' = 0 \Rightarrow l = l' \wedge (\forall i \in \{0..l-1\} \bullet s[i] = s'[i])) \wedge \\
\quad (lck' = 1 \Rightarrow l = l'+1 \wedge (\forall i \in \{0..l-2\} \bullet s[i] = s'[i]) \wedge s[l-1] = rnd\_y')
\end{array}
$$

The proof is a simple verification of the six conditions of class simulation with $CX_\sharp = magic$, $CX_\natural = magic$, $DX_\sharp = magic$, $DX_\natural = r$, and CI and DI the respective initializations.

7 Conclusions and Discussion

We have given a model for action-based concurrency with objects. Classes with attributes, methods, and actions serve as templates for objects. Class refinement supporting algorithmic, data, and atomicity refinement is defined based on trace refinement. Class refinement can be proved by a simulation rule. Early returns are a special form of atomicity refinement. Dynamic data structures allow objects to run concurrently.

The refinement rules have been developed in a most general form without considering some useful special cases. For example, for the refinement of classes with dynamically created objects each attribute reference goes via the heap. If aliasing can be excluded, the rule could be simplified. Another special case is superposition refinement. When a subclass is created by superposition, the original computation on the inherited attributes is left unchanged. Additional functionality is provided through new attributes. Deriving rules for such special cases is left as future work.

Class refinement for concurrent objects is defined here as an extension of class refinement defined in [26, 27], following the general model of classes as self-referential structures with a delayed taking of the fixed point of [31, 16]. As known from [26], inheritance is not monotonic with respect to the refinement of the base class, leading to the so called fragile base class problem. This problem persists in the concurrent setting. With the possibility of self- and super-references between actions, it extends to actions.

For expressing symmetric communication and synchronization among several objects, multi-party actions have been studied in [6]. They can be introduced here without further difficulties.

Many interesting questions are connected with early returns. The remainder of a method into which we introduce a **release** statement cannot modify global variables. Otherwise, multiple changes that were previously executed in one atomic step could now be performed in multiple steps. The definition of trace refinement does not permit this. Making intermediate states visible and even making modifications to other global variable before the remainder's changes to global variables are performed are not legal refinements.

Modifications to other objects in the remainder of a method is a useful concept studied by Jones [20]. This is allowed if there are no other references to those objects and hence those changes are not observable to the remaining program. To this aim, Jones uses unique references. Spinning the idea of non-observability even further, the global state could also be updated in multiple steps if parts of it could be locked and be

guaranteed not to be observed until the remainder has been executed. The incorporation of such refinement steps into our formalism is an open issue.

The main advantage of a **release** statement over a "manual" atomicity refinement are the readability (no need to syntactically split the method into parts and to syntactically clutter all guards and the split method with synchronization and variable save statement) and the automatic resource locking. A version without resource locking would be possible and would allow additional interleavings, but would lead to practically rather strong proof conditions, making it less attractive.

The **release** statement could also be introduced into action systems without objects, for example within procedures. Objects, however, have the advantage that they encapsulate tightly coupled state components and, thereby, make it in practice easier to lock resources accessed by the remainder.

Acknowledgments We would like to thank Ralph Back and Marina Waldén for a number of clarifying discussions. The insightful comments of the anonymous referees are also gratefully acknowledged.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer Verlag, 1996.
2. Pierre America. Designing an object-oriented programming language with behavioral subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop*, Lecture Notes in Computer Science 489, pages 60–90, 1991.
3. Ralph Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.
4. Ralph Back. Refining atomicity in parallel algorithms. In *PARLE Conference on Parallel Architectures and Languages Europe*, Eindhoven, June 1989. Springer-Verlag.
5. Ralph Back. Atomicity refinement in a refinement calculus framework. Technical Report on Computer Science & Mathematics, Ser. A. No 141, Åbo Akademi, 1993.
6. Ralph Back, Martin Büchi, and Emil Sekerinski. Action-based concurrency and synchronization for objects. In T. Rus and M. Bertran, editors, *Transformation-Based Reactive System Development, Fourth AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software*, Lecture Notes in Computer Science 1231, pages 248–262, Palma, Mallorca, Spain, 1997. Springer-Verlag.
7. Ralph Back and Reino Kurki-Suonio. Decentralization of process nets with centralized control. In *2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 131–142. ACM Press, 1983.
8. Ralph Back and Reino Kurki-Suonio. Distributed co-operation with action systems. *ACM Transactions on Programming Languages and Systems 10:513–554*, 1988.
9. Ralph Back and Kaisa Sere. Action systems with synchronous communication. In E.-R. Olderog, editor, *IFIP Working Conference on Programming Concepts, Methods, Calculi*, pages 107–126, San Miniato, Italy, 1994. North-Holland.
10. Ralph Back and Joakim von Wright. Trace refinement of action systems. In B. Jonsson and J. Parrow, editors, *CONCUR '94: Concurrency Theory*, Lecture Notes in Computer Science 836. Springer-Verlag, 1994.
11. Ralph Back and Joakim von Wright. *Refinement Calculus – A Systematic Introduction*. Springer-Verlag, 1998.

12. Ralph Back and Joakim von Wright. Products in the refinement calculus. Technical Report 235, Turku Centre for Computer Science, February 1999.
13. Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. An approach to object-orientation in action systems. In *Mathematics of Program Construction*, Lecture Notes in Computer Science 1422, Marstrand, Sweden, 1998. Springer-Verlag.
14. K. M. Chandy and J. Misra. *Parallel Program Design – A Foundation*. Addison Wesley, 1988.
15. Ernie Cohen and Leslie Lamport. Reduction in TLA. In *Proceedings of CONCUR'98*, Lecture Notes in Computer Science 1466, pages 317–331. Springer-Verlag, 1998.
16. William Cook and Jens Palsberg. A denotational semantics of inheritence and its correctness. In *ACM Conference Object Oriented Programming Systems, Languages and Applications*, ACM SIGPLAN Notices, Vol 14, No 10, pages 433–443, 1989.
17. J.W. de Bakker and E.P. de Vink. Bisimulation semantics for concurrency with atomicity and action refinement. *Fundamenta Informaticae*, 20(1):3–34, 1994.
18. H.-M. Järvinen and R. Kurki-Suonio. DisCo specification language: Marriage of action and objects. In *Proceedings of 11th International Conference on Distributed Computing Systems*, pages 142–151, Arlington, Texas, 1991. IEEE Computer Society Press.
19. Cliff B. Jones. An object-based design method for concurrent programs. Technical report, University of Manchester, Department of Computer Science, December 1992.
20. Cliff B. Jones. Accomodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.
21. Leslie Lamport. The temporal logic of actions. *ACM Transactions of Programming Languages and Systems*, 16(3):872–923, 1994.
22. Leslie Lamport and Fred B. Schneider. Pretending atomicity. Technical Report Research Report 44, Compaq Systems Research Center, May 1989.
23. Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.
24. Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
25. David C. Luckham and Norihisa Suzuki. Verification of array, record, and pointer operations in pascal. *ACM Transactions on Programming Languages and Systems*, 1(2):226–244, October 1979.
26. Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In Eric Jul, editor, *ECOOP'98 – 12th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science 1445, pages 355–382, Brussels, Belgium, 1998. Springer-Verlag.
27. Anna Mikhajlova and Emil Sekerinski. Class refinement and interface refinement in object-oriented programs. In John Fitzgerald, Cliff Jones, and Peter Lucas, editors, *Formal Methods Europe'97*, Lecture Notes in Computer Science 1313, pages 82–101, Graz, Austria, 1997. Springer-Verlag.
28. Jayadev Misra. A discipline of multiprogramming. *ACM Computing Surveys*, 28A(4), December 1996.
29. Caroll C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.
30. Kaisa Sere and Marina Waldén. Data refinement of remote procedures. In *Proceedings of TACS 97*, Lecture Notes in Computer Science 1281, pages 267–294. Springer-Verlag, 1997.
31. Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *European Conference on Object Oriented Programming*, Lecture Notes in Computer Science 322, pages 55–77. Springer-Verlag, 1988.

Appendix

An Introduction to B

Martin Büchi

An Introduction to B

Martin Büchi

May 10, 2000

1 Introduction

This appendix gives a short introduction to the B method. Some familiarity with a model-based specification method, such as VDM, Z, the refinement calculus, or ad-hoc pre-/postconditions, is assumed. The purpose of this appendix is to provide the necessary background material for Papers I and II, which have been presented in contexts assuming a certain familiarity with B. The reader is referred to [1, 10, 11, 17] for full introductions to B. Most examples in this appendix are drawn from Paper I.

The B method, invented by Jean-Raymond Abrial [1], is a state-based method built on set theory and predicate logic. B is similar to VDM and Z. A comparison with these two methods is provided in Sect. 6.

The availability of two commercial tools for B [16, 2] has lead to a relatively wide acceptance of B in industry and CS curricula. Both tools include analyzers, animators, proof obligation generators, proof tools, code translators, and documentation facilities.

2 Development Overview

A typical development process in B looks as follows: First, we gather the requirements and write them down in an informal notation, e.g. plain English. This step is like for any other development method. The use of B starts with the next step.

Based on the informal requirements, we create a formal specification, referred to as a machine in B. This machine captures the observable behavior of our program, but does not detail how it is implemented. We animate (execute with the help of the user) this machine to check that it corresponds to the informal requirements. Furthermore, we perform certain consistency proofs on it.

Next we create an implementation —also using the B notation. The implementation is supposed to exhibit the same observable behavior as the original machine. We formally prove this using the refinement rules of B. In contrast to the machine, the implementation contains enough details so that it can be automatically translated into executable code. The current tools create C, C++, or Ada code, which can then be compiled with a normal compiler.

Refinement in B can be described as invent and verify. We write an implementation as a new text. In the same text we state how we believe the implementation refines the original machine. Finally, we prove that refinement actually holds. Thus,

refinement doesn't mean that we get the implementation by (mechanically) applying transformation rules to the specification.

In practice, it is often too cumbersome to directly move from a specification to an implementation. Too many decisions would have to be taken all at once. Therefore, the B method allows us to create a number or intermediate refinements. These refinements allow us to add implementation details step by step. For each step we prove that the observable behavior is preserved.

Layered development provides another possibility to partition the implementation task. Instead of implementing everything with just the language primitives, we import other modules to provide some services. The development of these auxiliary modules proceeds in the same fashion as for the original program.

3 Machines

Machines express original specifications. They define the syntactic interface and the externally visible behavior of a module. They do not say how the functionality is implemented. B machines are somewhat similar to C++ header files or Modula-2 definition modules with semantic annotations. The main difference is that B machines contain formal semantic specifications. The latter lend themselves to unambiguous mechanic reasoning.

Machines encapsulate a state and operations thereon. Thus they are abstract data structure modules in the terminology of the thesis' introduction (Sect. 2.1). A machine consists of a header, a state definition, an initialization, and operations.

3.1 Header

Machines may be parameterized by simple scalars and finite non-empty sets. The following example introduces a machine *Bank* with parameters *maxCustomers* and *maxAccounts*:

MACHINE
Bank(*maxCustomers*, *maxAccounts*)

The values of the machine parameters may be constrained with a *CONSTRAINTS* clause, which is written as a predicate on the parameters:

CONSTRAINTS
$maxCustomers \in 1 \,..\, 100000 \land maxAccounts \in 1 \,..\, 200000$

Parameters are instantiated upon inclusion into another machine and import into an implementation (Sect. 5).

Machines may declare constants. Their names are introduced in the *CONSTANTS* clause:

CONSTANTS
maxOpenFiles

Constants are typed and constrained in the *PROPERTIES* clause. In the example, *NAT1* denotes the set of positive natural numbers.

PROPERTIES
$maxOpenFiles \in \mathbf{NAT1}$

The valuation of constants is deferred to the implementation. It is done in the *VALUES* clause of the implementation.

Enumerated sets can be defined in the *SETS* clause. Such sets correspond roughly to enumerations in Pascal and C.

SETS
$FILE\_MODE = \{READ\_WRITE, TRUNCATE\_WRITE, READ, \mathrm{WRITE}\}$

Textual macros with parameters, like in C, can be introduced in the *DEFINITIONS* clause:

DEFINITIONS
$READ\_MODE == \{READ\_WRITE, READ\};$

3.2 State

The state space of a machine is determined by a set of variables. The *VARIABLES* clause introduces variable names. For example, the declaration below introduces the variables *customers*, *customerName*, and *customerYob*:

VARIABLES
customers, *customerName, customerYob*

The *INVARIANT* clause assigns types to the variables and relates them to each other. Thus, the invariant restricts the set of legal states. It expresses the static laws of the machine. The invariant is written as a predicate on the variables:

INVARIANT
$customers \subseteq CUSTOMER \land$
$customerName \in customers \rightarrow STRTOKEN \land$
$customerYob \in customers \rightarrow \mathbf{NAT} \land$
$customerName \otimes customerYob \in customers \rightarrowtail (STRTOKEN \times \mathbf{NAT})$

Here, *CUSTOMER* and *STRTOKEN* are sets, the declarations of which are not shown. The first conjunct states that *customers* is a subset of *CUSTOMER*. The symbol '$\rightarrow$' denotes total functions. Hence, the second conjunct expresses that *customerName* is a total function from *customers* to *STRTOKEN*. Applied to a customer, it returns the customer's name. Similarly, *customerYob* is used to represent the customers' year of birth. These first three conjunct express only basic typing information.

The last conjunct is more interesting. It states that no two customers have both the same name and the same year of birth. This kind of information cannot be expressed in most programming languages. The last conjunct uses a product type ('$\times$'), a total injective function ('$\rightarrowtail$'), and a direct relational product ('$\otimes$').

Sets and constants

BOOL	booleans *TRUE* and *FALSE*
NAT	natural numbers
NAT1	positive natural numbers
MAXINT	largest natural number for a given computer (e.g. 2^{31}-1)

Relation symbols

$\leftrightarrow$	arbitrary relations
$\rightarrow$	total functions
$\rightarrowtail$	total injective functions
$\nrightarrow$	partial injective functions

Functions on sets and relations

card(s)	cardinality of set *s*
dom(r)	domain of relation *r*
ran(r)	range of relation *r*
r^{-1}	inverse of relation *r*
$u \lhd r$	restriction of relation *r* to domain *u*
r[u]	relational image
f(x)	function application
$r \otimes s$	direct relation product (see text)

Sequences (q with domain *1..len(q)*)

[]	empty sequence
len(q)	length of sequence *q*
perm(s)	permutations of set *s*

Others

(x, y)	tuple (also written as $(x \mapsto y)$)
bool	converts a predicate into a boolean expression (e.g. *x:=bool(y=z)*)

Figure 1: Summary of B symbols

The direct relational product of two relations $r \in a \leftrightarrow b$ and $s \in a \leftrightarrow c$ with identical domain is denoted by $r \otimes s$ and defined as follows:

$$\mathrm{r} \otimes \mathrm{s} \stackrel{\mathrm{def}}{=} \{(\mathrm{x}, (\mathrm{y}, \mathrm{z})) \mid (\mathrm{x}, (\mathrm{y}, \mathrm{z})) \in \mathrm{a} \times (\mathrm{b} \times \mathrm{c}) \wedge (\mathrm{x}, \mathrm{y}) \in \mathrm{r} \wedge (\mathrm{x}, \mathrm{z}) \in \mathrm{s}\}$$

The symbols used in Papers I and II are summarized in Fig. 1. Additional symbols have their normal mathematical meaning.

Abstract and concrete variables

B differentiates between abstract and concrete variables. Because abstract variables are much more common, they are usually referred to simply as variables.

Abstract variables in a machine describe a model state. This model state is introduced to specify the effects of operations. An implementation may use variables with different names and types instead. Therefore, abstract variables cannot be directly read or written by implementations that import the machine.

B also provides concrete variables, which can be directly read in client constructs. Direct write access is not granted because otherwise the invariant could not be proved on a modular base and independent refinement would be impossible. Concrete variables are rarely used because they constrain the implementation and make it more difficult to modify specifications and implementations. Concrete variables are introduced in the *CONCRETE_VARIABLES* clause and, like abstract variables, typed in the *INVARIANT* clause. Concrete variables can only be of scalar types.

3.3 Initialization

The initialization assigns initial values to all variables. Figuratively, the initialization is executed upon loading of the machine.

The initialization of a machine is written as a generalized substitution without sequential composition. Generalized substitution is the B term for a slight extension of Dijkstra's language of guarded commands. The initialization of the above variables might look as follows, where '||' denotes parallel composition:

INITIALISATION
$customers := \emptyset \;||\; customerName := \emptyset \;||\; customerYob := \emptyset$

The initialization must establish the invariant. This is a consistency proof obligation of every machine. In our example, $\emptyset \subseteq CUSTOMER$ holds because the empty set is a subset of any set. Furthermore, the empty sets of tuples represents functions with empty domains. The direct relational product of empty relations is also the empty relation, which corresponds to an empty set of tuples. Hence, the above initialization establishes the invariant.

3.4 Operations

The state of a machine is modified and inspected through operations. They correspond to functions in C or methods in Java.

An operation consist of a precondition and a body. The precondition expresses under which circumstances an operation may be called. The precondition is a predicate on the state of the machine and the parameters. For example, in the operation *NewCustomer* (Fig. 2), the precondition after the keyword *PRE* states that the type of the parameter *name* must be *STRTOKEN* and that the type of *yob* must be *NAT*. Furthermore, the precondition requires that no customer with the same name and year of birth exists already in the database and that the latter is not full.

It is the caller's responsibility to assure that the precondition holds for every call. This is a proof obligation dictated by B.

The body of an operation is, like the initialization, written as a generalized substitution without sequential composition. The example in Fig. 2 uses the *ANY* specification statement. The *ANY* statement nondeterministically chooses a value for the bound

NewCustomer(*name*, *yob*) =
  PRE
    name ∈ *STRTOKEN* ∧ *yob* ∈ **NAT** ∧
    (*name*, *yob*) ∉ **ran**(*customerName* ⊗ *customerYob*) ∧ customers ≠ *CUSTOMER*
  THEN
    ANY *newCustomer* **WHERE** *newCustomer* ∈ *CUSTOMER* - *customers* **THEN**
      customers := *customers* ∪ {*newCustomer*} ||
      customerName(*newCustomer*) := *name* || *customerYob*(*newCustomer*):= *yob*
    END
  END;

Figure 2: Specification of operation *NewCustomer*

variable (i.e. *newCustomer*) such that the predicate after *WHERE* is satisfied and then executes its body delimited by *THEN* and *END*.

In the sample specification, *customerName*(*newCustomer*) := *name* is an abbreviation for *customerName* := *customerName* ∪ {*newCustomer* ↦ *name*}. Other generalized substitutions include the standard *IF* statement and the nondeterministic assignment *x* :∈ *S*, which assigns to *x* an arbitrary value of the set *S*.

Operations may also contain calls to other operations of included or seen (Sect. 5) machines. However, no calls can be made to operations of the same machine. Return values of operation calls are assigned with '←' rather than ':='.

For the consistency of a machine, we have to prove that the operations preserve the invariant whenever they are called with parameters satisfying the precondition.

4 Refinements and Implementations

Machines give concise descriptions of what should be done and what a client can expect, but they do not state how this is done. In intermediate refinements and in implementations we can add more details of the original informal specification and replace data structures and algorithms with more efficient ones. The laws of refinement guarantee that the behavior observable by clients is preserved.

In B, a machine can be refined to an implementation directly or with intermediate steps. The following introduces an intermediate refinement of our machine:

REFINEMENT
  Bank_0(*maxCustomers*, *maxAccounts*)

REFINES
  Bank

Like machines, refinements contain a state and operations thereon. The names and types of the variables of a refinement may differ from those of the corresponding machine. In our case, we replace the set *customers* of the machine by *nextCustomer* of a subrange type of *NAT*.

The state spaces of the machine and of the refinement are related to each other in the invariant. This is sometimes referred to as the linking or gluing invariant or as the abstraction or data refinement relation. This relation helps us prove that the refinement exhibits the same, or a more deterministic behavior as the machine.

In our case, we state that the set *customers* is equal to the set of numbers from *0* up to *nextCustomer-1*:

VARIABLES
nextCustomer, ...

INVARIANT
$nextCustomer \in 0 \,..\, maxCustomers \wedge$
$customers = 0 \,..\, nextCustomer\text{-}1 \wedge \ldots$

The initialization of a refinement must establish the invariant. Since the states of the machine and the refinement are linked via the invariant, this also implies that the refinement is in a corresponding state after initialization.

INITIALISATION
$nextCustomer := 0 \;||\; \ldots$

A refinement provides exactly the operations specified in the corresponding machine. The parameters are the same, but they are not syntactically repeated:

NewCustomer =
BEGIN
$nextCustomer := nextCustomer + 1;$
...
END;

Operations in refinements may contain sequential composition ';'. The refined operations must behave like their abstract counterparts for any actual parameters satisfying the precondition stated in the machine. The exact proof obligations are given by B's rules of (forward) data refinement.

An implementation distinguishes itself from an intermediate refinement through a number of things. An implementation starts with the keyword *IMPLEMENTATION*. Implementations may not contain nondeterministic constructs, such as the *ANY* statement. Furthermore, only concrete, but not abstract variables are permitted. Thanks to these restrictions, implementations can be automatically translated into executable code.

WHILE loops are only permitted in implementations. The loops contain a variant function to prove termination and an invariant to establish properties of the post state.

5 Modular Development

B provides for modular specification, refinement, and implementation. When we refine or implement a machine, we only have to consider that machine. We don't have to be aware of which clients use it in which ways.

Furthermore, B allows us to utilize other modules knowing only their specification, as provided by the machine construct. For example, we could implement our bank with the help of a database. For this purpose, we would import the machine *Database* in our implementation *Bank_1*. Note that we import the machine *Database*, not an implementation thereof. When we code *Bank_1*, we only have to look at the machine *Database*. The latter provides all the information we need in form of a concise specification. We do not need to care how *Database* is implemented.

B has four mechanisms for composing modules, providing for both inclusion and sharing. The different mechanisms can be used in different constructs. They are described below.

5.1 Inclusion in machines and refinements

We can include an instance of a machine into another machine or refinement. For example, a machine *RobustBank* could include an instance of the above machine *Bank*:

MACHINE
RobustBank

INCLUDES
Bank(100, 200)

The result is almost the same as textually copying *Bank* into *RobustBank*. There are three key differences:

1. The operations of *RobustBank* can only read, but not directly modify the (abstract and concrete) variables of *Bank*. This guarantees that the invariant of *Bank* cannot be invalidated by operations of *RobustBank*.

2. In operations of *RobustBank* we can call operations of the included *Bank*. (Remember that constructs cannot call their own operations.) Thus, variables of *Bank* can be modified in a controlled way.

3. By default, operations of the included *Bank* are not visible to clients of *RobustBank*. It is, however, possible to explicitly make operations visible to clients. Individual operations can be made visible with *PROMOTES*. By using *EXTENDS* in place of *INCLUDES* all operations get promoted.

Upon inclusion, we must instantiate the parameters of the included machine. Above we have instantiated *maxCustomers* with 100 and *maxAccounts* with 200.

We can include several instances of the same machine. In this case we have to rename the different instances. For example, we could have included two copies of *Bank* into our *RobustBank*:

INCLUDES
first.Bank(10,20), *second.Bank(30,40)*

The result is the same as if we had made two textual copies of the machine *Bank* and prefixed all identifiers therein with *'first.'* and *'second.'*, respectively.

The including construct becomes the focus of refinement, the included machine doesn't have to be implemented —unless another instance of the included machine is imported somewhere else. In the above example, this means that we do not have to refine or implement *Bank*.

5.2 Import in implementations

Implementations can import instances of other machines. For example, the implementation *Bank_1* could import an instance of *Database*. Upon import, we have to instantiate the parameters, e.g. the postulated *maxSize* parameter with *300*:

IMPLEMENTATION
Bank_1(*maxCustomers*, *maxAccounts*)

REFINES
Bank

IMPORTS
Database(300)

The operations of the implementation *Bank_1* may invoke operations of *Database*. Furthermore, they can read concrete variables of *Database*.

Abstract variables of *Database* cannot be read or modified in operations of *Bank_1*. However, they can be referenced in the invariant of *Bank_1* to express relationships. For example, the names of the customers may be stored in the database. In this case, we need to relate *customerName* of the refined machine *Bank* to the variable of the database representing this information. Assuming the latter is called *dbStrings*, the invariant of *Bank_1* may look as follows:

INVARIANT
$(0\,..\,nextCustomer\text{-}1) \lhd customerName = (0\,..\,nextCustomer\text{-}1) \lhd dbStrings \wedge \ldots$

Referencing variables of an imported machine in the invariant does not affect the behavior of the implementation. This is a pure aid for proving certain properties.

Renaming can also be used with *IMPORTS* to import multiple instances of a given machine. Each instance of a machine is imported by exactly one implementation. Thus, we cannot share instances of imported machines with *IMPORTS* alone.

5.3 Sharing

Multiple machines, refinements, and implementations can share an instance of a machine with *SEES*. Every instance of a seen machine must be imported by some implementation, which instantiates the parameters of the shared machine.

SEES provides read-only access to the shared machine. This means that the seeing machine can only invoke inquiry, but not modification operations of the shared machine. This restriction is necessary to avoid interference with the invariant of the implementation importing the shared machine. Assume that the implementation *IO_1* sees *Database*, which is imported by *Bank_1*. If an operation of *IO_1* were to modify

the variable *dbStrings* of *Database*, then the above invariant of *Bank_1* might be invalidated. This interference could only be detected with global, but not with modular proofs.

For dual reasons, variables of the shared machine *Database* cannot be referenced in the invariant of *IO_1*. Otherwise, operations of *Bank_1* might falsify the invariant of *IO_1*. In conclusion, sharing is restricted by the single writer/multiple readers paradigm. The importing implementation is the single writer, which is allowed to reference variables of the shared machine in its invariant. The seeing constructs are the readers, which are not allowed to reference variables of the shared machine in their invariants.

B has a second sharing mechanism called *USES*. It provides for limited sharing on the specification level only and is also restricted to a single writer. Furthermore, the shared and all the sharing machines must be included into the same construct. Hence, the applicability of *USES* is very limited.

5.4 Summary of composition mechanisms

The sheer number of composition mechanisms in B might be intimidating at first. However, empirical evidence shows that the different mechanisms address real needs. Except for *USES*, all mechanisms are widely applied.

The *INCLUDES* and *USES* clauses can be considered as weak or syntactic relations [3]. Their aim is to combine text of machine specifications; this structure is not reflected in subsequent refinements or in the final implementation. *SEES* and *IMPORTS* on the other hand are strong relations as the shared code will remain visible as a module in the final implementation. Sharing in B is restricted by the single writer/multiple readers paradigm.

6 Comparison of B with VDM and Z

In this section we compare B with VDM [9] and Z [15]. The main aim is to help the reader familiar with one of these methods put B into perspective.

Coverage VDM and B cover the whole development process from specification to implementation. Z also has rules for refinement, but in practice it is mostly used for specification only. B and VDM are methods. Z is a notation.

Logical foundation VDM [9] is based on the logic of partial functions. Z and B are based on similar formulations of set theory and first order predicate logic.

Syntax VDM and B use keyword-based textual notations. Both have a pure ASCII notation and a mathematical publication form. Z uses a semi-graphical schema notation. ASCII equivalents also exist.

Operation specification In VDM and B, the preconditions of operations are explicitly specified. In Z, the preconditions are not explicitly stated, but may be calculated from the schema definitions.

Z express the effect of an operation with a postcondition. B uses statements instead —much like imperative programming languages. VDM has both postcondition-based implicit operation specifications and statement-based explicit operation specifications.

In B and in explicit VDM operation specifications, only the variables that are explicitly assigned to are modified, all others remain unchanged. In implicit VDM operation specifications, only the global variables that are listed as writable in the operation's externals clause may be modified. In Z, the postcondition has to explicitly list the variables that remain unchanged.

The invariant is an explicit conjunct of every postcondition of Z and of implicit VDM operations. Thus, feasibility is the only proof obligation for such operation specifications. For B and explicit VDM operations, to which the invariant is not conjoined, we have to prove that they preserve the invariant (consistency).

Only VDM contains special syntax for specifying exceptions.

Modularization B has a strong modularization concept supporting information hiding, separate refinement, and layered development. Standard VDM and Z offer little support for modularization. Though several modularization extensions have been proposed and implemented [5, 7, 12].

Standardization The VDM specification language achieved ISO Standardization in 1996 [13]. Z is undergoing ISO standardization. Abrial's book [1] is the de facto standard for B. However, both industrial tools available for B deviate slightly from Abrial's book and are not fully compatible with each other.

Tool support All three methods are supported by a number of tools. Support for syntax and type checking, pretty printing, documentation, animation, and proving exist for all three methods. Code generators are only available for B and VDM.

Extensions Several object-oriented extensions of VDM and Z have been proposed and implemented. VDM++ [4, 6], an extension of VDM, supports object orientation, concurrency, and real time. Object-Z [14] extends Z with object-oriented concepts. Sum [8] extends Z with modules and explicitly states operation preconditions.

Translations from action systems, CSP, and UML diagrams to B have been proposed. However, no object-oriented or concurrent extensions of B have been published.

References

[1] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] B-Core. *B-Toolkit*. England, 1995. http://www.b-core.com/.

[3] Didier Bert, Marie-Laure Potet, and Yann Rouzaud. A study on components and assembly primitives in B. In *Proceedings of the first B conference*, pages 47–62, 3 rue du Maréchal Joffre, BP 34103, 44041 Nantes Cedex 1, 1996. IRIN Institut de recherche en informatique de Nantes.

[4] E.H. Dürr and J. van Katwijk. VDM++ — a formal specification language for object-oriented designs. In *Computer Systems and Software Engineering, Proceedings of CompEuro'92*, pages 214–219. IEEE Computer Society Press, 1992.

[5] J.S. Fitzgerald and C. B. Jones. Modularizing the formal description of a database system. In *VDM'90: VDM and Z – Formal Methods in Software Development*, pages 189–210. LNCS 428, Springer Verlag, 1990.

[6] The VDM Tool Group. The IFAD VDM++ language. Technical report, IFAD, October 1998.

[7] I. J. Hayes and L. P. Wildman. Towards libraries for Z. In J. P. Bowen and J. E. Nicholls, editors, *Z User Workshop: Proceedings of the Seventh Annual Z User Meeting*, Workshops in Computing. Springer Verlag, 1993.

[8] Wendy Johnston and Luke Wildman. The Sum reference manual. Technical Report 99-21, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072, Australia, November 1999.

[9] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, 1986.

[10] Kevin Lano. *The B Language and Method: A Guide to Practical Formal Development*. Springer Verlag London, 1996.

[11] Kevin Lano and Howard Haughton. *Specification in B: An Introduction Using the B Toolkit*. Imperial College Press, London, 1996.

[12] Marie-Laure Potet Yves Ledru and Rémy Sanlaville. VDM Modules. In John Fitzgerald and Peter Gorm Larsen, editors, *VDM in Practice*, pages 1–12, September 1999.

[13] P. G. Larsen and B. S. Hansen and H. Brunn N. Plat and H. Toetenel and D. J. Andrews and J. Dawes and G. Parkin and others. Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language, December 1996.

[14] Graeme Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 1999.

[15] J.M. Spivey. *The Z Notation*. Prentice Hall, second edition, 1992.

[16] Stéria Méditerranée. *Atelier-B*. France, 1996. http://www.atelierb.societe.com.

[17] J. B. Wordsworth. *Software Engineering with B*. Addison-Wesley, 1996.

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

http://www.tucs.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**