Diss. ETH No. 21939

Specified and Verified Reusable Components

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by
NADIA POLIKARPOVA
Master of Applied Mathematics and Informatics, SPb SU ITMO, Russia

born on
May 20th, 1985

citizen of
Russia

accepted on the recommendation of

Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. K. Rustan M. Leino, co-examiner
Prof. Dr. Peter Müller, co-examiner
Dr. Natarajan Shankar, co-examiner

2014

Acknowledgments

I would like to thank Bertrand Meyer for giving me the opportunity to become part of his group at ETH Zurich[1]. Bertrand has always seen a greater potential in me than I have, and respected my opinion even when it was opposite to his own; the freedom he granted me helped me grow into the researcher I am now.

I am grateful to my amazing co-examiners—K. Rustan M. Leino, Peter Müller, and Natarajan Shankar—for the effort they have put into evaluating my work. Their insightful comments and attention to detail helped improve this thesis a great deal. I am honored and flattered to be accepted into the "grown-up" scientific community by such brilliant people.

There is one person whose contribution to the success of this work is impossible to overstate. Carlo A. Furia is my scientific big brother; he was always there for me, giving sound advice in technical and cultural matters, editing my writing, and somehow turning every seeming failure into a research result.

Throughout my PhD I have been lucky to collaborate with many wonderful people: Julian Tschannen, Scott West, Yu Pei, and Yi Wei from ETH, as well as Michał Moskal and Rustan Leino[2] from Microsoft Research. This thesis would not have been the same without their contributions. And of course, I would not have even started my PhD without the help of Ilinca Ciupa, Anatoly Shalyto, and Danil Shopyrin, who led me through my first steps as a researcher during my undergraduate and master's years.

I could not have made it without the support of my two families: the one that raised me and the one I found at ETH. Apart from the people I have already listed, this second one also includes: Cristiano Calcagno, Georgiana Caltais, Claudia Günthart, Alexey Kolesnichenko, Benjamin Morandi, Andreas Leinter, Sebastian Nanz, Đurica Nikolič, Martin Nordio, Michela

[1]I did not start this sentence with "first and foremost", since I know how Bertrand hates cliché.

[2]You guys so rock!

Pedroni, Marco Piccioni, Chris Poskitt, Andrey Rusakov, Mischael Schill, Jiwon Shin, and Stephan van Staden. My irreplaceable officemates—Scott West and Christian Estler—are more than brothers to me: we are basically the same person.

A special place in these acknowledgments—and in my heart—belongs to Marco Trudel, who never failed to remind me that it is going to be okay.

Financing

This work was partially supported by the Swiss SNF under the projects FullContracts (200021-137931) and ASII (200021-134976); I also acknowledge the support of the Swiss National Supercomputing Centre, which providing infrastructure for computationally intensive experiments.

Contents

Abstract

For *reusable software components*—program modules designed for black-box usage in arbitrary, *a priori* unknown contexts—quality assurance is particularly important and easily justified. Although this is widely agreed upon, the industry standard is still far removed from a perfect world, in which components are unambiguously documented and their correctness is established with certainty. This thesis aims at providing programmers with practical tools and techniques for assessing and improving the quality of reusable components at various stages of the software development process.

Formal specifications play a central role in quality assurance, documenting the interface between a component and its clients and acting as oracles for verification. Writing good interface specifications—those that include all relevant details and none of the irrelevant—is challenging in the absence of precise guidelines and formal assessment criteria. The present work addresses this challenge with *model-based contracts*: a specification methodology that enhances Design by Contract with mathematical models, and supports strong yet abstract specifications. The thesis assesses feasibility and costs of deploying such strong specifications, and demonstrates their benefits, which include boosting automated testing, preventing inconsistent library designs, and decreasing the density of implementation defects.

Achieving high confidence in the quality of component implementations requires formal correctness proofs. Although static program verification has made significant progress in recent years, existing methods and tools provide insufficient support for model-based contracts, and for the design patterns often found in object-oriented component libraries. This thesis advances the state of the art in program verification with two contributions. First, it proposes *semantic collaboration*: a new methodology to reason about class invariants in the presence of inter-object dependencies; the methodology is flexible enough to accommodate advanced design patters, but comprises useful default annotations, which reduce the specification overhead in common scenarios. Second, the thesis details a practical verification methodology for model-based contracts, featuring advanced support for model classes, and an approach to frame specifications that works well for complex inheritance hierarchies. Both proposed methodologies have been implemented in the AutoProof program verifier for the Eiffel programming language.

Another contribution of this thesis facilitates understanding and debugging failed verification attempts—one of the biggest remaining obstacles to usable program verification.

Practical solutions targeting reusable components must be evaluated on real software libraries. Two Eiffel data structure libraries, EiffelBase and its successor EiffelBase2, serve as case studies throughout the thesis. In particular, EiffelBase2—the first example of a data structure library developed from the start with strong specifications and verified (to a significant extent) for full functional correctness—embodies the vision of high-quality reusable components promoted in this work.

Zusammenfassung

Für *wiederverwendbare Software-Komponenten*—Programmmodule, die zur Black-Box-Nutzung in verschiedenen, im Voraus unbekannten Kontexten entwickelt werden—ist Qualitätssicherung besonders wichtig und der damit verbundene Aufwand gerechtfertigt. Obwohl diese Ansicht weitläufig akzeptiert wird, bleibt der Industriestandard weit von dem Idealfall entfernt, in dem alle Komponenten eine eindeutige Dokumentation haben und ihre Richtigkeit zweifellos sichergestellt ist. Ziel dieser Dissertation ist es, Programmierern anwendbare Werkzeuge und Techniken zur Verfügung zu stellen, die eine Bewertung und Verbesserung der Qualität von wiederverwendbaren Komponenten erlauben.

Formale Spezifikationen spielen bei der Qualitätssicherung eine zentrale Rolle, da sie die Schnittstelle zwischen Komponenten und ihre Kunden dokumentieren, und zur Verifikation von Implementierungen verwendet werden. Gute Schnittstellenspezifikationen zu schreiben—solche, die alle relevanten aber keine irrelevanten Details beinhalten—ist ohne genaue Richtlinien und formalen Bewertungskriterien schwierig. Die vorliegende Arbeit beschreibt einen Ansatz zur Überwindung dieser Schwierigkeiten mit Hilfe von *Modell-basierten Verträgen*: eine Spezifikationsmethode, die Design by Contract um mathematische Modelle ergänzt, und damit starke aber zugleich abstrakte Spezifikationen erlaubt. Die Dissertation evaluiert Durfürbarkeit und Kosten solch starker Spezifikationen, und zeigt ihren Nutzen auf, einschliesslich der Vorteile für automatische Testverfahren, der Vermeidung von inkonsistenten Bibliothek-Entwürfen, und der Abnahme der Häufigkeit von Implementierungsfehlern.

Um eine hohe Implementierungsqualität von Komponenten zu garantieren, benötigt man Richtigkeitsbeweise. Trotz bedeutenden Fortschritten, die statische Programmverifikation in den letzten Jahren gemacht hat, bieten existierende Methoden und Werkzeuge ungenügende Unterstützung für Modell-basierte Verträge, sowie für einige Entwurfsmuster, die häufig in den objektorientierten Komponenten-Bibliotheken zu finden sind. Diese Dissertation trägt zum Fortschritt der Programmverifikation in zweierlei Hinsicht bei. Erstens führt sie eine neuartige Methode, *Semantische Kollaboration* genannt, ein, um über Klasseninvarianten in Gegenwart von Abhängigkeiten zwischen Objekten schlussfolgern zu können; die Methode ist flexibel genug sich komplexen Entwurfsmustern anzupassen, enthält aber zugleich nützliche Standardwerte, die die Kosten einer Spezifikation in Normalfall gering halten. Zweitens stellt die Dissertation eine praktische Verifikationsmethode für Modell-basierte Verträge vor, die eine fortgeschrittene Unterstützung von Modell-Klassen bietet und einen Ansatz für Frame-Spezifikationen umfasst, der gut mit komplexen Vererbung-Hierarchien funktioniert. Beide Methoden wurden in AutoProof, einem Programmverifizierer für die Eiffel Programmiersprache, implementiert.

Ein weiterer Beitrag der vorliegenden Arbeit erleichtert das Verstehen und die Fehlerbehebung im Fall von misslungenen Beweisversuchen—eines der grössten verbleibenden Hindernisse für nutzbare Programmverifikation.

Praktische Lösungen, die auf wiederverwendbare Software-Komponenten zielen, müssen anhand echter Software-Bibliotheken evaluiert werden. Zwei Eiffel Datenstruktur-Bibliotheken, EiffelBase und ihr Nachfolger EiffelBase2, dienen in der gesamten Dissertation als Fallstudien. Insbesondere, EiffelBase2—das erste Beispiel einer Datenstruktur-Bibliothek, die von Anfang an mit starken Spezifikationen entwickelt wurde und deren volle funktionelle Richtigkeit grösstenteils bewiesen wurde—verkörpert die in dieser Arbeit verfolgte Vision von hochwertigen wiederverwendbaren Komponenten.

Being abstract is something profoundly different from being vague… The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.

Edsger W. Dijkstra

Chapter 1

Introduction

1.1 Motivation and Goal

Correctness is one of the most fundamental aspects of software quality [84]. Since one can only judge whether a software system is correct relative to a specification, ensuring correctness comprises two principal activities: *specifying* the system by formalizing informal (often implicit) requirements, and *verifying* that the implementation satisfies the specification.

The degree to which these activities are carried out in practice is a matter of cost-to-benefit ratio. Different verification techniques offer different trade-offs between the amount of effort they require and the level of assurance they provide. For example, fully automatic static and dynamic analyses support an inexpensive way of revealing simple errors. Code review and testing (manual or automated) provide a medium level of assurance: both are capable of revealing deeper errors than fully automatic analyses, but can never show their absence. At the high end of the assurance spectrum, formal proofs of full functional correctness can guarantee deep properties of the system, but require highly-trained experts and "heroic" effort [63, 71].

This motivates two important research goals in the area of software correctness: (*i)* reducing the amount of effort required by existing families of verification techniques, and (*ii)* increasing the diversity of techniques, so that the most appropriate solution can be chosen for a required assurance level. The present work contributes to both of those goals.

While in most areas of software industry high-assurance techniques are considered prohibitively expensive, several factors can justify their use. One example is extremely high cost of software errors, common for safety- and mission-critical systems. Another example is software reuse: *reusable soft-*

ware components,[1] are developed once and reused as a black box in a large number of systems, thus minimizing the amortized cost of quality assurance.

Practitioners recognize that quality is particularly important for reusable components (e.g. [14]), which makes such components attractive for verification in two ways: ensuring their correctness is not only more important than for system-specific code, but also more realistic, since the starting point is a module with well-defined interface and functionality, as well as high-quality implementation and documentation.

At the same time, verifying reusable components poses challenges that do not arise in whole-system verification, since the author of a component is not aware of the context in which it will be used. For specifications it raises the question of what properties they should express, since it is unknown which aspects of a component's behavior will be relevant in the client's context. For verification it limits the range of applicable techniques to *modular* approaches, where verifying each module of a system in isolation implies correctness of the system as a whole. In addition, unlike safety-critical systems, which might give up powerful programming constructs in the name of correctness [7], general-purpose libraries have to be written in an expressive programming language within a mainstream paradigm, and cannot be limited to using the most basic language features, data structures and algorithms.

The **goal** of this thesis is to advance the state the art towards making high-quality reusable components—with coherent design, detailed documentation, and provably correct implementations—an accepted software engineering practice. To this end, the thesis proposes practical approaches to specification and verification of reusable components at several levels of assurance.

1.2 Challenges and Contributions

The starting point for this work is Design by Contract [83]: a practical approach to specification, originally implemented for the Eiffel programming language, and more recently supported by other environments [28, 2]. Design by Contract enables programmers to write specifications in the form of *contracts*, using expressions from the programming language. Empirical studies show [20, 26, 106, 41] that Eiffel programmers indeed write contracts, but those contracts are sometimes incorrect and generally incomplete. We set out to find a way of improving these partial specifications, and making

[1]The term *component* is used in a broad sense, as a synonym of *module*; in an object-object oriented system, a component is typically a class or a library of classes.

use of better contracts to ensure software correctness by means of testing and proofs.

EiffelBase2. The present work aims for stronger specifications, while retaining the practical nature of Design by Contract; thus all proposed techniques have to be evaluated on a realistic software system of a substantial size. As the central case study of the thesis we have developed *EiffelBase2*: an Eiffel container library, which serves both as a motivation for the solutions proposed throughout the thesis, and as evidence of their feasibility. Containers are an archetype of reusable components, since a basic container library is a part of every programming language, and it is reused virtually in every program written in that language. In addition to serving as a testbed for techniques proposed in the present work, EiffelBase2 is a valuable artifact *per se*, offering high-quality reusable components to the Eiffel community, and contributing to the Verified Software Initiative [54].

Better specifications. We first investigate how to turn partial contracts into *strong specifications*, which would include everything a client might want to know about the behavior of a component. To express strong specifications, we adopt the *model-based* approach [58, 49, 98, 134], which uses well-understood mathematical concepts (sets, bags, relations, sequences, etc.) to define the semantics of program operations. The main challenge is giving developers precise guidelines and tools for constructing strong specifications and assessing their quality, which serves the purpose of both improving specifications and reducing their cost. To this end, we develop a specification methodology called *model-based contracts*, underpinned by four formally defined quality criteria, which guarantee that a specification is both as strong and as abstract as possible, for a given component interface. We evaluate practical feasibility of the methodology by specifying EiffelBase2.

Better testing. Strong model-based contracts only describe the interface of a software component, and are in general insufficient for a program verifier to carry out a correctness proof (which normally requires significant additional effort in the form of supplying *auxiliary annotations*). We suggest, however, that strong specifications can also benefit software quality with no additional effort; in particular they can play an important role in design and testing. Intuitively, checking stronger contracts at runtime should discover more (and deeper) faults. The question is whether this effect is significant in practice, and whether the runtime overhead of checking more complex specifications neutralizes the benefit of strong contracts given a fixed testing budget. As a result of an extensive empirical study we discovered that strong model-based contracts make very effective test oracles, while incurring only moderate specification overhead, which puts testing against such specifications in an attractive spot on the landscape of verification techniques. At the same time, through EiffelBase2 we observed that software developed with strong specifications from the start exhibits significantly fewer faults and architectural inconsistencies that software developed with traditional Design by Contract, which confirms the effectiveness of model-based contracts as a design methodology.

Better proofs. Next, we investigate full functional correctness proofs of reusable components. More precisely, we are interested in building an *auto-active program verifier* [76] that can prove, given appropriate auxiliary annotations, the kind of complete model-based contracts found in EiffelBase2. One central challenge we encountered is related to *class invariants*[2]—a notion that is inherent in object-oriented programming and thus a desirable feature to support in a verifier [121]. Existing invariant methodologies [9, 77, 11, 92, 80, 88, 120, 30] are either not directly applicable to automatic reasoning or provide insufficient support for non-hierarchical object structures, which arise in common design patters[3]. To tackle this problem we propose *semantic collaboration*: a new methodology to specify and reason about invariants of arbitrary object structures, which models inter-object dependencies by semantic means. To evaluate semantic collaboration we compiled a set of six benchmark problems, each of which contains a unique challenge for an invariant methodology.

The second set of challenges has to do specifically with model-based contracts. Supporting this style of specification in a verifier requires relating the concrete state of a class to its model in a way that is sound, practical, and concise (in terms of auxiliary annotations). We extend an existing program verifier for Eiffel, AutoProof [5], with support for models, as well as our new invariant methodology, and use it to verify the core of EiffelBase2.

Debugging verification. The final part of this work is devoted to making the verification process more incremental. When a program proof fails, it is often hard to understand what went wrong due to false positives and the absence of concrete executions that expose parts of the program responsible for the failure. To alleviate this issue, we propose a technique to automatically generate executions of programs annotated with complex specifications (such as those used in full functional correctness proofs). Our approach combines symbolic execution and SMT constraint solving to generate small tests that are easy to read and understand. We implement the technique for the Boogie intermediate verification language [73], which is used as a back-end in AutoProof. Our implementation is available as a tool called Boogaloo [15].

[2]Also known as *object invariants* or *representation invariants*.

[3]For example, the Iterator pattern, used extensively in EiffelBase2.

Summary of Contribution. The following list summarizes the main contributions of this thesis, along with the publications where they initially appeared.

1. *Model-based contracts*: a methodology for constructing strong specifications and assessing their quality.
   [Polikarpova N., Furia C., Meyer B., *Specifying Reusable Components*, VSTTE'10]

2. An empirical study showing that *testing against strong specifications* discovers significantly more faults than testing against traditional contracts, with reasonable annotation overhead.
   [Polikarpova N., Furia C., Pei Y., Wei Y., Meyer B., *What Good Are Strong Specifications?*, ICSE'13]

3. *Semantic collaboration*: a methodology to specify and reason about class invariants of arbitrary object structures, implemented in an auto-active verifier.
   [Polikarpova N., Tschannen J., Furia C., Meyer B., *Flexible Invariants through Semantic Collaboration*, FM'14]

4. A practical *verification methodology for model-based contracts*, implemented in an auto-active verifier.

5. *EiffelBase2*: a specified and partially verified container library, used in practice.

6. *Boogaloo*: a tool for debugging failed verification attempts.
   [Polikarpova N., Furia C., West S., *To Run What No One Has Run Before*, RV'13]

1.3 Terminology

In the rest of this thesis, we refer to common object-oriented programming constructs using the following terminology, which is established in the Eiffel community. A program is a collection of *classes*. Each class is composed of *attributes* (data) and *routines* (operations), collectively known as *features*. Routines with a return value are called *functions*, and otherwise *procedures*. Procedures intended for initialization of newly created objects are called *creation procedures* (analogous to constructors in Java and C#). The term *query* encompasses functions and attributes, while *command* is a non-creation procedure. The first (implicit) argument of a feature call is known as the *target*; a routine body can refer to its target though a built-in entity `Current` (`this` in Java and C#). A function can refer to its return value through a built-in variable `Result`.

A class C is *effective* if it supplies implementations of all of its features, otherwise it is *deferred* (abstract). C can directly inherit from zero or more *parents*; classes that directly inherit from C are called its *children*. *Ancestors* of C include C itself and, recursively, the ancestors of all its parents; symmetrically, its *descendants* include C and the descendants of all its children.

The built-in entity `Void` (`null` in Java and C#) denotes a reference that is not attached to any object.

1.4 Outline

The rest of the dissertation is organized as follows. Chapter 2 introduces the EiffelBase2 library, which is used to evaluate the techniques proposed in the following chapters. Chapter 3 details the specification and design methodology based on model-based contracts; in particular, it formalizes the notion of strong specifications. Chapter 4 describes our experiments with using model-based contracts in testing. Chapters 5 and 6 are devoted to correctness proofs, with the former focusing on class invariants, and the latter targeting model-based abstraction techniques and describing the verification of EiffelBase2. Chapter 7 introduces Boogaloo: a technique and tool for debugging verification, based on symbolic execution. Finally, Chapter 8 draws conclusions and discusses directions for future research.

Chapter 2

The EiffelBase2 Library

This chapter introduces the EiffelBase2 library, which serves to motivate and evaluate the techniques proposed in the rest of the thesis. Specification and verification of EiffelBase2 are presented in the following chapters, together with the corresponding techniques; here we only give a brief overview of the library design, scope, and scale.

2.1 Overview

EiffelBase2 is a general-purpose data structure library for Eiffel. It provides containers such as arrays, lists, sets, tables, stacks, queues, and binary trees; iterators to traverse these containers; and comparator objects to parametrize containers with respect to arbitrary equivalence and order relations on their elements. The current version of EiffelBase2 includes 72 classes totaling about 11'500 lines of code; these figures make EiffelBase2 a library of substantial size with realistic functionalities, comparable to those of .NET collections and java.util collections.

Since April 2012, EiffelBase2 is distributed together with EiffelStudio, the main Eiffel IDE. For three years, the library has been successfully employed in teaching introductory programming at ETH Zurich, each time used extensively by over 200 students. We have also relied on EiffelBase2 ourselves for the development of Traffic 4: a graphical library that models and visualizes public transportation in the city, also used in the introductory programming course [125]. These experiences confirm that EiffelBase2 is usable in practice.

The latest version of the library source code is available from the repository [40].

2.2 Design Goals

EiffelBase2 is intended as a replacement for the *EiffelBase* library, which has played a central role in Eiffel development for over twenty years. EiffelBase is representative of mature Eiffel code, extensively exploiting traditional Design by Contract, and widely used in Eiffel applications. This makes it an excellent reference point for comparative studies of new approaches in the areas of design, contracts, or implementation quality.

EiffelBase2 aims at providing the same general set of functionalities as EiffelBase; its ultimate goal, however, is specifying and proving full functional correctness—backward compatibility is not one of our primary concerns. This implies that EiffelBase2 revisits and solves any deficiency and inconsistency in the design of EiffelBase that impedes achieving full functional correctness or hinders the full-fledged application of formal techniques. In particular, we simplify the inheritance hierarchy of EiffelBase, which, as demonstrated in the following chapters, prevents strong specifications at high levels of abstraction and causes multiple instances of unexpected behavior.

Another design choice reconsidered in EiffelBase2 concerns the use of *internal cursors*: each EiffelBase container stores an additional piece of state—the cursor position, which enables iteration though the container without creating a separate iterator object. While convenient in simple cases, internal cursors do not support multiple simultaneous iterations, complicate the container abstraction, and impose a cumbersome save-restore policy in order to avoid side effects in functions.

2.3 The EiffelBase2 Architecture

EiffelBase2 consists of two (mutually dependent) libraries: the container library, which includes data structures proper, and the Mathematical Model Library (MML), which provides immutable classes used to express model-based contracts. In the following we concentrate on the container library, and present MML in Sect. 2.3.2.

The container part of EiffelBase2 is split into two class hierarchies: *containers* and *streams*. A container is a finite storage of values, while a stream provides linear access to a set of values. A stream is not necessarily bound to a container, e.g. a `RANDOM` stream observes an infinite sequence of pseudo-random numbers. Streams that traverse containers are called *iterators*.

Fig. 2.1 and 2.2 depict the class diagram of the two hierarchies. All EiffelBase2 class names start with `V_` (for Verified), but the prefix is omitted from the diagrams for brevity. The names of *deferred* (abstract) class are written

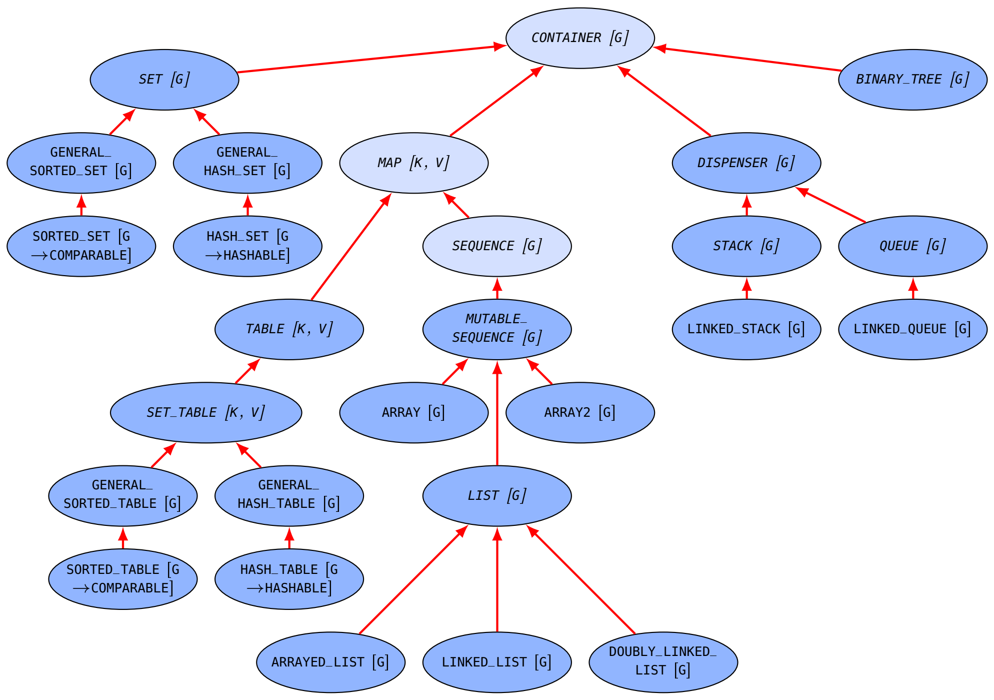


Figure 2.1: Container class hierarchy.

in ***ITALICS***; lighter fill color indicates that a class provides an *immutable* interface to the underlying data; in particular, it is impossible to change the content of a container through an immutable iterator.

The following table gives brief descriptions of EiffelBase2 container classes:

CLASS	DESCRIPTION
CONTAINER	Containers for a finite number of values.
SET	Container where all elements are unique with respect to some equivalence relation. Elements can be added and removed.
GENERAL_SORTED_SET	Sets implemented as binary search trees with arbitrary order relation and equivalence relation derived from order.
SORTED_SET	Sorted sets with order relation on keys provided by COMPARABLE.
GENERAL_HASH_SET	Hash set with arbitrary equivalence relation on keys and hash function. Implementation uses chaining.
HASH_SET	Hash sets with hash function provided by HASHABLE and with reference or object equality as equivalence relation on keys.
MAP	Containers where values are associated with keys. Keys are unique with respect to some equivalence relation.
SEQUENCE	Containers where values are associated with integer indexes from a continuous interval.
MUTABLE_SEQUENCE	Sequences where the value at a given index can be updated.

`ARRAY`	Indexable containers, whose elements are stored in a continuous memory area.
`ARRAY2`	Two-dimensional arrays.
`LIST`	Indexable containers, where elements can be inserted and removed at any position.
`ARRAYED_LIST`	Lists implemented as arrays.
`LINKED_LIST`	Singly linked lists.
`DOUBLY_LINKED_LIST`	Doubly linked lists.
`TABLE`	Maps where key-value pairs can be updated, added, and removed.
`SET_TABLE`	Tables implemented as sets of key-value pairs.
`GENERAL_SORTED_TABLE`	Tables implemented as binary search trees with arbitrary order relation on keys and equivalence relation on keys derived from order.
`SORTED_TABLE`	Sorted tables with order relation on keys provided by `COMPARABLE`.
`GENERAL_HASH_TABLE`	Hash tables with arbitrary equivalence relation on keys and hash function. Implementation uses chaining.
`HASH_TABLE`	Hash tables with hash function provided by `HASHABLE` and with reference or object equality as equivalence relation on keys.
`DISPENSER`	Containers that can be extended with values and make only one element accessible at a time.
`STACK`	Dispensers where the latest added element is accessible.
`LINKED_STACK`	Linked implementation of stacks.
`QUEUE`	Dispensers where the earliest added element is accessible.
`LINKED_QUEUE`	Linked implementation of queues.
`BINARY_TREE`	Binary trees (doubly linked implementation).

2.3.1 Specification and Verification Challenges

The semantics of data structures offered by EiffelBase2 is, in general, well-understood, and their implementations are rather straightforward. Nevertheless, these textbook data structures present commonly acknowledged challenges when it comes to specification and verification [56, 70]

First, the library creates and maintains complex object structures in the heap. Some of those structures are hierarchical (also called *aggregate objects*), where one object contains another as part of its internal representation; others are *collaborative*, where multiple objects at the same level of abstraction work together to achieve a common goal. The most prominent example of the latter is the collaboration between a container and its iterators.

Reasoning about complex object structures is complicated by aliasing, and describing their shapes (e.g. in routine footprints) requires powerful abstraction mechanisms. One particular challenge is specifying the consistency of object structures while preserving information hiding. For example, in order for an iterator to function properly, its state needs to be consistent with the state of the target container. Modifying the container can invalidate its

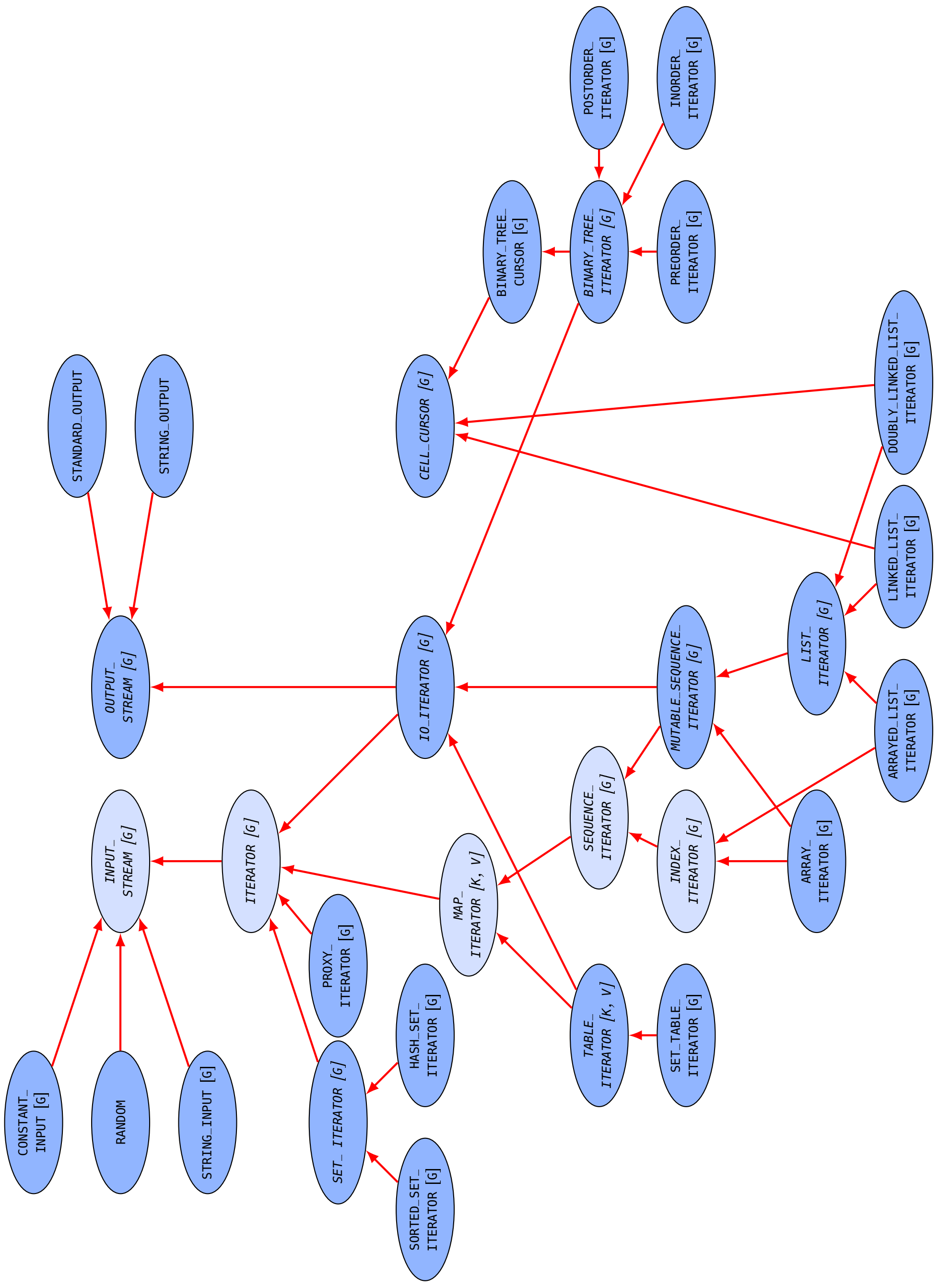


Figure 2.2: Stream and iterator class hierarchy.

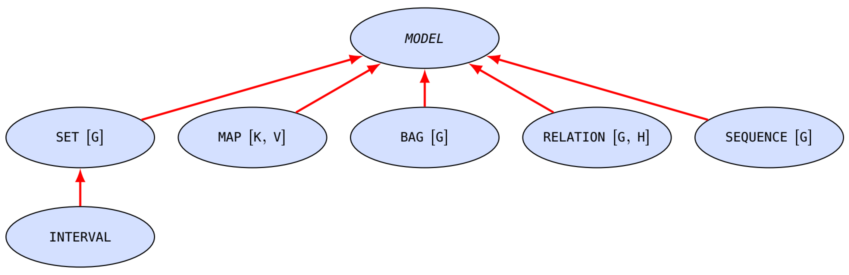


Figure 2.3: Class diagram of the Mathematical Model Library.

iterators, and lead to unexpected behavior. The challenge is to ensure that this never happens, while still allowing an unbounded number of iterators to be attached to a container at the same time.

Second, EiffelBase2 makes substantial use of inheritance. At the design and interface specification stages, one challenge is developing abstract components, such as `CONTAINER` or `DISPENSER`. On the one hand, they need to be general enough to naturally encompass several more specialized components; on the other hand, they need to be sufficiently self-contained and precise to be useful for clients and allow for strong specifications. For verification, inheritance poses additional challenges in terms of modularity and information hiding: most importantly, when the dynamic type of an object is unknown, precise definitions of its state and operations are unavailable, which aggravates the issues posed by aliasing and inter-object dependencies.

Finally, EiffelBase2 uses functional objects (called *agents* in Eiffel) to implement higher-order operations and to parametrize containers with various relations and functions on their elements. A practical and flexible way of reason about functional objects is still an open problem in verification.

2.3.2 The Mathematical Model Library

The Mathematical Model Library is a collection of immutable classes that serve as programming-language counterparts of the mathematical concepts used in model-based contracts. Similar libraries exist in other languages that support model-based specification style, e.g. JML [69]; the first version of MML in Eiffel was implemented by Schoeller [111].

Fig. 2.3 depicts the class diagram of MML (as in Fig. 2.1 and 2.2, the prefix `MML_` is omitted for brevity). The library consists of 7 classes, totaling 1'960 lines of code. Their features correspond to widely used mathematical operations, such as set union and sequence concatenation. The library provides implementations of those operations, which can be used for runtime checking of model-based contracts. The semantics given to the model classes by the program verifier is discussed in Chapter 6.

Chapter 3

Specifying Reusable Components

The first step towards ensuring correctness is creating a formal specification. In this chapter, we explore what constitutes a "good" specification for a reusable component, and offer guidelines for constructing such specifications in the context of an object-oriented programming language.

3.1 Introduction

The case for precise software specifications involves several well-known arguments; in particular, specifications help understand the problem before building a solution, and they are necessary for verifying implementations. In the case of a library of reusable software components, precise specifications have another application: providing client programmers with a detailed description of the component interface (the API); such descriptions are called *behavioral interface specifications* [50].

The main purpose of an interface specification for a reusable component is to enable practical reasoning about programs that use the component without having access to its implementation. This entails two important and somewhat conflicting "quality" objectives: on the one hand, an interface specification has to be *strong enough* to fully describe the relevant effects of component operations; on the other hand, it has to be *abstract enough* to allow several concrete implementations and to shield the clients from any complexity associated with a particular implementation (in other words, one should avoid implementation bias [58]). Creating high-quality specifications satisfying these objectives is hard in the absence of detailed guidelines and formal assessment criteria.

One of the most practical approaches to interface specifications is based on Design by Contract [83, 84]; it allows authors of reusable modules to equip them with *contracts*, usually in the form of routine pre- and postconditions, and class invariants. As one of its key features, Design by Contract relies on an assertion language embedded in the programming language to express contracts. This approach has two important advantages: specifications expressed in a programming language are more easily taught to programmers, and they can be checked at runtime, thus playing a major role in testing and debugging. Another aspect that contributed to the practical success of Design by Contract is incrementality: unlike most "all-or-nothing" formal development methods (e.g. [1]), Design by Contract takes the "anything goes" view: it embraces partial specifications, and provides incremental benefits starting with very few, very simple assertions.

This traditional focus on partial specifications, combined with insufficient expressiveness of the assertion language in comparison with more flexible mathematical notations, in practice results in weak contracts. As an example, the postcondition of a `push` operation on a stack in the existing standard Eiffel library specifies what the new top of the stack will be, and that the number of items will increase by one, but it does not state that the existing stack elements are unaffected. This example is typical: several empirical studies [20, 26, 106, 41] indicate that in practice Eiffel classes contain many contracts, but they cover only part of the programmer's informal understanding of the behavior. Thus, building high-quality behavioral interface specification on top of Design by Contract requires both increasing the expressiveness of the assertion language, and establishing more strict rules about what should actually go into contracts.

One of the first formal treatments of the quality aspect of interface specifications was developed for *algebraic specifications* of abstract data types (ADTs) [47, 48, 138]. Arguably the most influential work is by Guttag and Horning [48], which defines the notion of *sufficient completeness* of a set of axioms defining an ADT.

Algebraic specifications are very general, but hard to understand and construct [132]; an alternative *model-based* approach [58, 49, 98, 134], aims to simplify understanding and writing specifications by expressing them in terms of a fixed collection of well-understood and highly reusable mathematical theories. For the construction of a model-based specification, one of the most important design decisions is the choice of the mathematical model (also called "conceptual model" or "mental model" [96]) for a component; this choice largely determines the quality of a specification, and thus the quality of the component itself [133].

Integrating the model-based approach into Design by Contract led to the concept of *model classes* in Eiffel [112, 111] and a similar mechanism in the Java Modeling Language (JML) [69, 24][1]. Model classes represent mathematical theories in the programming language, and thus enable model-based specifications within the standard assertion language of Design by Contract. We will refer to such specifications as *model-based contracts*. The **goal** of this chapter is to explore and formalize quality criteria for model-based contracts, and to turn these criteria into concrete practices and tools.

Sect. 3.2 starts with some real-world examples of weak contracts in Eiffel libraries, which fail to prevent incorrect implementations and poor designs; it then demonstrates how strong model-based contracts address these issues. Sect. 3.3 formalizes high-quality model-based specifications in the language of abstract data types: it shows how an ADT defines unambiguously a notion of *abstract state space*, which in turn determines the *model* of the ADT; it also outlines the process of constructing model-based specifications, and finally proposes formal definitions of four quality criteria—*completeness*, *observability*, *closure*, and *controllability*—which underpin the intuitive notions of "strong" and "abstract" specifications. Sect. 3.4 applies these theoretical concepts in the context of Eiffel—an imperative, object-oriented language—resulting in a practical specification methodology for strong model-based contracts; it also discusses tool support for checking quality criteria, formalized in the previous section.

Sect. 3.5 describes two case studies that use the proposed specification methodology to develop libraries of data structures with strong contracts. The results show that the methodology is feasible and successful in delivering well-designed components with expressive—usually complete—specifications. Most advantages of standard Design by Contract are retained, while pushing a more accurate evaluation of design choices and an impeccable definition of interfaces.

3.2 Motivating Examples

3.2.1 Some limitations of Design by Contract

Let us demonstrate the shortcomings of traditional contracts on a couple of examples from the EiffelBase library [39] (see also Chapter 2).

Fig. 3.1a shows a portion of class `LINKED_LIST`, implementing a singly linked list. Features `count` and `index` record respectively the number of elements stored in the list and the current position of the internal cursor. Routine `put_right` inserts an element `v` to the right of the current position of the

[1]JML uses the terms *modeling types* or *immutable types*.

```
class LINKED_LIST [G]
  count: INTEGER
    -- Number of elements
  index: INTEGER
    -- Current cursor position

  item: G
    -- Value at cursor position
    require 1 ≤ index ≤ count
    do ... end

  put_right (v: G)
    -- Add ‘v’ to the right of cursor
    require 0 ≤ index ≤ count
    do ...
    ensure
      count = old count + 1
      index = old index
    end

  duplicate (n: INTEGER): LINKED_LIST
    -- Copy of sublist of length ‘n’
    -- beginning at current position
    require n ≥ 0
    do ...
    ensure Result.index = 0
    end
end
```

(a) Class LINKED_LIST.

```
deferred class TABLE [G, K]
  put (v: G ; k: K)
    -- Associate value ‘v’ with key ‘k’
    require valid_key (k)
    deferred
    end
end

class ARRAY [G]
inherit TABLE [G, INTEGER]
  put (v: G ; i: INTEGER)
    -- Replace ‘i’-th entry,
    -- if in index interval, by ‘v’
    do ...
    end
end

class HASH_TABLE [G, K →HASHABLE]
inherit TABLE [G, K]
  put (v: G ; k: K)
    -- Insert ‘v’ with key ‘k’
    -- if there is no other item
    -- associated with ‘k’
    do ...
    end
end
```

(b) Class TABLE with two descendants.

Figure 3.1: Snippets from EiffelBase classes.

cursor, without moving it. The postcondition of the routine (clause **ensure**) asserts that inserting an element increments `count` by one but does not change `index`. This is correct, but it does not capture the gist of the semantics of insertion: the list after insertion is obtained by all the elements that were in the list up to position `index`, followed by element `v` and then by all elements to the right of `index`.

Expressing such complex facts is tedious with the standard assertion language; as a result most specifications are *incomplete* in the sense that they fail to capture precisely the functional semantics of routines. Weak specifications hinder formal verification in two ways. First, establishing weak postconditions is simple, but confidence in the full functional correctness of a verified routine will be low: the quality of specifications limits the value of verification. For example, the above postcondition of `put_right` permits an implementation that replaces all existing list elements with `v`. Second, weak contracts impede client reasoning: it is impossible to establish what a routine r achieves, if r calls another routine s whose contract is not strong enough to document its effect within r precisely. This is particularly problematic if s is part of a reusable component, since the set of its potential clients is *a priori* unknown.

Weak assertions limit the potential of many other applications of Design by Contract. Specifications, for example, should document the abstract semantics of operations in deferred classes. Weak contracts cannot fully do so; as a result, programmers have fewer safeguards to prevent inconsistencies in the design and fewer chances to make deferred classes useful to clients through polymorphism and dynamic dispatching.

Feature `put` in class `TABLE` (Fig. 3.1b) is an example of this phenomenon. It is unclear how to express the abstract semantics of `put` with standard contracts. In particular, the absence of a postcondition leaves it undefined what should happen when an element is inserted with a key that is already in the table: should `put` replace the previous element with the new one or cancel the insertion? Indeed, some descendants of `TABLE` implement `put` with a replacement semantics (such as class `ARRAY`), while others disallow `put` to override preexisting mappings (such as class `HASH_TABLE`). Some classes (including `HASH_TABLE`) even introduce another feature `force` that implements the replacement semantics. This obscures the behavior of routines to clients and makes it questionable whether `put` has been introduced at the right point in the inheritance hierarchy.

3.2.2 Enhancing Design by Contract with models

To address the aforementioned problems we propose *model-based contracts*: a specification discipline enabled by the use of *model classes* [69, 111] (immutable classes representing mathematical concepts). Model-based contracts are a conservative extension of Design by Contract, aiming at providing strong specifications without the need to extend the notation, which remains the one familiar to programmers.

Fig. 3.2 and 3.3 show extensions of the examples in Fig. 3.1 with model-based contracts. `LINKED_LIST` (Fig. 3.2) is augmented with a query `sequence` that returns an instance of class `MML_SEQUENCE`, a model class representing a mathematical sequence of elements of homogeneous type. The query can be implemented either as an attribute, or as a function that builds `sequence` according to the actual content of the list; this choice is discussed in some more

```
class LINKED_LIST [G]
model sequence, index
  put_right (v: G)
    -- Add 'v' to the right of cursor
    require 0 ≤ index ≤ sequence.count
    modify [sequence] Current
    do ...
    ensure
      sequence = old (sequence.front (index)
       .extended (v) +
       sequence.tail (index + 1))
    end

  duplicate (n: INTEGER): LINKED_LIST [G]
    -- A copy of at most 'n' elements
    -- starting at cursor position
    require n ≥ 0
    do ...
    ensure
      Result.sequence = sequence
        .interval (index, index + n − 1)
      Result.index = 0
    end
```

```
  count: INTEGER
    -- Number of elements

  index: INTEGER
    -- Current cursor position

  item: G
    -- Value at cursor position
    require
      sequence.domain [index]
    ensure
      Result = sequence [index]
    end

  sequence: MML_SEQUENCE [G]
    -- Sequence of elements
     ...

invariant
  0 ≤ index ≤ sequence.count + 1
  count = sequence.count
end
```

Figure 3.2: A snippet from class `LINKED_LIST` with model-based contracts.

detail below. The **model** clause[2] declares the two features `sequence` and `index` as the model of the class; every contract will rely on the abstraction they provide. In particular, the postcondition of `put_right` can precisely describe the effect of the routine: the new `sequence` is the concatenation of the **old** `sequence` up to `index`, extended with element `v`, with the tail of the **old** `sequence` starting after `index`. The keyword **modify** introduces a routine's frame specification: a list of all model queries whose value is allowed to change after executing the routine. For example, routine `put_right` in Fig. 3.2 may only change the value of **Current**`.sequence`, but not **Current**`.index` and not any model query of an object different from **Current**[3] We can assert that the new postcondition—including

[2]Here and in the rest of the thesis, newly introduced language constructs are underlined. In practice we express these elements using Eiffel's **note** meta-annotation to avoid changing the syntax.

[3]In fact, `put_right` can also modify the abstract state of objects that constitute the internal representation of **Current**, as well as any newly allocated objects, but those changes do not concern the clients of `put_right`. The semantics of **modify** clauses used in this

```
deferred class TABLE [G, K]
model map
  map: MML_MAP [G, K]
    -- Map of keys to values
    ...
```

```
  put (v: G ; k: K)
    -- Associate value 'v' with key 'k'
    require map.domain [k]
    modify [map] Current
    deferred
    ensure map = old map.replaced_at (k, v)
    end
end
```

Figure 3.3: A snippet from class TABLE with model-based contracts.

the frame specification—is *complete* with respect to the model of the class, because it defines the effect of put_right on the model fully and deterministically. At the same time, the specification of LINKED_LIST is *abstract*, since the model does not reveal any representation details, beyond the information available through the public interface. These complementary notions of completeness and abstraction are a powerful guide to writing accurate specifications that makes for well-defined interfaces and verifiable classes.

A mathematical map—represented by the model class MML_MAP—is the natural model for the class TABLE (Fig. 3.3). Even though TABLE is deferred and does define a concrete data representation, the availability of a model supports complex specifications already at this abstract level. In particular, writing a complete postcondition for routine put requires committing to a specific semantics for insertion. The example in Fig. 3.3 chooses the replacement semantics; correspondingly, all descendants of TABLE will have to conform to this semantics, guaranteeing a coherent reuse of TABLE throughout the class hierarchy.

3.3 Foundations of Model-Based Contracts

This section formalizes the notions of "complete" and "abstract" model-based specifications using the mathematical language of *abstract data types* (ADT). Sect. 3.3.1 briefly reviews traditional algebraic ADT specifications and the notion of *sufficient completeness* of such specifications, as proposed in [48]; Sect. 3.3.2 introduces the concepts of abstract equality and abstract state space of an ADT; finally Sect. 3.3.3 shows the benefits of model-based ADT specifications over arbitrary algebraic specifications and presents sufficient

chapter is a simplification; formal details are presented in Chapters 5 and 6.

conditions to achieve consistency, sufficient completeness and abstraction.

3.3.1 Abstract Data Types

Following the classical work of Guttag and Horning [48], an abstract data type is a *type algebra* $\mathbf{T} = \langle \mathcal{I} + \{T\}, F \rangle$, where $\mathcal{I}$ is the set of *supporting phyla*, $T \notin \mathcal{I}$ is a distinguished phylum called the *type of interest* (TOI), and F is a finite set of finitary partial operations $f \colon V_1 \times \ldots \times V_n \nrightarrow V_0$, where each $V_i \in \mathcal{I} + \{T\}$ and at least one $V_i = T$.

A type algebra is a special case of a heterogeneous algebra; unlike a generic heterogeneous algebra, which defines multiple phyla by mutual recursion, a type algebra defines a single phylum—the type of interest—in terms of the supporting phyla. As a consequence, every $V \in \mathcal{I}$ is assumed to have a fixed structure, including an equality relation $=_V$ (we omit the subscript when the type of the operands is clear). The equality relation $=_T$ of the type of interest, on the other hand, is not predefined, but rather axiomatized, just like the operations F (the semantics of $=_T$ is discussed in more detail below).

The set of operations F can be split into *creators* (N), *queries* (Q) and *commands* (C), depending on whether T occurs in their signature only on the right, only on the left, or on both sides, respectively. The sets $\mathcal{I}$ and the operations F together define a set of *terms* $L_{\mathcal{I},F}$, which contains all "strings" $f(x_1, \ldots, x_n)$, where $f \in F$, each $x_i \in V \in \mathcal{I}$ or $x_i \in L_{\mathcal{I},F}$, and $\langle x_1, \ldots, x_n \rangle \in \text{dom}(F)$. The set of terms can be naturally extended to a *term algebra* $\mathbf{L}_{\mathcal{I},F} = \langle \mathcal{I} + \{L_{\mathcal{I},F}\}, F \rangle$.

The meaning of operations in F is given by a set of axioms $\mathcal{A}$, each of which is of the form $\forall x_1, \ldots, x_n \bullet lhs = rhs$. Here $x_1, \ldots, x_n$ are variables; lhs is a well-typed term over those variables, built using only operations in F with the nesting level of one or two; rhs is a well-typed term (of the same types as lhs) over $x_1, \ldots, x_n$, built using operations in F or predefined operations of $\mathcal{I}$, including the built-in operation `if then else` of the phylum $\mathbb{B}$ (Booleans). The meaning of $=$ in each axiom depends on the type of lhs. Since all free variables of lhs and rhs are always universally qualified, the quantifier can be omitted.

Example 3.1. We define an ADT `QUEUE` for queues of integers. Its supporting phyla are Booleans and integers ($\mathcal{I} = \{\mathbb{B}, \mathbb{Z}\}$), and its operations consist of a creator `new`, two queries: `is_empty` and `item`, and two commands: `put` and `remove` with the following signatures[4]:

$$\texttt{new} \colon \ \to \texttt{QUEUE}$$

[4] Here we use the same symbol `QUEUE` to denote both the ADT and its type of interest.

$$
\begin{aligned}
\texttt{is\_empty}\colon \texttt{QUEUE} &\to \mathbb{B}\\
\texttt{item}\colon \texttt{QUEUE} &\nrightarrow \mathbb{Z}\\
\texttt{put}\colon \texttt{QUEUE} \times \mathbb{Z} &\to \texttt{QUEUE}\\
\texttt{remove}\colon \texttt{QUEUE} &\nrightarrow \texttt{QUEUE}
\end{aligned}
$$

The domains of partial operations `item` and `remove` can be specified using precondition predicates: $P_{\texttt{item}}(q) \equiv P_{\texttt{remove}}(q) \equiv \neg\texttt{is\_empty}(q)$. The following axioms $\mathcal{A}$ give the meaning of queue operations:

$$
\begin{aligned}
\texttt{is\_empty}(\texttt{new}) &= \top\\
\texttt{is\_empty}(\texttt{put}(q, v)) &= \bot\\
\texttt{item}(\texttt{put}(q, v)) &= \textbf{if } \texttt{is\_empty}(q) \textbf{ then } v \textbf{ else } \texttt{item}(q)\\
\texttt{remove}(\texttt{put}(q, v)) &= \textbf{if } \texttt{is\_empty}(q) \textbf{ then } q \textbf{ else } \texttt{put}(\texttt{remove}(q), v)
\end{aligned}
$$

Definition 3.1 ([48]). The set of axioms $\mathcal{A}$ is a *sufficiently complete* axiomatization of $\mathbf{T} = \langle \mathcal{I} + \{T\}, N + Q + C \rangle$, if for every term $f(t_1, \dots, t_n) \in L_{\mathcal{I},F}$, such that $(f\colon V_1 \times \ldots \times V_n \nrightarrow V_0) \in Q$, there exists a value $u \in V_0$ such that

$$\mathcal{A} \quad \models \quad f(t_1, \dots, t_n) = u$$

$\mathcal{A}$ is *consistent* if such a u is unique.

The problem of establishing whether a given set of axioms is sufficiently complete is, in general, undecidable. Moreover, experience shows that constructing sufficiently complete axiomatizations is often nontrivial in practice; [48] gives a set of guidelines, which significantly restrict the shape of potential axioms, however some of those axioms are still rather non-obvious. Intuitively, the goal of an ADT specification is to express the effect of each command on the value of each query; however, axioms constructed according to the guidelines of [48] often involve multiple commands, and sometimes in a such a way that one side of the identity does not seem simpler than the other. An example is the last axiom of `QUEUE`, where the term $\texttt{remove}(\texttt{put}(q, v))$ is related to $\texttt{put}(\texttt{remove}(q), v)$, which does not immediately seem useful.

In the *model-based* approach, detailed below, axioms are more structured: the meaning of each ADT operation is expressed in terms of a single query m (called the *model query*), which makes the specification of the operations more independent, and thus more intuitive and easier to write.

3.3.2 Abstract equality and abstract state space

Associating the type of interest of an ADT with a concrete set would defeat its purpose of being abstract; instead T is usually defined as $T = L_{\mathcal{I},F} / =_T$:

a quotient set of $L_{\mathcal{I},F}$ by some congruence relation. In general, there can be multiple congruence relations compatible with a given axiomatization, and thus multiple ways to define the type of interest. Some early work on ADTs [47, 138] makes the assumption that the values of T must be considered different unless provably equal, thus choosing $=_T$ to be the *smallest* congruence compatible with the axioms. Guttag and Horning [48] take the opposite view: they note that the significance of values in T rests solely upon the difference they make—in terms of the predefined $=_V$ for $V \in \mathcal{I}$—for the results of queries Q; thus it makes sense to regard elements of T as equal unless demonstrably distinct.

Example 3.2. Consider an ADT SET for sets of integers, with three operations: new: $\to$ SET, has: SET $\times \mathbb{Z} \to \mathbb{B}$, and put: SET $\times \mathbb{Z} \to$ SET. A sufficiently complete (and intuitively pleasing) axiomatization of SET is given by the following two identities:

$$\begin{aligned} \texttt{has}(\texttt{new}, x) &= \bot \\ \texttt{has}(\texttt{put}(s, y), x) &= \textbf{if } x = y \textbf{ then } \top \textbf{ else } \texttt{has}(s, x) \end{aligned}$$

The axioms make no mention of $=_{\texttt{SET}}$, thus the smallest possible congruence on SET is the (term) identity. With this approach the terms $\texttt{put}(\texttt{put}(\texttt{new}, 0), 1)$ and $\texttt{put}(\texttt{put}(\texttt{new}, 1), 0)$ are considered different, since there is no way to prove otherwise using the axioms. Taking the view of [48], however, those two terms—and in general, any two sets that agree on the value of has for all integers—are considered equal, which is consistent with our intuitive understanding of sets.

In the present work, we formalize this view of [48] through the following *distinguishability* relation[5].

Definition 3.2. Two terms $x, y \in L_{\mathcal{I},F}$ are *distinguishable* by the set of axioms $\mathcal{A}$ (written $x \not\approx_{\mathcal{A}} y$), if there exists a nonempty sequence of appropriately typed operations $f_1, \ldots, f_n \in F$ with $(f_1 \colon T \nrightarrow V) \in Q$, and two values $u, v \in V$ with $u \neq v$, such that

$$\begin{aligned} \mathcal{A} &\models (f_1 \circ \ldots \circ f_n)(x) = u \land \\ \mathcal{A} &\models (f_1 \circ \ldots \circ f_n)(y) = v \end{aligned}$$

Otherwise x and y are *indistinguishable* by $\mathcal{A}$: $x \approx_{\mathcal{A}} y$.

[5]For clarity of presentation, hereafter we restrict all ADT operations to take exactly one argument; an extension to multiple arguments is straightforward.

We will omit the subscript $\mathcal{A}$ whenever the set of axioms is unambiguous. Note that in an ADT with no queries ($Q = \emptyset$), all terms are indistinguishable. Another important observation concerns sufficiently complete axiomatizations:

Proposition 3.1. If $\mathcal{A}$ is consistent and sufficiently complete by Definition 3.1, then $x \approx_{\mathcal{A}} y$ is equivalent to $\forall f_1, \ldots, f_n \in F$ with $f_1 \in Q$:

$$\mathcal{A} \quad \models \quad (f_1 \circ \ldots \circ f_n)(x) = (f_1 \circ \ldots \circ f_n)(y).$$

Example 3.3. Consider two terms from $L_{\texttt{QUEUE}}$: $x \equiv \texttt{put}(\texttt{put}(\texttt{new}, 0), 0)$ and $y \equiv \texttt{put}(\texttt{put}(\texttt{new}, 0), 1)$. Given the axiomatization of Example 3.1, $x \not\approx y$, since we can prove that $\texttt{item}(\texttt{remove}(x)) = 0$ and $\texttt{item}(\texttt{remove}(y)) = 1$.

Now imagine the `QUEUE` ADT without the `remove` operation (and the corresponding axiom). In this new type $x \approx y$, since there is no way to observe any of the queue elements beyond the first one. In this example the distinguishability relation contradicts our intuition about the ADT, providing a clue that some operations might be missing, even though the axiomatization of the existing operations is sufficiently complete.

Lemma 3.1. If an axiomatization $\mathcal{A}$ of an ADT $\mathbf{T} = \langle \mathcal{I} + \{T\}, F \rangle$ is consistent and sufficiently complete, then $\approx_{\mathcal{A}}$ is a congruence relation on $\mathbf{T}$, compatible with $\mathcal{A}$.

Proof. Since $\mathcal{A}$ is consistent and sufficiently complete, for $\approx_{\mathcal{A}}$ we can use the formula from Proposition 3.1. This relation is reflexive, since all $f \in F$ are functions over terms; it is also symmetric and transitive, since $=_V$ for all $V \in \mathcal{I}$ is symmetric and transitive. Hence $\approx_{\mathcal{A}}$ is an equivalence relation.

This relation is compatible with queries Q and commands C. For arbitrary terms x and y such that $x \approx_{\mathcal{A}} y$, and an arbitrary query $q \in Q$, we have $\forall f_1, \ldots, f_n \in F$ with $f_1 \in Q$, $\mathcal{A} \models (f_1 \circ \ldots \circ f_n)(x) = (f_1 \circ \ldots \circ f_n)(y)$; instantiating $n = 1$, $f_1 = q$, we get $q(x) = q(y)$. Similarly, for an arbitrary command $c \in C$, instantiating f_n with c we get $\mathcal{A} \models (f_1 \circ \ldots \circ f_{n-1})(c(x)) = (f_1 \circ \ldots \circ f_{n-1})(c(y))$ for any $f_1, \ldots, f_{n-1} \in F$, thus $c(x) \approx_{\mathcal{A}} c(y)$. Hence $\approx_{\mathcal{A}}$ is a congruence.

Finally, if $\mathcal{A} \models x =_T y$ and $\mathcal{A} \models (f_1 \circ \ldots \circ f_n)(x) = u$, then by equational reasoning we get $\mathcal{A} \models (f_1 \circ \ldots \circ f_n)(y) = u$, which means $x \approx_{\mathcal{A}} y$. Thus $\approx_{\mathcal{A}}$ is compatible with $\mathcal{A}$. $\square$

Indistinguishability has another useful property: it is bigger than all congruences compatible with the structure of the supporting phyla. Consider a congruence $\sim$, such that $(\sim) \not\subseteq (\approx)$; then there exist $x, y \in L_{\mathcal{I},F}$ such that $x \sim y$ but $x \not\approx y$. Hence $\exists f_1, \ldots, f_n \in F$ with $(f_1 \colon T \nrightarrow V) \in Q$ such that

$\mathcal{A} \models (f_1 \circ \ldots \circ f_n)(x) \neq_V (f_1 \circ \ldots \circ f_n)(y)$, even though $x \sim y$; in other words, $\sim$ is not compatible with $=_V$ and thus is not a congruence on $\mathbf{T}$.

From the above discussion and Lemma 3.1 it follows, that given a consistent and sufficiently complete axiomatization, indistinguishability is the biggest ("most abstract") congruence relation on $\mathbf{T}$. For that reason we also refer to $\approx$ as *abstract equality*, and to the quotient set $L_{\mathcal{I},F}/\approx$ as the *abstract state space* of an ADT. Given the goal to achieve maximum abstraction, it makes sense to define $=_T$ as $\approx$, and the type of interest T as the abstract state space. This turns $\mathbf{T}$ into a quotient algebra of the term algebra $\mathbf{L}_{\mathcal{I},F}$ by $\approx$: $\mathbf{T} = \mathbf{L}_{\mathcal{I},F}/\approx$.

For incomplete axiomatizations $\mathcal{A}$, $\approx_{\mathcal{A}}$ is not necessarily transitive, and thus not an equivalence relation. For example, without the last axiom of the QUEUE ADT in Example 3.1, one cannot distinguish $\texttt{remove}(\texttt{put}(\texttt{new}, 0))$ from *any* other term, including, for instance $\texttt{put}(\texttt{new}, 0)$ and $\texttt{put}(\texttt{new}, 1)$, which are distinguishable from each other.

3.3.3 Model-Based Specifications

The goal of model-based approaches to specification [58, 49, 98, 134] is to simplify reasoning about an ADT by mapping it to a concrete set, called its *model*, and expressing the semantics of operations in a more structured manner, in terms of the model.

Definition 3.3. Given an ADT $\mathbf{T} = \langle \mathcal{I} + \{T\}, N + Q + C \rangle$, let us augment the set of supporting phyla $\mathcal{I}$ with the *model phylum* M, and the set of queries Q with the *model query* $m \colon T \to M$. The set of axioms $\mathcal{A}_M$ is called a *functional model-based specification* of $\mathbf{T}$ if it has the following form (where each ϕ_i is a composition of operations defined in $\mathcal{I} + \{M\}$):

- for each $q \in Q$, $\mathcal{A}_M$ contains an axiom $\forall x \bullet q(x) = \phi_q(m(x))$;
- for each $c \in C$, $\mathcal{A}_M$ contains an axiom $\forall x \bullet m(c(x)) = \phi_c(m(x))$;
- for each $n \in N$, $\mathcal{A}_M$ contains an axiom $\forall i \bullet m(n(i)) = \phi_n(i)$;
- those are the only axioms in $\mathcal{A}_M$.

It is easy to show that a functional model-based specification is always consistent and sufficiently complete: by induction on the term structure one can prove that any term $f(t_1, \ldots, t_n)$ where $t_i \in L_{\mathcal{I},F}$ and $f \in Q$ is reducible to a term containing only ϕ_q, ϕ_c and ϕ_n.

We say that a model-based specification $\mathcal{A}_M$ is *compatible* with the set of axioms $\mathcal{A}$ of a general form, if they agree on all terms $f(t) \in L_{\mathcal{I},F}$ with $(f \colon T \nrightarrow V) \in Q$, that is, if $\exists u \in V \bullet \mathcal{A} \models f(t) = u$, then $\mathcal{A}_M \models f(t) = u$.

Choosing a Model

Finding a good model for a component is the central problem of all model-based specification methodologies. In our case, given an ADT $\mathbf{T}$ with a consistent and sufficiently complete axiomatization $\mathcal{A}$, how does one pick the phylum M and construct $\mathcal{A}_M$ compatible with the initial axioms? Informally to make a functional model-based specification possible, M must have "enough" values to differentiate between the distinguishable elements of T; on the other hand, it should not contain "too many" values, since that would defy the pursuit of abstraction. Fortunately, the notion of abstract state space, introduced above, provides exactly the right level of detail. Thus, M is a good model if there exists a set of operations Φ, such that the "model algebra" $\langle \mathcal{I} + \{M\}, \Phi \rangle$ is *isomorphic* to the quotient algebra $\mathbf{L}_{\mathcal{I},F}/\approx$; the operations Φ can then be used in the specification $\mathcal{A}_M$ as ϕ_q, ϕ_c and ϕ_n. This condition on M is captured more accurately by the following definition.

Definition 3.4. Given an ADT $\mathbf{T} = \langle \mathcal{I} + \{T\}, F \rangle$ with a consistent and sufficiently complete axiomatization $\mathcal{A}$, a surjective function $\mu \colon L_{\mathcal{I},F} \to M$ is a *model function* for $\langle \mathbf{T}, \mathcal{A} \rangle$ if

$$\forall x, y \in L_{\mathcal{I},F} \bullet x \approx_{\mathcal{A}} y \;\Leftrightarrow\; \mu(x) = \mu(y).$$

We can show that whenever such μ exists, it is possible to define a set of operations Φ and produce a functional model-based specification $\mathcal{A}_M$, in the sense that those operations are indeed functional (univalent). The latter can be expressed as $\forall q \in Q, x, y \in L_{\mathcal{I},F} \bullet \mu(x) = \mu(y) \Rightarrow q(x) = q(y)$ and $\forall c \in C, x, y \in L_{\mathcal{I},F} \bullet \mu(x) = \mu(y) \Rightarrow \mu(c(x)) = \mu(c(y))$, and follows immediately from $\approx_{\mathcal{A}}$ being a congruence. Of course, the fact that operations in Φ are functional does not formally guarantee that they can be expressed as a composition of the predefined operations of M and $\mathcal{I}$, but since the premise of model-based specifications is the existence of sufficiently expressive model types, these operations are likely available.

Given a functional model-based specification $\mathcal{A}_M$ based on operations Φ, and an interpretation $\mu \colon L_{\mathcal{I},F} \to M$ of the model query m, it follows directly from the shape of the axioms that μ is a homomorphism from $\mathbf{L}_{\mathcal{I},F}$ to the model algebra $\langle \mathcal{I} + \{M\}, \Phi \rangle$. Thus, by the first isomorphism theorem [18], the image of μ ($\mathrm{im}(\mu)$) is isomorphic to the quotient algebra $\mathbf{L}_{\mathcal{I},F}/\sim_\mu$ (where $x \sim_\mu y$ means $\mu(x) = \mu(y)$). If in addition μ is a model function, that is, $(\sim_\mu) = (\approx)$ and $\mathrm{im}(\mu) = M$, then the model algebra $\langle \mathcal{I} + \{M\}, \Phi \rangle$ is isomorphic to $\mathbf{L}_{\mathcal{I},F}/\approx$, which corresponds to our intuitive definition of a "good" model. In conclusion, choosing a good model for a given $\langle \mathbf{T}, \mathcal{A} \rangle$ boils down to finding a model function.

In Definition 3.4, the $\Leftarrow$ direction (terms with the same model are indistinguishable) intuitively means that the model is “expressive enough” or “not too coarse”. Failure to comply with this constraint results in the inability to express the operations ϕ_f for some $f \in Q + C$, and thus to complete $\mathcal{A}_M$. The $\Rightarrow$ direction (terms with different models are distinguishable) guarantees that the model is “abstract enough” or “not too fine-grained”. This property is also known as *observability* [133]; failure to establish it leads to models with too much detail. Finally, the requirement that the model function be surjective entails that the model phylum M does not contain any “garbage” values, that do not correspond to abstract values of the ADT; this property is also known as *controllability* [133].

Since it is unreasonable to expect that the predefined collection of model types contains precisely the image of every useful model function μ, in practice it is convenient to represent the model phylum M as $\{z \in Z \mid I_M(z)\}$, where Z is a predefined type and I_M is an *invariant predicate*, which restricts the possible values of the model to $\mathrm{im}(\mu)$.

Example 3.4. Recall the `QUEUE` ADT defined in Example 3.1. As the model phylum for queues we choose integer sequences, $\mathbb{Z}^*$, with a standard set of operations, including index-based access ($s[i]$), slicing ($s[i..j]$), construction of empty ($[]$) and singleton ($[v]$) sequences, and concatenation ($s_1 + s_2$). We can argue that $\mathbb{Z}^*$ is a suitable model by providing an informal definition of the model function $\mu(q)$ as “the sequence of elements stored in q in the order of their insertion”. According to our intuition, the operations of `QUEUE` cannot distinguish two queues with the same sequence of elements, and conversely, two queues with different element sequences can be distinguished; in addition, for any sequence of values, we can build a queue storing those values; this makes μ a valid model function.

After ensuring that the model is chosen properly, constructing a functional model-based specification for `QUEUE` is straightforward:

$$\begin{aligned} m(\texttt{new}) &= [] \\ \texttt{is\_empty}(q) &= (m(q) = []) \\ \texttt{item}(q) &= m(q)[1] \\ m(\texttt{put}(q, v)) &= m(q) + [v] \\ m(\texttt{remove}(q)) &= m(q)[2..] \end{aligned}$$

This specification is guaranteed to be consistent and sufficiently complete *independently of our choice of the model.* In addition, if our reasoning about μ was sound, this specification is also abstract and consistent with the intended semantics.

Now consider `QUEUE` without `remove` (see Example 3.3). The model $\mathbb{Z}^*$ is too fine-grained for this ADT, since any two queues with the same first element cannot be distinguished. One appropriate model for such a queue is a set, which is empty for an empty queue, and contains its first element otherwise; this model can be specified through an invariant predicate $I_M \equiv |m(q)| \leq 1$. At the same time, for the original `QUEUE` ADT, this new model is obviously too coarse, and as a result the operation $\phi_{\texttt{remove}}$ cannot be defined.

Assessing Quality

After the specification $\mathcal{A}_M$ has been constructed, one can emancipate it from the original axioms $\mathcal{A}$ and define the following *internal quality criteria*:

Definition 3.5. A model-based specification $\mathcal{A}_M$ of an ADT $\mathbf{T} = \langle \mathcal{I} + \{T\}, N + Q + C \rangle$ with $M = \{z \in Z \mid I_M(z)\}$ and a model query m is:

1. *complete* if $\forall x, y \in L_{\mathcal{I},F} \bullet x \approx_{\mathcal{A}_M} y \;\Leftarrow\; m(x) = m(y)$;
2. *observable* if $\forall x, y \in L_{\mathcal{I},F} \bullet x \approx_{\mathcal{A}_M} y \;\Rightarrow\; m(x) = m(y)$;
3. *closed* if $\forall (n\colon V \nrightarrow T) \in N, i \in V \bullet P_n(i) \Rightarrow I_M(\phi_n(i))$ and $\forall c \in C, z \in Z \bullet P_c(z) \wedge I_M(z) \Rightarrow I_M(\phi_c(z))$, where P_f is the precondition of f;
4. *controllable* if $\forall z \in Z \bullet I_M(z) \Rightarrow \exists x \in L_{\mathcal{I},F} \bullet (\mathcal{A}_M \models m(x) = z)$.

Completeness always holds for functional model-based specifications. Verifying observability requires providing, for any two terms $x, y \in L_{\mathcal{I},F}$ such that $m(x) \neq m(y)$, a witness sequence of operations that distinguishes them. Closure and controllability together express that the invariant I_M describes precisely the range of m. Closure always holds for models with a trivial invariant. Checking controllability requires providing a witness term that has a given model.

The fact that $\mathcal{A}_M$ satisfies the criteria of Definition 3.5 does not tell us anything about its relation to the initial specification $\mathcal{A}$; on the other hand, it does guarantee that $\mathcal{A}_M$ is consistent and sufficiently complete, and that M is abstract and minimal. Such internal quality criteria are important in practice, since normally the specification process starts with $\mathcal{A}$ as the developer's *informal* understanding of the ADT semantics. Guided by this understanding, the developer chooses a model phylum M and a model function μ that maps intuitively distinguishable objects to different values, and indistinguishable objects to the same value. With the model at hand, the developer constructs a *formal* model-based specification $\mathcal{A}_M$, following a rather strict format of Definition 3.3. At this point, the quality of $\mathcal{A}_M$ can be checked formally, according to Definition 3.5. A failure to verify one of the criteria calls for strengthening the specification, changing the model, or even adding or removing some ADT operations.

Example 3.5. Let us evaluate the quality of the model-based specification of the `QUEUE` ADT from Example 3.4.

1. *Completeness* holds trivially since the specification is functional.
2. *Observability:* Given queues q_1, q_2 with $m(q_1) = s_1$, $m(q_2) = s_2$ and $s_1 \neq s_2$, either exactly one of the sequences is empty (hence, from the specification of `is_empty`, $\texttt{is\_empty}(q_1) \neq \texttt{is\_empty}(q_2)$) or both sequences are nonempty. In that case, either $s_1[1] \neq s_2[1]$ (and hence, from the specification of `item`, $\texttt{item}(q_1) \neq \texttt{item}(q_2)$), or $s_1[2..] \neq s_2[2..]$. In the latter case, we can apply the same reasoning recursively to the queues $\texttt{remove}(q_1)$, $\texttt{remove}(q_2)$ and sequences $s_1[2..]$, $s_2[2..]$
3. *Closure* holds trivially since $I_M \Leftrightarrow \top$.
4. *Controllability*: Given an empty sequence s, by the specification of `new` we have $m(\texttt{new}) = s$. Now suppose that for any sequence of length k we can construct the corresponding queue; given a sequence s of length $k + 1$, we use the induction hypothesis to produce a q such that $m(s) = s[..k]$, and then use the specification of `put` to verify that $m(\texttt{put}(q, s[k + 1])) = s$.

Relational Model-Based Specifications

When the model phylum M (together with the supporting phyla $\mathcal{I}$) does not provide enough operations to represent some $\phi \in \Phi$ as their composition, the solution is to axiomatize ϕ, which leads to specifications syntactically different from functional model-based specifications.

Definition 3.6. For an ADT $\mathbf{T} = \langle \mathcal{I} + \{T\}, N + Q + C \rangle$, augmented with M and $m \colon T \to M$, the set of axioms $\mathcal{A}_M$ is called a *relational model-based specification* if it has the following form (where each ρ_i is a composition of operations defined in $\mathcal{I} + \{M\}$):

- for each $q \in Q$, $\mathcal{A}_M$ contains an axiom $\forall x \bullet \rho_q(m(x), q(x))$;
- for each $c \in C$, $\mathcal{A}_M$ contains an axiom $\forall x \bullet \rho_c(m(x), m(c(x)))$;
- for each $n \in N$, $\mathcal{A}_M$ contains an axiom $\forall i \bullet \rho_n(i, m(n(i)))$;
- those are the only axioms in $\mathcal{A}_M$.

Although syntactically different, relational model-based specifications can be semantically equivalent to functional specifications discussed above, if every ρ_f for $f \in N + Q + C$ is functional (univalent, right-unique). It is easy to show that under this condition, a relational model-based specification is complete as per Definition 3.5.

Example 3.6. Suppose that our specification language does not offer sequence slicing, but instead provides a sequence length operation ($|s|$) and quantification over integer intervals ($\textbf{all}\ i \in n..m : p(i)$). We have to replace the specification of `remove` in the `QUEUE` ADT with a relational equivalent $\rho_{\texttt{remove}}(m(q), m(\texttt{remove}(q)))$, where

$$\rho_{\texttt{remove}}(s_1, s_2) \quad \equiv \quad |s_1| = |s_2| + 1 \ \wedge\ \textbf{all}\ i \in 1..|s_2| : s_2[i] = s_1[i+1].$$

We can show that $\rho_{\texttt{remove}}$ is univalent, since it defines uniquely the length of s_2 and fixes its elements at positions from 1 to $|s_2|$. Hence, this new, relational model-based specification of `QUEUE` still satisfies the internal quality criteria of Definition 3.5.

3.4 Model-Based Contracts in Practice

This section demonstrates how the specification process and quality criteria defined in Sect. 3.3 can be applied to imperative object-oriented programs.

3.4.1 From Abstract Data Types to Classes

The notion of *class* in object-oriented programming shares many similarities with abstract data types: the purpose of both is to define a type in terms of operations that can be performed on it, and both rely on existing types to do so. Similarly to the ADT operations, the features of a class can be separated into creators (which initialize new objects), commands (which transform the object state), and queries (which relate objects to values of other types). These similarities prompt the reuse of ADT specification techniques for describing the meaning of classes.

The established approach to reasoning about routines in imperative programs is based on pre- and postconditions [52]. Algebraic specifications of ADTs, as proposed in [48], cannot be directly rewritten in this form, since they often relate multiple commands to each other; for instance, it is unclear how to express the axiom $\texttt{remove}(\texttt{put}(q, v)) = \dots$ from Example 3.1 as a postcondition of either `remove` or `put`[6]. On the other hand, the restricted form of

[6]An entirely different view on algebraic specifications in an imperative context is out-

model-based specifications, where each operation is defined by an independent axiom in terms of the model query, makes them easily applicable to imperative programs, in particular, in the context of Design by Contract.

There are, however, some differences between ADTs and classes (as present in modern object-oriented languages), which affect the way clients reason about the behavior of a reusable component. Most importantly, since the ADT notation is stateless, the signature of an ADT operation defines precisely its input and output, as a tuple of abstract values. Classes have state, which is stored on the heap; an "abstract value" of an object is a function of the heap, and each feature includes the heap as a hidden input and output. The programming language usually imposes no restrictions on the set of heap locations a feature may modify, and as a result it may change the abstract value of virtually any object in the system. Since the goal of interface specifications is to simplify client reasoning, we would like to encapsulate the complexity associated with the heap and express the semantics of a feature as a function from abstract values to abstract values (just like an ADT operation). For reasons explained above, however, defining the output of a feature in terms of abstract values is challenging. Our technique for relating the heap to abstract values is detailed in Chapters 5 and 6, devoted to verification. What matters from the interface specification standpoint, is that in addition to pre- and postconditions, a feature has to be equipped with a *modify clause*, which essentially restricts the abstract values the feature "outputs".

Fig. 3.4 shows how one can turn a model-based specification of the `QUEUE` ADT, defined in Example 3.4, into model-based contracts for class `QUEUE`. The rest of the section explains various aspects of model-based contracts in more detail, using class `QUEUE`, as well as `LINKED_LIST` (Fig. 3.2) and `TABLE` (Fig. 3.3), as illustrative examples.

Model classes

In practice it is convenient to express the model of a class C using a tuple $M_C = \langle M_C^1, M_C^2, \ldots, M_C^n \rangle$ of model types. Model types include elementary sorts (such as Booleans, integers, and object references), functions over other model types (encoded in Eiffel as agents [95]), and *model classes*: immutable classes designed for specification purposes, which wrap rigorously defined mathematical structures, such as sets, bags, relations, maps, and sequences. The MML library presented in Chapter 2 provides a variety of such model classes, equipped with features that correspond to common operations on

lined in [132]: one can define an immutable type that corresponds directly to an ADT and then use functions of this type to specify the class features.

```
class QUEUE

create new
model sequence

  sequence: MML_SEQUENCE [INTEGER]
    -- Sequence of elements

  is_empty: BOOLEAN
    -- Is the queue empty?
    ensure Result =
      sequence.is_empty
    end

  item: INTEGER
    -- Head of the queue
    require not sequence.is_empty
    ensure Result = sequence [1]
    end

  new
    -- Create an empty queue
    modify [sequence] Current
    ensure sequence.is_empty
    end

  put (v: INTEGER)
    -- Add 'v' to the tail
    modify [sequence] Current
    ensure sequence =
      old sequence.extended (v)
    end

  remove
    -- Remove the head of the queue
    require not sequence.is_empty
    modify [sequence] Current
    ensure sequence = old sequence.but_first
    end
end
```

Figure 3.4: Class `QUEUE` annotated with model-based contracts, equivalent to the model-based specification of a `QUEUE` ADT in Example 3.4.

the mathematical structure they represent. For example, class `MML_SET` models finite sets; it includes features for operations such as membership test, union, intersection, difference, etc. `MML_SET` also supports a restricted form of quantification through Eiffel's *loop expressions*: one can write **all** x ∈ s : B(x) for universal and **some** x ∈ s : B(x) for existential quantification, where s is a set expression and B(x) is a Boolean expression over x.

In Fig. 3.4 the model of a queue is expressed using a single model class `MML_SEQUENCE` (for finite sequences). To represent the model of a linked list with internal cursor (see Fig. 3.2), we can combine `MML_SEQUENCE` with the elementary sort `INTEGER` (to encode the cursor position); this assumes that no information about the pointer structure of the list in the heap is accessible through the interface of the class.

Model Queries

Every class C defines a tuple of public *model queries* $\langle m_C^1, m_C^2, \ldots, m_C^n \rangle$, one for each component of the model M_C^i. Like in the ADT case, each model query has exactly one argument—the target object—and is total. The clause **model** $m_C^1, m_C^2, \ldots$ lists all model queries of the class (see Fig. 3.2 and 3.4 for examples).

It is likely that some model queries are equally suitable for use in the implementation; for example `LINKED_LIST.index` in Fig. 3.2. Other model queries, such as `LINKED_LIST.sequence`, are added solely for specification purposes and are not meant to be used in executable code.

Specification-only model queries can be implemented either as attributes or as functions, both with their own pros and cons. The former approach requires augmenting routine bodies with bookkeeping instructions that update model attributes. Attributes are also more difficult to "erase" from the system when contract checking is disabled: they always occupy memory at run time, unless the programming language provides a **ghost** construct that tells the compiler to omit given declarations and instructions during code generation. Implementing models as functions enforces a cleaner division between implementation and specification, but can incur performance overhead for runtime checking (when a model is calculated multiple times in the same concrete state), and impede verification in several ways (see Chapter 6). Note that the choice of implementation is immaterial from the interface specification standpoint, and will only become important in later chapters.

Routine contracts

Just like preconditions and axioms define the meaning of operations in an ADT, routine contracts (preconditions, postconditions, and frame specifications) express the semantics of class routines. The rest of the section contains guidelines for creating these specification elements, assuming all involved classes are equipped with model queries.

Similarly to the ADT case, the *precondition* of a feature is a constraint on the abstract values of its arguments. If an argument is of a model type, its abstract and concrete values coincide; otherwise the abstract value of an argument is given by its model (the tuple of values of its model queries). For example, the precondition **not** `sequence.is_empty` of feature `item` in Fig. 3.4 is a predicate over the model of the feature's sole argument (the target object).

Note that an argument x of a (non-model) reference type actually stands for two different abstract values: the model of the object that x is attached to, and the reference x itself. Depending on developer's intention, either one or even both may appear in contracts. Consider the precondition `map.domain [k]` of feature `put` in class `TABLE` (Fig. 3.3): it refers to the model of the target object (through the model query `map`), and to the actual reference `k`.

It is good practice to partition features into queries and commands;

queries are functions of the object state, whereas commands modify the object state but do not return any value. This *command-query separation* [84] is not enforced by object-oriented programming languages, most importantly, because completely preventing functions from modifying the heap is too restrictive. The desired property is known as *abstract purity* (also called *observational purity*): executing a query should leave unchanged the abstract state of all objects accessible to the client. This is a methodological principle, which can be expressed more precisely through specification guidelines. In particular, the model-based contracts methodology prescribes different postconditions and frame specifications for queries and commands.

The *postcondition of a command* defines a relation between the abstract pre-state and the abstract post-state of its arguments (i.e. the state before and after executing the command). For example the postcondition `sequence = old sequence.extended (v)` of `put` in Fig. 3.4 relates the new abstract state of the target object (`sequence`) to its old abstract state (`old sequence`) and the value of the reference `v`. This postcondition is a direct equivalent of the axiom $m(\texttt{put}(q, v)) = m(q) + [v]$ from Example 3.4.

Since the programming language allows a command r to modify the abstract state of any object in the system, completing the specification of r necessitates providing a *modify clause*. A clause of the form `modify` $[m_C^1, m_C^2, \ldots]$ o in the specification of a command r gives it permission to modify model queries $m_C^1, m_C^2, \ldots$ of the object o and nothing else (instead of a single object o, one can specify an arbitrary set of objects, encoded as an instance of `MML_SET`). For example, the modify clause `modify [sequence] Current` restricts the effect of `put` in Fig. 3.4 to the abstract state of the target object. In the corresponding ADT operation of Example 3.4, this property is encoded in the signature, `QUEUE` $\times$ `INTEGER` $\to$ `QUEUE`, which means that the operation only produces a queue and nothing else.

The *postcondition of a query* defines the abstract state of its result in terms of the abstract state of its arguments. Once again, if the result y of a query is of a reference type, its abstract value is a pair of a reference y and the model of the object that y is attached to. For example, compare the postcondition of query `item` from class `LINKED_LIST` (Fig. 3.2), which defines a reference to the returned list element, with the postcondition of query `duplicate` in the same class, which specifies the model of the returned list.

Models assign a precise meaning to the notion of “abstract value” of an object, whereby giving a formal interpretation to the concept of abstract purity. In the model-based contracts methodology, a feature is (abstractly) pure if it does not have a modify clause, which implies that it does not alter the model of any object accessible to the client; this is the recommended default for queries.

Apart form queries and commands, there is a third kind of features, which corresponds to creators in an ADT. *Creation procedures* are used to initialize newly created objects, and are designated in the class interface using a **create** clause (see Fig. 3.4 for an example). Contracts of creation procedures are similar to those of commands, except that they cannot mention the abstract pre-state of the target object.

Class Invariants

Model-based contracts prescribe three types of *class invariant* clauses.

Model constraints restrict the values of model queries to match precisely the abstract state space of the class; they correspond to the invariant predicate I_M defined for ADTs in Sect. 3.3.3. For example, model queries sequence and index of LINKED_LIST in Fig. 3.2 are constrained by an invariant clause $0 \leq \text{index} \leq \text{sequence.count} + 1$.

Attribute definitions play the same role for public attributes as model-based postconditions do for functions. Public attributes, from the class interface standpoint, are indistinguishable from public functions, and thus their values should be defined in terms of model queries. Syntactically, Eiffel allows attributes to have postconditions; they are, however, ignored during runtime checking and their semantics for verification is unclear, since the responsibility to maintain those properties lies on the routines of the class and not on the attributes themselves. Hence our methodology prescribes placing public attribute definitions in the class invariant. For example, in Fig. 3.2 the attribute count of LINKED_LIST is given a definition $\text{count} = \text{sequence.count}$.

Finally, *linking invariants* are related to inheritance and are detailed in the next section.

Inheritance and model-based contracts

A class C' that inherits from a parent class C may or may not re-use C's model queries to represent its own abstract state. It is often desirable to *replace* a parent's model query with another one, for example, because the richer interface of C' gives rise to an extended abstract state space. It is also possible to *abandon* a parent's model query entirely, when the child class imposes additional constraints, making some dimensions of the abstract space obsolete. Naturally, C' can also *add* new model queries, orthogonal to the parent's model.

For every model query m_C of the parent class that is not among the child's model queries, C' should provide a *linking invariant* to guarantee consistency in the inheritance hierarchy. The linking invariant is a formula that defines

```
deferred class COLLECTION [G]
model bag
  bag: MML_BAG [G]

  is_empty: BOOLEAN
    ensure Result = bag.is_empty end

  wipe_out
    modify [bag] Current
    ensure bag.is_empty end

  put (v: G)
    modify [bag] Current
    ensure bag = old bag.extended (v) end
end
```

```
deferred class DISPENSER [G]
inherit COLLECTION [G]
model sequence

  sequence: MML_SEQUENCE [G]

invariant
    -- Linking invariant:
  bag.domain = sequence.range
  all x ∈ bag.domain :
    bag [x] =
      sequence.occurrences (x)
end
```

Figure 3.5: Snippets of classes `COLLECTION` (left) and `DISPENSER` (right) with model-based contracts.

the value of m_C in terms of the model queries of the C'. This guarantees that the new model is indeed a specialization of the previous model, in accordance with the notion of sub-typing inheritance.

A properly defined linking invariant ensures that every inherited feature has a definite semantics in terms of the new model. However, the new semantics may be weaker in that a command whose contract in the parent class was deterministic becomes underspecified in the child class. In general, the quality criteria of Sect. 3.3.3, whose application to model-based contracts is discussed in more detail in the next section, might not be preserved in this case and have to be re-verified.

Consider class `COLLECTION` in Fig. 3.5, a generic container of elements whose model is a bag. Class `DISPENSER` inherits from `COLLECTION` and specializes it by introducing a notion of insertion order; correspondingly, it replaces the parent's model with a sequence. The linking invariant of `DISPENSER` defines the value of the inherited feature `bag` in terms of the new feature `sequence`: the domain of `bag` coincides with the range of `sequence`, and the number of occurrences of any element `x` in `bag` corresponds to the number of occurrences of the same element in `sequence`.

The linking invariant ensures that the semantics of features `is_empty` and `wipe_out` is unambiguously defined also in `DISPENSER`. On the other hand, the model-based contract of `put` in `COLLECTION` and the linking invariant are insufficient to characterize the effects of `put` in `DISPENSER`, as the position within the sequence where the new element is inserted is irrelevant for the bag.

3.4.2 Assessing Quality in Practice

The internal quality criteria for model-based ADT specifications formulated in Definition 3.5, carry over rather straightforwardly to model-based contracts, and can be encoded in a program verifier in order to enable formal checks. Such checks provide feedback on consistency and usefulness of interface specifications early in the development process (before constructing an implementation and attempting a correctness proof), and may point to poorly chosen conceptual models, missing contract elements and even missing features. For each of the four quality criteria, this section shows a possible encoding in a verifier, using the examples of `QUEUE` (Fig. 3.4) and `LINKED_LIST` (Fig. 3.2); it then discusses the implications of checking the criterion for the development process. Implementing the proposed encoding and evaluating the feasibility and usefulness of formal quality checks is part of future work.

The four criteria can be split into two groups according to the way they are checked. *Completeness* and *closure* are proved individually for each feature (the latter only for commands and creation procedures), which reduces to showing correctness of an automatically generated routine.[7] *Observability* and *controllability* are instead checked once per class; since the properties involve an existential quantifier over programs, these checks amount to providing a provably correct implementation for a given specification.

Completeness

Consider command `put_right` of class `LINKED_LIST`; we can show that its postcondition is complete by verifying the correctness of `put_right_complete` procedure in Fig. 3.6. Indeed, if the code is correct, executing `put_right` on two arbitrary lists with the same model produces lists whose model is again the same. Since modular verification only relies on the contract of `put_right` to complete the proof, we have shown that the postcondition of `put_right` is univalent—that is, defines the new model of the list as a mathematical function of its old model and the argument `v`—which corresponds to the definition of completeness given above. Similarly, for queries, we have to show that their result is defined as a mathematical function of the arguments; an example for query `item` is shown in Fig. 3.6.

[7]In auto-active verifiers [76], this is an established way of checking properties that do not directly correspond to user-written specifications elements, such as well-formedness of expressions or admissibility of class invariants.

```
put_right_complete
  (l1, l2: LINKED_LIST [G]; v: G)
  require
    l1.sequence = l2.sequence
    l1.index = l2.index
    0 ≤ l1.index ≤ l1.sequence.count
  do
    l1.put_right (v)
    l2.put_right (v)
  ensure
    l1.sequence = l2.sequence
    l1.index = l2.index
  end
```

```
item_complete (l1, l2: LINKED_LIST [G]):
  (v1, v2: G)
  require
    l1.sequence = l2.sequence
    l1.index = l2.index
    l1.sequence.domain [l1.index]
  do
    v1 := l1.item
    v2 := l2.item
  ensure
    v1 = v2
  end
```

Figure 3.6: Encoding completeness checks in a verifier: correctness of procedures `put_right_complete` and `item_complete` implies that postconditions of `put_right` and `item` in Fig. 3.2, respectively, are complete.

Query postconditions in the form `Result = exp (m, a)` or `Result.m = exp (m, a)` and command postconditions in the form `m = exp (old m, a)`—where `exp` is a side-effect free expression, `m` denotes a generic model query, and `a` is an argument—correspond to functional model-based specifications for ADTs and are trivially complete. Even when postconditions are written in a different form, verifying completeness only relies on the background theories that encode model types in the verifier; thus, given rich enough theories, it should be possible in most cases to verify completeness fully automatically.

How useful is completeness in practice? As a norm, it is a valuable yardstick to evaluate whether the contracts are sufficiently detailed for any inference a client might want to make, given the chosen mathematical model. Incompleteness calls for strengthening the postcondition, and the impossibility of systematically writing complete contracts is a strong indication that the model is chosen poorly.

While complete postconditions should be the norm, there are recurring cases where incomplete postconditions are unavoidable or even preferable. Three major sources of benign incompleteness are the following.

- Inherently *nondeterministic* abstractions, including those that accept input from the outside world; for example, a random number generator or a network socket.

- *Partial abstractions*, resulting from the use of *inheritance* to factor out

```
put_right_closed (s1, s2: MML_SEQUENCE [G]; i1, i2: INTEGER; v: G)
  require
    0 ≤ i1 ≤ s1.count + 1 -- Invariant (of 'LINKED_LIST')
    0 ≤ i1 ≤ s1.count -- Precondition (of 'put_right')
    s2 = s1.front (i1).extended (v) + s1.tail (i1 + 1)) -- Postcondition
    i2 = i1 -- Modify clause
  do
  ensure
    0 ≤ i2 ≤ s2.count + 1 -- Invariant
  end
```

Figure 3.7: Encoding a closure check in a verifier: correctness of procedure `put_right_closed` implies that the invariant of `LINKED_LIST` in Fig. 3.2 is closed under the postcondition of `put_right`.

common parts of (complete) specifications. For example, class `DISPENSER` in Fig. 3.5 is a common ancestor of `STACK` and `QUEUE`. If its interface includes features `item`, `put` and `remove`, its model must be isomorphic to a sequence. Then, it becomes impossible to write a complete postcondition for `put` in `DISPENSER`: the specification of `put` cannot define precisely where an element is added to the sequence; a choice compatible with the semantics of `STACK` will be incompatible with `QUEUE` and vice versa.

- Imperfections in *information hiding*. For example, class `ARRAYED_LIST` is an array-based implementation of lists which exports a query `capacity` returning the size of the underlying array; this piece of information is then part of the model of the class. Default constructors set `capacity` to an initial fixed value. Their postconditions, however, do not mention this default value, hence they are incomplete. The rationale behind not revealing this information is that clients should not rely on the exact size of the array when they invoke the constructor.

In all these cases, reasoning about completeness is still likely to improve the understanding of the classes and to question constructively the choices made for interfaces and inheritance hierarchies. A benign incompleteness can be indicated through user annotations in order to suppress the verifier warning.

Closure

For the same command `put_right` of `LINKED_LIST` checking closure amount to proving procedure `put_right_closed` in Fig. 3.7. The procedure takes as arguments two copies of a list model, $\langle \mathtt{s1}, \mathtt{i1} \rangle$ and $\langle \mathtt{s2}, \mathtt{i2} \rangle$; it assumes that the first copy satisfies the precondition of `put_right` and the model constraint I_M

```
distinguish (q1, q2: QUEUE)
  require
    q1.sequence ≠ q2.sequence
  modify [sequence] q1, q2
  do
    ...
  ensure
    q1.is_empty ≠ q2.is_empty or else
      (not q1.is_empty and
        q1.item ≠ q2.item)
  end
```

```
...
do
  if not q1.is_empty and not q2.is_empty
      and q1.item = q2.item then
    q1.remove
    q2.remove
    distinguish (q1, q2)
  end
ensure
...
```

Figure 3.8: Encoding an observability check in a verifier: existence of a provably correct implementation of procedure `distinguish` implies that the specification of class `QUEUE` in Fig. 3.4 is observable; one example of such an implementation is shown on the right.

extracted from the invariant of `LINKED_LIST`, and moreover, it is related to the second copy through the postcondition of `put_right`. We aim to prove that the second copy $\langle$`s2`, `i2`$\rangle$ also satisfies I_M. Similarly to completeness, showing closure in most cases need not require user interaction.

For routines with incomplete postconditions, a failing closure check is of limited value. If, however, the postcondition has been proven complete, failure to verify closure most likely indicates a missing precondition clause, or an incorrect (or missing) invariant clause.

Observability

Let us check observability of the `QUEUE` interface specification, given in Fig. 3.4. We can reason that this check amounts to constructing a provably correct implementation of routine `distinguish` (shown in Fig. 3.8 on the left) that always executes the same sequence of commands on its two arguments. Recall that the model of a queue is given by `sequence`, and its only two (non-model) queries are `is_empty` and `item`. Given two arbitrary queues with different models, the body of `distinguish` must drive them into a state where at least one of the queries q is applicable to both queues and returns a different result. Thus, for each concrete input, the executed sequence of commands, together with the query q, is precisely the sequence of operations required for observability in Definition 3.5. In general, the implementation of `distinguish` must be provided by the developer, but it might be possible to find heuristics

```
build (s: MML_SEQUENCE [INTEGER]): QUEUE
  require
    True -- Invariant of QUEUE
  do
    ...
  ensure
    Result.sequence = s
  end
```

```
do
  if s.is_empty then
    create Result.new
  else
    Result := build (s.but_last)
    Result.put (s.last)
  end
ensure
...
```

Figure 3.9: Encoding a controllability check in a verifier: existence of a provably correct implementation of procedure `build` implies that the specification of class `QUEUE` in Fig. 3.4 is controllable; one example of such an implementation is shown on the right.

for generating such programs automatically in common cases. A syntactic check can make sure that any command call on one of the arguments is immediately followed by a call to the same command on the other argument.

The importance of observability for understandable specifications is discussed in detail in [133]; most importantly, non-observable specifications are confusing for clients, because they make irrelevant distinctions in the mental model. When an observability check fails, the course of action depends on whether the developer can provide an implementation of `distinguish` that is *intuitively* correct. If yes, the postconditions of the features involved in `distinguish` should be strengthened; otherwise, the developer should reconsider his choice of model, or add more features to the class interface. Lack of observability can be justified for nondeterministic and partial abstractions, described above in connection with completeness.

Controllability

Showing that the interface specification of class `QUEUE` is controllable amounts to providing an implementation of routine `build` that conforms to the contract given in Fig. 3.9 on the left. For every valid value of the model `sequence` (in the case of `QUEUE`, a model is trivially valid), such program must construct a `QUEUE` object with the given model.

Most observations about `distinguish` also apply to `build`. In particular, a failed verification can be attributed either to weak postconditions, or to missing commands or model constraints. Apart from nondeterministic and partial abstractions, lack of controllability can be justified for immutable interfaces, which do not provide any commands, but nevertheless are commonly considered useful. Additionally, deferred classes are deemed uncontrollable according to the definition of `build` given above, because an object of a deferred class normally cannot be created. An alternative definition of `build` could take an extra argument `q: QUEUE` and ensure `q.sequnce = s`.

3.5 Experimental Evaluation

This section describes experiments in developing model-based contracts for real object-oriented software written in Eiffel. The experiments target two non-trivial case studies based on data-structure libraries with the goal of demonstrating that deploying high-quality model-based contracts is feasible, practical, and useful.

3.5.1 Case studies

The first case study targeted EiffelBase: a mature Eiffel data structure library, extensively exploiting traditional Design by Contract. We selected 7 classes from EiffelBase, for a total of 304 features (254 of them are public) over more than 5700 lines of code. The 7 classes include 3 widely used container data structures (`ARRAY`, `ARRAYED_LIST`, and `LINKED_LIST`) and 4 auxiliary classes used by the containers (`INTEGER_INTERVAL`, `LINKABLE`, `ARRAYED_LIST_CURSOR`, and `LINKED_LIST_CURSOR`). Our experiments systematically introduced models and conservatively augmented the contracts of all public features in these 7 classes with model-based specifications.

The second case study developed EiffelBase2, a new general-purpose data structure library, designed from the start with expressive model-based specifications. The container part of EiffelBase2 consists of 61 classes with a total of 689 features (558 of them public). For more detail on EiffelBase2, see Chapter 2.

3.5.2 Results and discussion

This section addresses the following two research questions based on the experience with EiffelBase and EiffelBase2:

1. How many different model classes are needed to write model-based contracts?

2. Are the four criteria of model-based contract quality applicable in practice?

```
class SET [G]
model set, eq
  has (v: G): BOOLEAN
    -- Does this set contain 'v'?
    ensure
      Result = some x ∈ set : eq (x, v)
    end

  set: MML_SET [G]
    -- The set of elements

  eq: PREDICATE [G, G]
    -- Equivalence relation on elements
end
```

(a) Class SET.

```
class BINARY_TREE [G]
model: map
  add_root (v: G)
    -- Add a root with value 'v'
    -- to an empty tree
    require
      map.is_empty
    ensure
      map.count = 1
      map [Empty] = v
    end

  map: MML_MAP [MML_SEQUENCE[BOOLEAN], G]
    -- Map of paths to elements
end
```

(b) Class BINARY_TREE.

Figure 3.10: Examples of nonobvious models in EiffelBase2.

How many model classes?

One of the biggest objections to the model-based approach to specification is that its reliance on a fixed collection of predefined mathematical theories severely limits the domain where the approach is applicable. Our experiments suggest that a moderate number of well-understood mathematical models suffices to specify a general-purpose library of data structures. In particular, model-based contracts for EiffelBase used elementary model types such as Booleans, integers, and references, as well as model classes for (finite) sets, relations, and sequences. EiffelBase2 additionally required (finite) maps and bags, as well agents for modeling (infinite) equivalence and order relations.

Determining to what extent this is generalizable to software other than data-structure libraries is an open question which belongs to future work. Domain-specific software may indeed require complex domain-specific model classes (e.g., real-valued functions, stochastic variables, finite-state machines), and application software that interacts with a complex environment may be less prone to accurate documentation with models.

Another interesting observation is that the correspondence between the limited number of model classes needed in our experiments and the classes using these model classes is far from trivial: data structures are often more complex than the mathematical structures they implement. Consider, for example, class SET in Fig. 3.10a: EiffelBase2 sets are parameterized with respect to an equivalence relation, hence the model of SET is a pair of a mathematical set and a predicate. Another significant example is `BINARY_TREE` (Fig. 3.10b): instead of introducing a new model class for trees or graphs, `BINARY_TREE` concisely represents a tree as a map of paths to values; the model of a path is in turn a sequence of Booleans.

Are quality criteria applicable?

To determine if model-based contracts in EiffelBase and EiffelBase2 satisfy the quality criteria of completeness, observability, closure, and controllability, we reasoned informally, but rigorously, guided by the check procedures outlined in Sect. 3.4.2.

The experiment mostly focused on assessing completeness, since it is the main indicator of strong specifications, and also the precursor to precise evaluation of the other three criteria. Only 7% of public features in EiffelBase with model-based contracts and 4% of public features in EiffelBase2 have incomplete postconditions. All of them are examples of benign incompleteness mentioned in Sect. 3.4.2. EiffelBase2, in particular, was designed trying to minimize the number of incomplete postconditions, and does not exhibit incompleteness due to imperfections in information hiding.

These results indicate that model-based contracts make it feasible to write systematically complete contracts; in most cases this was even relatively straightforward to achieve. Unsurprisingly, using model-based contracts dramatically increases the completeness of contracts in comparison with standard Design by Contract. For example, 66% of public features of class `LIST` in the original version of EiffelBase (without model-based contracts) have incomplete postconditions, including 31% without any postcondition.

For other quality criteria our analysis was limited to the EiffelBase2 library. We only found two examples of closure and observability violations: classes `INPUT_STREAM` and `RANDOM`, both of which are highly nondeterministic abstractions. Informal reasoning about closure revealed 6 cases of missing class invariant clauses. For some container classes, two objects with different models can only be distinguished using an iterator; intuitively such classes should be deemed observable, since the container and its iterator are really parts of the same reusable component. As part of future work, the observability check proposed in Sect. 3.4.2 should be adapted for this case.

On the other hand, only about 57% of EiffelBase2 classes are controllable as per the encoding in Sect. 3.4.2. Another 15% are uncontrollable only due to their deferred status, which can be fixed by adopting an alternative definition of controllability discussed at the end of Sect. 3.4.2. The rest of the classes provide immutable and partially mutable interfaces.

3.6 Related Work

The ultimate goal of this chapter is to help developers design better interfaces and write better interface specifications for reusable components. Literature in various areas, within and outside software engineering, offers plenty of informal advice on good interface design: [96, 14, 84] to name just a few. In the present work, we are interested in *formal* principles, expressed in terms of formal behavioral specifications and amenable to mechanized (ideally, automatic) reasoning.

Consistency and completeness of interface specifications were first formally explored for algebraic specifications of ADTs [48, 47, 138]; these results were discussed in Sect. 3.3.1. [59] introduces a definition of *expressiveness* of the operation set of an ADT, which captures whether an ADT defines enough operations for a client to implement all computable functions on its values. A notion similar to our indistinguishability relation is defined in [13]. All these lines of work share the primary goal with the present chapter, but since their specifications are algebraic, the results are not directly applicable to imperative object-oriented programs.

A more practical approach to strong specifications of reusable components is provided by the model-based style, pioneered in specification languages such as Larch [49], Z [134], VDM [58], and RESOLVE [98]. Central to quality of model-based specifications is the problem of choosing an appropriate model for specifying a given interface. For example, [58] introduces a semi-formal notion of an "unbiased" or "sufficiently abstract" model.

The most detailed and formal treatment of quality criteria for model-based specifications can be found in the context of the RESOLVE framework [132, 114, 133]. The latter is probably the work most related to ours; it discusses definitions of observability and controllability for components implemented in an imperative programming language. The given definitions are very similar to the checks we propose in Sect. 3.4.2, but there are two major differences in terms of method and scope. First, [133] directly encodes the informal intuition behind observability and controllability in the programming language, and then explores subtle differences between various encodings; the present work first defines the quality criteria at a more abstract level, in terms of ADTs, which promotes better understanding of the principles before introducing the full complexity of an imperative programming language. With this approach, we managed to recast some well-known properties of algebraic specifications for model-based specifications, and to come up with a precise mathematical characterization of a "good" model (as a homomorphic image of an ADT with a uniquely defined kernel), which encompasses and unifies both observability and controllability, as well as two complementary quality criteria of closure and completeness.[8] The second difference is that we assume a mainstream object-oriented programming language, with dynamic memory and aliasing, while in RESOLVE aliasing does not arise, which simplifies the mapping between program objects and their abstract values. In particular, RESOLVE specifications do not need modify clauses.

The specification notation of this chapter is a direct extension of the work by Schoeller on model-based contracts [112, 111]. In terms of scope, our approach strives to be more methodological and systematic, with the primary target of fully contracting a complete library of data structures. Outside of Eiffel, the Java Modeling Language (JML) [69, 68] is likely the notation that shares the most similarities with ours: JML annotations are based on a subset of the Java programming language and the JML framework provides a library of model classes mapping mathematical concepts. Previous research on models in JML [17, 24, 33] focuses mostly on their role in runtime checking and verification, and does not discuss guidelines for writing model-based specifications or criteria for assessing their quality.

3.7 Summary and Future Work

This chapter introduced a methodology to write strong behavioral interface specifications for reusable object-oriented components. The methodology is based on the idea of expressing abstract states of objects though models, and features formal definitions of four complementary quality criteria, which together guarantee that a specification is both as strong and as abstract as possible, for a given component interface. The application of the methodology to two libraries of general-purpose data structures demonstrates its practical feasibility and usefulness.

One important direction for future work is implementing the proposed quality checks in an auto-active verifier, such as AutoProof [5], and evaluating whether these checks capture our intuitive understanding of the quality criteria for real software components. A technical challenge is to automate the checks as much as possible: in particular, propose heuristics for generating witness programs, required in observability and controllability proofs.

Another direction of future work is exploring feasibility of strong model-based contracts beyond libraries of general-purpose data structures.

[8]We should remark that [133] does not consider completeness desirable.

Chapter 4

Testing against Strong Specifications

In the previous chapter, we have presented a methodology for equipping object-oriented components with *strong* contracts. Whether such contracts can be adopted in practice depends on their cost-to-benefit ratio. This chapter explores the benefits of strong contracts for automated testing and estimates their cost in terms of annotation overhead.

4.1 Introduction

Many years of progress in the theory and practice of formal methods notwithstanding, writing software specifications still seems to be "disliked by almost everyone" [104]. In many cases, this aversion is a consequence of a high cost/benefit ratio—perceived or real—of writing and maintaining accurate specifications on top of the code. After all, developers *will* write specifications as long as they are simple, and help them write and debug code better and faster. One example is Design by Contract, discussed in detail in the previous chapter, where simple executable specifications support design, incremental development, and testing and debugging. Another one is test-driven development [12], where rigorously defined test cases play the role of specifications in defining correct and incorrect behavior. Experiences with these techniques show that providing lightweight specifications is an accepted practice when it brings tangible benefits and integrates well with the overall development process.

But what about *strong* specifications—like the ones proposed in the previous chapter—which attempt to capture the entire (functional) behavior of the software? Should they be deemed impractical on the grounds that the effort required to write them is not justified against the benefits they bring in the majority of mundane software projects? This chapter studies the impact of deploying strong behavioral interface specifications, for detecting errors in software by means of automatic testing.

Using strong contracts involves costs and possible benefits. Among the former, we have the programming effort necessary to write such strong specifications and the runtime overhead of checking them during execution. The benefits may include finding more errors, finding more subtle errors, finding errors more quickly, and exposing errors in ways that are easier to understand and correct. Our contributions address the cost factors—by measuring and trying to mitigate them—and assess the benefits:

- Sect. 4.3 proposes an extension to the specification methodology presented in Chapter 3, which enables runtime checking of model-based contracts using standard Eiffel mechanisms and tools; the extension is supported by a contract pre-processor we implemented.

- Sect. 4.4 and 4.5 describe an extensive empirical study that evaluates the use of strong contracts for real software and measures their costs and benefits in terms of defect detection.

The bulk of our empirical study targets EiffelBase: Eiffel's standard container library (see Chapter 2). The production version of EiffelBase includes traditional (partial) contracts, which are nonetheless quite effective at finding implementation bugs automatically using contract-based random testing [86], where executable contracts serve as oracles and enable a push-button testing process. In this study, we augment the simple contracts that come with EiffelBase using the methodology of Chapter 3, with a few extensions discussed in Sect. 4.3. The result is EiffelBase+: a version of EiffelBase with identical implementation but strong (mostly complete) specifications.

In an extensive set of experiments, we compare the effectiveness of random testing on EiffelBase and EiffelBase+, with the goal of assessing whether the additional effort invested into the strong contracts pays off in terms of quantity and complexity of the bugs found. Our experiments show that these measures dramatically increase when deploying strong specifications: random testing found twice as many bugs in EiffelBase+, and the simple contracts of EiffelBase would have uncovered none of the newly found bugs. The overhead size of specifications, in contrast, remains moderate, with the specification-to-code ratio going from 0.2 to 0.46.

Our approach to writing strong specifications that are effective for testing is not limited to Eiffel programs. In a companion set of experiments, we applied the same technique to writing strong specifications for the DSA

```
merge_right (other: LINKED_LIST [G])
  require
    not after
    other ≠ Void
    other ≠ Current
  ensure
    count = old count + old other.count
    index = old index
  end
```

(a) Standard specification.

```
merge_right (other: LINKED_LIST [G])
  require
    not after
    other ≠ Void
    other ≠ Current
  modify [sequence] Current
  ensure
    sequence = old (sequence.front (index)
      + other.sequence
      + sequence.tail (index + 1))
  end
```

(b) Model-based specification.

Figure 4.1: Specification of routine `merge_right` in `LINKED_LIST`.

C# library [36], and tested the result using Pex [124]; in this case too we discovered new bugs with reasonable additional effort.

Somewhat orthogonally to the goals of the rest of the chapter, our third set of experiments applies automated testing to EiffelBase2: a new general-purpose data structure library, designed from the start with expressive model-based specifications (see Chapter 2). This experiment discovered significantly fewer defects than the one targeting EiffelBase+, which serves as evidence of the important role strong specifications play in software design.

4.2 A Motivating Example

The following example illustrates and justifies the use of strong specifications in testing. Consider the EiffelBase class `LINKED_LIST`—Eiffel's standard implementation of linked lists. Like all containers in EiffelBase, `LINKED_LIST` includes an internal cursor to iterate over elements of the list. The query `index` gives the cursor's position, which can be on any element of the list in positions 1 through `count`, or take the special boundary values 0 ("`before`" the list) and `count` + 1 ("`after`" the list). The attribute `count` denotes the number of elements in the list.

Fig. 4.1a shows the EiffelBase specification of the routine `merge_right` from `LINKED_LIST`. The routine inserts another list `other` passed as argument into the current list immediately after the cursor position. For example, if **`Current`** stores the sequence of elements `b·a·r·t` with cursor positioned on the "`r`" (`index` = 3) and `other` stores `o·n·e`, `merge_right` changes **`Current`** to `b·a·r·o·n·e·t`. The precondition specifies that the routine cannot be called when the cursor is `after`: there is no valid position to the right of it. It also demands that `other` be non-`Void` and not aliased with the `Current` list: otherwise, merging is not well defined. The postcondition describes some expected effects of executing `merge_right`: the `Current` list will contain as many elements as it contained before the call to `merge_right` plus the number of elements of the `other` list; and the cursor's position `index` will not change.

The contracts in Fig. 4.1a are a good example of the kind of specification that Eiffel programmers normally write [106]: it is correct and nontrivial, and it can help detect errors in the implementation, such as performing partial merges or incorrectly leaving the cursor at a different position. Unfortunately the specification is also incomplete, because it does not precisely describe the expected state of the list after merging. In fact, the current implementation of `merge_right` contains an error that is undetectable against the specification of Fig. 4.1a. The error occurs in the special case of calling `merge_right` with cursor `before` the list (`index` = 0): the implementation will insert `other` at the second rather than at the first position. For example, merging f·o·l·d and u·n when the cursor is `before` yields f·u·n·o·l·d instead of the correct u·n·f·o·l·d.

Chapter 3 presented a methodology to write, with moderate effort, strong specifications that extend and, whenever possible, complete this kind of partial specification. Fig. 4.1b shows the strong specification obtained by applying the methodology to `merge_right`, the way it appears in EiffelBase+. As is common in most Eiffel projects, the programmer who wrote `merge_right` did a good job with the precondition, which is sufficiently detailed and need not be strengthened. The postcondition, however, turns into a single assertion that defines the `sequence` of elements stored in the list after calling `merge_right` as the concatenation (operator +) of three segments: `Current`'s original sequence up until position `index` (written `sequence.front (index)`), followed by `other`'s element sequence, followed by the original sequence from position `index` + 1 (written `sequence.tail (index + 1)`). In order to complete the specification of `merge_right`, we add the clause `modify [sequence] Current`, which means that `merge_right` may only modify the sequence of elements in the `Current` list and *nothing else*. Using the strong postcondition in Fig. 4.1b, completely automatic testing with the AutoTest tool [86] detected the error that occurs in `merge_right` when the cursor is `before`.

4.3 Strong Specifications for Testing

This section discusses additional annotations and tool support necessary to exploit the full potential of model-based contracts for runtime checking and testing. We refer to the resulting extended methodology, as well as to the supporting tool, as RunMBC (for *runtime model-based contracts*).

```
class LINKED_LIST [G]
model sequence, index

  index: INTEGER
    -- Current cursor position

  count: INTEGER
    -- Number of elements

 ...
-- Implementation:
  first_cell: LINKABLE [G]
    -- First cell of the list

  last_cell: LINKABLE [G]
    -- Last sell of the list
```

```
-- Specification:
  sequence: MML_SEQUENCE [G]
    -- Sequence of elements
     ...

invariant
  -- Model constraint
  0 ≤ index ≤ sequence.count + 1
  -- Attribute definition
  count = sequence.count
  -- Linking invariant
  bag = sequence.to_bag
  -- Internal representation constraint
  not sequence.is_empty implies
    last_cell.item = sequence.last
end
```

Figure 4.2: Excerpt of the model-based specification of `LINKED_LIST` in EiffelBase+.

We first introduce the RunMBC extensions for class invariants in order to discover more faults (Sect. 4.3.1) and avoid spurious violations (Sect. 4.3.2); Sect. 4.3.3 discusses the choice of implementation for model classes and model queries, as well as tool support for runtime checking of framing specifications and other additional annotations.

4.3.1 Representation Constraints

Chapter 3 details three kinds of class invariants present in model-based contracts: *model constraints*, prescribing which abstract object states are valid; *definitions of public attributes*, expressing attribute values in terms of model queries; and *linking invariants*, enabling the reuse of ancestor specifications stated in terms of a different model. Fig. 4.2 gives examples of all three kinds of invariants used in the class `LINKED_LIST` from EiffelBase+.

RunMBC introduces another kind of class invariant clauses: *internal representation constraints*, which relate the values of model queries to the private attributes of the class. For example, the last invariant clause in Fig. 4.2 says that the item stored in the `last_cell` of the linked list coincides with the last element of list's `sequence` (whenever the sequence is not empty).

Unlike other model-based specifications, invariants of this type do not describe the public interface of the class and usually cannot be made complete without revealing unnecessary implementation details in the model.

However, even in this limited form, they turned out to be very effective at revealing errors that corrupt object's internal representation (see Sect. 4.5.1).

4.3.2 Avoiding False Positives

The semantics of class invariants becomes tricky in the presence of callbacks, inter-object dependencies, and state updates that might make objects temporarily inconsistent. A sound verification methodology must ensure that invariants always hold at certain program points, while allowing them to be broken at other points. The next chapter of this thesis develops one such invariant methodology, building on top of much related work [9, 77, 11, 92, 80, 30]. Although diverse, all those methodologies have something in common: in return for the soundness guarantee, they demand a significant amount of auxiliary annotations.

In a dynamic setting, where the goal is finding faults rather than showing their absence, soundness concerns are largely irrelevant. Some aspects of the tricky invariant semantics are, however, also harmful for testing, since they may cause *false positives*: spurious runtime violations that arise when an invariant is being checked at a program point where it need not hold. To address this problem we introduce additional dedicated constructs for complex invariant properties. We borrow some ideas from existing verification methodologies (e.g., [11, 80], among many)[1]; unlike these sophisticated techniques, the present solution for class invariants does not target soundness or comprehensiveness, but requires very few annotations and is sufficient in practice to avoid spurious invariant violations.

Standard Eiffel mechanisms check class invariants at the beginning and at the end of every qualified[2] call on an object of the class. This rule successfully prevents checking the invariant whenever routines of a class call one another within the boundaries of a single object, in order to accomplish a common task, as the object will normally be inconsistent ("open") until all operations are completed. When circular dependencies between objects arise, this semantics may lead to spurious invariant violations: a problem known in the Eiffel community as the *dependent delegate dilemma* [85].

Consider an example derived from real code in EiffelBase: a binary tree data structure, where each node has a link to its `parent` and `left` and `right` children. The **`Current`** node is executing one of its routines and is temporarily in a state that violates the invariant; to restore it, it makes a qualified call on, say, its right child. The object `right`, however, does not know that its `parent` is

[1]Chapter 5 develops these same ideas into a full-fledged verification methodology.

[2]A call `t.r` is *qualified* when the target `t` is an object other than **`Current`**.

in the middle of executing a call; if `right` calls back to **`Current`**, then, it detects an invariant violation even if `right`'s call does not rely on the invariant.

RunMBC deploys a runtime semantics where these spurious invariant violations do not occur. Objects are implicitly equipped with a Boolean attribute `open` that is set to true at the entrance of every public routine call on the object and restored to its previous value when the routine terminates; class invariants are checked only if `open` is false. This automatically solves the dependent delegate problem in the presence of callbacks: when `right` calls back to **`Current`**, the latter is open, and hence its invariant is not checked.

This "implicit opening" mechanism is not sufficient to avoid spurious invariant violations when an object's invariant depends on the state of other objects. Consider again binary trees; an invariant states that the **`Current`** node is its parent's left or right child:

```
parent ≠ Void implies (parent.left = Current or parent.right = Current)
```

Routine `prune_left` removes **`Current`**'s left child as follows:

```
old_left := left
left := Void
if old_left ≠ Void then
  old_left.set_parent (Void)
end
```

When `old_left.set_parent (Void)` is called to remove the back-link from **`Current`**'s child, `old_left`'s class invariant is violated: its parent's `left` is already set to **`Void`** and `old_left` is not open; in fact, the very reason for calling `set_parent` is to remove this inconsistency. RunMBC provides the keyword **`depend`** to declare that an invariant clause depends on the state of an attribute, and hence it should be checked only if the object attached to the attribute is closed. Annotating the invariant in the example with **`depend`** `parent` removes the spurious invariant violation (`old_left.parent` is **`Current`**, which is open).

In the few cases when fine-grained control over the opening of objects is necessary, RunMBC provides the **`unwrap`** clause for routines, which *explicitly* opens the objects attached to some of the routine's arguments when the routine begins execution and restores them when the routine terminates (as we discussed, the target is always opened implicitly). Consider a variant of the binary tree example where nodes have an attribute `is_root` that should be true when their `parent` node is **`Void`**:

```
parent = Void implies is_root
```

In this variant, `prune` takes an argument of class `NODE` that is supposed to be its left or right child and removes it as follows:

```
prune (n: NODE)
```

```
do
  if left = n then
    left.set_parent (Void) ; left.set_root (True) ; left := Void
  end
  if right = n then ... end -- analogously to 'right'
end
```

When prune's call to left.set_parent returns, the invariant of left is violated : left.parent = Void but left.is_root is still false. Annotating prune with unwrap n sets n.open to true at the beginning of the routine, and restores it to its previous value at the end, which suspends checking of n's invariant until prune terminates, thus removes the spurious invariant violation.

As we discuss in Sect. 4.4, in EiffelBase+ we had to deploy explicit depend and unwrap annotations only in a very few cases, limited to doubly-linked list nodes, and binary and n-ary trees.

4.3.3 Tool support

Most model-based contracts can be checked at runtime and used in testing out of the box: with the same tools and user experience as standard Eiffel contracts. In our experiments, model queries introduced for specification purposes are implemented as regular functions that compute the abstract model value from the concrete object state. Compared to an alternative, attribute-based implementation, function-based approach is usually more concise, less error-prone, and enforces a cleaner separation between the implementation and the specification, since model queries do not require explicit initialization or updates in the code. The specification classes we provide in MML are also regular Eiffel classes, implemented in a functional style. Even though this approach to implementation of model queries and model classes potentially incurs a high runtime overhead, the experiment results in Sect. 4.5 confirm its feasibility for contract-based testing.

Checking frame specifications and complex class invariants requires additional tool support. At the same time, those specifications are *conservative*: if executed with standard Eiffel mechanisms, they are simply ignored instead of producing spurious violations. Newly introduced specification constructs, such as modify, depend and unwrap, do not have any effect in the standard Eiffel semantics, since they are specified using note meta-annotations (similar to Javadoc or C#'s meta-data). To specify subtle invariant clauses in a conservative way, we exclude them from the original class invariant and encapsulate them into *invariant functions*: regular Boolean functions tagged with a dedicated note annotation.

We have developed a simple tool that rewrites the annotations described above into plain Eiffel, according to the following rules.

- A *frame specification* `modify [f] x`, where `x` is an argument of a routine `r`, generates postcondition clauses `x.g = old x.g` for every model query `g` $\neq$ `f` declared in the class of `x`; as well as a clause `y` $\neq$ `x implies y.h = old y.h` for all other arguments `y` of `r`. This does not capture the full semantics of the `modify` clause, which prohibits changing the model of any object different from `x`, not only those attached to the routine's arguments; full frame checks at runtime are, however, infeasible.

- A Boolean attribute `open` is added to every class under test. The body of every public routine is extended with new instructions, which set `open` to `True` in the beginning, and restore its old value at the end.

- The body of every routine annotated with `unwrap x` is extended with new instructions, which set `x.open` to `True` in the beginning, and restore its old value at the end.

- Every invariant clause `inv` is replaced with `not open implies inv`. Every invariant function `f` generates an invariant clause `not open implies f`. If an invariant function `f` is annotated with `depend x`, it generates an invariant clause `(x` $\neq$ `Void and not open and not x.open)implies f`.

Our experiments show (Sect. 4.5) that postconditions and invariants automatically generated by RunMBC are effective at finding faults and successfully avoid common sources of spurious violations, while requiring very little effort compared to writing equivalent assertions manually.

4.4 Experiments

We performed an extensive experimental evaluation to assess the benefits of using strong specifications for finding errors in software.

4.4.1 Research Questions

The overall goal of this evaluation is assessing and comparing the advantages and the cost of deploying strong specifications in the form of model-based contracts when applied to automatic contract-based testing of real software.

This materializes into the following research questions:

1. Are strong specifications effective for finding faults in software?

2. Do strong specifications find subtle and complex faults?

3. Do strong specifications find faults in little testing time?
4. What is the performance overhead of checking strong specifications at runtime?
5. What is the development effort required to provide strong specifications for existing software?

To answer these questions, we conducted two sets of experiments, targeting software written in Eiffel (Sect. 4.4.2) and C# (Sect. 4.4.3). In both cases, we selected an open-source library, specified it following the RunMBC methodology, and extensively tested it with a standard automatic testing tool. The rest of this section discusses the experiments; Sect. 4.5 presents the results.

We also report on a preliminary study that applied automated testing to software designed with strong contracts from the start (Sect. 4.4.4). This study is targeting a different research question: namely, if using strong specifications at the design stage improves software quality.

4.4.2 Eiffel Experiments

The main experiments target EiffelBase (rev. 506)—Eiffel's standard base library—from which we selected 21 classes of varying size and complexity. Using the facilities of the EiffelStudio IDE, we built the *flat* version of each class, which is a self-contained implementation including all inherited members explicitly in the class text. This simplified the task of writing specifications without being distracted by EiffelBase's deep multiple inheritance hierarchy. For each of the 21 classes in their flat version, Tab. 4.1 lists the size (in LOC) and the number of public routines (PR), possibly also including helper classes directly used in the class implementation. Since different classes may share some parent or helper classes, the totals at the bottom of the table are in general less than the sum of the elements in each column.

Like most Eiffel software, EiffelBase comes with partial specification in the form of contracts: the 21 classes include 561 precondition clauses, 985 postcondition clauses, and 250 class invariant clauses. In EiffelBase+ we completely replaced EiffelBase's original postconditions and class invariants with model-based annotations, but we kept EiffelBase's preconditions (with a few exceptions discussed below)[3]. EiffelBase+'s strong specification includes 589 precondition clauses, 1066 postcondition clauses, and 164 class invariant clauses (21% model constraints, 23% attribute definitions, 10% linking

[3]All the code developed as part of the study, as well as descriptions of found faults are publicly available online [123].

Table 4.1: Eiffel classes under test and results.

	EiffelBase							EiffelBase+					
Class	LOC	PR	TC	Spec	Inc	Real	New	LOC	PR	TC	Inc	Real	New
ARRAY	831	53	2.8	2	0	2	1	986	59	1.2	0	3	2
ARRAYED_LIST	1840	86	3.5	0	0	0	0	2037	92	1.7	0	1	1
ARRAYED_QUEUE	537	32	1.8	0	0	2	0	648	37	3.8	0	2	0
ARRAYED_SET	1960	49	5.8	3	1	8	0	2053	58	5.4	0	16	8
BINARY_TREE	1122	64	1.0	2	5	6	0	1366	70	1.1	0	16	10
BOUNDED_QUEUE	558	32	1.4	0	0	2	0	659	37	3.8	0	2	0
HASH_TABLE	1345	51	0.9	1	0	1	0	1626	63	0.9	0	2	1
HASH_TABLE_ITERATOR	217	15	0.4	0	0	0	0	248	15	0.5	0	0	0
INDEXABLE_ITERATOR	186	14	1.0	2	0	0	0	228	15	2.7	0	0	0
INTEGER_INTERVAL	519	42	4.3	1	1	0	0	637	45	0.9	0	3	3
LINKED_LIST	1759	69	2.0	0	0	2	0	1942	77	2.5	0	5	3
LINKED_LIST_ITERATOR	311	15	0.7	0	0	0	0	357	16	0.7	0	0	0
LINKED_SET	2128	83	5.4	5	2	7	0	2410	94	4.8	0	24	17
LINKED_SET_ITERATOR	311	15	0.7	0	0	0	0	357	16	0.7	0	0	0
LINKED_STACK	1077	27	1.0	0	0	3	1	1078	32	3.2	0	6	4
TWO_WAY_LIST	2007	71	0.8	0	0	3	0	2184	79	2.2	0	6	3
TWO_WAY_LIST_ITERATOR	412	15	0.7	0	0	0	0	462	16	0.7	0	0	0
TWO_WAY_SORTED_SET	2706	91	5.3	5	2	9	0	2983	102	4.8	1	34	25
TWO_WAY_SORTED_SET_ITERATOR	412	15	0.7	0	0	0	0	462	16	0.7	0	0	0
TWO_WAY_TREE	2548	90	1.4	4	4	22	5	2865	101	1.3	0	29	12
TWO_WAY_TREE_ITERATOR	412	15	0.7	0	0	0	0	462	16	0.7	0	0	0
Total	**17841**	**1033**	**42.5**	**15**	**12**	**48**	**7**	**19400**	**1164**	**44.4**	**1**	**103**	**62**

LOC: Lines of code, PR: Public routines, TC: Test cases drawn (million)
Spec: Specification errors found, Inc: Inconsistency errors found, Real: Real faults found, New: Faults found only in this experiment

invariants, 46% internal representation constraints), as well as 278 **modify**, 4 **depend** and 7 **unwrap** clauses. Tab. 4.1 shows the size (in LOC and PR) of EiffelBase+, which also includes model definitions and implementations of the model queries.

Preconditions. In all but two EiffelBase+ classes, we kept the same preconditions as in EiffelBase. Within the specific setup of our experiments, where we compare traditional contracts and strong contracts, it is important to have the same preconditions in the two artifacts under comparison. Preconditions define the valid calling contexts of routines (in particular, contract-based testing tools use them to select valid test cases). Changing preconditions would change the semantics of classes in a way similar to changing implementation: strengthening a precondition may reduce the number of faults detectable for the routine, since it would move obligations from the routine to its clients; weakening a precondition may increase the number of faults, since it would impose a heavier burden on its implementation. We treat preconditions as developers' design decisions, which we normally take at face value. This policy makes the experiments with EiffelBase and EiffelBase+ fully comparable.

The only exception occurred with four routines of class BINARY_TREE and eight routines of class TWO_WAY_TREE that insert new nodes into a tree. In these twelve cases, we strengthened the preconditions to disallow creating cycles among nodes in the tree. Without the strengthening, tree instances can be driven into inconsistent states with cycles where the whole specification of trees would be inapplicable. These changes in preconditions are conservative: the EiffelBase+ experiments using these stronger preconditions miss a few faults that are detected in EiffelBase, because the new preconditions rule out some previously valid failing test cases. Since these changes affect only a small fraction of all the experiments, the results with EiffelBase and EiffelBase+ remain comparable.

Specification correctness. To create strong specifications, we analyzed the original implementation, contracts, and comments in EiffelBase, and relied on our informal knowledge of the semantics of data structures and their implementation. To increase our confidence in the correctness of the new specification, we ran a series of short preliminary testing sessions with the goal of detecting inconsistencies and inaccuracies. All our changes were conservative, in that whenever a new contract forbade a behavior that was not clearly forbidden by the comments, standard contracts, or informal knowledge, we weakened the specification to allow the behavior. In all, we reached a high confidence that EiffelBase+'s specification is correct and strong enough. The results of the main testing sessions (Sect. 4.5) corroborate this informal

Table 4.2: C# classes under test and results.

Class	DSA LOC	DSA PR	DSA+ LOC	DSA+ PR	Testing T	Testing F
AvlTree	345	6	391	7	23	1
BinarySearchTree	205	5	213	5	21	1
CommonBinaryTree	419	13	536	18	83	0
Deque	201	14	231	15	145	0
DoublyLinkedList	408	17	458	19	171	3
Heap	371	11	390	12	61	1
OrderedSet	136	9	158	11	10	0
PriorityQueue	186	13	216	14	65	0
SinglyLinkedList	439	20	492	22	148	3
Total	**3043**	**133**	**3486**	**149**	**727**	**9**

LOC: Lines of code, PR: Public routines
T: Testing time (minutes), F: Faults found

assessment.

Testing experiments. We ran a large number of random testing sessions with the AutoTest framework [86] on a computing cluster of the Swiss National Supercomputing Centre, configured to allocate a standard 1.6 GHz core and 4 GB memory to each parallel AutoTest session. The experiments totalled 1680 hours of testing time that generated nearly 87 million test cases; the TC columns in Tab. 4.1 list the million of test cases drawn when testing each class in EiffelBase and in EiffelBase+. The testing of every class was split into 30 sessions of 80 minutes, each with a new seed for the random number generator, such that corresponding sessions in EiffelBase and EiffelBase+ use the same seeds. This thorough testing protocol guaranteed statistically significant results [4].

4.4.3 C# Experiment

A smaller set of experiments targets 9 classes from DSA (v. 0.6)—an open-source data structure and algorithm library written in C# [36]. Support for contracts in C# appeared only recently, through the Code Contracts framework [28]; therefore, most C# projects (including DSA) do not have any formal specification. This was a chance to extend the validation of the model-based contracts methodology to other languages and to projects without pre-existing specification.

We instructed one of our bachelor's students to follow the methodology of Chapter 3 and create DSA+: a variant of DSA with the same implementation but equipped with strong model-based contracts. DSA+'s specification includes 6 precondition clauses, 143 postcondition clauses and 23 class invariant clauses. For each of the 9 classes, Tab. 4.2 shows the size (in LOC and PR) of both DSA and DSA+, inclusive of all specification elements and model query implementations. As in Tab. 4.1, the count also includes (possibly shared) helper classes. Flattening was not necessary in this case because the inheritance hierarchy is shallow.

Specification correctness. We manually inspected the DSA+ specification written by our student, and assessed its quality to be comparable to that of EiffelBase+ in terms of correctness and completeness. Since DSA was not designed with contracts in mind, it makes recurrent usage of defensive programming, throwing exceptions to signal invalid arguments. The experiment setup is consistent with this programming style: we do not consider such exceptions to be faults.

Testing experiments. We performed automatic testing with the Pex concolic testing framework [124] running on a Windows box equipped with a 2.16 GHz Intel Core2 processor and 3 GB of memory. In order to achieve better test coverage we provided Pex with factory classes, which construct nontrivial instances of classes under test, and tweaked the exploration parameters. The experiments ran for about 12 hours; column T in Tab. 4.2 reports the breakdown per class in minutes. The testing time is different from class to class because Pex testing sessions by default are limited by coverage criteria rather than duration. We only tested DSA+ since DSA has no formal specification elements usable as automated testing oracles.

The C# experiment is less extensive than the Eiffel experiment and intended as a control mechanism to identify any potential dependency of the results on the Eiffel language, libraries (EiffelBase) or tools.

4.4.4 EiffelBase2 Experiment

In an independent preliminary study we targeted EiffelBase2 (Chapter 2)—a new container library for Eiffel, developed from scratch using the model-based contracts methodology. We have grouped all effective classes of EiffelBase2 available at the time of the study into 10 groups, each including a data structure implementation together with its iterators and helper classes. For each of the 10 data structures, Tab. 4.3 lists the total size of the group in its flattened form (in LOC) and the number of public routines (PR). We did not alter the existing specifications in EiffelBase2, since the library already comes with strong model-based contracts.

Testing experiments. We performed random testing using the AutoTest framework, on a Windows box equipped with a 2.6 GHz Intel Core2 processor and 6 GB of memory. For every data structure, we ran 30 testing

Table 4.3: EiffelBase2 classes under test and results.

CLASS	LOC	PR	SPEC	REAL
ARRAY	1254	65	2	1
ARRAYED_LIST	1594	78	0	1
BINARY_TREE	3197	173	1	2
HASH_SET	1101	53	0	2
HASH_TABLE	1376	25	0	0
LINKED_LIST	1812	82	1	2
LINKED_QUEUE	618	42	1	0
LINKED_STACK	618	42	1	0
SORTED_SET	1012	55	0	3
SORTED_TABLE	1348	27	0	0
Total	**8508**	**642**	**2**	**5**

LOC: Lines of code, PR: Public routines
SPEC: Specification errors found
REAL: Implementation errors found

sessions, 60 minutes each. Even though the setup differs slightly from the EiffelBase experiments (in particular, with respect to the testing time and the version of AutoTest used), it still enables a qualitative comparison of the two libraries in terms of the total number of faults found, especially since most faults are revealed at the beginning of a testing session.

4.5 Results

This section discusses the result of the experiments, focusing on the larger EiffelBase experiments, with Sect. 4.5.1 through Sect. 4.5.5 targeting the research questions 1–5 of Sect. 4.4.1. Then, Sect. 4.5.6 and 4.5.7 briefly discuss the experiments with C# and EiffelBase2, respectively, and Sect. 4.5.8 presents possible threats to validity of the results.

4.5.1 Faults Found

AutoTest found 75 faults in EiffelBase and 104 in EiffelBase+; these are *unique*, that is they identify distinct and independent errors. We classified them in three categories.

Specification faults correspond to violations of *wrong* contracts (meaning that in our judgement they specify the expected behavior of the program incorrectly). We found 15 specification faults in EiffelBase (column SPEC in Tab. 4.1) and none in EiffelBase+, which increased our confidence that the preliminary testing sessions mentioned in Sect. 4.4.2 were sufficient to achieve correct specifications. We consider specification faults spurious in our study, because we are not comparing the correctness of the specification in EiffelBase and EiffelBase+ but rather their effectiveness at finding real errors in the implementation.

Inconsistency faults correspond to failures triggered by calls on objects in inconsistent states, which are not captured by a partial class invariant. For example, `LINKED_SET` may be driven into a state where the container stores duplicate elements; calling `remove (x)` in such a state triggers a failure (only one occurrence of `x` is removed), but `remove` is not to blame for it, since it is due to previous erroneous behavior that went undetected. While inconsistency faults are genuine errors, we classify them separately because understanding and locating the ultimate source of an inconsistency is normally harder. Additionally, a single inconsistency fault often results in many failing test cases (potentially in all routines of the class that rely on the broken invariant), requiring additional effort from the developer when analyzing the testing results.

We found 12 inconsistency faults in EiffelBase and a single one in EiffelBase+ (columns Inc in Tab. 4.1); the ultimate source of the latter fault is a class invariant not including all internal representation constraints (see Sect. 4.3.1), which would have required exposing implementation details in the model. The other inconsistency faults of EiffelBase are not detected in EiffelBase+, because, due to stronger class invariants, their *real* source is detected instead. In the `LINKED_SET` example above, instead of the inconsistency fault in `remove`, model-based contracts report a fault in routine `replace`, which does not check if the new value is already present in the set, thereby introducing duplicates. The results in this category indicate that strong specifications report faults in a way that is easier to understand and debug.

All other errors are *real faults* which correspond to genuine errors directly traceable to the code. We found 48 real faults in EiffelBase and 103 in EiffelBase+ (columns Real in Tab. 4.1); 41 of them are found in both sets of experiments, 7 only in EiffelBase, and 62 only in EiffelBase+. We submitted bug reports for all the 110 faults found in our experiments. The Eiffel Software developers in charge confirmed 107 (97%) of them as real bugs to be fixed. This is evidence that we are dealing with genuine faults in our evaluation. The remaining three faults not taken on by the developers also arguably highlight real problems in the implementation, but they are probably not so likely to occur during "normal" runs. The rest of the discussion focuses on real faults unless stated otherwise.

Only seven faults are found in EiffelBase but not in EiffelBase+ (columns New in Tab. 4.1). Four of them are prevented by the strengthened preconditions in the tree classes (Sect. 4.4.2); two are shadowed by new failures occurring earlier; and one disappears due to an unintentional side-effect of a

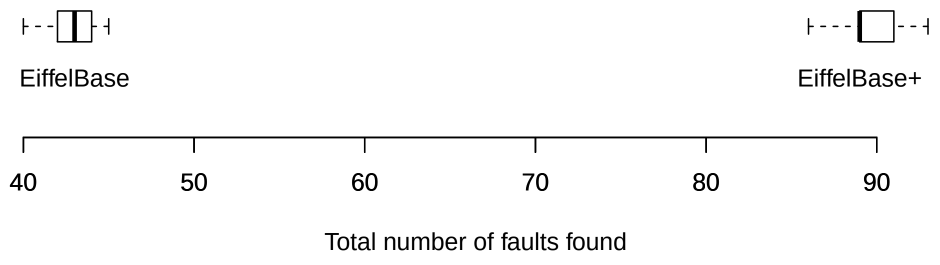


Figure 4.3: Unique real faults found in all classes over 80-minute testing sessions.

model query that amends an invariant violation. None of these faults found only in EiffelBase show inherent deficiencies of strong specifications or of the RunMBC method. In contrast, the 62 faults found only in EiffelBase+ are undetectable in EiffelBase.

Except for the two ITERATOR classes (no faults in both cases) and the two QUEUE classes (the same two faults in both cases), the number of faults found is consistently higher in EiffelBase+ *in each class.* As evident from the boxplot in Fig. 4.3, the difference is highly significant: the Mann-Whitney U test gives $U = 0$ (testing EiffelBase+ outperforms testing EiffelBase in *all* sessions), and $p = 2 \cdot 10^{-11}$ overall and $p \leq 2.1 \cdot 10^{-11}$ for every class (except the ITERATORs and QUEUEs). The difference remains highly statistically significant even if we aggregate the experiments in sessions of different length.

To sum up, testing with strong specifications detected 55 more—**twice as many**—unique real faults than testing with standard, partial contracts. 62 (56%) of the faults are detected only with strong specifications.

4.5.2 Fault Complexity

Although it is to some extent subjective whether a fault is “deep” or “subtle”, faults violating postconditions or class invariants are arguably more complex because so are the violated properties. While there is no significant difference in the percentage of class invariant violations between EiffelBase and EiffelBase+ (33% in both cases), postconditions trigger 42% of violations in EiffelBase+ but only 11% in EiffelBase: the Wilcoxon signed-rank test among all classes gives $W = 0$ and $p = 6 \cdot 10^{-3}$ both for postconditions alone and for postconditions and class invariants counted together, which demonstrates that strong specifications systematically detect more complex errors. 76% of faults in EiffelBase+ are detected thanks to postconditions or invariants—a direct consequence of the effectiveness of the model-based methodology for writing them.

One example of a fault detected by a model-based postcondition was already discussed in Sect. 4.2. Here we give two other examples to demonstrate that they are indeed subtle yet understandable:

- Routine `ARRAY.force(v, i)` inserts value `v` at position `i` into an array, extending its bounds if necessary. All elements in between the old bound and `i` are supposed to be initialized with default values, however `force` contains an off-by-one error, and in a particular scenario fails to initialize one element. This is missed by the original postcondition `item(i)=v`, which only takes care of the newly inserted element, but detected by the complete model-based postcondition, which, following the methodology, specifies array elements at all positions.

- Both `ARRAYED_SET` and `LINKED_SET` inherit most of their implementation from the corresponding list classes, including the implementation of `is_equal`: the object equality function. As a result, two sets with the same elements in a different order are considered different. The original postcondition only states that equal sets must have the same size and that equality is symmetric, which does not capture the specifics of set equality.

It is revealing that 11 faults in EiffelBase+ are detected due to violations of contracts generated automatically by the RunMBC tool, including 5 observable side effects in functions (classified as errors, since in Eiffel all functions are supposed to be pure). These faults are practically out of the scope of regular contracts, as specifying the corresponding properties explicitly is extremely onerous.

Another interesting observation is that almost 40% of faults can be classified as *inheritance-related*, that is, caused by incorrect reuse or redefinition of inherited features. This confirms our intuition that Eiffel's advanced multiple inheritance mechanisms, together with the size of the EiffelBase class hierarchy, systematically leads to software errors, in the absence of strong contracts acting as safeguards against inconsistencies.

Throughout the whole experiment, we encountered one violation of an invariant that could be later restored before the enclosing public routine call terminates. Strictly speaking, such violation is spurious, and to eliminate it we would have to extend the notation for **`unwrap`** clauses, in order to support opening arbitrary expressions rather than just routine arguments. However, in reality, this particular invariant was *not* restored, so the violation pointed

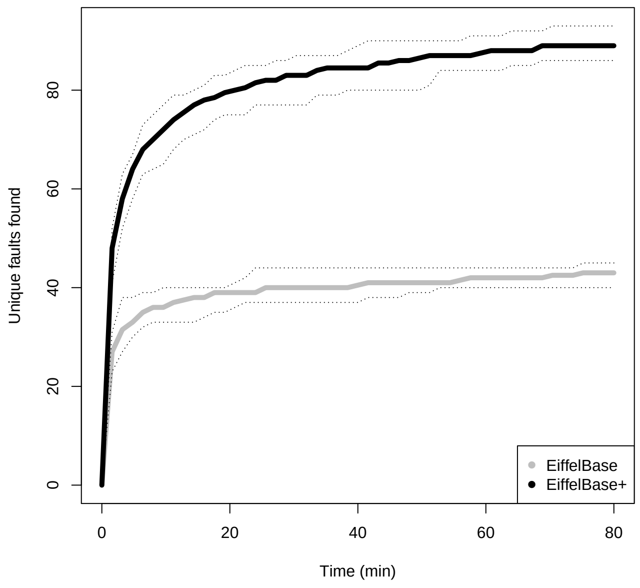


Figure 4.4: Median number of faults, aggregated from all classes, in time. Dotted lines show minimum and maximum for each case.

to a real fault. This example suggests that if an object is too "far away" in the object structure from the call target to be mentioned in the **unwrap** or **depend** clause, it is likely that a developer forgets to restore its invariant anyway, because the object is not in the area of immediate interest for the routine.

4.5.3 Usage of Testing Time

Fig. 4.4 plots the number of faults detected in EiffelBase and EiffelBase+ over a median 80-minute session; it is clear that the behavior with strong specifications dominates over standard contracts after only a few minutes. Dominance is observed consistently in all classes (with the usual exception of ITERATORs and QUEUEs): a median session with strong contracts finds more faults than a median session with standard contracts after a time between two seconds and five minutes depending on the class under test; after a time between 13 seconds and 20 minutes, testing with strong contracts finds more faults than testing with standard contracts will find in the whole session.

Testing with standard contracts also seems to exhaust earlier its fault-finding potential: given any time from 20 minutes on, there are more Eiffel-

Base sessions than EiffelBase+ sessions that have found all the faults they ever will by this time. This may indicate that standard contracts are good to find "quick to detect" faults, but they also soon run out of steam.

We considered other differences between experiments with EiffelBase and with EiffelBase+ in the usage of testing time: repeatability of testing session history, and the presence of *rare* faults triggered only in a small number of cases. Our experiments with strong specifications are slightly less repeatable and include a few more rare faults, but the differences with standard contracts are not statistically significant.

4.5.4 Runtime Performance Overhead

Runtime checking of strong specifications based on models often requires traversing the whole data structure to construct an object of a model class, whenever a contract element is exercised. As a rule, this demands more computational resources than executing the simple checks involved in standard contracts. To measure the runtime overhead of checking model-based specifications in automated testing, we compared the number of test cases generated by AutoTest in the same amount of time when testing EiffelBase and EiffelBase+. Contrary to our expectations, the overhead is small in many cases and not significant overall (see column TC of Tab. 4.1). A possible interpretation of this data is that the overhead of strong specifications grows as larger data structures are instantiated; because random testing most of the time only exercises small data structures, this overhead does not show.

We did not find a significant correlation between the variation of overhead for different classes and any source code metrics we considered. On the other hand, some AutoTest heuristics that decide to discard previously created objects are activated more often for classes where strong specifications are faster to check.

4.5.5 Specification Writing Overhead

Creating EiffelBase+ required roughly one person-month, plus one person-week of preliminary testing for fine-tuning the specification, which puts the overall ratio benefit/effort at about four defects detected per person-day. Tab. 4.4 measures the amount of work produced in this time: for each specification item, including preconditions, postconditions, class invariants, frame properties (**`modify`** clauses), and model query implementations, we compare the number of *tokens* in EiffelBase+ against those in EiffelBase (when applicable) and give the OVERHEAD of strong specifications as the ratio of the two values. The last line also shows the overall specification to code ratios.

Table 4.4: Specification overhead

# TOKENS	EIFFELBASE	EIFFELBASE+	OVERHEAD
Preconditions	1514	1696	1.12
Postconditions	5410	11837	2.19
Invariants	1508	1587	1.05
Frames		1893	
Model queries		2268	
Total	**8432**	**19281**	**2.29**
Spec/code	0.20	0.46	

Reflecting the importance our methodology gives to strong postconditions and the more restricted role of class invariants, 67% of all *new* specification in EiffelBase+ are postconditions, whereas only 9% are class invariants. Frame properties total 11%; they are however straightforward to write and more concise than corresponding semantically equivalent postconditions. Model query implementations account for the remaining 13%.

These numbers suggest that the specification overhead of model-based contracts is moderate and abundantly paid off by the advantages in terms of errors found and quality of documentation. The specification to code ratio also compares favorably to other approaches to improving software quality. Detailed quantitative data about industrial experiences with test-driven development is scarce, but few references indicate [12, 94, 82] that it is common to have between 0.4 and 1.0 lines of tests per line of application code for projects of size comparable to EiffelBase. Correctness proofs are normally much more demanding, as they require between 1.5 and 20 specification elements per implementation element [62, 38, 42, 63].

4.5.6 C# Experiments

Pex found 9 unique faults in DSA+ violating the model-based specification (column F in Tab. 4.2). Unfortunately, we could not get an evaluation of these faults by the original code developers. We have confidence, however, that the faults uncover some obvious errors and, even in the most benign interpretation, some instances of bad object-oriented design.

The fault rates (faults per line of executable code) are comparable in the Eiffel and C# experiments, being respectively $6 \cdot 10^{-3}$ and $3 \cdot 10^{-3}$. The fault complexity is also qualitatively similar for the two languages. The testing time (column T in Tab. 4.2) is instead incomparable, as Pex and AutoTest implement very different testing algorithms.

Writing specifications to create DSA+ required roughly 50 person-hours, plus another 8 person-hours used by the student to learn the specification methodology on small examples. The specification/code ratio is perceptibly higher in DSA+ compared to EiffelBase+ (0.9); this is largely due to the verbose syntax of Code Contracts which are a library, as opposed to Eiffel's native language support for contracts.

4.5.7 EiffelBase2 Experiments

AutoTest found 7 unique faults in EiffelBase2: 2 of them in specification and 5 in the implementation. A comparison with the EiffelBase+ experiments, which revealed 110 unique faults, supports a qualitative claim that the new library contains significantly fewer errors. This is true even after accounting for the size difference: the fault rate of EiffelBase2 is roughly 10^{-3} faults per line of executable code, against $6 \cdot 10^{-3}$ for EiffelBase+.

These results tentatively confirm the effectiveness of model-based contracts as a design methodology. In particular, low number of specification errors shows that getting strong specifications right is not harder than for traditional contracts. Also, only one of the 7 errors is inheritance-related, which suggests that strong specifications in general succeed in "taming" inheritance.

4.5.8 Threats to Validity

Threats to internal validity of our findings come from the usage of randomized testing tools, whose behavior may change in different sessions. We designed the experimental protocol [4] to reduce this threat to a minimum: we ran a large number of repeated experiments and we performed suitable non-parametric statistical tests of significance for all differences we observed.

Threats to external validity refer to the generalizability of our findings. While RunMBC leads to very good results in our experiments, applying it to programs in application domain other than data structures might be more difficult or require an extension of the technique. Our results remain significant, however, if compared to the state of the art in deploying strong specifications. The generalizability to other languages and analysis tools is partially addressed by our experiments targeting two languages (Eiffel and C#) and two automatic testing technologies (random and concolic). Future work will experiment with even more approaches and notations.

4.6 Related Work

This section discusses the most significant related work in three areas: using formal specifications for testing, using inferred specifications to improve testing, and runtime checking of model-based specifications.

4.6.1 Formal specifications for testing.

The idea of using formal specifications for testing has a history that stretches back more than three decades; see [51] for a comprehensive survey. Various proposals targeted different specification formalisms including algebraic datatypes [45, 22], logic-based notations [119], UML Statecharts [97] and other state machines, and contracts and similar forms of embedded assertions [81, 25, 16, 86]. In these applications, formal specifications provide reliable—often automated—testing oracles [118] and can also guide test planning and test case generation.

This extensive experience is evidence that formal specifications can improve the testing process. From a software engineering viewpoint, however, an outstanding open issue is finding optimal trade-offs between the effort required to provide formal specifications and the improvements (in efficiency and effectiveness) they bring to the testing of real software. The evidence—empirical [90] or anecdotal [104]—is scarce in this area: most successful experiences do not explicitly take into account the effort required to produce reliable specifications against the benefits gained for testing (e.g., [37]); or they only target partial specifications, which have the advantage of being easy to write (e.g., [16, 86]). In contrast, the present work targets the high-hanging fruit of deploying strong specifications, explicitly addressing the difficulties of writing and using such specifications for existing software. Our results that strong specifications reveal complex (design) errors corroborate Hoare's view that the real value of tests is that "they detect inadequacy in the [development] methods" [53].

4.6.2 Inferred specifications for testing.

When specifications can be inferred automatically from the code, the deployment effort is negligible compared to the benefits they bring. Therefore, a number of recent works (e.g., [101, 135, 137, 131]) developed sophisticated techniques for inferring specifications from program executions with the intent of using them to improve testing. The experiments reported in these papers show that inferred specifications can boost automated testing [32]; on the other hand, even the most accurate inferred specifications only express the code from a different angle, and hence cannot take the developer's intent fully into account and are necessarily limited to detecting certain types of inconsistencies. Combining inferred and manually written specifications is an interesting endeavor that belongs to future work (see [106, 130] for some preliminary studies).

4.6.3 Model-based specifications at runtime.

Java Modeling Language (JML) [69] contains a wide variety of specification constructs beyond traditional assertions [21], some of them non-executable; to handle this variety, the JML compiler is equipped with multiple advanced runtime checking features. One example is the support for *model variables* [24], which involves generating implementations for model variables defined through constraints, and attaching such implementations to Java interfaces. Neither of those problems arises in RunMBC, since it defines model queries as regular Eiffel functions, and thus can rely on built-in assertion checking mechanisms.

There are several tools targeting automated testing of Java code against JML specifications, e.g. Jartege [100] and JmlUnit [139]; we are not aware of any large-scale empirical studies of these testing techniques, in particular, exploring the benefits of strong specifications involving model classes.

4.7 Summary

This chapter presented an extension of the model-based contracts methodology, which supports more accurate checking of such contracts at runtime. We carried out an extensive empirical evaluation, using software written in Eiffel and C#, to determine the benefits of using strong specifications in testing with automatic tools. We found twice as many bugs in the software with strong specifications as in the same software specified with standard partial contracts. At the same time, the effort required to write strong specifications for testing is moderate, since they are limited to interface properties and do not include extensive auxiliary annotations used by static techniques. We have also provided evidence that software developed with strong specifications from the start exhibits considerably fewer defects than software developed in a traditional way.

Another contribution of this chapter is the RunMBC tool, which translates advanced model-based specifications into regular Eiffel contracts, amenable to runtime checking. To our knowledge, this is the first tool that uses ideas from the Spec# verification methodology [9] to enable runtime checking of class invariants in the presence of callbacks and inter-object dependencies.

Chapter 5

Flexible Invariants for Complex Object Structures

The previous chapter focused on quality assurance by means of automated testing. Although testing against strong specifications can reveal many software faults, it can never guarantee their absence. In this and the next chapter, we target correctness proofs: a verification technique that is more demanding but ensures software correctness with mathematical certainty. The current chapter, in particular, addresses one the central issues in the area of correctness proofs of object-oriented programs: reasoning about class invariants.

5.1 Introduction

Class invariants are here to stay [121]—even with their tricky semantics in the presence of callbacks and inter-object dependencies, which make reasoning so challenging [102]. The main reason behind their widespread adoption is that they formalize the notion of a *consistent* class instance, which is inherent in object-oriented programming, and thus naturally present when reasoning, even informally, about program behavior.

The distinguishing characteristic of invariant-based reasoning is *stability*: it should be impossible for an operation r to violate the invariant of an object o without modifying o itself. Stability promotes information hiding and simplifies client reasoning about preservation of consistency: without invariants, a client would need to know which other objects o's consistency depends on, while with invariants it is sufficient that it check whether r modifies o—a piece of information normally available as part of r's frame specification. The goal of an *invariant methodology* (also called *protocol*) is thus to achieve stability even in the presence of inter-object dependencies—where the consistency of o depends on the state of other objects, possibly recursively or in a circular fashion (see Sect. 5.2 for concrete examples).

The numerous methodologies introduced over the last decade, which we review in Sect. 5.3, successfully relieve several difficulties involved in reasoning with invariants; but there is still room for improvement in terms of flexibility, usability, and automated tool support. In this chapter, we present *semantic collaboration* (SC): a new methodology for specifying and reasoning about invariants in the presence of inter-object dependencies that combines flexibility and usability, and is implemented in a program verifier.

A standard approach to inter-object invariants is based on the notion of *ownership*, which has been deployed successfully in several invariant methodologies [9, 77, 92] and is available in tools such as Spec# [10] and VCC [29]. Under this model, the invariant of an object o only depends on the state of the objects explicitly owned by o. Ownership is congenial to object-orientation because it supports a strong notion of encapsulation; however, not all inter-object relationships are hierarchical and hence reducible to ownership. Multiple objects may also *collaborate* as equals, mindful of each other's consistency; a prototypical example is the Observer pattern [44] (see Sect. 5.2).

Semantic collaboration naturally complements ownership to accommodate invariant patterns involving collaborating objects. Most existing methodologies support collaboration through dedicated specification constructs and syntactic restrictions on invariants [77, 11, 88, 120]; such disciplines tend to work only for certain classes of problems. In contrast, SC relies on standard specification constructs—ghost state and invariants—to keep track of inter-object dependencies, and imposes *semantic* conditions on class invariant representations. It builds upon the philosophy of *locally-checked invariants* (LCI) [30]: a low-level verification method based on two-state invariants. LCI has served as a basis for other specialized, user- and automation-friendly methodologies for ownership and shared-memory concurrency. SC can be viewed as an improved specialization of LCI for object collaboration in a sequential setting. To further improve usability, SC comprises useful "defaults", which characterize typical specification patterns. As we argue in Sect. 5.5 based on several challenge problems, the defaults significantly reduce the annotation burden without sacrificing flexibility in the general case.

We implemented SC as part of AutoProof [5], an automated verifier for Eiffel. The implementation provides more concrete evidence of the advantages of SC compared to other methodologies to specify collaborating objects (e.g., [11, 80, 120, 88] all of which currently lack tool support).

The presentation of this chapter is based on examples of non-hierarchical object structures, customarily used in the literature. Sect. 5.2 presents the examples and the challenges they embody; and Sect. 5.3 discusses the approaches taken by main existing invariant methodologies. Sect. 5.4 introduces SC, demonstrates its application to the running examples, and outlines a soundness proof. Sect. 5.5 evaluates both SC and existing protocols on an extended set of examples, including challenge problems from the SAVCBS workshop series [109]. The evaluation demonstrates that SC is the only methodology that supports *(a)* collaboration with unknown classes, while preserving stability, and *(b)* invariants depending on unbounded sets of objects, possibly unreachable in the heap. The collection of problems of Sect. 5.5—available at [110] together with our solutions—could serve as a benchmark to evaluate invariant methodologies for non-hierarchical object structures. The website [110] also gives access to AutoProof through a web interface.

5.2 Motivating Examples

The *Observer* and *Iterator* design patterns are widely used programming idioms [44], where multiple objects depend on one another and need to maintain a global invariant. Their interaction schemes epitomize cases of inter-object dependencies that ownership cannot easily describe; therefore, we use them as illustrative examples throughout the chapter, following in the footsteps of much related work [80, 102, 88].

5.2.1 Observer pattern

Fig. 5.1 shows the essential parts of an implementation of the Observer design pattern in Eiffel. An arbitrary number of `OBSERVER` objects (called "subscribers") monitor the public state of a single instance of class `SUBJECT`. Each subscriber maintains a copy of the subject's relevant state (integer attribute `value`) into one of its local variables (attribute `cache`). The subscribers' copies are cached values that must be consistent with the state of the subject, formalized as the invariant clause `cache = subject.value` of class `OBSERVER`, which depends on another object's state. This dependency is not adequately captured by ownership schemes, since no one subscriber can have exclusive control over the subject.

In the Observer pattern, consistency is maintained by means of explicit collaboration: the subject has a list of `subscribers`, updated whenever a new subscriber registers itself by calling `register (Current)` on the subject. Upon every change to its state (routine `update`), the subject takes care of explic-

```
class SUBJECT

  value: INTEGER
  subscribers: LIST [OBSERVER]

  make (v: INTEGER) -- Constructor
    do
      value := v
      create subscribers
    ensure
      subscribers.is_empty
    end

  update (v: INTEGER)
    do
      value := v
      across subscribers as o do
        o.notify
      end
    ensure
      value = v
    end

feature {OBSERVER}
  register (o: OBSERVER)
    require
      not subscribers.has (o)
    do
      subscribers.add (o)
    ensure
      subscribers.has (o)
    end
end
```

```
class OBSERVER

  subject: SUBJECT
  cache: INTEGER

  make (s: SUBJECT) -- Constructor
    do
      subject := s
      s.register (Current)
      cache := s.value
    ensure
      subject = s
    end

feature {SUBJECT}
  notify
    do
      cache := subject.value
    ensure
      subject = old subject
      cache = subject.value
    end

invariant
  subject.subscribers.has (Current)
  cache = subject.value
end
```

Figure 5.1: The *Observer pattern*: an observer's `invariant` depends on the state of the `SUBJECT`, which reports its state changes to all its `subscribers`. The clients of the subscribers must be able to rely on their `cache` always being consistent, while oblivious of the update/notify mechanisms that preserve invariants.

itly notifying all registered subscribers (using an `across` loop that calls `notify` on every `o` in `subscribers`). This explicit collaboration scheme—called "considerate programming" in [120]—ensures that the subscribers' state remains consistent (i.e., the class invariant holds) between calls to the public routines of the object structure.

Fig. 5.1 uses Eiffel's *selective exports*[1] to separate the public interface of the classes from the features internal to the object structure: **feature** {OBSERVER} denotes that routine register is only available to instances of class OBSERVER, and **feature** {SUBJECT} similarly limits the visibility of notify to the subject. While selective exports help emphasize collaboration patterns, they are not crucial for the proposed methodology, which is applicable to any object-oriented language regardless of the available visibility specifiers.

A methodology to verify the Observer pattern must ensure invariant stability; namely, that clients of OBSERVER can rely on its invariant without knowledge of the register/notify mechanism. Another challenge is dealing with the fact that the number of subscribers attached to the subject is not fixed *a priori*, and hence we cannot produce explicit syntactic enumerations of the subscribers' cache attributes. We must also be able to verify update and notify without relying on the class invariant as precondition—in fact, those routines are called on inconsistent objects precisely to restore consistency.

5.2.2 Iterator Pattern

In the Iterator pattern, an arbitrary number of iterator objects traverse a collection of elements. Fig. 5.2 sketches an implementation where the COLLECTION uses an ARRAY of elements as underlying representation. The ITERATOR's main capability is to return the item at the current position index in the target collection[2]. item's precondition specifies that this is possible only when the iterator points to a valid element of target, that is index is between 1 and target.count (included); otherwise, if index is 0 the iterator is before the list, and if it equals target.count + 1 it is after the list. The invariant of class ITERATOR defines the public state components before and after in terms of the internal state component index, as well as the acceptable variability range for index.

Since the iterator's invariant depends on the state of the target collection, modifying the collection (for example, by calling remove_last) may *disable* the iterator (make it inconsistent). This is aligned with the intended usage of iterators, which should be discarded after traversing a collection without changing it. A verification methodology should ensure that clients of ITERATOR only access iterators in a consistent state, without knowledge of the iterator's internal state index or of its relation to the target collection. In fact, the selective exports used in Fig. 5.2 hide the details of ITERATOR's

[1]Similar to friend classes in C++.

[2]We omit the description of other necessary operations, such as advancing the iterator, since they are irrelevant for our discussion about invariants.

```
class COLLECTION [G]

  count: INTEGER

  make (capacity: INTEGER)
    -- Constructor
    require
      capacity ≥ 0
    do
      create elements(1, capacity)
    ensure
      elements.count = capacity
      count = 0
    end

  remove_last
    require
      count > 0
    do
      count := count − 1
    ensure
      count = old count − 1
    end

feature {ITERATOR}
  elements: ARRAY [G]

invariant
  0 ≤ count ≤ elements.count
end
```

```
class ITERATOR [G]

  target: COLLECTION [G]
  before, after: BOOLEAN

  make (t: COLLECTION)
    -- Constructor
    do
      target := t
      before := True
    ensure
      target = t
      before and not after
    end

  item: G
    require
      not (before or after)
    do
      Result := target.elements [index]
    end

feature {NONE}
  index: INTEGER

invariant
  0 ≤ index ≤ target.count + 1
  before = index < 1
  after = index > target.count
end
```

Figure 5.2: The *Iterator pattern*: an iterator's invariant depends on the state of the collection it traverses, which is oblivious of the iterators. Verification must prove that clients do not access disabled iterators, without knowing collection's and iterator's internal states.

invariant from its clients (the visibility of an invariant clause is determined by its least visible subexpression, and **feature** {NONE} denotes purely private members). An additional obstacle to verification comes from the fact that considerate programming would be at odds with the ephemeral nature of iterators compared to observers: collections are normally implemented unaware of the iterators operating on them; a flexible invariant methodology should allow such implementations.

5.3 Existing Approaches

This section reviews the main existing methodologies for specifying and reasoning about class invariants; based on their most important features and limitations. Sect. 5.4 will present our own methodology. Since we are interested in verifying reusable components, we only discuss methodologies that support modular reasoning (where local checks on individual classes or small groups of classes subsume global program correctness).

A crucial issue is deciding *when* (at which program points) class invariants should hold: state-changing operations normally consist of sequences of elementary updates, which individually may break the class invariant temporarily. For example, in Fig. 5.1 the invariants of `subscribers` might be broken after the first instruction of `update`. To deal with this problem, some methodologies restrict the program points where class invariants are expected to hold; others interpret the invariants in a weakened form, which holds vacuously at intermediate steps during updates (and fully at crucial points).

Methodologies based on **visible-state semantics** only require invariants to hold when no operation is being executed on their objects, that is in states visible to clients. This idea was introduced for Eiffel [84], and later also adopted by JML [67]. Without additional mechanisms, visible-state semantics can't achieve modularity in the presence of callbacks (the client making the callback is unaware of ongoing operations that may affect the invariant) and of inter-object dependencies (if o_1's invariant depends on o_2, the former is also affected by operations on o_2 invisible to clients of o_1). Existing solutions adopt aliasing control measures [92] to deal with hierarchical object structures described by ownership. Other solutions [85, 87, 88, 120], for collaborative invariants, either infer through static analysis or require the developer to explicitly indicate which objects might be inconsistent at routine call boundaries; for example, routine `register (o: OBSERVER)` of class `SUBJECT` in Fig. 5.1 would be annotated with `broken o` to specify that argument `o`'s invariant may not hold when executing `register`.

These two families of solutions—for hierarchical and for collaborative object structures—based on visible-state semantics are not easily combined; this is a practical limitation, since many object-oriented systems consist of an interplay between both types of structure. For example, continuing with Fig. 5.1, objects of class `SUBJECT` collaborate with `OBSERVER` objects but also own a `subscribers` list as part of their representation. Thus, when reasoning about routine `register`, we should be able to deal with the call `subscribers.add (o)` whose argument `o` is inconsistent (and hence `add` cannot assume `o`'s invariant); however, annotating `LIST`'s `add` by declaring its argument `broken` goes against modularity, as class `LIST` should not need to know how and where it is used. The difficulty of integrating hierarchical and collaborative models is the main limitation of visible-state methodologies, and likely a reason why, to our knowledge, they have not been implemented in any program verifier.

Another family of methodologies, collectively known as **Spec# methodologies** after the program verifier where they have originally been implemented, follow the approach of weakening the default semantics of invariants so that they can be evaluated only when appropriate. In a nutshell, all classes include a ghost Boolean attribute closed,[3] which denotes whether an object is in a consistent state; an invariant inv is then interpreted as the weaker closed⇒inv, which vacuously holds for open (i.e., not closed) objects. Routines explicitly indicate whether they expect relevant objects to be closed or open; this approach is more conducive to modularity than visible-state semantics: it does not impose consistency by default at routine call boundaries and thus does not require routines to list *all* possibly inconsistent objects in the entire program.

The original methodologies from this family, as implemented in the Spec# system [10], are mainly based on *syntactic* mechanisms to express ownership relations. For example, following [9], we would annotate attribute elements of class COLLECTION in Fig. 5.2 with rep, to denote that it belongs to COLLECTION's internal representation; thus, modifying elements is only possible if its COLLECTION owner has been opened—a situation where closed⇒count ≤ elements.count vacuously holds. This solution only supports representations based on bounded sets of objects directly accessible through attributes. Follow-up work [77] partially relaxes this restriction introducing a form of quantification predicating over an owner ghost attribute (which goes up the ownership hierarchy), and a mechanism to transfer ownership. The additional expressiveness comes with a price to pay mainly in terms of complex invariant admissibility conditions (hence, it may be hard to understand what is expressible and how) and complicated soundness proofs of the methodology.

In contrast, the VCC verifier [29] implements a Spec#-style methodology where ownership is encoded on top of LCI's *semantic* approach [30]. Objects include an additional ghost attribute, owns, storing the set of all owned objects; ghost code modifies this set explicitly when the owner object is open. In the example of Fig. 5.2, instead of annotating attribute elements with rep, we would introduce a first-order formula, such as owns = {elements}, in the invariant of COLLECTION to express that elements is part of the representation. The advantage of this approach becomes apparent with linked structures, where owned elements are accessible only by following chains of references

[3]We follow VCC's terminology [29] whenever applicable; other works use different names.

(e.g., a linked list owns all reachable cells). In fact, semantic approaches to ownership provide the flexibility necessary to specify an unbounded number of owned objects, which may even be not directly attached to the owner, as well as to implement ownership transfers without need for ad hoc mechanisms. They also simplify the rules of reasoning; for example, invariant admissibility becomes a simple proof obligation that all objects whose state is mentioned in the invariant are bound, by the same invariant, to belong to `owns`. These features have contributed to making VCC applicable to real-world systems [71].

In addition to ownership, some Spec# methodologies also deal with collaborating objects. [77] introduces the notion of *visibility-based* invariants, which requires that a class be aware of the types and invariants of all objects concerned with its state[4]. For example, in Fig. 5.1 `SUBJECT` must declare its `value` attribute with a modifier `dependent OBSERVER`. Whenever the subject changes its `value`, it has to check that all potentially affected `OBSERVER`s are open. If aware of the `OBSERVER`'s invariant, it can show that the only affected observers are $\{o\colon$ `OBSERVER` $\mid o.$`subject` $=$ **`Current`**$\}$. Such indirect representations of the concerned objects complicate discharging the corresponding proof obligations; and relying on knowing the concerned objects' invariants introduces tight coupling between the collaborating classes.

To lift these complications, [11] suggests instead to introduce a ghost attribute `deps` storing the set of all concerned objects. It also introduces *update guards*, allowing a concerned object to state conditions under which its invariant is preserved without revealing the invariant itself. Both approaches [77, 11] have shortcomings that derive from their reliance on syntactic mechanisms and conditions: collaboration invariants can only depend on a bounded number of objects known *a priori* and accessible through attributes (called "pivot fields" in [11]); the types of the concerned objects must be known explicitly; and the numerous ad hoc annotations (e.g., `friend` and `keeping`) and operations (e.g., to modify `deps`) make the methodologies harder to present and use. One of the main goals of our methodology (Sect. 5.4) is to lift these shortcomings by dealing with collaborative invariants by *semantic* rather than syntactic means—similarly to what VCC did to the classic syntactic treatment of ownership. The semantic approach makes SC very flexible, capable of accommodating disparate object-oriented design patterns without requiring ad hoc mechanisms.

Somewhat orthogonally to other Spec#-family approaches, the *history invariants* methodology [80] provides for more loose coupling between the

[4]We say that an object o is *concerned* with an attribute a of another object s if updating $s.a$ might affect o's invariant.

collaborating classes, but gives up stability of invariants.

5.4 Semantic Collaboration

Our new invariant methodology belongs to the Spec# family; as we illustrated in Sect. 5.3, this entails that objects can be *open* or *closed*, and class invariants have to hold only for closed objects. On top of semantic mechanisms for ownership, similar to those developed for VCC (see Sect. 5.3), our methodology also provides a semantic treatment of dependencies among collaborating objects; hence its name *semantic collaboration*. The keywords and constructs specific to SC are `underlined` in the following.

Overview of semantic collaboration. To specify collaboration patterns, we equip every object `o` with ghost fields `subjects` and `observers`. As their names suggest,[5] `o.subjects` stores the set of objects on which `o`'s invariant might depend; conversely, `o.observers` (analogous to `deps` in [11]) stores the set of objects whose invariant might depend on `o`, in other words, objects potentially *concerned* with `o`.

The methodology achieves modularity by reducing global validity (all closed objects satisfy their invariants) to local checks of two kinds: *(i)* all concerned objects are stored in `observers`; and *(ii)* updates to the attributes of an object `o` maintain the validity of `o` and its observers. Check *(i)* becomes an admissibility condition that every declared class invariant must satisfy. Check *(ii)* holds vacuously for open observers, thus one way to satisfy it is to "notify" all observers of a potentially destructive update by opening them. For more flexibility, the methodology also allows subjects to skip "notifying" observers whenever the attribute update satisfies its *guard* (a notion also inspired by [11]). This option is supported by another admissibility condition: an invariant must remain valid after updates to subjects that comply with their update guards.

5.4.1 Preliminaries and Definitions

As it is customary, the following presentation targets fundamental programming constructs, while ignoring those that do not affect reasoning about invariants (e.g., control structures). We also largely ignore issues related to inheritance, but we briefly come back to them in Sect. 5.6.

[5]While the names are inspired by the Observer pattern, they are also applicable to other collaboration patterns, as we demonstrate in Sect. 5.4.4. The `underlining` in the formatting avoids confusion.

Our methodology assumes a generic object-oriented programming language, where a program is a collection of classes, and a class is a collection of attributes, routines, and side-effect free *logic functions*.[6] Any of those constructs can be declared *ghost* if it is meant to be used only in specifications.

A pair $x\,\mathbf{.}\,a$ of an object identifier x and an attribute a is called a *location*. A set of locations is called a *frame*.

Built-in attributes. Every class is implicitly equipped with ghost attributes: `closed` (to encode consistency); `owns` and `owner` (to encode the ownership hierarchy); and `subjects` and `observers` (to encode collaboration). We also define the shorthands: `o.open` for ¬`o.closed`; `o.free` for `o.owner` = **`Void`**; and `o.wrapped` for `o.closed` ∧ `o.free`. The *ownership domain* of an object `o` is {`o`} if `o` is open, and the reflexive transitive closure of `o.owns` if `o` is closed. Attributes `closed` and `owner` are only changed indirectly through the implicitly defined ghost routines `wrap` and `unwrap`, whose semantics is defined below.

Specifications. The specification of a *logic function* `f` consists of a *definition*—a side-effect free expression defining the function value—and zero or more **`read`** clauses of the form **`read`** $[a_1, \dots, a_n]\ x_1, \dots, x_m$, where each x_i is an expression that denotes either a set of objects or an individual object (interpreted as a singleton set), and each a_j is an attribute name. Such a clause denotes a frame $\{x\,\mathbf{.}\,a \mid a \in \{a_1, \dots, a_n\}, x \in \bigcup_{i \in 1 \dots m} x_i\}$ (if the attribute list is omitted, all attributes of the listed objects are included in the frame); the union of frames for all **`read`** clauses gives the *read frame* of `f`: the set of locations that `f` may depend on.

The specification of a *routine* `r` consists of a **`require`** clause (a precondition), an **`ensure`** clause (a postcondition), and zero or more **`modify`** clauses of the form **`modify`** $[a_1, \dots, a_n]\ x_1, \dots, x_m$. Such a clause denotes a frame, defined analogously to the **`read`** clause, except it also includes the union of the ownership domains of all listed objects; the union of frames for all **`modify`** clauses gives the *write frame* of `r`: the set of locations that `r` may modify.

The specification of an *attribute* `a` consists of an *update* `guard`—a Boolean expression over the **`Current`** object, new attribute value `y`, and generic observer object `o`, written guard(**`Current`**.a := y, o).

The specification of a *class* includes its invariant `inv`.

The choice to implicitly include ownership domains into the write frame of a routine is a key abstraction mechanism in ownership-based verification methodologies: clients of a routine `r` should not be able to distinguish between `r` modifying `x` and modifying an object in the internal representation of

[6]Inspired by the `function` construct of Dafny [74], and `def` construct of VCC. Logic functions need not be a dedicated language construct, and can instead be expressed with pure routines; these details are irrelevant for the present discussion.

x, since that would break information hiding. This specification convention can also be used to avoid explicit usage of sets in frame specifications—an approach promoted by the *dynamic frames* family of techniques [60, 117, 74]—and restrict the syntax of **modify** clauses to (finite) lists of objects. The framing notation we adopted as the basis of SC allows arbitrary set expressions, and thus fully supports the dynamic frames style, at the same time, making use of information hiding mechanisms offered by ownership. Sect. 5.4.7 discusses implications of adopting a similar convention for **read** clauses of logic functions.

Expressions. We consider standard programming language expressions, extended with bounded quantification, exemplified by Eiffel's loop expressions **all** x ∈ s : B(x) and **some** x ∈ s : B(x) (see Chapter 3). We treat **Void** as an object that is always allocated and open.

Expressions are evaluated in a *heap*, which maps locations to values. The current heap H is normally clear from the context and left implicit. Otherwise, e_h denotes the value of expression e in heap h; and $h[\texttt{x.a} \mapsto e]$ denotes the heap that agrees with h everywhere except possibly about the value of x.a, which is e. Since we ignore deallocation, our heaps have no dangling references: only allocated objects are reachable from allocated objects.

The *read frame* reads(e) of a primitive expression is defined as follows: for an access x.a to attribute a, reads(x.a) = {x ▪ a}; for an application x.f (y) of logic function f, reads(x.f (y)) is given by f's read frame after argument substitution. The read frame of a compound expression e is the union of the read frames of e's subexpressions.

Instructions. For the present discussion, we only have to consider routine calls x.r (y), as well as *heap update instructions*: **create** x (allocate an object and attach it to x); x.a := y (update attribute a); and x.wrap and x.unwrap (opening and closing an object).

The *write frame* writes(s) of a primitive instruction s is defined as follows: for an update x.a := y of attribute a, writes(x.a := y) = {x ▪ a}; for opening or closing an object x, writes(x.unwrap) = writes(x.wrap) = {x ▪ closed} ∪ {o ▪ owner | o ∈ x.owns}; for a call x.r (y) to routine r, writes(x.r (y)) is given by r's write frame after argument substitution. The write frame of a compound instruction is the union of write frames of its sub-instructions.

5.4.2 Semantic Collaboration: Goals and Proof Obligations

The **goal** of any invariant methodology is to provide *modular* proof obligations to establish *global* validity: the property that every object in the program is *valid* at every program point. Following SC's approach, an object is valid if it satisfies its invariant when closed; thus global validity is defined as:

$$\forall o : o.\underline{\texttt{closed}} \Rightarrow o.\texttt{inv} \tag{G1}$$

Additionally, maintaining ownership-based invariants requires strengthening global validity with the property that whenever a parent object p is closed, all its owned objects are closed (and their $\underline{\texttt{owner}}$ attributes point back to p):

$$\forall o, p : p.\underline{\texttt{closed}} \wedge o \in p.\underline{\texttt{owns}} \Rightarrow o.\underline{\texttt{closed}} \wedge o.\underline{\texttt{owner}} = p \tag{G2}$$

Proof obligations. The proof obligations specific to SC consist of two types of checks: *(i)* every class invariant is *admissible* according to Definition 5.1; and *(ii)* every heap update instruction satisfies its precondition. These proof obligations are *modular* in that they only mention the state of the current object, its observers and owned objects. Sect. 5.4.3 describes how establishing the local proof obligations entails global validity, that is, subsumes checking (G1) and (G2).

Admissibility captures the requirements that class invariants respect ownership and collaboration relations, as well as update guards.

Definition 5.1. An invariant inv is *admissible* iff:

1. inv only depends on attributes of **Current**, its owned objects, and its subjects (except built-in attributes $\underline{\texttt{closed}}$ and $\underline{\texttt{owner}}$):

$$\texttt{inv} \Rightarrow \mathsf{reads}(\texttt{inv}) \subseteq \{x\,.\,a \mid x \in \{\texttt{Current}\} \cup \underline{\texttt{owns}} \cup \underline{\texttt{subjects}} \quad \wedge \quad a \neq \underline{\texttt{closed}}, \underline{\texttt{owner}}\} \tag{A1}$$

2. All subjects of **Current** are aware of it as an observer:

$$\texttt{inv} \Rightarrow \forall s : s \in \underline{\texttt{subjects}} \Rightarrow \texttt{Current} \in s.\underline{\texttt{observers}} \tag{A2}$$

3. inv is preserved by any update s.a := y that conforms to its guard:

$$\forall s, a, y : s \in \underline{\texttt{subjects}} \wedge \texttt{inv} \wedge \mathsf{guard}(s.a := y, \texttt{Current}) \Rightarrow \texttt{inv}_{\mathsf{H}[s.a \mapsto y]} \tag{A3}$$

The specifications of the heap update instructions are given below; the instructions only modify objects and attributes mentioned in the postconditions.

Allocation creates a free open object, with no observers:

```
create x        require           ensure
                True              x.open
                                  x.owner = Void
                                  x.observers = {}
```

Unwrapping opens a wrapped object and frees all objects in its `owns` set:

`x.unwrap`	**`require`**	**`ensure`**
	`x.wrapped`	`x.open`
		`all o ∈ x.owns : o.wrapped`

Attribute update operates on an open object and preserves validity of its observers:

`x.a := y`	**`require`**	**`ensure`**
	`a ≠ closed`	`x.a = y`
	`x.open`	
	`all o ∈ x.observers : o.open or` guard(`x.a := y, o`)	

Wrapping closes a consistent open object and gives it ownership over all objects in its `owns` set:

`x.wrap`	**`require`**	**`ensure`**
	`x.open`	`x.wrapped`
	`x.inv`	`all o ∈ x.owns : o.owner = x`
	`all o ∈ x.owns : o.wrapped`	

Other proof obligations. Proof obligations unrelated to invariants are the usual ones of axiomatic reasoning: every call to a routine `r` occurs in a state that satisfies `r`'s precondition; executing a routine `r` in a state that satisfies its precondition leads to a state that satisfies `r`'s postcondition; the **`read`** clause of every logic function `f` is consistent (i.e., the read frame of `f`'s definition is a subset of `f`'s **`read`** clause); the **`modify`** clause of every routine `r` is consistent (i.e., the write frame of `r`'s body is a subset of `r`'s **`modify`** clause, closed under ownership domains); and the definitions of logic functions are terminating.

5.4.3 Soundness of the Methodology

To establish soundness of the proposed invariant methodology we have to show that every program satisfying the proof obligations of SC is always globally valid, that is satisfies (G1) and (G2). We outline a proof of this fact in three parts, of which only the third is specific to SC, while the first two can be reused for other invariant protocols.

The first part concerns ownership: every methodology that, like SC, imposes a suitable discipline of wrapping and unwrapping to manage ownership domains reduces (G2) to local checks.

Lemma 5.1. Consider a methodology M whose proof obligations verify the following:

a. freshly allocated objects are $\underline{\texttt{open}}$;

b. whenever $\texttt{x.}\underline{\texttt{owner}}$ is updated or $\texttt{x.}\underline{\texttt{closed}}$ is set to **False**, $\texttt{x.}\underline{\texttt{owner}}$ is $\underline{\texttt{open}}$;

c. whenever $\texttt{x.}\underline{\texttt{closed}}$ is set to **True**, every object `o` in $\texttt{x.}\underline{\texttt{owns}}$ is $\underline{\texttt{closed}}$ and satisfies $\texttt{o.}\underline{\texttt{owner}} = \texttt{x}$;

d. whenever an attribute `x.a` (with $\texttt{a} \notin \{\underline{\texttt{closed}}, \underline{\texttt{owner}}\}$) is updated, object `x` is $\underline{\texttt{open}}$.

Then every program that satisfies M's proof obligations also satisfies (G2) everywhere.

Proof. The proof is by induction on the length of program traces.

The base case is the trace only consisting of the initial heap where no object is allocated but for an open object **Void**; thus (G2) holds initially. For the inductive step, let h be the final heap of a trace where (G2) invariably holds. Consider a heap update instruction s that yields heap h' if executed on h, and assume that h' does not satisfy (G2):

$$\exists o, p \in alloc(h') : p.\underline{\texttt{closed}}_{h'} \wedge o \in p.\underline{\texttt{owns}}_{h'} \wedge (\neg o.\underline{\texttt{closed}}_{h'} \vee o.\underline{\texttt{owner}}_{h'} \neq p)$$

(where $alloc$ denotes the set of objects allocated in a heap). Since h' differs from h in exactly one location, or in allocation status of exactly one object, s must be one of the following:

- **create** o: in this case $o \notin alloc(h) \wedge o \in p.\underline{\texttt{owns}}_h$ which is a contradiction, since there are no dangling references.

- **create** p: here $p \notin alloc(h) \wedge p.\underline{\texttt{closed}}_{h'}$, which contradicts rule a.

- $p.\underline{\texttt{owns}} := \ldots$: in this case $p.\underline{\texttt{closed}}_h \wedge o \notin p.\underline{\texttt{owns}}_h \wedge o \in p.\underline{\texttt{owns}}_{h'}$, which contradicts the requirement that of rule d that p be open.

- $o.\underline{\texttt{owner}} := \ldots$: here $p.\underline{\texttt{closed}}_h \wedge o.\underline{\texttt{owner}}_h = p \wedge o.\underline{\texttt{owner}}_{h'} \neq p$, which contradicts the requirement that of rule b that $o.\underline{\texttt{owner}}_h$ be open.

- $x.\underline{\texttt{closed}} := \ldots$ (for some x) could validate the first or the third conjunct in the above formula, or even both if $p = o$. The latter is impossible since that implies $\neg p.\underline{\texttt{closed}}_h \wedge o.\underline{\texttt{closed}}_h$. If the first disjunct is validated, then $\neg p.\underline{\texttt{closed}}_h \wedge p.\underline{\texttt{closed}}_{h'} \wedge o \in p.\underline{\texttt{owns}}_h \wedge (\neg o.\underline{\texttt{closed}}_h \vee o.\underline{\texttt{owner}}_h \neq p)$, which contradicts rule c. Finally, if the third disjunct is validated, then $p.\underline{\texttt{closed}}_h \wedge o.\underline{\texttt{closed}}_h \wedge o.\underline{\texttt{owner}}_h = p \wedge \neg o.\underline{\texttt{closed}}_{h'}$, which contradicts the requirement that of rule b that $o.\underline{\texttt{owner}}_h$ be open. □

The second part applies to any kind of inter-object invariants and assumes a methodology that, like SC, checks that attribute updates preserve validity of all concerned objects; we show that such checks subsume (G1). How a methodology identifies concerned objects is left unspecified as yet.

Lemma 5.2. Consider a methodology M whose proof obligations verify the following:

a. freshly allocated objects are `open`;

b. whenever `x.closed` is updated to **`True`**, `x.inv` holds;

c. whenever an attribute `x.a` (with $\texttt{a} \neq \texttt{closed}$) is updated to some `y`, every concerned object satisfies $(\texttt{o.closed} \wedge \texttt{o.inv}) \Rightarrow \texttt{o.inv}_{\texttt{H[x.a}\mapsto\texttt{y]}}$;

d. class invariants depend neither on attribute `closed` nor on the allocation status of objects.

Then every program that satisfies M's proof obligations also satisfies (G1) everywhere.

Proof. The proof is by induction on the length of program traces.

The base case is the trace only consisting of the initial heap where no object is allocated but for an open object **`Void`**; thus (G1) holds initially. For the inductive step, let h be the final heap of a trace where (G1) invariably holds. Consider an instruction s that yields heap h' if executed on h. Without loss of generality, let $h' \neq h$; therefore, s is either an allocation of a new object or an attribute update. If s allocates a new object `x`, (G1) still holds in h': `x` is open (rule a) and no other object's invariants depends on it, since `x` has just been created and class invariant do not know about allocation status (rule d). If s sets to **`False`** some o.`closed` in (G1)'s antecedent, then (G1) vacuously hold. If s sets to **`True`** some o.`closed` in (G1)'s antecedent, then o.`inv` holds (rule b); thus (G1) holds too. Also, updates to some o.`closed` cannot concern the invariants of objects other than o (rule d). If s updates some `x.a`, with $\texttt{a} \neq \texttt{closed}$, let o be any object concerned with the update; either o is open, or it is closed and o.`inv` holds in h by the induction hypothesis, so rule c applies. Either way, (G1) holds in h' for o. □

The third part of the soundness proof argues that SC satisfies the hypotheses of Lemmas 5.1 and 5.2, and hence ensures global validity.

Theorem 5.1. Every program that satisfies the proof obligations of SC also satisfies (G2) and (G1) everywhere.

Proof. SC satisfies the hypotheses of Lemma 5.1: allocation satisfies rule a; unwrapping satisfies rule b and wrapping satisfies rules b and c (we assume that `unwrap x` first updates `x.closed`, and then sets the `owner` attribute of every object in `x.owns`; `wrap` performs the updates in the reverse order). Remember that `closed` and `owner` are only changed by `wrap` and `unwrap`. Attribute update satisfies rule d.

It also satisfies the hypotheses of Lemma 5.2: allocation satisfies rule a; wrapping satisfies rule b; invariant admissibility and the rules of language syntax satisfy rule d. Rule c requires more details. First note that invariant admissibility requires that no invariant mention `owner`; thus no object is concerned with wrapping or unwrapping, which therefore vacuously satisfy rule c. Now, consider an update $\texttt{x.a} := \texttt{y}$ with $\texttt{a} \neq \texttt{owner}$ and $\texttt{a} \neq \texttt{closed}$, and let `o` be any concerned object. Assuming `o.closed` and `o.inv` hold for a generic heap h, we have to show that `o.inv` also holds of the heap $h' = h[\texttt{x.a} \mapsto \texttt{y}]$. By definition of read frame, $\texttt{x} \in \mathsf{reads}(\texttt{o.inv})$; `o.inv` is also admissible and hence it satisfies (A1); therefore $\texttt{x} \in \{\texttt{o}\} \cup \texttt{o.owns} \cup \texttt{o.subjects}$. However, the second precondition of the attribute update rule says that `x` is open; thus $\texttt{x} \neq \texttt{o}$ because `o` is closed. We already proved that h satisfies (G2); for $p = \texttt{o}$ this entails that all objects in `o.owns` are closed; therefore, $\texttt{x} \notin \texttt{o.owns}$ as well. We conclude that $\texttt{x} \in \texttt{o.subjects}$ which, combined with condition (A2) for `o.inv`'s admissibility, implies that $\texttt{o} \in \texttt{x.observers}$ holds in h. Finally, the third precondition of the attribute update rule establishes $\mathsf{guard}(\texttt{x.a} := \texttt{y}, \texttt{o})$, and thus by admissibility condition (A3), `o.inv` still holds in in the heap h'. □

A variant of the soundness proof above has been mechanized in Dafny [74] is available online [110].

As a closing remark, we note that another way to show soundness of SC is via reduction to LCI. To encode collaboration in LCI on top of the ownership encoding detailed in [30], we add the following clauses to the invariant of each class: one stating that all `subjects` know **`Current`** for an observer (the consequent of (A2)), and for each attribute of **`Current`**, another one stating that all `observers` approve of the changes to this attribute.

5.4.4 Examples

We illustrate SC on the two examples of Sect. 5.2: Fig. 5.3 and 5.4 show the Observer and Iterator patterns fully annotated according to the rules of Sect. 5.4.2. We use the shorthands `wrap_all (s)` and `unwrap_all (s)` to denote calls to `wrap` and `unwrap` on all objects in a set `s`. As we discuss in Sect. 5.5, several annotations of Fig. 5.3 and 5.4 are subsumed by the defaults mentioned in Sect. 5.4.5. We postpone to Sect. 5.4.6 dealing with update guards

```
class SUBJECT

  value: INTEGER
  subscribers: LIST [OBSERVER]

  make (v: INTEGER) -- Constructor
    require open
    modify Current
    do
      value := v
      create subscribers
      owns := { subscribers }
      wrap
    ensure
      subscribers.is_empty
      wrapped
    end

  update (v: INTEGER)
    require
      wrapped
      all o ∈ observers : o.wrapped
    modify Current, observers
    do
      unwrap ; unwrap_all (observers)
      value := v
      across subscribers as o do o.notify end
      wrap_all (observers) ; wrap
    ensure
      value = v
      wrapped
      all o ∈ observers : o.wrapped
      observers = old observers
    end

feature {OBSERVER}
  register (o: OBSERVER)
    require
      not subscribers.has (o)
      wrapped
      o.open
    modify Current
    do
      unwrap
      subscribers.add (o)
      observers := observers + { o }
      wrap
    ensure
      subscribers.has (o)
      wrapped
    end
```

```
invariant
  observers = subscribers.range
  owns = { subscribers }
  subjects = {}
end

class OBSERVER

  subject: SUBJECT
  cache: INTEGER

  make (s: SUBJECT) -- Constructor
    require
      open
      s.wrapped
    modify Current, s
    do
      subject := s
      s.register (Current)
      cache := s.value
      subjects := { s }
      wrap
    ensure
      subject = s
      wrapped
      s.wrapped
    end

feature {SUBJECT}
  notify
    require
      open
      subjects = {subject}
      subject.observers.has (Current)
      observers = {}
      onws = {}
    modify Current
    do
      cache := subject.value
    ensure
      inv
    end

invariant
  cache = subject.value
  subjects = { subject }
  subject.observers.has (Current)
  observers = {}
  owns = {}
end
```

Figure 5.3: The *Observer pattern* using SC annotations (underlined).

and the corresponding admissibility condition (A3).

Observer pattern. The OBSERVER's invariant is admissible (Definition 5.1) because it ensures that subject is in subjects (A1) and that Current is in the subject's observers (A2). Constructors normally wrap freshly allocated objects after setting up their state. Public routine update must be called when the whole object structure is wrapped and makes sure that it is wrapped again when the routine terminates. This specification style is convenient for public routines, as it allows clients to interact with the class while maintaining objects in a consistent state, without having to explicitly discharge any condition. Routines such as register and notify, with restricted visibility, work instead with open objects and restore their invariants so that they can be wrapped upon return. Since notify explicitly ensures inv, update does not need the precise definition of the observer's invariant in order to wrap it (it only needs to know enough to establish the precondition of notify). Thus the same style of specification would work if OBSERVER were an abstract class and its subclasses maintained different views of subject's value.

Let us illustrate the intuitive reason why an instance of SUBJECT cannot invalidate any object observing its state. On the one hand, by the attribute update rule, any change to a subject's state (such as assignment to value in update) must be reconciled with its observers. On the other hand, any closed concerned OBSERVER object must be contained in its subject's observers set: a subject cannot surreptitiously remove anything from this set, since such a change would require an attribute update, and thus, again, would have to be reconciled with all current members of observers.

Note that we had to restate the first invariant clause of OBSERVER from Fig. 5.1 in terms of observers instead of subscribers. In general, collaboration invariants have to be expressed directly in terms of attributes of subjects and cannot refer to their ownership domains. This is not a syntactic restriction but follows from the fact that it is rarely possible to establish a subject/observer relation with the whole domain (in this example, we would have to require LIST to allow OBSERVER objects in its observers set). Sect. 5.4.7 discusses an extension to SC that would allow such dependencies in invariants; note, however, that this limitation can always be easily circumvented by introducing a ghost attribute in the subject that mirrors the requires state.

Iterator pattern. The main differences in the annotations of the Iterator pattern occur in the COLLECTION class whose non-ghost state is, unlike SUBJECT above, unaware of its observers. Routine remove_last has to unwrap its observers according to the update rule. However, it has no way of restoring their invariants (in fact, a collection is in general unaware even of the *types* of the iterators operating on it). Therefore, it can only leave them in an

```
class COLLECTION [G]

  count: INTEGER

  make (capacity: INTEGER) --
       Constructor
    require
      open
      capacity ≥ 0
    modify Current
    do
      create elements(1, capacity)
      owns := { elements } ; wrap
    ensure
      elements.count = capacity
      count = 0
      observers = {}
    end

  remove_last
    require
      count > 0
      wrapped
      all o ∈ observers : o.wrapped
    modify Current, observers
    do
      unwrap ; unwrap_all (observers)
      observers := {}
      count := count − 1
      wrap
    ensure
      count = old count − 1
      wrapped
      observers = {}
      all o ∈ old observers : o.open
    end

feature {ITERATOR}
  elements: ARRAY [G]

invariant
  0 ≤ count and count ≤ elements.count
  owns = { elements }
  subjects = {}
end
```

```
class ITERATOR [G]

  target: COLLECTION [G]
  before, after: BOOLEAN

  make (t: COLLECTION) -- Constructor
    require
      open and t.wrapped
    modify Current, t
    do
      target := t
      before := True
      t.unwrap
      t.observers := t.observers +
        { Current }
      t.wrap
      subjects := { t }
      wrap
    ensure
      target = t
      before and not after
      wrapped
    end

  item: G
    require
      not (before or after)
      wrapped and t.wrapped
    do
      Result := target.elements [index]
    end

feature {NONE}
  index: INTEGER

invariant
  0 ≤ index and index ≤ target.count + 1
  before = index < 1
  after = index > target.count
  subjects = { target }
  target.observers.has (Current)
  observers = {} and owns = {}
end
```

Figure 5.4: The *Iterator pattern* using SC annotations (underlined).

inconsistent state and remove them from the `observers` set. Public routines of `ITERATOR`, such as `item`, normally operate on wrapped objects, and hence in general cannot be called after some operations on the collection has disabled its iterators. The only way out of this is if the client of collection and iterators can prove that a certain iterator object `i_x` was not in the modified collection's `observers`; this is possible if, for example, the client directly created `i_x`. The fact that now clients are directly responsible for keeping track of the `observers` set is germane to the iterator domain: iterators are meant to be used locally by clients.

5.4.5 Default Annotations

The annotation patterns shown in Sect. 5.4.4 occur frequently in object-oriented programs. To reduce the annotation burden in those cases, we suggest the following defaults.

Pre- and postconditions: public procedures require and ensure that the `Current` object, its `observers`, and routine arguments be `wrapped`.

Frames: procedures modify `Current`; functions modify nothing.

Invariants: Built-in ghost set attributes (such as `owns`) are invariably empty if they are not mentioned in the programmer-written invariant.

Wrapping: public procedures start by unwrapping `Current` and terminate after wrapping it back.

Built-in set manipulation: if a built-in ghost set attribute s is only mentioned in an invariant clause of the form $s =$ `expr`, then s is considered *implicit*; correspondingly, every `wrap` of objects enclosing s will implicitly perform an assignment s:= `expr`.[7]

These defaults encourage considerate programming: unless explicitly specified otherwise, an object is always required to restore the consistency of its observers at the end of a public routine. This is a useful property, since the considerate paradigm promotes encapsulation and is convenient for the clients. Nevertheless, the defaults are only optional suggestions that can be overridden by providing explicit annotations; this ensures that they do not tarnish the flexibility and semantic nature of our methodology.

[7]This is inspired by the default “static” treatment of `owns` sets in VCC.

5.4.6 Update guards

Update guards are used to distribute the burden of reasoning about attribute updates between subjects and observers, depending on the intended collaboration scheme. At one extreme, if a $\mathsf{guard}(\mathtt{x.a} := \mathtt{y}, \mathtt{o})$ is identically **`False`**, the burden is entirely on the subject, which must check that all observers are open whenever `a` is updated; in contrast, the admissibility condition (A3) holds vacuously for the observer `o`. At the other extreme, if a guard is identically **`True`**, the burden is entirely on the observer, which deals with (A3) as a proof obligation that its invariant does not depend on `a`; in contrast, the subject `x` can update `a` without particular constraints.

Another recurring choice for a guard is $\mathtt{inv(o)} \Rightarrow \mathtt{inv(o)}_{\mathsf{H}[\mathtt{x.a} \mapsto \mathtt{y}]}$. For its flexibility, we chose this as the default guard of SC. Just like **`False`**, this guard also does not burden the observer, but is more flexible at the other end: upon updating, the subject can establish that each observer is either open or its invariant is preserved. The subject can rely on the latter condition if the observer's invariants are known, and ignore it otherwise.

When it comes to built-in ghost attributes, $\underline{\mathtt{owns}}$ and $\underline{\mathtt{subjects}}$ are guarded with **`True`**, since other objects are not supposed to depend on them, while $\underline{\mathtt{observers}}$ has a more interesting guard, namely $\mathsf{guard}(\mathtt{x.}\underline{\mathtt{observers}} := \mathtt{y}, \mathtt{o}) = \mathtt{o} \in \mathtt{y}$. This guard reflects the way this attribute is commonly used in collaboration invariants, while leaving the subject with reasonable freedom to manipulate it; for example, adding new observers to the set $\underline{\mathtt{observers}}$ without "notifying" the existing ones (this is used, in particular, in the `register` routine of Fig. 5.3).

5.4.7 Extensions

Dependency on ownership domains. In the core SC methodology presented so far, an invariant of `x` can only depend on `x` and objects directly contained in $\mathtt{x.}\underline{\mathtt{owns}}$ and $\mathtt{x.}\underline{\mathtt{subjects}}$. Extending this dependency to the ownership domain of `x` (that is, taking transitive closure of $\underline{\mathtt{owns}}$) does not present any difficulty; but what about invariants that rely on the domain of a subject $\mathtt{s} \in \mathtt{x.}\underline{\mathtt{subjects}}$? Allowing such invariants is desirable, since ownership-based methodologies strive to make an object and its ownership domain indistinguishable for clients. Unfortunately, current rules of SC are unsound for such invariants: it is easy to set up an example where an object `o` from the domain of `s` is being modified and is unaware of `x`, which is closed, thus threatening to invalidate it.

One way to address this problem is to modify the built-in operations and proof obligations of SC as follows. The main idea is to decouple the updates to closed from the updates owner, both of which are currently performed by the wrap and unwrap operations. The new semantics of wrap and unwrap is to only set the closed attribute, while owner manipulation is performed by two new dedicated operation:

Freeing: frees all owned objects of x after its observers have been opened:

```
x.free_owns     require                          ensure
                x.open                           all o ∈ x.owns : o.wrapped
                all y ∈ x.observers : y.open
                all o ∈ x.owns : o.owner = x
```

Reclaiming: gives x the ownership over wrapped objects in x.owns:

```
x.reclaim_owns  require                          ensure
                x.open                           all o ∈ x.owns : o.owner = x
                all o ∈ x.owns : o.wrapped
```

Wrapping additionally requires that owned objects of all subjects are reclaimed:

```
x.wrap          require                          ensure
                x.open                           x.wrapped
                x.inv
                all y ∈ x.subjects + {x} :
                  all o ∈ y.owns : o.owner = y
```

In order to operate on an object o owned by s, one now ought to first unwrap s, then unwrap all observers of s, and finally call s.free_onws, which makes o wrapped, and hence ready to be opened and modified. Since closing an observer of s would require o to be reclaimed (and thus not open or wrapped), this approach maintains a global invariant that all observers of s are open while o is open.

This flavor of SC is more congenial to ownership and has an additional benefit that the semantics of read clauses of logic functions can be made consistent with the semantics of modify clauses of routines (in core SC, including ownership domains into read frames of functions would effectively ban them from invariants). Unfortunately, the proof obligations of this extension are more complex, and hard to establish in practice: in particular, wrapping any object requires reasoning about the ownership domains of its subjects. As a result of the experimental evaluation of Sect. 5.5 we decided to stick to the core SC and leave this extension to future work.

Inheritance. Inheritance poses additional challenges for invariant protocols, since the dynamic type of an object, and thus the precise definition of its invariant, is rarely known. In SC, this issue arises when reasoning about calls to `wrap`. Rather than imposing severe restrictions on how invariants can be strengthened in descendants, we prefer to re-verify those inherited routines that wrap the `Current` object to make sure they still properly re-establish the invariant. We maintain that this approach achieves a reasonable trade-off.

As a way to abstract from the invariant definition, routines may use the built-in predicate `inv` explicitly in their contracts (see Fig. 5.3), which enables wrapping objects without knowing their dynamic type. This observation generalizes to other logic functions, which can be redefined in descendants, supporting specifications that are flexible yet strong enough. This approach to inheritance is similar in spirit to *abstract predicates* in separation logic [103], and has been used before in auto-active verifiers (e.g. [117]).

Concurrency. When it comes to reasoning about invariants, sequential and concurrent programs each have their distinctive challenges. In a sequential setting, one typically performs state updates in a series of steps that temporarily break object consistency; this is acceptable since intermediate states are not visible to other objects. A sequential invariant protocol must adequately support such update schemes, while making sure that invariants hold at "crucial" points. Concurrent invariant protocols deal with different schemes, and hence have different goals. For this reason, we do not recommend extending SC to deal with concurrent programs; rather, it could be *combined* with an invariant protocol for concurrent programs, as done in VCC [29].

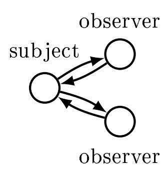
5.5 Experimental Evaluation

We arranged a collection of representative challenge problems involving inter-object collaboration, and we specified and verified them using our SC methodology. This section presents the challenge problems (Sect. 5.5.1), and discusses their solutions using SC (Sect. 5.5.2), as well as other methodologies (Sect. 5.5.3). See [110] for full versions of problem descriptions, together with our solutions, and a web interface to the AutoProof verifier.

5.5.1 Challenge Problems

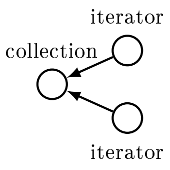
Beside using it directly to evaluate SC, the collection of challenge problems described in this section can be a benchmark for other invariant methodologies. The benchmark consists of six examples of varying degree of difficulty, which capture the essence of various collaboration patterns often found in object-oriented software. The emphasis is on non-hierarchical structures that maintain a global invariant.

We briefly present the six problems in roughly increasing order of difficulty in terms of the shape of references in the heap, state update patterns, and challenges posed to preserving encapsulation.



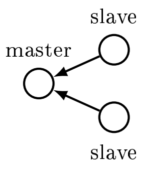
Observer [80, 102, 88] (see also SAVCBS '07 [109], and Sect. 5.2). The invariants of the observer objects depend on the state of the subject. Verification must ensure that the subject reports all its state changes to all observers, so that their clients can always rely on a globally consistent state. *Additional challenge*: combination with ownership (the subject keeps references to its observers in a collection, which is a part of its representation).

Variants: a simplified version where the number of observers is fixed (thus collections of observers are not needed); a more complex version with multiple observer classes related by inheritance, each class redefining the class invariant and the implementation of `notify`.



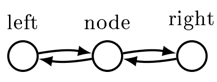
Iterator [80] (see also SAVCBS '06, and Sect. 5.2). Unlike observers in the Observer pattern, the implementation of a collection is not aware of the iterators operating on it. Specification must still be able to refer to the iterators attached to the collection while avoiding global reasoning. *Additional challenge*: we cannot rely on the implementation following considerate programming (where objects must be in consistent states at public call boundaries).

Variants: a more complex version where iterators modify the collection.

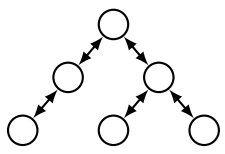


Master clock [11, 80]. The time stored by a master clock can increase (public routine `tick`) or be set to zero (public routine `reset`). The time stored locally by each slave clock must never exceed the master's but need not be perfectly synchronized. Therefore, when the master is `reset`, its slaves are disabled until they synchronize (similar to iterators); when the master increments the time, its slaves remain in a consistent state without requiring synchronization. *Additional challenges*: `tick`'s frame does not include slaves; perform reasoning local to the master with only partial knowledge of the slaves' invariants.

Variants: a version without `reset` (slaves cannot become inconsistent).

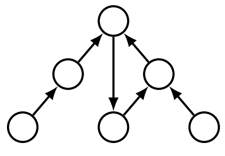


Doubly-linked list [77, 87]. The specification expresses the consistency of the `left` and `right` neighbors directly attached to each `node`. Verification establishes that updates local to a node (such as inserting or removing a node next to it) preserve consistency. Unlike in the previous examples, the heap structure is recursive; the main challenge is thus avoiding considering the list as a whole (such as to propagate the effects of local changes).



Composite [121, 120, 70, 108], (see also SAVCBS '08). A tree structure maintains consistency between the values stored by parent and children nodes (for example, the value of every node is the maximum of its children's). Clients can add children anywhere in the tree; therefore, ownership is unsuitable to model this example. Two new challenges are that the node invariant depends on an unbounded number of children; and that the effects of updates local to a node (such as adding a child) may propagate up the whole tree involving an unbounded number of nodes. Specification deals with these unbounded-size footprints; and verification must also ensure that the propagation to restore global consistency terminates. Clients of a tree can rely on a globally consistent state while ignoring the tree structure.

Variations: a simplified version with n-ary trees for fixed n (the number of children is bounded); more complex versions where one can also remove nodes or add whole subtrees.



PIP [121, 120, 108]. The Priority Inheritance Protocol [113] describes a compound whose nodes are more loosely related than in the Composite pattern: each node has a reference to at most one parent node, and cycles are possible. Unlike in the Composite pattern, the invariant of a node depends on the state of objects not directly accessible in the heap (parents do not have references to their children). New challenges derive from the possible presence of cycles, and the need to add children that might already be connected to whole graphs; specifying footprints and reasoning about termination of update operations are trickier.

5.5.2 Results and Discussion

We specified the six challenge problems using SC, and verified the annotated Eiffel programs with AutoProof. Tab. 5.1 shows various metrics about our solutions: the SIZE of each annotated program; the number of TOKENS of EXECutable code, REQuirements specification (the given functional specification to be verified), and AUXiliary annotations (specific to our methodology, both with and without default annotations); the SPEC/EXEC overhead, i.e., (REQ + AUX)/EXEC; and the verification time in AutoProof. The overhead is roughly between 1.5 (for Observer) and 6 (for PIP), which is comparable

Table 5.1: The challenge problems specified and verified using SC.

	SIZE	TOKENS (no defaults)				TOKENS (defaults)		TIME
PROBLEM	(LOC)	EXEC	REQ	AUX	$\frac{\text{SPEC}}{\text{EXEC}}$	AUX	$\frac{\text{SPEC}}{\text{EXEC}}$	(sec.)
Observer	129	156	52	296	2.2	185	1.5	7
Iterator	177	168	176	315	2.9	247	2.5	7
Master clock	130	85	69	267	4.0	190	3.1	5
DLL	147	136	83	435	3.8	320	3.0	8
Composite	188	124	270	543	6.6	427	5.6	12
PIP	152	116	310	445	6.5	402	6.1	12
Total	923	785	960	2301	4.2	1771	3.5	50

with that of other verification methodologies applied to similar problems (see e.g. [57]). The default annotations of Sect. 5.4.5 reduce the overhead by a factor of 1.3 on average.

The PIP example is perfectly possible using ghost code, contrary to what is claimed elsewhere [121]. In our solution, every node includes a ghost set `children` with all the child nodes (inaccessible in the non-ghost heap); it is defined by the invariant clause `parent` $\neq$ **`Void`** $\Rightarrow$ **`Current`** $\in$ `parent.children`, which ensures that `children` contains every closed node n such that n.`parent` $=$ **`Current`**. Based on this, the fundamental consistency property is that the `value` of each node is the maximum of the values of nodes in `children` (or a default value for nodes without children), assuming maximum is the required relation between parents and children.

The main challenge in Composite and PIP is reasoning about framing and termination of the state updates that propagate along the graph structure. For framing specifications, we use a ghost set `ancestors` with all the nodes reachable following `parent` references. Proving termination in PIP requires keeping track of all visited nodes and showing that the set of ancestors that haven't yet been visited is strictly shrinking.

5.5.3 Comparison with Existing Approaches

We outline a comparison with existing approaches (focusing on those discussed in Sect. 5.3) on our six challenge problems. Tab. 5.2 reports how each methodology fares on each challenge problem: $-$ for "methodology not applicable", $+$ for "applicable", and $\oplus$ for "applicable and used to demonstrate the methodology when introduced".

Only SC is applicable to all the challenges, and other methodologies often have other limitations (notes in Tab. 5.2). Most approaches cannot deal

Table 5.2: Comparison of invariant protocols on the challenge problems.

	VISIBLE-STATE		SPEC# METHODOLOGIES			
	Cooperation [88]	Considerate [120]	Visibility-based [77]	Friends [11]	History [80]	SC
Observer	$\oplus$	$+$	$+$	$\oplus$	$\oplus^{d}$	$\oplus$
Iterator	$-^{a}$	$-^{a}$	$+$	$+$	$\oplus^{d}$	$\oplus$
Master clock	$-^{a}$	$-^{a}$	$+$	$\oplus$	$\oplus^{d}$	$\oplus$
DLL	$+$	$+$	$\oplus$	$+$	$+^{d}$	$\oplus$
Composite	$-^{b}$	$\oplus^{c}$	$-^{b}$	$-^{b}$	$-^{b}$	$\oplus$
PIP	$-^{b}$	$\oplus^{c}$	$-^{b}$	$-^{b}$	$-^{b}$	$\oplus$

[a] Only considerate programming [b] Only bounded set of reachable subjects
[c] No framing specification [d] No invariant stability

with unbounded sets of subjects, and hence are inapplicable to Composite and PIP. The methodology of [120] is an exception as it allows set comprehensions in invariants; however, it lacks an implementation and does not discuss framing, which constitutes a major challenge in Composite and PIP. Both methodologies [88, 120] based on visible-state semantics are inapplicable to implementations which do not follow considerate programming; they also lack support for hierarchical object dependencies, and thus cannot verify implementations that rely on library data structures (e.g., Fig. 5.1 and 5.2).

Another important point of comparison is the level of coupling between collaborating classes, which we can illustrate using the Master clock example. In [77], class `MASTER` requires complete knowledge of the invariant of class `CLOCK`, which breaks information hiding (in particular, `MASTER` has to be re-verified when the invariant of `CLOCK` changes). The update guards of [11] can be used to declare that slaves need not be notified as long their master's time is increased; this provides abstraction over the slave clock's invariant, but class `MASTER` still depends on class `CLOCK`—where the update guard is defined. In general, the syntactic rules of [11] require that subject classes declare all potential observer classes as "friends". In SC, update guards are defined in subject classes; thus we can prove that `tick` maintains the invariants of all observers without knowing their type. Among the other approaches, only history invariants [80] support the same level of decoupling, but they cannot preserve stability with the `reset` routine.

Reasoning without invariants. Other, more fundamental verification methodologies not based on invariants, such as dynamic frames [60] and separation logic [107], can fully handle all the six benchmark problems. The generality they achieve is, however, not without costs, as one loses stability of consistency properties (e.g., SUBJECT is not required to notify all its observers). The first automatic verification of the PIP example was described in [108]: the solution relies on reusable specification elements called *stereotypes* to achieve low annotation overhead, however the reported verification times are significantly higher than in our experiments.

SAVCBS workshops solutions. SC also fares favorably compared against the solutions submitted to the SAVCBS workshops [109] challenges (Iterator, Observer, and Composite). Considering only solutions for general-purpose languages and targeting complete requirement specifications, there are two solutions to the Iterator problem and two to the Composite problem. One solution to the Iterator uses JML and ESC/Java2; the collaborating parts of the invariants are, however, described by pre- and postconditions. One solution to the Composite also uses JML; it is hard to compare it to our solution as it is based on model programs and proves invariant preservation only for routines that refine the model program used as specification. The other solution to the Composite [57] uses separation logic and VeriFast; the specification overhead for clients is higher than in our solution, but there is no ghost state in the nodes (which otherwise would have had to be updated during global modifications), thus it has advantages and disadvantages compared to our solution. The second solution to the Iterator problem [65] uses higher-order separation logic, and thus is not amenable to automated reasoning.

5.6 Summary

This chapter presented *semantic collaboration*: a new methodology for specifying and verifying invariants of arbitrary object structures. Compared to existing invariant protocols, it offers considerable flexibility and conceptual simplicity, as it does not syntactically restrict the form of invariants. We implemented semantic collaboration as part of the AutoProof Eiffel program verifier. Our experiments with six challenge problems demonstrate the wide applicability of the methodology.

Chapter 6

Verifying Reusable Components

In the previous chapter we presented a general verification methodology for object-oriented programs, which can support a variety of specification styles. This chapter extends the methodology with support for model-based contracts: a particular specification method introduced in Chapter 3.

6.1 Introduction

Data abstraction is crucial for making reusable components truly reusable: abstract specifications provide clients with a simpler, more useful interface, and shield them from the evolution of the component's implementation.

Techniques presented in the previous chapter provide the basis of our verification methodology, but for the most part ignore data abstraction; this chapter extends the methodology with support for abstraction mechanisms, based on *models*. The ultimate goal is being able to verify interface specifications in the form of model-based contracts, proposed in Chapter 3.

Supporting model-based specifications in a program verifier involves three major challenges. First, such specifications rely on *model classes* and their operations to describe component's behavior at an abstract level; those operations need to be assigned meaning in the underlying logic of the verifier. Second, model-based contracts use *model queries* to map concrete program states to abstract values; a verification methodology has to encode this mapping, called the *abstraction function*, in a sound and modular way, and find the right trade-off between ease of automated reasoning and annotation overhead. Third, *frame specifications* utilize models to express that only some part of an object's state is modified, without revealing its internal representation; such frame specifications need an interpretation is terms of concrete heap locations.

Once again, EiffelBase2 (see Chapter 2) serves as a testbed for the techniques proposed in this chapter. Verifying a realistic library comes with its own set of challenges, such as, for example, a nontrivial inheritance hierarchy, where models can be reused, replaced, and combined.

We integrated the support for model-based contracts into *AutoProof*, an auto-active [76] verifier for functional properties of programs written in Eiffel. AutoProof checks that a program adheres to a specification in the form of inline assertions, routine contracts (pre- and post-conditions, and frame specifications), and class invariants; is also proves termination of loops and recursion using variant functions. Advanced specification constructs of AutoProof include pure routines and logic functions, which can be used in specifications, ghost variables and code, and frame specifications for loops. AutoProof works by generating verification conditions from the program and its specification, via the Boogie intermediate language [8, 73]; these conditions are discharged by a reasoning engine of choice, usually Z3 [35]. For more information on AutoProof we refer the reader to the project website [5] and previous work [128, 129, 126].

After having enhanced AutoProof with support for models, we used it to verify the core of the EiffelBase2 library, which constitutes the main contribution of this part of the work. Other contributions include a practical approach to adding mathematical types to a program verifier by means of model classes, and a flexible encoding of model-based frame specifications with support for model adaptation due to inheritance.

The rest of the chapter is structured as follows. Sect. 6.2 gives an example of model-based specifications that we aim to verify; the example is used to illustrate the methodology throughout the chapter. Sect. 6.3 details our solutions to the three major challenges outlined above. The experimental evaluation of the techniques on the EiffelBase2 library is presented in Sect. 6.4, while Sect. 6.5 reviews related work and Sect. 6.6 concludes.

6.2 A Motivating Example

Fig. 6.1 illustrates the kind of model-based specifications we aim to verify, using a simplified example from EiffelBase2. In the example, a deferred class `CONTAINER` declares a `bag` of elements as its model; the container's interface consists of two queries: `count` (for the number of elements) and `is_empty`, and one command: `wipe_out`, which empties the container. Class `STACK` inherits

```
deferred class CONTAINER [G]
model bag
  bag: ghost MML_BAG [G]

  count: INTEGER
    deferred
    ensure
      Result = bag.count
    end

  is_empty: BOOLEAN
    do
      Result := count = 0
    ensure
      Result = bag.is_empty
    end

  wipe_out
    modify model [bag] Current
    deferred
    ensure
      bag.is_empty
    end
end

class MML_SEQUENCE [G]
  count: INTEGER
    ...

  last: G
    require count > 0
    ...

  extended (x: G): MML_SEQUENCE [G]
    alias "&"
    ...

  to_bag: MML_BAG [G]
    ...
end
```

```
class STACK
inherit CONTAINER
model sequence
  list: LIST [G]
  sequence: ghost MML_SEQUENCE [G]

  item: G
    require
      not is_empty
    do
      Result := list.item (count)
    ensure
      Result = sequence.last
    end

  count: INTEGER
    do
      Result := list.count
    end

  push (v: G)
    modify model [sequence] Current
    do
      list.extend_back (v)
    ensure
      sequence = old sequence & v
    end
   ...

  wipe_out
    do ...
    end
invariant
  bag = sequence.to_bag
  list ≠ Void
  owns = {list}
  sequence = list.sequence
end
```

Figure 6.1: Excerpt from class STACK, which inherits from CONTAINER and uses a LIST as its internal representation. The specification of STACK uses the model class MML_SEQUENCE, also shown in the listing, to refer to the mathematical sequence of stack elements.

from `CONTAINER` and replaces its model with a `sequence` of elements (hence the linking invariant `bag = sequence.to_bag`); it relies on an instance of class `LIST` as the concrete internal representation; its interface additionally includes a query `item` (which returns the top element) and a command `push` (which pushes a given value on top).

Although small, this example showcases various features of model-based contracts that a verification methodology has to support. First, the verifier needs to know how to interpret instances of `MML_BAG` and `MML_SEQUENCE`, and operations over those instances. Second, the routines of class `STACK` are specified in terms of the abstract state (the `sequence`), and implemented in terms of the concrete state (the `list`); verification of such routines has to rely on the abstraction function, expressed in the last class invariant clause. Finally, frame specifications in the example are expressed in terms of the abstract state (denoted by **`modify model`**). We would like to let `push` modify the `list`, without having to mention it in the frame. Also, the frame specification of `wipe_out` in class `CONTAINER` is expressed in terms of the model query `bag`, which is replaced in the child class. We would like to allow the implementation of `wipe_out` in `STACK` to modify `sequence`, even though it was not declared as part of the frame in the original version.

6.3 Verification Methodology for Model-Based Contracts

This section presents our solutions to the challenges posed by model-based contracts, and illustrates them using the `STACK` example from Fig. 6.1.

6.3.1 Encoding of Model Classes

Runtime checking of model-based contract requires expressing the semantics of model classes in a language understood by the compiler, that is, giving them implementations. On the other hand, using model classes in deductive proofs requires explaining their semantics to the proof engine (in our case, Boogie). An effective way to achieve this [70, 33] is to map model classes directly to *theories* in the underlying logic of the verifier. In Boogie, a theory is a collection of types and uninterpreted functions operating on those types, which are axiomatized in first order logic.

State-of-the-art auto-active verifiers, such as Dafny [74], often include a small, predefined set of mathematical types and operations. This approach is easy to implement, since the syntax and semantics of all mathematical operations can be hard-coded. The goal of AutoProof is to support the

```
class MML_SEQUENCE [G]
inherit ITERABLE [G]
maps_to Seq
theory "sequence.bpl"
typed_sets Seq#Range

  count: INTEGER
    maps_to: Seq#Length

  last: G
    require count >0
    ...

  range: MML_SET [G]
    ...

  extended (x: G): MML_SEQUENCE [G]
    alias "&"
    ...

  to_bag: MML_BAG [G]
    ...

  shorter_equal (s: MML_SEQUENCE [G]): BOOLEAN
    alias "≤"
    ...

  new_cursor: CURSOR [G]
    maps_to Seq#Range
end
```

Figure 6.2: Model class `MML_SEQUENCE` equipped with AutoProof *logic class* annotations. Features with no **`maps_to`** clause use the default naming scheme: for example, `last` maps to `Seq#Last`.

whole Mathematical Model Library (MML, see Chapter 2), which is more diverse than a typical set of built-in mathematical types: after all, the design of MML was driven by the EiffelBase2 specification effort. In addition, we would like to be able to add new types and operations relatively easily.

To address those concerns we extended AutoProof with *logic classes*, whose instances and operations are translated into Boogie in a special way. To declare a logic class, it is enough to add a **`maps_to`** clause to the class definition,[1] specifying the Boogie type, to which instances of this class will belong. Each constructor and function of the class can be mapped to a particular Boogie function in a similar way, using a **`maps_to`** clause; in the absence of such a clause, the Boogie function name is derived from the Eiffel feature name. AutoProof's **`maps_to`** clauses are inspired by the work of Darvas and Müller [33], who proposed a similar construct for JML. Fig. 6.2 gives an examples of using **`maps_to`** clauses to define the semantics of `MML_SEQUENCE`.

Boogie types and functions used in a logic class are encoded manually in one or more separate files. The **`theory`** clause (see an example in Fig. 6.2) allows one to specify Boogie files to be included into the translation generated by AutoProof whenever the corresponding logic class is used in an Eiffel pro-

[1]Recall that in the actual code we use Eiffel's **`note`** meta-annotation to introduce new specification constructs.

```
// Sequence type
type Seq T;

// Sequence length
function Seq#Length<T>(Seq T): int;

// Element at a given index
function Seq#Item<T>(Seq T, int): T;

// Last element
function Seq#Last<T>(q: Seq T): T
{ Seq#Item(q, Seq#Length(q)) }

// Set of values
function Seq#Range<T>(Seq T): Set T;
axiom (∀<T> q: Seq T, o: T • Seq#Has(q, o) ⟺Seq#Range(q)[o]);

// Sequence extended with x at the end
function Seq#Extended<T>(s: Seq T, val: T): Seq T;
axiom (∀<T> s: Seq T, v: T •
  Seq#Length(Seq#Extended(s,v)) =1 + Seq#Length(s));
axiom (∀<T> s: Seq T, i: int, v: T •
  (i =Seq#Length(s) + 1 ⟹Seq#Item(Seq#Extended(s,v), i) =v) ∧
  (i ≤Seq#Length(s) ⟹Seq#Item(Seq#Extended(s,v), i) =Seq#Item(s, i)));

// Sequence converted to a bag
function Seq#ToBag<T>(Seq T): Bag T;
axiom (∀<T> • Seq#ToBag(Seq#Empty(): Seq T) =Bag#Empty(): Bag T);
axiom (∀<T> s: Seq T, v: T •
  Seq#ToBag(Seq#Extended(s, v)) =Bag#Extended(Seq#ToBag(s), v));

// Is |q0| ≤|q1|?
function Seq#LessEqual<T>(q0: Seq T, q1: Seq T): bool
{ Seq#Length(q0) ≤Seq#Length(q1) }
```

Figure 6.3: An excerpt from `sequence.bpl`, which contains a manual Boogie encoding of `MML_SEQUENCE`.

gram. Fig. 6.3 shows an extract from a Boogie sequence theory `sequence.bpl`, which contains custom translations of `MML_SEQUENCE` operations.

AutoProof also implements a number of advanced features, which aim at making custom logic classes as expressive and convenient as built-in mathematical types:

- *Quantifiers.* Eiffel's iteration mechanism makes it possible to use an arbitrary object `s` in a loop expression, such as **all** `x ∈ s : B(x)`, as long as the class of `s` inherits from `ITERABLE` and implements a feature `new_cursor`. AutoProof can translate such loop expressions into quantification if the class of `s` is logical and its `new_cursor` feature is mapped to a set-valued function (whose result is then taken as the quantification domain). For example, the declaration in Fig. 6.2 enables quantification over sequences, translated into quantification over the elements of a sequence (its `range`).

- *Ordering.* By convention, a feature with a $\leq$ alias (such as `shorter_equal` in Fig. 6.2) defines a well-founded order on the values of a logic class, which makes it possible to use them as variants in termination proofs.

- *Element types.* It is a common technique [73] to map all reference types of a source language to a single Boogie type, while encoding more precise type information in automatically generated predicates. Since many logic classes represent mathematical "collections" of elements, it is often necessary to encode the fact that all elements of such collection belong to a certain type. For example, an Eiffel declaration `s: MML_SEQUENCE [PERSON]` should give rise to an axiom stating that all elements of `s` are of type `PERSON` (or its descendants). The **`typed_sets`** clause achieves exactly that: for every generic parameter `G` of a logic class, it defines a set-valued function that returns the set of all objects of type `G` "stored" in an instance of the class. Fig. 6.2 defines the required property for sequences; as another example, class `MML_MAP [K, V]`, with two generic parameters, declares two typed sets: corresponding to the domain and the range of a map.

- *Constraints.* Sometimes a logic class imposes constraints on its values, which are guaranteed to hold whenever a value of the class is chosen nondeterministically. AutoProof supports such constraints through **`where`** clauses. For example, class `MML_BAG` is equipped with a **`where`** clause stating that the number of occurrences of each element is positive.

- *Boogie maps.* Map types are built-in Boogie types, useful in encoding various mathematical constructs. It is often desirable to map a logic class to a map type, rather than to an uninterpreted, user-defined type. If one of the features of a logic class is mapped to brackets (**`maps_to`** `"[ ]"`), AutoProof associates this feature with map access in Boogie, and infers the map type from its signature. For example, in `MML_SET [G]` function `has (x: G): BOOLEAN` is mapped to brackets; thus the logic class is translated as a Boogie map from `G` to Booleans.

To our knowledge, AutoProof is the only auto-active verifier with support for extensible mathematical types that enjoy the same level of language integration as built-in types in other verifiers.

6.3.2 Encoding of Model Queries

The interface specification methodology of Chapter 3 does not detail how the abstract state of a class is related to its concrete state. For testing purposes (Chapter 4), we had to decide between attribute-based and function-based encoding of specification-only model queries, based on the specification overhead and runtime performance of the two approaches. For deductive verification, however, the choice of encoding has different implications.

One approach to representing abstract state in verification is based on *model variables* (also called *model fields*) [24, 68, 72, 91]. Programs cannot assign directly to model variables; instead their semantics is defined by an *abstraction function*, i.e. a mapping from the concrete state. In fact, the traditional treatment of model variables is much closer to that of argument-less functions, rather than attributes, with the only difference that model variables need not have a functional definition and can be specified relationally.

This traditional approach is known to have soundness and modularity issues [78]. First, relational specifications of model variables can be unsatisfiable; even if an abstraction function is well-defined whenever the concrete state is consistent, it may be undefined in intermediate states, where the object invariant is temporarily violated. Second, translating frames expressed in terms of model variables into sets of heap locations is far from straightforward (for example, in JML they are interpreted using data groups [24]). The reasoning is complicated by the fact that any update to the concrete state can have an *instant effect* on the values of model variables elsewhere in the system, which is detrimental to modularity.

One final concern involves verifying postconditions expressed in terms of model variables. Verification of nontrivial routines (for example, those involving loops) often requires introducing auxiliary ghost variables to represent local or intermediate abstract state; thus, the main benefit of model variables compared to ordinary ghost variables—no bookkeeping—does not make a difference for such routines.

To preserve simplicity and avoid issues discussed above, AutoProof encodes model queries as regular (possibly, ghost) attributes. The relation between the abstract and concrete states of a class is given in the class invariant—see the last invariant clause of Fig. 6.1 for an example. Since our methodology only requires invariants to hold for closed objects (see Chapter 5), this treatment automatically supports partial abstraction functions, while giving routines freedom to modify the abstract and the concrete states of an object independently when the object is open. Unsatisfiable constraints on abstract state are easily detected, since they would prevent wrapping the object. The meaning of model queries in frame specifications also becomes straightforward: we discuss it in detail in Sect. 6.3.3.

In order to reduce the bookkeeping overhead, particularly noticeable for unused inherited model queries, we propose to reuse the *implicit attribute* mechanism, which is already implemented in AutoProof for built-in ghost sets owns, subjects and observers, and was described in Sect. 5.4.5. The idea is very simple and inspired by VCC's "static owns": if a model query m is mentioned in an invariant clause of the form $m =$ expr, then every wrap of objects enclosing m will implicitly perform an assignment m:= expr[2]. The order of assignments to model queries within the same class is usually irrelevant, since they are supposed to express independent dimensions of the abstract state; unused inherited model queries are updated at the end, reflecting the typical structure of linking invariants. For example, routines push and wipe_out of class STACK in Fig. 6.1 need not explicitly update the model queries sequence and bag; instead, the following instructions are implicitly executed at the end of their bodies:

```
sequence := list.sequence
bag := sequence.to_bag
wrap
```

Note that, like all AutoProof defaults, this behavior can be easily overridden when inapplicable.

This new mechanism is a conservative extension of its original version, since the built-in ghost attributes of semantic collaboration (closed, owns, owner, subjects, and observers) are automatically added to the list of model queries of every class.

We believe that implicit attributes successfully address the excess annotation overhead of ghost state in simple cases, while retaining conceptual simplicity and flexibility required in more sophisticated scenarios.

6.3.3 Abstract Framing

The **modify** and **read** clauses of semantic collaboration (Sect. 5.4.1) support frame definitions in terms of whole objects or individual (concrete) locations. At the whole-object level, they provide sufficient abstraction mechanisms, such as set expressions and ownership domains; at the same time, they lack the means to describe a part of the object state without revealing its concrete

[2]Guarded by the condition that m is writable.

representation.

In model-based specifications, it is natural to restrict a frame to particular model queries. To this end, we introduce a new kind of write and read frames, called *abstract frames* and specified through the keywords `modify model` and `read model`, respectively (see, for example, modify clauses of `wipe_out` and `push` in Fig. 6.1).

A clause `modify model` $[m_1, \dots, m_n]$ s in the specification of a routine r gives r permission to modify a location o.a only in one of two cases: *(1)* $o \in s$ and either a is one of $m_1, \dots, m_n$, or a is *not among the model queries* declared in the static type of o, or *(2)* $o \notin s$ but o belongs to the ownership domain of one of $o' \in s$. Case 1 is specific for abstract frames, while case 2 expresses closure under ownership domains and is common for both standard and abstract modify clauses. The semantics of `read model` is defined in a similar way, except case 2 does not apply (for a discussion see Sect. 5.4.7). As before, in place of a set expression s we allow individual objects, as well as lists containing objects and sets.

The `modify model` construct achieves the desired data abstraction: routines are allowed to update the concrete state without revealing it in specifications, while clients do not get any guarantees regarding the concrete state, which enforces information hiding. Note that abstract frames constitute an incremental change to the existing framing methodology: they rely on the same encoding of frames (as sets of locations), and the same proof obligations and frame axioms; the only difference is in the way modify (read) clauses are translated into write (read) sets.

Since abstract state only makes sense for closed objects, and it is impossible to modify an object without unwrapping it, every useful `modify model` clause has to include the model query `closed`. To reduce annotation overhead, AutoProof implicitly adds this attribute to all abstract write frames. For example, routine `push` in Fig. 6.1 is allowed to unwrap `Current`—which happens implicitly at the beginning of its body—because its frame specification is automatically expanded into `modify model [sequence, closed] Current`. (It is also allowed to modify any attribute of the `list` object, since it is contained in the ownership domain of `Current` at the entry to the routine.)

Abstract Framing and Inheritance

As we have seen in Chapter 3, and again in the motivating example of Sect. 6.2, models can change with inheritance. When a class B inherits from A, it defines its set of model queries $\mathsf{mq}(B)$ from scratch, which can lead to reusing A's model queries, adding new model queries (to express new dimensions of the abstract state), replacing A's model queries (to express existing dimensions in a different way), or even abandoning some of A's model queries, when a dimension becomes redundant. For an abstract frame of a routine r declared in class A, this raises a question: in which context should it be interpreted when reasoning about class B? An interpretation in the context of A would express the effect of r in terms of A's model rather than B's model, which is not very useful for clients of B. On the other hand, if we choose to translate the abstract frame specification into a set of locations differently in A and B, according to the substitutability principle, we have to make sure that B's interpretation is a subset of A's: otherwise the frame axiom A's clients rely on is unsound. To sum up, the challenge is to come up with an interpretation of inherited abstract frames that is sound yet useful, and does not incur a high annotation overhead in common cases.

Coming back to the motivating example of Fig. 6.1, class `STACK` reuses the built-in model queries of `CONTAINER` (`closed`, `owns`, etc.), and replaces its remaining model query `bag` with `sequence` (so far, the fact that `bag` is replaced rather than abandoned is not reflected in the specification). Interpreting the modify clause of `wipe_out` in the context of `CONTAINER` gives it permission to modify all attributes of **`Current`** that are not among `CONTAINER`'s model queries, and thus it does not give the clients of `STACK` any guarantees about its new model queries.

To support the desired semantics of abstract frames, we introduce the notion of *flat model queries* of a class C, $\mathsf{fmq}(C)$, which include model queries declared in C and all its ancestors: $\mathsf{fmq}(C) = \mathsf{mq}(C) \cup \bigcup_{P \in parents(C)} \mathsf{fmq}(P)$. By definition, the set of flat model queries can only grow with inheritance: $\mathsf{fmq}(A) \subseteq \mathsf{fmq}(B)$ whenever B inherits from A.

Next, we refine case 1 in the above definition of **`modify model`** $[m_1, \ldots, m_n]\ s$ as follows: $o.a$ can be modified if $o \in s$ and either a is one of $m_1, \ldots, m_n$, or a is not among the *flat model queries* declared in the static type of o. Now, instead of interpreting the frame specification in the context of a parent class A, where it is written, we can interpret it in the child class B, where it is used; in particular, if o is **`Current`**, its static type inside B is B, and thus the interpretation of the abstract frame automatically shrinks, excluding all newly introduced model queries of B. This semantics of abstract frames is consistent with subtyping, and works well when a child class introduces independent model queries; unfortunately, it is overly restrictive when parent's model queries are replaced. For example, this semantics would prevent the version of `wipe_out` in `STACK` from modifying `sequence`.

To address this problem, AutoProof allows additional **`modify model`** clauses in routine redefinitions, with the effect of adding new locations to the inherited frame. The soundness requirement gives rise to a proof obligation that the new set of locations be a subset of the original frame, as seen by the parent. Effectively, this allows the child class to prevent the inherited frame from shrinking "too much", while still abiding by the promise given to the clients of the parent. In line with the the rest of the methodology, the check is semantic, which offers maximum flexibility.

As an illustration of this mechanism, we can extend the inherited frame of `wipe_out` in class `STACK` by adding a clause **modify model** [sequence] **Current**. This will allow the implementation of `wipe_out` to modify `sequence`, while generating a proof obligation that **Current**.`sequence` is contained in the write set of `CONTAINER`'s `wipe_out` (which is true, since $\mathtt{sequence} \notin \mathsf{fmq}(\mathtt{CONTAINER})$).

To further reduce the annotation overhead in a common case of replacing parent's model query, AutoProof provides the following shortcut: annotating a child's model query m with **replace** m' will automatically add m to all inherited abstract frames that mention m' (even if the enclosing routine is not redefined). The soundness proof obligation in this case is reduced to a simple static check that $m \notin \mathsf{fmq}(P)$, for any parent class P, which implies that m could not be excluded by any inherited abstract frame. In terms of our motivating example, it is enough to equip `sequence` with a clause **replace** `bag` to achieve the desired framing for routine `wipe_out`.

As a side-effect of abstract framing, it is not sufficient anymore to re-verify an inherited routine r only if it wraps the **Current** object, since r might call another routine s, which has been redefined and given a larger frame. This issue can be avoided by abandoning arbitrary frame redefinitions in favor of the **replace** mechanism; this, however, goes against the semantic nature of the rest of the verification methodology. At the moment, AutoProof re-verifies all inherited routines, which seems to work reasonably well in practice.

6.4 Experimental Evaluation

We added the support for models—most notably, abstract framing—to AutoProof, and used it to verify the core of the EiffelBase2 container library (see Chapter 2). For each verified class of EiffelBase2, in addition to the interface specifications written earlier, we had to provide auxiliary annotations connecting the class model to its concrete state, and describing object structures according to the semantic collaboration methodology. The annotated source code is available from the `verified` directory in the repository [40].

The library verification effort is still ongoing. To date, 17 out of the 61 container library classes (28%) have been verified. These numbers include widely used data structures, such as `LINKED_LIST`, together with their iterators, as well as various deferred classes at top levels of abstraction.

For each of the verified classes, Tab. 6.1 lists its total SIZE; the num-

Table 6.1: EiffelBase2 verification results.

Class	Size	Tokens			Routines		Time
	LOC	Exec	Spec	$\frac{\text{spec}}{\text{exec}}$	Exec	Spec	(sec)
CONTAINER	136	140	294	2.1	3	1	4.6
INPUT_STREAM	74	49	157	3.2	1	0	3.8
OUTPUT_STREAM	102	111	397	3.6	2	0	4.4
ITERATOR	286	212	723	3.4	5	0	5.6
MAP	74	72	168	2.3	3	1	5.0
MAP_ITERATOR	57	54	157	2.9	5	0	5.9
SEQUENCE	241	224	905	4.0	11	2	8.5
SEQUENCE_ITERATOR	54	95	47	0.5	7	0	6.9
MUTABLE_SEQUENCE	239	352	1'169	3.3	19	0	19.8
IO_ITERATOR	67	54	169	3.1	9	0	8.9
MUTABLE_SEQUENCE_ITERATOR	44	54	47	0.9	9	2	11.8
LIST	418	347	1'719	5.0	22	5	57.6
LIST_ITERATOR	133	80	745	9.3	10	0	13.7
CELL	27	32	16	0.5	1	0	4.6
LINKABLE	35	46	28	0.6	2	0	5.5
LINKED_LIST	734	970	2'890	3.0	38	11	210.1
LINKED_LIST_ITERATOR	574	558	1'507	2.7	34	1	114.5
Total	3'295	3'450	11'138	3.2	181	23	491.2

ber of TOKENS in EXECutable code and SPECifications, and the annotation overhead as their ratio; the number of EXECutable and SPECification ROUTINES verified; and the verification time in AutoProof. Specification routines are logic functions, lemma procedures, and procedures added specifically for modifying ghost state. Since AutoProof re-verifies inherited features, the routine count and the verifications times correspond to "flattened" versions of classes.

The average annotation overhead is 3.2, which is consistent with our previous experience with AutoProof (see Chapter 5). The highest overhead corresponds to deferred classes with complex specifications and little or no executable code (such as LIST and LIST_ITERATOR). This, however, pays off in their descendants (such as LINKED_LIST and LINKED_LIST_ITERATOR), which provide the routine implementations, while reusing the inherited specification to a large extent; the payoff is particularly significant if a deferred class has multiple implementations. Our specifications make extensive use of the ownership features of the invariant methodology, and rely on its collaboration features to implement iterators. Verifying all 17 classes with AutoProof takes under 9 minutes.

The main **limitation** of the effort at the moment is due to insufficient support for *agents* in AutoProof, which hinders the verification of higher-order routines, as well as classes that rely on agents to encode equivalence and order relations on their elements (instead, we verify simplified versions of such classes without agents). Integrating complete support for agents into AutoProof and verifying the corresponding parts of EiffelBase2 are both parts of future work.

6.5 Related Work

This section reviews previous work in three areas, corresponding to the three challenges addressed in this chapter: encoding of model classes, encoding of model queries, and abstraction techniques for framing.

Model classes. The technique of mapping model classes directly to theories of the underlying reasoning engine was pioneered in JML [23, 69, 70]. Darvas and Müller [33] propose a flexible notation for specifying such mappings inside model classes. Their notation inspired AutoProof's `maps_to` clauses, but, to our knowledge, it was not implemented in an auto-active verifier and did not target tight integration of model classes into the specification language, which is a contribution of the present work.

Defining semantics of model classes is listed among the specification challenges in [70]. One direction of previous work explores the relation between the representation of model classes in the source language and their encoding in the reasoning engine. For example, the technique by Darvas and Müller [33, 34] allows one to prove that the encoding of a model class is consistent with its contracts. The present work is not concerned with contracts of model classes, and considers it acceptable to relay their semantics to programmers informally. A more pressing issue is verifying that the implementation of a model class, used for runtime checking, is consistent with its Boogie encoding. Apart from ensuring that static and dynamic verification techniques always produce the same results, this might uncover unsound axioms in our Boogie theories (in contrast, Darvas and Müller assume that the background theories are consistent). We leave verification of model classes to future work.

Model queries. A lot of related work studies the semantics of *model variables* (or *model fields*)—a specification construct present in JML [21, 24] and other notations [72, 91]. A model variable has the appearance of a variable to clients, but its value is defined by a constraint (for example, using JML's `represents` clause), and need not be explicitly updated.

Early verification techniques for model variables [72, 17] suffered from a number of soundness and modularity issues, mentioned in Sect. 6.3.2. Leino and Müller [78] propose an approach that addresses those issues in the context of the Boogie methodology (see Sect. 5.3) by encoding model variables as attributes and automatically updating them whenever the enclosing object is being wrapped, to avoid bookkeeping. In order to detect unsatisfiable constraints, they require the reasoning engine to come up with a concrete solution, which unfortunately is not practical for nontrivial constraints. The present work encodes model queries as regular attributes, and uses a simple heuristic to reduce the bookkeeping overhead in simple cases, without jeopardizing automated reasoning in general. As an additional benefit, our straightforward encoding does not require any special syntax (such as `represents` clauses) or admissibility rules; in particular, the definition of a valid model variable constraint proposed in [78] is subsumed by our invariant admissibility.

Abstract Framing. Abstraction techniques for specifying heap topologies and frames are the focus of a whole area of verification research; they appear as a specifications challenge in [70], while the most prominent existing solutions are surveyed in detail in [50]. One family of solutions, which includes dynamic frames [60, 74] and region logic [6], represents frames explicitly, using specification expressions of type "set of objects" or "set of locations"; in this case, regular abstraction mechanisms, such as, logic functions, pure methods, or ghost variables maybe be used to make frame representations abstract. Another family of solutions, represented by separation logic [107, 99] and implicit dynamic frames [116, 79], uses specialized logics, where each specification expression is implicitly associated with a set of locations; abstraction is achieved through logic functions (in particular, predicates). Finally, ownership-based techniques offer a specialized abstraction mechanism for frames: closure under ownership domains (see Sect. 5.4.1).

The goal of any framing methodology is to provide a concise notation for describing unbounded sets of locations (normally made up by a *fixed* set of attributes of an *unbounded* set of objects). Some notations (e.g. Dafny [74]) only support the level of granularity of a whole object, which already goes a long way, since the intra-object structure is finite and can be specified by other means. At the same time, most approaches to framing can incorporate partial objects seamlessly: for example, in dynamic frames, sets of locations can be treated as first-class objects [117, 115], while separation logic and implicit dynamic frames use individual locations in the interpretation of their predicates.

The present work follows a different path, of separating frame descriptions into the object dimension (expressed using a combination of dynamic frames and ownership), and the attribute dimension (represented via fixed lists). To achieve abstraction for the latter dimension, we propose listing model queries instead of concrete attributes, which integrates well with the rest of the model-based specification methodology. A similar technique is used in JML [24], which allows model variables in `assignable` clauses. Each such model variables is associated with a data group, inferred from its `represents` clause. Thus JML's abstract frames are more restrictive than ours: they only allow changing concrete locations that are part of the representation of listed model variables, while AutoProof does not restrict updates to concrete locations. In our experimental evaluation we did not encounter examples that would benefit from the more restrictive semantics; thus we opted for the simpler interpretation.

6.6 Summary and Future Work

This chapter presented an extension of our basic verification methodology with support for models. The extension combines a flexible mechanism for seamless integration of new mathematical constructs into the specification language, a simple yet relatively concise encoding of model queries, and a data abstraction technique for frame specifications, which is natural to use in model-based contracts. We implemented all three features as part of the AutoProof verifier, and used it to prove correctness of a significant subset of the EiffelBase2 library, which serves as evidence of the feasibility and usefulness of the proposed techniques.

One direction of **future work** is finishing the verification of EiffelBase2, which, as discussed above, requires further extending AutoProof with support for agents. Another direction is verifying model classes, that is, checking consistency of their implementations and their Boogie encoding. A possible approach is to associate with each model class `M` a single model query (of type `M`), and use it to express postconditions of `M`'s routines, which can then be verified in the usual way. For example, the class `MML_SET [G]` can be implicitly equipped with a model query `set: MML_SET [G]`, and every routine r of the class can be automatically supplied with a postcondition `Result.set` $=$ $[\![r]\!]$(`Current.set`), where $[\![r]\!]$ is the Boogie function prescribed by r's `maps_to` clause. `MML_SET` is implemented using EiffelBase2 arrays, which in turn are equipped with model-based contracts, expressed in terms of a model query `map`; after adding `set = array.map.range` to the class invariant, we can proceed with verification of `MML_SET` as if it were a regular container class.

Chapter 7

Debugging Failed Verification Attempts

Practical methodologies, like the one proposed in Chapters 5 and 6, are necessary but insufficient to make deductive verification usable. This chapter tackles another crucial aspect of the verification process: understanding and debugging failed proof attempts.

7.1 Introduction

One of the biggest remaining obstacles to usable program verification is understanding failed proof attempts [76]. Deductive verification techniques—such as those discussed in Chapters 5 and 6—are necessarily incomplete, since they target undecidable problems. Incompleteness implies that program verifiers are "best effort": when they fail, it is no conclusive evidence of error. It may as well be that the specification is sound but insufficient to prove the implementation correct; for example, a loop invariant may be too weak to establish the postcondition. Even leaving the issue of incomplete specifications aside, the feedback provided by failed verification attempts is often of little use to understand the ultimate source of failure. A typical error message states that some executions might violate a certain assertion but, without concrete input values that trigger the violation, it is difficult to understand which parts of the programs should be adjusted. And even when verification is successful, it would still be useful to have "sanity checks" in the form of concrete executions, to increase confidence that the written specification is not only consistent but sufficiently detailed to capture the intended program behavior.

Dynamic verification techniques are natural candidates to address these shortcomings of static program proving, since they can provide concrete executions that conclusively show errors and help narrow down probable causes. For example, a concrete execution of a loop that violates its invariant but satisfies its postcondition shows that the invariant must be fixed, while increasing our confidence that the implementation is correct. Traditional dynamic techniques based on *testing* are, however, poor matches to the capabilities of static provers. Testing, at best, targets lightweight executable specifications, such as contracts. Program provers, in contrast, work with very expressive specification and implementation languages supporting features such as nondeterminism, unbounded quantification, infinitary structures (sets, sequences, etc.), and complex first- or even higher-order axioms; none of these is *executable* in the traditional sense. As we argue in Sect. 7.2, however, even relatively simple programs may require such complex specifications. Program provers also support modular verification, where sufficiently detailed specifications of modules or routines are used in lieu of missing or incomplete implementations; this is another scenario where runtime techniques fall short because they require complete implementations.

In this chapter, we propose a technique to generate executions of programs annotated with complex specifications using features commonly supported by program provers: nondeterminism, unbounded quantification, partial implementations, etc. The technique combines symbolic execution with SMT constraint solving to generate small and readable test cases that expose errors (failing executions) or validate specifications (passing executions).

The proposed approach supports executing both imperative and declarative program elements, which accommodates the *implementation* semantics of loops and procedure calls, defined by their bodies, as well as their *specification* semantics, used in modular verification, where the effect of a procedure call is defined solely by the procedure's pre- and postcondition and the effect of a loop by its invariant. The implementation semantics is useful to discriminate between inconsistent and incomplete specifications; while the specification semantics makes it possible to generate executions in the presence of partial implementations, as well as to expose spurious executions permitted by incomplete specifications.

Existing static and dynamic symbolic execution techniques are not directly applicable to programs we target, since path constraints in such programs include unbounded quantification and other complex specification constructs, listed above. An attempt to solve such constraints directly, without additional guidance in the form of quantifier instantiation heuristics, often leads to the solver getting bogged down. Our technique simplifies the constraints passed to the SMT solver, instantiating quantifiers and unrolling recursive definitions, targeting the values required for a particular symbolic execution. The simplification greatly improves the predictability of test case generation. Combined with model minimization techniques, it produces short—often minimal-length—executions that are quite easy to read. While constraint simplification might also produce false positives (infeasible executions), the evaluation of Sect. 7.5 shows that this rarely happens in practice: the small risk amply pays off by producing easy-to-understand executions, symptomatic of the rough patches in the implementation or specification that require further attention. We also identify a subset of the annotation language for which no infeasible executions are generated.

We implemented our technique for the Boogie intermediate verification language, used as back-end of AutoProof, introduced in previous chapters, as well as numerous other program verifiers [29, 74, 75]. Working atop an intermediate language opens up the possibility of reusing the tool with multiple high-level languages and verifiers that already translate to Boogie. It also ensures that our technique is sufficiently general: Boogie is a small yet very expressive language (including both specification and imperative constructs), designed to support translations of disparate notations with their own supporting methodologies. Our implementation is available as a tool called Boogaloo, available for download [15] and through a web interface [31]. For simplicity, hereafter we will use "Boogaloo" to denote the execution generation technique as well as its implementation, and will employ the self-explanatory Boogie syntax in the examples.

7.2 A Motivating Example

We give a concise overview of the capabilities of Boogaloo using a simple verification example: finding the maximum element in an integer array.[1] Fig. 7.1 shows a Boogie procedure `Max`, which inputs an integer `N`, denoting the array size, and a map[2] `a` that represents the array elements `a[0]`, . . ., `a[N−1]`; it returns an integer `max` for `a`'s maximum. The procedure includes specification in the form of two postconditions (**`ensures`**), formalizing the definition of maximum: `max` should be no smaller than any element of `a` (line 2); and it should be an element of `a` (line 3).

What happens if we try to verify procedure `Max`, as shown in Fig. 7.1, using Boogie? Verification fails with a vague error message ("`Postconditions on lines 2 and 3 might not hold.`") which is inconclusive and of little help to

[1]The tool output messages in this section are abridged without sacrificing the gist of the original.

[2]In general, maps have an infinite domain in Boogie.

```
1  procedure Max(N: int, a: [int] int) returns (max: int)
2    ensures (∀ j: int • 0 ≤j ∧j < N ⟹a[j] ≤max);
3    ensures (∃ j: int • 0 ≤j ∧j < N ∧a[j] =max);
4  {
5    var i: int;
6    i :=0;
7    max :=0;
8    while (i < N) {
9      if (a[i] > max) { max :=a[i]; }
10     i :=i + 1;
11   }
12 }
```

Figure 7.1: Boogie procedure `Max` that finds the maximum element in an array. Both the specification and the implementation contain errors, and no loop invariant is provided.

understand the source of failure. Rather, running Boogaloo on the same input generates concrete inputs that make the program fail; we get the message "`Postcondition on line 3 violated with N -> 0, a -> [], max -> 0`", which clearly singles out a problem with `Max`: the maximum of an empty array is undefined.

We can formalize the intuition that `Max` is undefined for empty arrays as the precondition **requires** `N` > 0. Boogie's output, however, does not change if we add this precondition: it still cannot establish either postcondition since it would need a loop invariant to reason about loops—no matter how simple they are. Instead, running Boogaloo on `Max` annotated with the precondition shows another input that triggers a failure: "`Postcondition on line 3 violated with N -> 1, a -> [0 -> -1], max -> 0`". This time the problem is with the implementation rather than the specification: when `a` contains a single negative value, initializing `max` to 0 (line 7) does not work. With this concise concrete counterexample, it is easy to understand that the same problem occurs with any array containing only negative elements. Designing a correction is also routine: we change the initialization on line 7 to `max` :=a[0], which is well-defined thanks to the precondition `N` > 0.

We can see that the modified program—including precondition and new initialization of `max`—is finally correct. However, Boogie's behavior on it does not change at all: without a loop invariant, it still fails to prove either postcondition. Boogaloo, in contrast, can generate a number of test cases (1024 by default, which takes just a few seconds with the running example on standard desktop hardware) and successfully check all of them against the specification. While this still falls short of a formal correctness proof, it provides evidence that the program is indeed correct, and that all we have to do is strengthen the specification by adding a suitable loop invariant.

While we selected a simple example which can be briefly presented, we were able to demonstrate, in a nutshell, several fundamental issues of working with static program verifiers such as Boogie, and how Boogaloo can complement their weaknesses. Specifically, Boogaloo's capabilities to provide concrete inputs that show errors or amass evidence for correctness; and to work with the same programs used for verification including elements such as first-order quantification (lines 2 and 3), but without requiring specifications at all costs (a loop invariant). Another distinguishing, and practically crucial, feature of Boogaloo is that it produces small (often minimal) tests: in the example, the smallest arrays and the smallest integer values exposing faults and discrepancies.

7.2.1 Comparison with other approaches

To further demonstrate the unique features of Boogaloo, let us consider the behavior of other approaches to complementing static program verification on the same example of procedure `Max`.

Assuming `Max` were a Boogie encoding produced from some high-level programming language, we could use standard testing tools on the source program to generate concrete inputs and discover failures. One problem is that first-order quantifications (and other features used by Boogie) are inexpressible using the simple Boolean expressions of standard programming languages. While the quantifications used in `Max` are bounded, and hence expressible using executable constructs such as finite iterations over arrays or list comprehensions, getting rid of quantifiers and other non-executable constructs is neither possible nor desirable in general. As soon as we look at examples more complex than `Max`, we need to express abstract properties potentially involving infinitely many elements, such as for framing and for reasoning about unbounded sequences of pointers to heap-allocated data. Even in an example as simple as sorting, if a sorting procedure takes a function pointer as argument to denote the comparison function, we need to express that it encodes a total order—something involving quantification over a potentially unbounded domain. More generally, we designed Boogaloo to work with the same proof-oriented annotated programs used by static verifiers, which involve features difficult to execute and normally not found in high-level programming languages.

Another option to debug `Max` is using the Boogie Verification Debugger (BVD [46]), which extracts concrete counterexamples from failed verification attempts. The relevance of such counterexamples is, however, limited in the presence of loops and procedure calls with incomplete specifications. On

`Max` as in Fig. 7.1, BVD returns the assignment "`N = 1, a = [], max = -900`"; after adding `N` $>$ `0` as precondition, it returns "`N = 797, a = [], max = -900`"; after fixing the implementation, it returns "`N = 797, a = [0 -> -901], max = -901`". These examples fail to point out the two errors in `Max`, because according to modular reasoning [73] and in the absence of an invariant, any loop is equivalent to assigning arbitrary values to program variables. While BVD's modular semantics helps debug incompleteness in specifications, it also enforces an "all-or-nothing" development style, where developers first have to get right the most complicated part (the invariants), before they can proceed with debugging the rest of the program. This lack of incrementality is what makes modular verification so hard in the first place.

It is possible to make Boogie use loop and procedure bodies instead of their specification by *unrolling* loops and *inlining* procedures U times, for a given $U \geq 0$. With unrolling, BVD finds counterexamples for executions where `N` $\leq U$, and in particular the same counterexamples constructed by Boogaloo. The approach, however, has its limitations. First, unrolling and inlining require users to guess a suitable U; since all longer executions are ignored, verification vacuously succeeds when the shortest counterexample requires $> U$ iterations or nested calls, without providing any concrete feedback. Second, unrolling of complex loops and inlining of recursive procedures scale poorly, as they consist of literally rewriting the code U times; Boogaloo, in contrast, uses symbolic execution techniques, which are less likely to incur blow up. Building a debugger on top of the Boogie verifier also means that it cannot generate passing executions (Boogie does not produce a model in case verification succeeds) and cannot help when the theorem prover gets bogged down. In contrast, Boogaloo uses simpler verification conditions, designed for predictable generation and readability of counter examples as opposed to sound proofs.

7.3 A Runtime Semantics of Boogie Programs

This section describes the syntax of Boogaloo programs (Sect. 7.3.1) and their operational semantics (Sect. 7.3.2). We use the following notation: $\mathbb{Z}$ is the set of mathematical integers; and $\mathbb{B}$ is the set $\{\top, \bot\}$ of Boolean values. A *map* m is a mathematical function from a domain $D_1 \times \cdots \times D_n$, for $n > 0$, to a codomain D_0; square brackets denote map applications. Whenever convenient, we see m as a set of $(n+1)$-tuples: $m \subset D_1 \times \cdots \times D_n \times D_0$ such that $(d_1, \ldots, d_n, d_0) \in m$ iff $m[d_1, \ldots, d_n] = d_0$. $\mathrm{dom}(m)$ and $\mathrm{rng}(m)$ denote the *domain* and *range* of m; m is *total* if $\mathrm{dom}(m) = D_1 \times \cdots \times D_n$, and *finite* if $|\mathrm{dom}(m)| \in \mathbb{Z}$; $m[d_1, \ldots, d_n \mapsto d]$ denotes a map m' identical to m except that $m'[d_1, \ldots, d_n] = d$. We overload this notation to denote variable substitution: if $e, y_1, \ldots, y_n$ are expressions, and $x_1, \ldots, x_n$, are distinct variable names, $e[x_1, \ldots, x_n \mapsto y_1, \ldots y_n]$ denotes e with all occurrences of x_k replaced by y_k, for $k = 1, \ldots, n$.

7.3.1 Input Language

Boogaloo desugars generic Boogie programs [73] into the simpler language described in Fig. 7.2. Programs P are lists of declarations D, whose order is immaterial. Declarations include uninterpreted `type`s, global `var`iables, and `procedure`s with input parameters, output parameters (`returns`), global variables the procedure may modify (`modifies` clause), and body (between braces). Procedure bodies consist of local variable declarations followed by a list of labeled statements S. The latter include sequential composition, regular and nondeterministic assignment ($:=$ and `havoc`, possibly in parallel to multiple variables), procedure `call`, `assume`, nondeterministic `goto` a set of label identifiers, and abrupt `return` to the caller procedure, as well as *directives* R described shortly. Expressions must be properly typed as `bool`eans, `int` egers, maps $[T_1, \ldots, T_n]T_0$ from arbitrary domain $(T_1, \ldots, T_n)$ and codomain T_0 types, and user-defined uninterpreted types[3]. Expressions E include literal constants C, variables V, map applications $m[t_1, \ldots, t_n]$, map updates $m[t_1, \ldots, t_n := t]$, `old` expressions which refer to the value of an expression when the procedure was entered, plus the usual applications of unary operators *UOp*, binary operators *BOp*, a ternary `if`/`then`/`else` operator, and quantifications and lambda expressions *QOp*.

The *directives* `halt`, `abort`, and `pick` are Boogaloo-specific and characterize symbolic executions: `halt` terminates the current execution with success (marking *passing* executions); `abort` also terminates the current execution but with error (marking *failing* executions); `pick` forces the interpreter to resolve nondeterminism by trying to build a concrete state out of the current symbolic constraints. Boogaloo automatically inserts a `halt` at the end of every control path in the input program; and uses `abort` to desugar `assert` statements as follows. A Boogie statement `assert B`, where `B` is a Boolean expression, indicates that `B` must hold in every correct execution reaching the statement; `assume B`, on the other hand, indicates that only executions where `B` holds upon reaching the `assume` are feasible. Therefore, Boogaloo expresses the semantics of `assert B` using `assume`, `abort`, and nondeterministic choice as follows:

[3]While Boogaloo supports Boogie's type constructors with arguments, as well as type parameters in procedures and maps, we do not include them in the discussion for simplicity.

$$
\begin{array}{rcl}
P & ::= & D^* \\
D & ::= & \mathtt{type}\ tid \mid \mathtt{var}\ V : T \\
 & \mid & \mathtt{procedure}\ pid\ (\ \langle V : T\rangle^*\ )\ \mathtt{returns}\ (\ \langle V : T\rangle^*\ )\ \mathtt{modifies}\ (V^*) \\
 & & \langle \{\ \langle V : T\rangle^*\ \langle lid : S\rangle^*\ \}\rangle^? \\
S & ::= & S\ ;\ S \mid \mathtt{havoc}\ V^+ \mid V^+ := E^+ \mid \mathtt{call}\ V^* := pid\ (E^+) \\
 & \mid & \mathtt{assume}\ E \mid \mathtt{goto}\ lid^+ \mid \mathtt{return} \mid R \\
R & ::= & \underline{\mathtt{halt}} \mid \underline{\mathtt{abort}} \mid \underline{\mathtt{pick}} \\
T & ::= & \mathtt{bool} \mid \mathtt{int} \mid [\,T^+\,]\,T \mid tid \\
E & ::= & C \mid V \mid E\,[\,E^+\,] \mid E\,[\,E^+ := E\,] \mid \mathtt{old}\ E \\
 & \mid & UOp\ E \mid E\ BOp\ E \mid \mathtt{if}\ E\ \mathtt{then}\ E\ \mathtt{else}\ E \mid QOp\ \langle V : T\rangle^+ \bullet\ E \\
V & ::= & vid \\
C & ::= & \mathtt{false} \mid \mathtt{true} \mid 0 \mid 1 \mid 2 \mid \cdots \\
UOp & ::= & - \mid \neg \\
BOp & ::= & + \mid - \mid \cdots \mid < \mid \leq \mid = \mid \cdots \mid \wedge \mid \vee \mid \cdots \\
QOp & ::= & \exists \mid \forall \mid \lambda
\end{array}
$$

Figure 7.2: Desugared language supported by Boogaloo, consisting of programs P, declarations D, statements S, types T, and expressions E. Angular brackets $\langle\ \rangle$ are part of the grammar metalanguage, used to mark optional (?) or repeated ($*$, $+$) expressions.

```
goto T, F;
F: assume ¬B; abort;
T: assume B;
```

Boogaloo also injects a `pick` statement right before every `halt` and `abort`, so that every terminating execution gets a concrete state. Boogaloo automatically instruments programs with the directives R, based on different strategies (see Sect. 7.5) so that one can use Boogie programs without additional annotations.

The rest of the desugaring of Boogie into the language of Fig. 7.2 is fairly standard. We rewrite function declarations `function` f$(T_1, \ldots, T_n)$ `returns`(T_0) into constants `const` f: $[T_1, \ldots, T_n]\,T_0$ of map type, and express the corresponding function definitions as axioms. In turn, we express axioms and other specification constructs—`where` clauses (used to constrain the values of uninitialized variables), pre- and post-conditions, and loop invariants—using `assume` and `assert` reflecting the standard semantics. We replace constants with variables. Finally, we transform procedure bodies into sets of *basic blocks* (labeled sequential blocks of code that end with a `return` or `goto`) using standard techniques [73]. For example, the Boogie loop at lines 8–11 in

Fig. 7.1 roughly becomes:

```
         goto done, body;
body :   assume i < N; goto thn, els;
 thn :   assume a[i] > max; max :=a[i]; goto inc;
 els :   assume ¬(a[i] > max); goto inc;
 inc :   i :=i + 1; goto done, body;
done :   assume ¬(i < N);
```

7.3.2 Runtime Operational Semantics

We now describe an operational semantics for the language in Fig. 7.2. The presentation focuses on the most interesting aspects while omitting standard details.

Let us start with an informal overview of the basic concepts. The operational semantics describes the effect, on the symbolic state, of executing statements. The symbolic state associates symbolic values to program variables in scope. Executing some statements may involve enforcing constraints between symbolic values; the most obvious example is that of **assume** P: the symbolic values associated with variables mentioned in P must satisfy P in every computation that continues after the statement. Therefore, the symbolic state includes *constraints* which are updated as execution progresses. Finally, **pick** directives select *concrete* values that satisfy the current constraints; executions continue after **pick** with the selected concrete state components replacing the corresponding symbolic state components (but subsequent statements will be executed symbolically until the next **pick**). In this sense, symbolic executions are *speculative*, in that the constraints may not have a solution (infeasible executions), and *nondeterministic*, in that the constraints may have more than one solution (multiple feasible executions); **pick** forces the interpreter to resolve nondeterministic choice before continuing. Another source of nondeterminism comes from executing **goto**s with multiple labels; such choices are resolved immediately, resulting in explicit path enumeration. Since Boogaloo injects **pick** statements at every terminating location, it can provide concrete input and output values for every feasible execution, while still availing of symbolic representation to limit the combinatorial explosion introduced by the inherently nondeterministic nature of specifications.

The main source of complexity in executing Boogie programs lies in solving constraints, in particular when they involve universal quantifiers and uninterpreted maps with infinite domains. Even though state-of-the-art SMT solvers can decide satisfiability of quantified formulas in many practical cases, they can hardly generate readable "natural" infinite models. In light of these difficulties, we drop Boogie's standard interpretation—where all maps are total—and replace it with a *finitary* interpretation where maps have finite domains. Finite, small instances are sufficient to expose errors and inconsistencies in most programs; Alloy's techniques are based on a similar "small scope" hypothesis [55]. We also treat universally quantified constraints in a special way: the `pick` directive *finitizes* them, that is turns them into simpler quantifier-free constraints. Finitization is in general unsound, but Sect. 7.5 demonstrates that the precision loss is normally acceptable, especially if the goal is finding inconsistencies and errors.

Concrete values. Each Boogie type corresponds to a set of concrete values: `bool` corresponds to $\mathbb{B}$, `int` corresponds to $\mathbb{Z}$; each user-defined `type U` corresponds to a countable uninterpreted set U; each map type $[\mathtt{T}_1, \ldots, \mathtt{T}_n]\, \mathtt{T}_0$ corresponds to the set of all *finite* maps from $T_1 \times \cdots \times T_n$ to T_0, where T_k is the set of concrete values of type $\mathtt{T}_k$, for $k = 0, \ldots, n$. K denotes the union of all concrete value sets.

Symbolic values. The set of symbolic values Σ is defined as:

$$\Sigma \ ::= \ K \mid L \mid \mathit{UOp}\, \Sigma \mid \Sigma\, \mathit{BOp}\, \Sigma \mid \textbf{if}\ \Sigma\ \textbf{then}\ \Sigma\ \textbf{else}\ \Sigma\,,$$

where K is the set of concrete values defined above; unary UOp and binary BOp operators are defined in Fig. 7.2, and L denotes a set of *logical variables* of the same types as the concrete values. A logical variable ℓ of type T corresponds to a symbolic placeholder (a "promise") for a concrete value of type T. To represent quantifiers in constraints, we also introduce a set of *universal symbolic values* $\Sigma_\forall ::= \forall \langle V : T\rangle^{+} \bullet\ \Sigma_V$, where Σ_V is a symbolic value, except it can also include bound variables V. Given a set X of expressions, $\mathrm{LV}(X)$ is the set of all logical variables appearing in X.

Symbolic states. A symbolic state (environment) is a tuple

$$\mathcal{E} = \langle \sigma, \lambda, \mu, \kappa, \upsilon, \tau \rangle,$$

where the store $\sigma : V \to \Sigma$ maps variables to symbolic values; the logical store $\lambda : L \to K$ maps scalar logical variables to concrete values; the map store $\mu : L \to (\Sigma^* \to \Sigma)$ maps map logical variables to symbolic maps; $\kappa \subset \Sigma$ is a finite set of *simple state constraints*; $\upsilon \subset \Sigma_\forall$ is a finite set of *universal state constraints*; and τ is one of ✧, ✓, ✗, denoting an intermediate state (✧), or the final state of a passing (✓) or failing (✗) execution. The map store associates logical variables of map type to *symbolic maps*: finite

$$\text{LOG-IN}\,\frac{\ell \in \mathrm{dom}(\lambda)}{\llbracket \ell \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}}\lambda[\ell]} \qquad \text{LOG-OUT}\,\frac{\ell \notin \mathrm{dom}(\lambda)}{\llbracket \ell \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}}\ell}$$

$$\text{VAR-IN}\,\frac{v \in \mathrm{dom}(\sigma) \qquad \llbracket \sigma[v] \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}} e}{\llbracket v \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}} e} \qquad \text{VAR-OUT}\,\frac{v \notin \mathrm{dom}(\sigma) \qquad \ell \text{ is fresh} \qquad \sigma' = \sigma[v \mapsto \ell]}{\llbracket v \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'} \ell}$$

$$\text{SEL-IN}\,\frac{\llbracket (m,\vec{a}) \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'} (\ell_m, \vec{a}') \qquad \vec{a}' \in \mathrm{dom}(\mu'[\ell_m]) \qquad \llbracket \mu'[\ell_m][\vec{a}'] \rrbracket\ {}^{\mathcal{E}'}{=}^{\mathcal{E}'} e}{\llbracket m[\vec{a}] \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'} e}$$

$$\text{SEL-OUT}\,\frac{\ell \text{ is fresh} \qquad \llbracket (m,\vec{a}) \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}_1} (\ell_m, \vec{a}_1) \qquad \vec{a}_1 \notin \mathrm{dom}(\mu_1[\ell_m]) \qquad m' = [\mu_1[\ell_m][\vec{a}_1] \mapsto \ell]}{\llbracket m[\vec{a}] \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'} \ell \qquad \mathcal{E}' = \langle \sigma_1, \mu_1[\ell_m \mapsto m'], \upsilon_1 \rangle}$$

$$\text{UPD}\,\frac{\ell \text{ is fresh} \qquad \llbracket (m,\vec{a},e) \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}_1} (\ell_m, \vec{a}_1, e_1) \qquad m' = [\mu_1[\ell_m][\vec{a}_1] \mapsto e_1]}{\llbracket m[\vec{a} := e] \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'} \ell \qquad \mathcal{E}' = \langle \sigma_1, \mu_1[\ell \mapsto m'], \upsilon_1 \cup \{\forall \vec{x} \bullet\ \vec{x} \neq \vec{a}_1 \Rightarrow \ell[\vec{x}] = \ell_m[\vec{x}]\} \rangle}$$

$$\text{LAMBDA}\,\frac{\ell \text{ is fresh} \qquad \llbracket e \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}_1} e_1 \qquad \sigma_1(\vec{x}) = \vec{\ell}_1}{\llbracket \lambda\, \vec{x} \bullet\ e \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'} \ell \qquad \mathcal{E}' = \langle \sigma_1 \setminus \{(\vec{x}, \vec{\ell}_1)\}, \mu_1, \upsilon_1 \cup \{\forall \vec{x} \bullet\ \ell[\vec{x}] = e_1[\vec{\ell}_1 \mapsto \vec{x}]\} \rangle}$$

$$\text{QUANT-T}\,\frac{\mathsf{Skolem}[Q_1 x_1 \cdots Q_n x_n \bullet\ q]\ {}^{\mathcal{E}}{=}^{\mathcal{E}_1} \forall \vec{y} \bullet\ q_1 \qquad \llbracket q_1 \rrbracket\ {}^{\mathcal{E}_1}{=}^{\mathcal{E}_2} q_2 \qquad \sigma_2(\vec{y}) = \vec{\ell}}{\llbracket Q_1 x_1 \cdots Q_n x_n \bullet\ q \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'} \top \qquad \mathcal{E}' = \langle \sigma_2 \setminus \{(\vec{y}, \vec{\ell})\}, \mu_2, \upsilon_2 \cup \{\forall \vec{y} \bullet\ q_2[\vec{\ell} \mapsto \vec{y}]\} \rangle}$$

$$\text{QUANT-F}\,\frac{\llbracket \widetilde{Q}_1 x_1 \cdots \widetilde{Q}_n x_n \bullet\ \neg q \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'} \top}{\llbracket Q_1 x_1 \cdots Q_n x_n \bullet\ q \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'} \bot}$$

Figure 7.3: Symbolic evaluation (significant cases).

maps whose domain and range are in Σ; symbolic maps extend their finite domains as execution progresses; `pick` concretizes their domain and range, turning symbolic maps into concrete ones.

Expression evaluation. Let E denote the set of all expressions defined by E in Fig. 7.2 but whose atoms range over $C \cup V \cup L$ (i.e., including logical variables L). The *evaluation* of an expression $e \in \mathsf{E}$ in an environment $\mathcal{E}$ is a symbolic value in Σ. We use the notation: $\llbracket e \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'} e'$ to denote that $e \in \mathsf{E}$ evaluates in $\mathcal{E}$ to $e' \in \Sigma$. As we detail shortly, evaluating an expression may change the environment; correspondingly, $\mathcal{E}'$ denotes the updated environment, whose components are written $\langle \sigma', \lambda', \mu', \kappa', \upsilon', \tau' \rangle$. When convenient, we extend this notation to *sequences* $\vec{e} = e_1, \ldots, e_n$ of expressions, evaluated one after another as follows: $\llbracket \vec{e} \rrbracket\ {}^{\mathcal{E}}{=}^{\mathcal{E}'} \vec{e}'$ iff $\llbracket e_k \rrbracket\ {}^{\mathcal{E}_k}{=}^{\mathcal{E}'_k} e'_k$ for $k = 1, \ldots, n$, $\mathcal{E}'_k = \mathcal{E}_{k+1}$ for $k = 1, \ldots, n-1$, and $\vec{e}' = e'_1, \ldots, e'_n$. Fig. 7.3 shows the evaluation rules for the most interesting expression kinds. Since evaluation does not change the λ, κ, and τ environment components, Fig. 7.3 omits them. Also notice that evaluating a symbolic value never changes the environment, and every concrete value evaluates to itself.

Rules LOG-IN and LOG-OUT describe the simple cases of evaluating a logical variable ℓ: if $\lambda[\ell]$ is defined, it yields ℓ's evaluation; otherwise, ℓ

evaluates to itself.

Rules VAR-IN and VAR-OUT describe the evaluation of a (program) variable v. If it has already been initialized, the evaluation of $\sigma[v]$ gives its symbolic value. Otherwise (VAR-OUT), such as when v enters the scope or after executing `havoc` v, $\sigma[v]$ gets initialized to a fresh logical variable ℓ.

The rules for map selection are similar to those for variables but target the map store μ: if a map selection has already been evaluated, its symbolic value is returned (SEL-IN); otherwise, a fresh logical variable is generated and stored in μ (SEL-OUT).

Rules UPD and LAMBDA deal with evaluating expressions of map type for updates and lambda abstractions. Both rules introduce a fresh map logical variable and add to υ a universally quantified constraint that defines the map. Thus, map expressions (variables, updates, and lambdas) always evaluate to a logical variable; this justifies using the evaluation ℓ_m of m as an index in μ in the premises of SEL-IN, SEL-OUT, and UPD.

The rules for quantified expressions are non-deterministic. Consider an expression $\mathcal{Q} = Q_1 x_1 \cdots Q_n x_n \bullet\, q$ in prenex normal form, where $n > 0$, Q_k is one of $\forall$ and $\exists$ for each k, and q is quantifier-free. Rule QUANT-T evaluates $\mathcal{Q}$ to true and adds it to the universal constraints υ after the following transformation. First, $\mathcal{Q}$ is Skolemized as $\forall \vec{y} \bullet\, q_1$, where $\vec{y}$ is the subsequence of $x_1, \ldots, x_n$ including only those x_k's for which Q_k is $\forall$; $\mathcal{E}_1$ is the environment after Skolemization, which contains fresh logical variables for the Skolem functions introduced by the process. Evaluating q_1 in $\mathcal{E}_1$ yields some q_2 where the bound variables $\vec{y}$ map to fresh logical variables $\vec{\ell}$; after performing the substitution $q' = q_2[\vec{\ell} \mapsto \vec{y}]$, $\forall \vec{y} \bullet\, q'$ is added to υ. In the special case where $\mathcal{Q}$ is purely existential, Skolemization yields a quantifier-free formula, and the corresponding q_2 is directly added to κ. Rule QUANT-F, which evaluates $\mathcal{Q}$ to false, follows by duality ($\widetilde{\forall}$ denotes $\exists$, and $\widetilde{\exists}$ denotes $\forall$).

Procedure call semantics. The precise semantics of procedure calls involves several details to support recursion—mainly, maintaining a stack of environments and correspondingly keeping track of scope. We overlook these tedious aspects and focus on the gist of the semantics of a call to a generic procedure `P` (before desugaring):

$$\texttt{procedure P } (\vec{a})\,\texttt{returns } (\vec{o})\,\texttt{requires } p \;\; \texttt{ensures } q \;\; \texttt{modifies}(\vec{m}) \quad \langle\{B\}\rangle^?$$

with formal input $\vec{a}$ and output $\vec{o}$ parameters, modified global variables $\vec{m}$, body B, and pre- and postcondition p and q. The desugaring of Sect. 7.3.1 turns pre- and postcondition into checks at the call site:

$$\texttt{assert } p[\vec{a} \mapsto \vec{u}];\ \texttt{call } \vec{v} := \texttt{P}(\vec{u});\ \texttt{assume } q[\vec{a}, \vec{o} \mapsto \vec{u}, \vec{v}];$$

$$\text{SEQ}\ \frac{\tau = \text{✧} \qquad \mathcal{E} \ {-}\mathtt{I}{\rightsquigarrow}\ \mathcal{E}_1 \qquad \mathcal{E}_1 \ {-}\mathtt{J}{\rightsquigarrow}\ \mathcal{E}'}{\mathcal{E} \ {-}\mathtt{I};\mathtt{J}{\rightsquigarrow}\ \mathcal{E}'} \qquad \text{GOTO}\ \frac{\tau = \text{✧} \qquad k \in \{1, \dots, n\} \qquad \mathcal{E} \ {-}@\mathsf{x}_k{\rightsquigarrow}\ \mathcal{E}'}{\mathcal{E} \ {-}\mathbf{goto}\ \mathsf{x}_1, \dots, \mathsf{x}_n{\rightsquigarrow}\ \mathcal{E}'}$$

$$\text{RETURN}\ \frac{\tau = \text{✧} \qquad \mathcal{E} \ {-}@\mathsf{caller}{\rightsquigarrow}\ \mathcal{E}'}{\mathcal{E} \ {-}\mathbf{return}{\rightsquigarrow}\ \mathcal{E}'} \qquad \text{ASSUME}\ \frac{\tau = \text{✧} \qquad [\![\mathsf{P}]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ p}{\mathcal{E} \ {-}\mathbf{assume}\ \mathsf{P}{\rightsquigarrow}\ \langle \sigma', \lambda', \mu', \kappa' \cup \{p\}, \upsilon', \tau' \rangle}$$

$$\text{HAVOC}\ \frac{\tau = \text{✧}}{\mathcal{E} \ {-}\mathbf{havoc}\ \mathsf{v}{\rightsquigarrow}\ \langle \sigma \setminus \{(\mathsf{v}, \sigma[\mathsf{v}])\}, \lambda, \mu, \kappa, \upsilon, \tau \rangle}$$

$$\text{ASSIGN}\ \frac{\tau = \text{✧} \qquad [\![\mathsf{e}]\!]\ {}^{\mathcal{E}}{=}^{\mathcal{E}'}\ e'}{\mathcal{E} \ {-}\mathsf{v} :=\mathsf{e}{\rightsquigarrow}\ \langle \sigma[\mathsf{v} \mapsto e'], \lambda', \mu', \kappa', \upsilon', \tau' \rangle}$$

$$\text{HALT}\ \frac{\tau = \text{✧}}{\mathcal{E} \ {-}\underline{\mathbf{halt}}{\rightsquigarrow}\ \langle \sigma, \lambda, \mu, \kappa, \upsilon, \checkmark \rangle} \qquad \text{ABORT}\ \frac{\tau = \text{✧}}{\mathcal{E} \ {-}\underline{\mathbf{abort}}{\rightsquigarrow}\ \langle \sigma, \lambda, \mu, \kappa, \upsilon, \text{✗} \rangle}$$

$$\text{PICK}\ \frac{\begin{array}{ccc} \tau = \text{✧} \quad \kappa' = \kappa \cup \kappa_\mu \cup \Phi(\upsilon) & & \mathrm{dom}(\Lambda) = \{\ell \in \mathrm{LV}(\kappa') \mid \ell \text{ is scalar}\} \\ \mathcal{E}' = \langle \sigma, \lambda \cup \Lambda, \mu, \emptyset, \upsilon, \tau \rangle & & [\![\bigwedge \kappa']\!]\ {}^{\mathcal{E}'}{=}^{\mathcal{E}'}\ \top \end{array}}{\mathcal{E} \ {-}\underline{\mathbf{pick}}{\rightsquigarrow}\ \mathcal{E}'}$$

Figure 7.4: Symbolic execution: operational semantics. All rules describe transformations of a generic symbolic state $\mathcal{E} = \langle \sigma, \lambda, \mu, \kappa, \upsilon, \tau \rangle$.

(For brevity, we do not discuss the handling of **old** expressions in postconditions.) It also generates a modified procedure body B' to reflect the implementation or specification semantics, according to whether P has a body or not: if B is defined, B' adds an **assert**q before each **return** statement in B; if B is not defined, B' consists of the single statement **havoc**$\vec{o}, \vec{m}$ (which, combined with assuming the postcondition at the call site, renders the specification semantics). The effect of the call statement is then given by $B'[\vec{a} \mapsto \vec{u}]$@entry where $B'[\vec{a} \mapsto \vec{u}]$ is a shorthand for B' with actual arguments replaced for formals and @entry denotes the basic block of statements at procedure P's entry. Even though Boogaloo defaults to implementation semantics whenever a body is available, one can always switch to the specification semantics for a particular procedure by commenting out its body.

Operational semantics. Fig. 7.4 describes the operational semantics of statements other than procedure calls, using the notation $\mathcal{E} \ {-}\mathsf{s}{\rightsquigarrow}\ \mathcal{E}'$ to denote that executing statement s changes the environment $\mathcal{E}$ into $\mathcal{E}'$. Rules are applicable only if $\tau = \text{✧}$, that is if the computation has not terminated yet; after rules HALT or ABORT have changed τ to passing ✓ or failing ✗ no rule is applicable and hence the computation terminates.

Rules SEQ for sequential composition, GOTO for branch, and RETURN for abrupt termination are standard, using the notation @caller to denote the basic block beginning after the current call in the caller procedure. Rule GOTO is clearly nondeterministic.

Rule ASSUME adds the assumed Boolean formula to the set κ of state constraints. Rule HAVOC "forgets" the symbolic value of the variable `v`, as if uninitialized. Rule ASSIGN updates the symbolic value in σ associated to the assigned variable `v`.

The most interesting rule is PICK, which details how **`pick`** concretizes symbolic states. It extends κ into κ', adding map instance constraints $\kappa_\mu = \{m[\vec{x}] = y \mid (m, \vec{x}, y) \in \mu\}$, which express the information in μ about symbolic maps; as well as *finitized* universal constraints $\Phi(\upsilon)$.s It then picks a *solution* $\Lambda : L \to K$ of κ': an assignment of concrete values to scalar logical variables for which the conjunction of constraints in κ' evaluates to true. It finally adds the picked solution to λ and drops the solved constraints. The rule is nondeterministic, as κ' might have multiple solutions. When κ' has no solutions, the rule cannot apply and executions gets stuck at **`pick`**: we call such executions *infeasible*. The rule is also agnostic with respect to the exact method of solving simple constraints, as well as to the finitization mapping Φ. To solve constraints, one can leverage an external constraint solver or even a brute force enumeration; even if the solving process is unsound, one can always evaluate κ' in $\mathcal{E}'$ and discard solutions that do not simplify to $\top$. The only requirement on $\Phi : \Sigma_\forall{}^* \to \Sigma$ is that it is an over-approximation: any valid solution of υ is also a solution of $\Phi(\upsilon)$. In practice, Φ performs quantifier instantiation: it replaces a quantified formula $\forall \vec{x} \bullet q$ with a finite set of quantifier-free formulas $\{q[\vec{x} \mapsto \vec{e}] \mid e \in R\}$, for some finite set R of "relevant" symbolic values. The challenge is to choose an R that is large enough to describe all relevant values in the current environment, yet small enough to produce constraints that can be solved efficiently. Sect. 7.4 gives more details about Boogaloo's finitization procedure.

Boogaloo vs. Boogie semantics. How does the operational semantics discussed in this section compare with the original Boogie semantics? For this discussion, a *semantics* of a program P is a set of sequences of concrete states, corresponding to its feasible terminating executions; a (concrete) state $\mathcal{C} = \langle \sigma, \tau \rangle$ consists of a store σ (involving finitely many variables) and a termination flag $\tau \in \{✧, ✓, ✗\}$. A state $\mathcal{C}$ is *finitary* if it involves only finite maps: for all $m \in \text{dom}(\sigma)$, $|\text{dom}(\sigma[m])|$ is finite; otherwise, it is *infinitary*. A state $\mathcal{C}_F$ *finitizes* another state $\mathcal{C}$ (written $\mathcal{C}_F \sqsubseteq_{\mathcal{F}} \mathcal{C}$) iff $\mathcal{C}_F$ is finitary, $\tau_F = \tau$, $\text{dom}(\sigma_F) = \text{dom}(\sigma)$ and, for all map variables $m \in \text{dom}(\sigma)$, $\sigma_F[m] \subseteq \sigma[m]$. A sequence e of states is finitary (infinitary) if all its elements are finitary (infinitary); e finitizes another sequence e' if every state of e finitizes the corresponding state of e'.

For a program P, $\mathcal{I}[P]$ denotes the Boogie semantics defined in [73], which is infinitary since all maps are total ($\mathcal{I}$ is for "infinitary"); and $\mathcal{F}[P]$ denotes the finitary semantics of this chapter, where all maps have finite domains.[4] Assuming perfect constraint-solving capabilities, the only aspect where $\mathcal{F}$ may drop information w.r.t. $\mathcal{I}$ is in the rule PICK, and more precisely in the finitization mapping Φ. The requirement that Φ be an over-approximation implies that every Boogie execution is finitized by *some* Boogaloo execution. The converse does not hold in general: in particular, for some programs S, $\mathcal{I}[S] = \emptyset$ but $\mathcal{F}[S] \neq \emptyset$ contains executions (which we regard as spurious). For example, the following program:

```
var a: [int] int;
assume (∀ i, j: int • i < j ⟹a[i] < a[j]);
assume a[0] =0 ∧a[1000] =1;
```

has no feasible executions in $\mathcal{I}$, while the current implementation of Boogaloo produces a passing execution where the quantified constraint is only instantiated for $\mathtt{i} = 0$ and $\mathtt{j} = 1000$. Sect. 7.5 demonstrates that such unsound executions are infrequent in practice, and, even when they occur, workarounds are possible, for example forcing the evaluation on more points by accessing them in a loop. Also, Boogaloo's implementation of Φ does not produces spurious executions for programs where all quantified constraints are derived from terminating recursive function definition (see Sect. 7.4).

There is an additional source of discrepancies between $\mathcal{I}$ and $\mathcal{F}$, due to the fact that Boogie always uses the specification semantics for loops and procedure calls, while Boogaloo defaults to the implementation semantics, which might contain fewer executions. This discrepancy between the two semantics is a useful feature, which makes it possible to debug programs in the presence of incomplete specifications. The specification semantics is still available on demand in Boogaloo: it is sufficient to replace an imperative construct with its specification.

7.4 Boogaloo: Implementation Details

This section presents some details of the Boogaloo tool—our prototype implementation of the approach described in the previous sections. The tool takes as input a Boogie source file and a procedure name as entry point, and produces a set of feasible executions, characterized by their concrete initial and final states. Boogaloo is implemented in Haskell, and uses the SMT solver Z3 [35] in the back-end.

Finitizing universal constraints. The choice of the finitization mapping Φ plays an important role. Our experiments suggest that the powerful

[4]The soundness of desugaring implies that all feasible executions both in the Boogie and in the Boogaloo semantics agree on being passing or failing.

quantifier instantiation strategies available in SMT solvers such as Z3 have some downsides when applied to solve constraints generated by executing Boogie programs, as their performance is unpredictable unless additional user input (in the form of "triggers") is provided. Instead, Boogaloo preprocesses constraints using a simple strategy, based on the observation that universally quantified formulas are typically used to axiomatize uninterpreted maps; since we are only interested in finitely many points (those stored in μ), we just instantiate the bound variables at those points. Quantified constraints that do not contain map applications are simply ignored; the examples in Sect. 7.5 suggests that this finitization strategy is not too restrictive on typical verification examples.

This is how Boogaloo implements Φ for a formula $\forall \vec{x} \bullet P(\vec{x})$. For each term $m[\vec{y}]$ in P such that $\vec{y}$ includes some bound variable (i.e., $\vec{y} \cap \vec{x} \neq \emptyset$), Boogaloo extracts a parametrized map constraint of the form $(m, Q(\vec{y}, m[\vec{y}]))$, where Q is a subformula of P including the term, and $\vec{y}$ are the parameters free in Q; if $\vec{x} \not\subseteq \vec{y}$, then Q is itself quantified. For example, $\forall i \bullet a[i] > i \wedge b[i, 0] = 1$ generates two parametrized constraints: $(a, a[i] > i)$ and $(b, j = 0 \Longrightarrow b[i, j] = 1)$; whereas $\forall i, j \bullet i < j \Longrightarrow c[i] < c[j]$ generates a single parametrized constraint with a quantifier: $(c, \forall j \bullet i < j \Longrightarrow c[i] < c[j])$.

Boogaloo evaluates parametrized constraints for a given map store μ iteratively: pick an element $p = (m, \vec{e}, s)$ of μ, instantiate all parametrized constraints for m with $\vec{e}$ and evaluate them, and mark p; repeat until all elements of μ are marked. If a Q in a parametrized constraint (m, Q) contains quantifiers, instantiating m triggers the generation of new parametrized constraints from Q.

Since evaluating a parametrized constraint may add new points to μ, termination of the evaluation procedure is not guaranteed in the presence of recursive formulas, which generate constraints $(m, Q(\vec{y}, m[\vec{y}]))$ where Q also contains applications of m to elements other than $\vec{y}$. For example, an axiomatization of the factorial function f as

$$f[0] = 1$$
$$\forall i \bullet i > 0 \Longrightarrow f[i] = i \cdot f[i-1]$$

generates the constraint

$$q \equiv (f, i > 0 \Longrightarrow f[i] = i \cdot f[i-1]).$$

If initially $\mu[f] = \{(\ell_0, \ell_1)\}$, evaluating q for $i = \ell_0$ introduces a new map application at $\ell_0 - 1$, which then leads to an application at $\ell_0 - 2$, and so on.

Boogaloo evaluates such recursive constraints using fair unrolling similarly to [122], based on the notion of guard: a parametrized constraint is *guarded* if has the form $(m, G(\vec{y}) \implies B(\vec{y}))$. When Boogaloo's iterative evaluation picks an element $p = (m, \vec{e}, s)$, it nondeterministically chooses a subset D of all guarded constraints for m and "disables" them in the evaluation determined by p: for a parametrized constraint $q = (m, G \implies B)$, it evaluates the constraint $\neg G$ if $q \in D$, and $G \wedge B$ otherwise. For the "right" selection of D, recursive definitions are disabled, so that they do not add new points to μ and evaluation terminates. In the factorial example, there are two choices for $f[\ell_0]$: disabling or enabling the single guarded constraint. Disabling it terminates the finitization process, producing an execution with $\ell_0 = 0$; enabling the guarded constraints produces one iteration (for $f[\ell_0 - 1]$), which in turn recursively leads to the same two choices, and so on. Unlike [122], which works only with function definitions and thus assumes that guards are mutually exclusive and cover all cases, Boogaloo's fair enumeration applies to guards of any form and constraints other than equality; it also provides an option to limit the number of unrollings, because recursive constraints may be not well-founded (a sufficient condition for termination).

Nondeterminism. There are four sources of nondeterminism in Boogaloo semantics: evaluation of quantified expressions (rules QUANT-T and QUANT-F), **goto**s, and **pick**—involving the disabling of guarded constraints in Φ and constraint solving to select a solution Λ. Boogaloo enumerates nondeterministic choices using backtracking monads (e.g. [61]). The command-line interface currently offers depth-first and fair exploration strategies, but the implementation can easily accommodate others parametrically.

When executing **goto** statements, the order in which labels are tried may affect progress: if the first chosen label leads back to the same statement, execution gets stuck in an infinite loop. To avoid this situation, Boogaloo keeps track of how often each label was taken along the current execution path, and always tries labels in ascending order of their frequencies (least frequent first). This strategy also has the nice effect of enumerating shorter executions before longer ones in the long run. A similar strategy applies to disabling parametrized constraints.

Since symbolic computation is speculative, it introduces the risk that long computations are constructed only to realize, when solving the symbolic constraints, that they are infeasible. This risk is mitigated by the enumeration technique, which produces short execution first. Moreover, whenever a constraint evaluates to the concrete value $\perp$, the current execution path is immediately aborted. This mitigates the overhead incurred by nondeterministic evaluation of quantified expressions: such expressions are likely to occur inside **assume** statements, thus branches where they evaluate to false are immediately abandoned.

Additionally, Boogaloo transparently tests the satisfiability of the current constraints κ at various points during an execution, and proceeds only if the constraints are satisfiable; unlike `pick` which may enumerate multiple solutions, a satisfiability check does not cause additional nondeterminism. One can still explore the trade-off between few expensive symbolic executions and many cheap concrete executions by adding `pick` directives at arbitrary points.

Minimization. In addition to producing short executions first, Boogaloo also uses a minimization technique based on binary search (similar to the one in [64]) in order to favor *small* integers for concrete values. In our experience, this significantly improves readability: for example, a constraint "x is positive and divisible by 5" with minimization produces the most natural solution $x = 5$ as first witness.

7.5 Experimental Evaluation

We evaluated Boogaloo on a choice of 15 examples from various sources[5]. Tab. 7.1 lists the programs and some data about them. The bulk of the evaluation targets the *verification* of algorithms of various kinds, listed in the top part of the table. For each of these problems, we constructed a correct version equipped with consistent but generally incomplete specifications, and a buggy one, obtained by injecting implementation or specification errors. We ran Boogaloo on both versions, with the goal of generating executions: passing executions for the correct programs, and failing executions exposing the bug for the buggy programs. The rest of the programs, in the bottom part of Tab. 7.1, are examples of *declarative* programming, which exercise Boogaloo's constraint solving capabilities to generate outputs satisfying given properties, in the absence of imperative implementations. We now briefly mention the most interesting features of our examples, and summarize the experimental results.

Verification. The majority of the programs in the top part of Tab. 7.1 are slightly adapted examples from the Boogie project repository[6], verification competitions [62], or previous work [43, 46]; they contain features that exercise various aspects of the test-case generation process. Strong preconditions (such as an array being sorted in BinarySearch or being a permutation in Invert) make generating valid executions challenging using standard testing enumeration techniques. Inlining (available in Boogie) scales poorly with

[5] Available online at `http://se.inf.ethz.ch/people/polikarpova/boogaloo/`

[6] `http://boogie.codeplex.com/`

Table 7.1: Programs tested with Boogaloo

PROGRAM	FEATURES	LOC	FUN	ANNOTATIONS						TIME			BUG		
				A	S	U	R	E	I	N	t_Σ	t_C	N	t_Σ	t_C
ArrayMax	see Sect. 7.2	33	0	0	0	1	1	2	1	46	0.5	0.4	0	0.0	0.0
ArraySum	recursive definition	34	1	0	0	1	1	1	2	46	0.3	0.4	1	0.0	0.0
BinarySearch	complex precondition	49	1	0	0	2	2	3	2	46	0.1	0.0	0	0.1	0.2
BubbleSort	complex postcond. and invariants	74	1	0	0	2	1	4	5	11	9.3	113.9	2	0.1	0.3
DutchFlag	user-defined types [43]	96	3	0	0	2	2	8	6	20	3.8	7.6	1	0.0	0.0
Fibonacci	recursive procedure	40	1	3	1	0	2	0	0	19	93.9	3.7	0	0.0	0.0
Invert	complex pre- and post-conditions	37	0	0	0	3	3	2	1	10	48.1	1.4	2	0.0	0.1
ListTraversal	heap model	49	3	2	0	0	1	1	1	20	61.8	2.5	2	0.0	0.0
ListInsert	see [46]	52	1	0	0	2	1	1	0	4	2.4	4.1	1	0.0	0.0
QuickSort	helper and recursive procedures	89	3	0	0	2	1	6	0	15	8.4	177.9	2	∞	0.1
QuickSort PI	partial implementation	79	3	0	0	2	2	9	0	4	0.2	42.1	2	0.1	∞
TuringFactorial	unstructured control flow	37	1	2	5	0	1	1	0	21	0.2	0.2	3	0.0	0.0
Split	linear arithmetic [66]	22	0	0	0	0	1	3	0	–	0.0	0.0			
SendMoreMoney	fixed-size array constraints [64]	36	1	0	0	15	0	0	0	–	0.3	0.3			
Primes	recursive definition [64]	31	2	0	0	0	0	2	0	8	0.2	0.9			
NQueens	variable-size array constraints	37	2	1	0	3	0	0	0	15	1.2	31.8			

LOC: lines of code, FUN: number of specification functions,
ANNOTATIONS: number of annotations (axioms A, asserts S, assumes U, preconditions R, postconditions E, loop invariants I)
TIME: time to generate passing executions
BUG: time to generate failing executions for the buggy version
(Times in seconds, rounded to the nearest integer: for a given input size N, time t_Σ with fully symbolic execution and t_C with concretization. ∞ denotes a timeout of 180 seconds).

the recursive procedure calls of Fibonacci and QuickSort. The specifications of BinarySearch, BubbleSort, QuickSort, and Invert use nested universal quantifiers with bound variables mentioned in different predicates. QuickSort PI (partial implementation) is a variant of QuickSort whose partitioning procedure has a complete pre- and postcondition but no implementation. This may represent an intermediate development step where we want to validate the overall logic of QuickSort before proceeding with implementing the partitioning procedure. Boogaloo simulates array partitioning based only on its specification—something unachievable with traditional testing techniques.

The injected bugs are mostly off-by-one errors and missing preconditions, both of which frequently occur in practice; the bugs in BinarySearch are among those found in textbooks [105].

Declarative programming. The other four examples come from previous work on constraint programming and code synthesis [66, 64], and involve linear arithmetic, recursively defined functions, and quantification over variable-sized arrays. Constraints are declared using `assume` statements or procedures without implementation; Boogaloo generates program outputs satisfying the constraints.

Experimental results. All problems in Tab. 7.1 but two include a parameter N that defines the input size (the input array or list for most problems). Column TIME displays the value of N used in the experiments; and the time required to generate a passing execution with different concretization strategies: t_Σ corresponds to fully symbolic executions where the state is concretized only once after terminating; t_C, instead, corresponds to executions where the state is concretized before every jump statement. Column BUGGY displays the same time measures for the buggy programs; for these, the value of N corresponds to the input size exposing the bug found by Boogaloo (which is the smallest possible for all programs). In all experiments we imposed a timeout of 180 seconds, to reflect the expectation to use Boogaloo with good responsiveness; the value of N for passing executions was then chosen as the largest (among those tried) where neither the symbolic (t_Σ) nor the concrete (t_C) strategy timed out.

Concretizing before jumping makes executions order-of-magnitude faster for some problems, and order-of-magnitude slower for others. This strategy may cause heavy backtracking when the constraints on a given logical variable are imposed incrementally, with one or more concretization points in between, producing potentially lengthy combinatorial enumerations. When constraints are "local", on the other hand, it can speed things up by operating on concrete values (for example, path exploration in a loop becomes deterministic if the number of loop iterations is fixed in advanced). Even though the current implementation has a big potential for improving performance, the experimental results are encouraging: in particular, exposing bugs—the primary purpose of Boogaloo—is fast, even in the presence of partial implementations.

7.6 Related Work

Debugging failed verification attempts. While still an incipient research area, a few techniques have recently been proposed to help understand and debug failed attempts of program verifiers. Sect. 7.2 already mentioned the Boogie Verification Debugger (BVD, [46]); the Spec# debugger [93] implements similar functionalities which construct concrete counterexamples from failed Boogie runs. Two-step verification [129] compares verification with different semantics (based on unrolling and inlining) to attribute verification failures to either inconsistent or incomplete specifications.

The fact that all these approaches are built around the output provided by a program verifier determines their main limitations compared to Boogaloo. As we demonstrated in Sect. 7.2.1, when verification fails because of insufficient specification, the counterexamples generated by BVD or similar tools are typically uninformative or even misleading, because they ignore the implementation even when it is correct (e.g., a loop), unless it is comes with an accurate specification (e.g., a loop invariant). Boogaloo supports a more incremental approach, where users can concentrate on fixing major bugs first. Sect. 7.2.1 also discussed how inlining and unrolling (available in Boogie and automatically used in two-step verification) ameliorate these problems, but they are also not directly comparable to Boogaloo, since they scale poorly and require to know explicit unrolling bounds. Of course, the finitary semantics implemented by Boogaloo comes with its own shortcomings: if the shortest counterexamples are very long, it may be infeasible to generate them by enumeration, whereas a static verifier's modular reasoning is insensitive to the length of concrete execution paths since it is entirely symbolic; tools such as BVD can directly work on any failed verifier attempts.

Another approach to produce readable counterexamples is restricting the input language (e.g., [122]), trading off expressiveness for decidability. Bounded model-checking techniques (e.g., [27]) also target standard programming languages and the verification of properties that do not include features such as infinite mappings and unbounded quantification. Boogaloo follows a different course: it supports the entire Boogie language as used in practice, which does not restrict expressiveness *a priori*, but may produce spurious counterexamples.

Testing. Testing is the process of executing programs to make them fail. Since it is based on execution, it is typically limited to violations of simple properties that can be efficiently evaluated at runtime and are implicit in the programming language semantics (e.g., null dereferencing). Languages such as Eiffel (used in the present work), JML [69], and Jahob [136] incorporate a richer language for annotations that is still executable, so as to extend the applicability of standard testing techniques. Another line of research in testing is the combination with static techniques, with the goal of complementing each other's strengths to search the input state space more efficiently. For example, [127] combines testing with program proving at a high level. A different array of techniques integrates testing with symbolic execution; see the recent survey [19]. Boogaloo is also based on symbolic execution, but with a different overall goal; as future work, we will leverage other techniques from symbolic execution to improve the enumeration of executions.

Constraint programming. This programming style supports program definitions based on declarative constraints, describing properties of the solution, rather than on traditional imperative constructs. Logic programming extends functional programming languages [3]; more recent approaches combine declarative constraints with imperative languages [89, 64]. All these approaches restrict the expressiveness of the constraint language to have predictable performance and some guarantees about soundness, completeness, or both. As briefly demonstrated in Sect. 7.5, Boogaloo can also be used as a Boogie-based constraint programming language. Unless we also restrict the language of assertions, we cannot offer strong guarantees about properties of the executions generated by Boogaloo (see the end of Sect. 7.3.2 for a discussion). However, the usage as a constraint programming language brings much flexibility to Boogaloo as a testing environment for Boogie programs, since users can achieve different trade-offs between modularity and scalability opting for the implementation or the specification semantics.

7.7 Summary and Future Work

We presented a technique and a prototype implementation to execute programs with complex specifications and nondeterministic constructs, written in the Boogie intermediate verification language. We also evaluated the main applications of the technique—understanding and debugging failed verification attempts by producing concrete simple counterexamples, and executing partial implementations—on several benchmark examples.

The main direction for **future work** is improving the performance of the tool up to the point where it is applicable to Boogie programs generated automatically by source language verifiers, such as AutoProof. Those programs tend to be bigger and more complex than the code written by hand, since they always include the encoding of the heap and various verification methodologies (e.g. for framing and class invariants), even if the source program does not make use of them. In its present capacity Boogaloo is mainly useful in teaching verification; in fact, in 2013 is was successfully applied in the Software Verification course at ETH Zurich.

Chapter 8

Conclusions

This thesis presented a comprehensive approach to improving the quality of reusable software components, in the context of sequential object-oriented systems.

For early stages of the software development process, the thesis advocates using an abstraction mechanism based on *models* as a design tool to define coherent interfaces and organize components into consistent hierarchies. Specifying the behavior of a component using *model-based contracts* provides precise documentation for its clients and enables automated verification at later stages of development, through both static and dynamic techniques. To assess the quality of such specifications the thesis proposes formally defined criteria of *completeness*, *observability*, *closure*, and *controllability*, which together guarantee that the contracts are strong yet abstract.

The present work shows that deploying strong—mostly complete—behavioral interface specifications in the form of model-based contracts is feasible for realistic component libraries. The overhead of such specifications is moderate: less than half a line of specification per line of executable code; but the benefits they can bring are significant: used as oracles in automated testing, strong specifications reveal twice as many faults as traditional Eiffel contracts. Additionally, using strong specifications as part of the development process seems to produce software with fewer faults "by construction".

For static verification, the present work gives particular importance to *class invariants*, which describe consistency of individual objects and object structures, and relate objects to their models. The thesis proposes a new verification methodology with support for invariants of complex object structures, dubbed *semantic collaboration*. Experimental evaluation on a set of benchmarks shows that the new methodology compares favorably to existing approaches in terms of flexibility and modularity: it supports invariants that depend on an unbounded number of objects, possibly unreachable in the heap, and invariants that depend on unknown classes, without sacrificing guarantees given to clients. Semantic collaboration is implemented in AutoProof: an auto-active program verifier for Eiffel.

Next, the thesis extends semantic collaboration with support for models. The support includes *logic classes*, which allow one to easily extend AutoProof with new mathematical types, and still enjoy several features normally available only for built-in types. The second extension is a simple heuristic for reducing bookkeeping overhead of ghost state, which complements AutoProof's straightforward encoding of model queries. Finally, the new methodology includes a model-based abstraction mechanism for frame specifications, with an intuitive semantics in the presence of inheritance. The thesis demonstrates practical applicability of the overall verification methodology using EiffelBase2: a realistic container library, used in practice.

The present work also proposes an approach to generating concrete test cases for programs equipped with complex specifications, normally used in static verification. The approach is based on the combination of symbolic execution and SMT solving, and is implemented in a tool called Boogaloo. The evaluation on several program verification examples demonstrates that the proposed test case generation technique can help understand failed verification attempts in conditions where traditional testing is not applicable, thus making formal verification techniques easier to use in practice.

Despite recent advances in software specification and verification, to which this thesis makes a useful addition, the battle for "correctness as a matter of course" is far from being won. Among various directions for **future research**, a particularly important one is creating more versatile and robust verification tools. In the design phase, such tools could evaluate consistency and expressive power of interface specifications, pointing to incorrect or missing specification elements. Later during development, the tools would employ a whole range of static and dynamic verification techniques to present the developer with the most accurate and useful information about the correctness of the system, including minimal failing test cases, hints about missing auxiliary annotations, and specification elements that lead to performance problems in deductive verification.

Bibliography

[1] Jean-Raymond Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.

[2] Andrei Alexandrescu. *The D Programming Language.* Pearson Education, 2010.

[3] Sergio Antoy and Michael Hanus. Functional logic programming. *Commun. ACM*, 53(4):74–85, 2010.

[4] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*, pages 1–10. ACM, 2011.

[5] AutoProof project. `http://se.inf.ethz.ch/research/autoproof/`.

[6] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, pages 387–411, 2008.

[7] J. Barnes. *High integrity Ada: the SPARK approach.* Addison-Wesley, 1997.

[8] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[9] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3, 2004.

[10] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.

[11] Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, pages 54–84, 2004.

[12] Kent Beck. *Test-Driven Development.* Addison-Wesley, 2002.

[13] Gilles Bernot, Michel Bidoit, and Teodor Knapik. Observational specifications and the indistinguishability assumption. *Theoretical Computer Science*, 139:275–314, 1995.

[14] Joshua Bloch. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 506–507, New York, NY, USA, 2006. ACM.

[15] Boogaloo project. `https://bitbucket.org/nadiapolikarpova/boogaloo/`.

[16] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *ISSTA*, pages 123–133, 2002.

[17] Cees-Bart Breunesse and Erik Poll. Verifying JML specifications with model fields. Technical report, ETH Zurich, 2003.

[18] S. Burris and H.P. Sankappanavar. *A course in universal algebra.* Graduate texts in mathematics. Springer-Verlag, 1981.

[19] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[20] Patrice Chalin. Are practitioners writing contracts? In *Rigorous Development of Complex Fault-Tolerant Systems*, pages 100–113, 2006.

[21] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *FMCO*, pages 342–363, 2005.

[22] Juei Chang and Debra J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *ESEC/FSE*, pages 285–302, 1999.

[23] Julien Charles. Adding native specifications to JML. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2006.

[24] Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: cleanly supporting abstraction in design by contract. *Softw. Pract. Exper.*, 35(6):583–599, 2005.

[25] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP*, pages 231–255, 2002.

[26] Ilinca Ciupa, Bertrand Meyer, Manuel Oriol, and Alexander Pretschner. Finding faults: Manual testing vs. random+ testing vs. user reports. In *Proceedings of ISSRE (International Symposium on Software Reliability) 2008*, 2008.

[27] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS*, volume 3440 of *LNCS*, pages 570–574, 2005.

[28] Code Contracts. `http://research.microsoft.com/en-us/projects/contracts/`.

[29] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[30] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *CAV*, pages 480–494, 2010.

[31] Boogaloo web interface. `http://cloudstudio.ethz.ch/comcom/#Boogaloo`.

[32] Marcelo d'Amorim, Carlos Pacheco, Tao Xie, Darko Marinov, and Michael D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *ASE*, pages 59–68, 2006.

[33] Ádám Darvas and Peter Müller. Faithful mapping of model classes to mathematical structures. *IET Software*, 2(6):477–499, 2008.

[34] Ádám Darvas and Peter Müller. Proving consistency and completeness of model classes using theory interpretation. In *FASE*, pages 218–232, 2010.

[35] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[36] DSA library. `http://dsa.codeplex.com/`.

[37] Lydie du Bousquet, Yves Ledru, Olivier Maury, Catherine Oriat, and Jean-Louis Lanet. Reusing a JML specification dedicated to verification for testing, and vice-versa: Case studies. *J. Autom. Reasoning*, 45(4):415–435, 2010.

[38] François Dupressoir, Andrew D. Gordon, Jan Jürjens, and David A. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. In *IEEE Computer Security Foundations Symposium*, 2011.

[39] EiffelBase library. `http://freeelks.svn.sourceforge.net`.

[40] EiffelBase2 library. `https://bitbucket.org/nadiapolikarpova/eiffelbase2/`.

[41] H.-Christian Estler, Carlo A. Furia, Martin Nordio, Marco Piccioni, and Bertrand Meyer. Contracts in practice. In *Proceedings, 19th International Symposium on Formal Methods (FM 2014)*, May 2014.

[42] Jean-Christophe Filliâtre. Verifying two lines of C with Why3: an exercise in program verification. In *VSTTE*, LNCS, pages 83–97, 2012.

[43] Carlo A. Furia, Bertrand Meyer, and Sergey Velder. Loop invariants: Analysis, classification, and examples. `http://arxiv.org/abs/1211.4470`, 2012.

[44] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[45] John D. Gannon, Paul R. McMullin, and Richard G. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, 1981.

[46] Claire Le Goues, K. Rustan M. Leino, and Michał Moskal. The Boogie verification debugger (tool paper). In *SEFM*, volume 7041 of *LNCS*, pages 407–414. Springer, 2011.

[47] Joseph A. Gougen, James W. Thatcher, and Eric G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume IV, pages 80–149. Prentice Hall, 1978.

[48] John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta Inf.*, 10:27–52, 1978.

[49] John V. Guttag, James J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. Larch: Languages and tools for formal specification. In *Texts and monographs in computer science*. Springer-Verlag, 1993.

[50] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16, 2012.

[51] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy A. Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2), 2009.

[52] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[53] C. A. R. Hoare. How did software get so reliable without proof? In *FME*, pages 1–17, 1996.

[54] C. A. R. Hoare, Jayadev Misra, Gary T. Leavens, and Natarajan Shankar. The verified software initiative: A manifesto. *ACM Comput. Surv.*, 41(4), 2009.

[55] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, 2006.

[56] Bart Jacobs, Joseph Kiniry, and Martijn Warnier. Java program verification challenges. *Formal Methods for Components and Objects*, pages 202–219, 2003.

[57] Bart Jacobs, Jan Smans, and Frank Piessens. Verifying the composite pattern using separation logic. In *Proceedings of the Specification and Verification of Component-Based SystemsâĂŤChallenge Track*, 2008.

[58] Cliff B. Jones. *Systematic software development using VDM.* Prentice-Hall, 2nd edition, 1990.

[59] Deepak Kapur and Srivas Mandayam. Expressiveness of the operation set of a data abstraction. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '80, pages 139–153, New York, NY, USA, 1980. ACM.

[60] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, pages 268–283, 2006.

[61] Oleg Kiselyov, Chung-Chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *ICFP*, pages 192–203. ACM, 2005.

[62] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st verified software competition. In *FM*, volume 6664 of *LNCS*, 2011. Extended version at `www.vscomp.org`.

[63] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220. ACM, 2009.

[64] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. In *POPL*, pages 151–164, 2012.

[65] Neelakantar R. Krishnaswami. Reasoning about iterators with separation logic. In *5th International Workshop on Specification and Verification of Component-Based Systems*, page 83âĂŞ86. ACM Press, 2006.

[66] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *PLDI*, pages 316–329. ACM, 2010.

[67] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. 1999.

[68] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[69] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55(1-3):185–208, 2005.

[70] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.*, 19(2):159–189, 2007.

[71] Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proceedings of the 2nd World Congress on Formal Methods*, FM'09, pages 806–809, Berlin, Heidelberg, 2009. Springer-Verlag.

[72] K. Rustan M. Leino. *Toward Reliable Modular Programs.* PhD thesis, Caltech, 1995.

[73] K. Rustan M. Leino. This is Boogie 2. `http://goo.gl/QsH6g`, 2008.

[74] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[75] K. Rustan M. Leino. Verifying concurrent programs with Chalice. In *VMCAI*, page 2, 2010.

[76] K. Rustan M. Leino and Michał Moskal. Usable auto-active verification. In *Usable Verification Workshop.* `http://fm.csl.sri.com/UV10/`, 2010.

[77] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *ECOOP*, pages 491–516, 2004.

[78] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In *ESOP*, pages 115–130, 2006.

[79] K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In *ESOP*, pages 378–393, 2009.

[80] K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. In *ESOP*, pages 80–94, 2007.

[81] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *ASE*, page 22, 2001.

[82] E. Michael Maximilien and Laurie Williams. Assessing test-driven development at IBM. In *ICSE*, pages 564–569, 2003.

[83] Bertrand Meyer. Applying design by contract. *Computer*, 25(10):40–51, 1992.

[84] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall, 2nd edition, 1997.

[85] Bertrand Meyer. The dependent delegate dilemma. In *Engineering Theories of Software Intensive Systems*, pages 105–118. Springer, 2005.

[86] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *Computer*, 42(9):46–55, 2009.

[87] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik J. Luit. Cooperation-based invariants for OO languages. *Electr. Notes Theor. Comput. Sci.*, 160:225–237, 2006.

[88] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik J. Luit. Invariants for non-hierarchical object structures. *Electr. Notes Theor. Comput. Sci.*, 195:211–229, 2008.

[89] Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, and Daniel Jackson. Unifying execution of imperative and declarative code. In *ICSE*, pages 511–520. ACM, 2011.

[90] Matthias M. Müller, Rainer Typke, and Oliver Hagner. Two controlled experiments concerning the usefulness of assertions as a means for programming. In *ICSM*, pages 84–92, 2002.

[91] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[92] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006.

[93] Peter Müller and Joseph N. Ruskiewicz. Using debuggers to understand failed verification attempts. In *FM*, volume 6664 of *LNCS*, pages 73–87. Springer, 2011.

[94] Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat, and Laurie Williams. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *ESE*, 13:289–302, 2008.

[95] Martin Nordio, Cristiano Calcagno, Bertrand Meyer, Peter Müller, and Julian Tschannen. Reasoning about function objects. In J. Vitek, editor, *TOOLS-EUROPE*, Lecture Notes in Computer Science, 2010.

[96] Donald A. Norman. *The Design of Everyday Things.* Basic Books, 2002.

[97] A. Jefferson Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *UML*, pages 416–429, 1999.

[98] William F. Ogden, Murali Sitaraman, Bruce W. Weide, and Stuart H. Zweben. The RESOLVE framework and discipline. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, 1994.

[99] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. *ACM Trans. Program. Lang. Syst.*, 31(3), 2009.

[100] Catherine Oriat. Jartege: A tool for random generation of unit tests for Java classes. In *Proceedings of the First International Conference on Quality of Software Architectures and Software Quality, and Proceedings of the Second International Conference on Software Quality*, QoSA'05, pages 242–256, Berlin, Heidelberg, 2005. Springer-Verlag.

[101] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP*, pages 504–527, 2005.

[102] Matthew J. Parkinson. Class invariants: the end of the road? In *IWACO*. ACM, 2007.

[103] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *POPL*, pages 75–86, 2008.

[104] David Lorge Parnas. Precise documentation: The key to better software. In *The Future of Software Engineering*, pages 125–148. Springer, 2011.

[105] Richard E. Pattis. Textbook errors in binary searching. In *SIGCSE*, pages 190–194. ACM, 1988.

[106] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 93–104, New York, NY, USA, 2009. ACM.

[107] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[108] Arsenii Rudich. *Automatic Verification of Heap Structures with Stereotypes*. PhD thesis, ETH Zurich, 2011.

[109] SAVCBS workshop series. `http://www.eecs.ucf.edu/~leavens/SAVCBS/`, 2001–2010.

[110] Semantic Collaboration. `http://se.inf.ethz.ch/people/polikarpova/sc/`.

[111] Bernd Schoeller. *Making classes provable trough contracts, models and frames*. PhD thesis, ETH Zurich, 2007.

[112] Bernd Schoeller, Tobias Widmer, and Bertrand Meyer. Making specifications complete through models. In *Architecting Systems with Trustworthy Components*, pages 48–70, 2004.

[113] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

[114] Murali Sitaraman, Lonnie R. Welch, and Dounglas E. Harms. On specification of reusable software components. *International Journal of Software Engineering and Knowledge Engineering*, 03(02):207–229, 1993.

[115] Jan Smans, Bart Jacobs, and Frank Piessens. VeriCool: An automatic verifier for a concurrent object-oriented language. In *FMOODS*, pages 220–239, 2008.

[116] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, pages 148–172, 2009.

[117] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for Java-like programs based on dynamic frames. In *FASE*, pages 261–275, 2008.

[118] Matt Staats, Michael W. Whalen, and Mats Per Erik Heimdahl. Programs, tests, and oracles. In *ICSE*, pages 391–400, 2011.

[119] Phil Stocks and David A. Carrington. Test templates: A specification-based testing framework. In *ICSE*, pages 405–414, 1993.

[120] Alexander J. Summers and Sophia Drossopoulou. Considerate reasoning and the composite design pattern. In *VMCAI*, pages 328–344, 2010.

[121] Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. The need for flexible object invariants. In *IWACO*, pages 1–9. ACM, 2009.

[122] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *SAS*, volume 6887 of *LNCS*, pages 298–315. Springer, 2011.

[123] Testing with strong specifications. `http://se.inf.ethz.ch/people/polikarpova/mbctesting`.

[124] Nikolai Tillmann and Jonathan de Halleux. Pex—white box test generation for .NET. In *TAP*, pages 134–153, 2008.

[125] Traffic repository. `https://bitbucket.org/nadiapolikarpova/traffic/`.

[126] Julian Tschannen, Carlo A. Furia, and Martin Nordio. AutoProof meets some verification challenges. *International Journal on Software Tools for Technology Transfer*, 2014.

[127] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. In *SEFM*, volume 7041 of *LNCS*. Springer, 2011.

[128] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Verifying Eiffel programs with Boogie. In *BOOGIE workshop*, 2011. `http://arxiv.org/abs/1106.4700`.

[129] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Program checking with less hassle. In *VSTTE*, pages 149–169, 2013.

[130] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring better contracts. In *ICSE*, pages 191–200, 2011.

[131] Yi Wei, Hannes Roth, Carlo A. Furia, Yu Pei, Alexander Horton, Michael Steindorfer, Martin Nordio, and Bertrand Meyer. Stateful testing: Finding more errors in code and contracts. In *ASE*, pages 440–443, 2011.

[132] Bruce W. Weide, William F. Ogden, and Stuart H. Zweben. Reusable software components. *Advances in Computers*, 33:1–65, 1991.

[133] B.W. Weide, S.H. Edwards, Wayne D. Heym, T.J. Long, and W.F. Ogden. Characterizing observability and controllability of software components. In *Software Reuse, 1996., Proceedings Fourth International Conference on*, pages 62–71, April 1996.

[134] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[135] Tao Xie and David Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Autom. Softw. Eng.*, 13(3):345–371, 2006.

[136] Karen Zee, Viktor Kuncak, Michael Taylor, and Martin C. Rinard. Runtime checking for program verification. In *RV*, volume 4839 of *LNCS*, pages 202–213. Springer, 2007.

[137] Andreas Zeller. Mining specifications: A roadmap. In *The Future of Software Engineering*, pages 173–182. Springer, 2010.

[138] Stephen N. Zilles. Introduction to data algebra. In *Abstract Software Specifications*, pages 248–272, 1979.

[139] Daniel M. Zimmerman and Rinkesh Nagmoti. JMLUnit: The next generation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software*, FoVeOOS'10, pages 183–197, Berlin, Heidelberg, 2011. Springer-Verlag.

CURRICULUM VITAE

Nadia Polikarpova
Chair of Software Engineering
Department of Computer Science, ETH Zurich
ETH Zentrum RZ J8, Clausiusstrasse 59, 8092 Zürich, Switzerland
Phone: +41 44 632 4723
Email: nadia.polikarpova@inf.ethz.ch
Homepage: se.ethz.ch/people/polikarpova/

Research positions

PhD Student and Research Assistant Chair of Software Engineering, Department of Computer Science, ETH Zurich (Switzerland)	09/2008–present
Research Intern Microsoft Research (Redmond, USA) Mentor: Michał Moskal	05–08/2011
Undergraduate Researcher Chair of Computer Technologies, Department of Information Technologies and Programming, SPbSU ITMO (Saint-Petersburg, Russia)	10/2005–08/2007

Education

PhD, Computer Science ETH Zurich (Switzerland) Thesis: *Specified and Verified Reusable Components* Advisor: Prof. Bertrand Meyer	09/2008–04/2014

MSc, Applied Mathematics and Informatics 09/2006–05/2008
SPbSU ITMO (Saint-Petersburg, Russia)
Thesis: *Dynamic Assertion Inference in a Programming Language with Design by Contract Support (Eiffel Case Study)*
Advisor: Prof. Anatoly Shalyto
Co-advisor: Ilinca Ciupa (ETH Zurich)
Grade: 5 / 5

BEng, Applied Mathematics and Informatics 09/2002–05/2006
SPbSU ITMO (Saint-Petersburg, Russia)
Thesis: *Object-oriented approach to modeling and specification of entities with complex behavior*
Advisor: Dr. Danil Shopyrin
Grade: 5 / 5

Research interests

My research interests lie in the area of *software correctness*. My expertise covers a range of topics in *formal methods* and *software engineering*; in particular, my research has contributed to auto-active verification, behavioral interface specifications, automated testing, dynamic invariant inference, and user interface for verification.

Projects

I have been involved in the development of the following tools and libraries:

AutoProof co-developer, Eiffel
An auto-active program verifier for Eiffel
`http://se.inf.ethz.ch/research/autoproof`

EiffelBase2 main developer, Eiffel
A specified and verified data structure library for Eiffel
`https://bitbucket.org/nadiapolikarpova/eiffelbase2`

Boogaloo main developer, Haskell
An interpreter and run-time assertion checker for Boogie
`https://bitbucket.org/nadiapolikarpova/boogaloo`

Dafny contributor, C#
A Language and Program Verifier for Functional Correctness
`http://research.microsoft.com/en-us/projects/dafny`

CITADEL main developer, Eiffel
An Eiffel front-end for the Daikon assertion detector
`http://se.inf.ethz.ch/people/polikarpova/citadel`

Publications[1]

International conferences and journals

P9. Polikarpova N., Tschannen J., Furia C., Meyer B., *Flexible Invariants Through Semantic Collaboration*, Proceedings of FM'14: 19th International Symposium on Formal Methods, (Singapore), *To Appear*

P8. Polikarpova N., Furia C., West S., *To Run What No One Has Run Before* Proceedings of RV'13: Fourth International Conference on Runtime Verification, (Rennes, France), September 2013

P7. Polikarpova N., Furia C., Pei Y., Wei Y., Meyer B., *What Good Are Strong Specifications?* Proceedings of ICSE'13: 35th International Conference on Software Engineering, (San Francisco, California, USA), May 2013

P6. Leino K. R. M., Polikarpova N., *Verified calculations*, Proceedings of VSTTE'13: Verified Software: Theories, Tools and Experiments, (Atherton, California, USA), May 2013

P5. Polikarpova N., Moskal M., *Verifying implementations of security protocols by refinement*, Proceedings of VSTTE'12: Verified Software:

[1]Publications are available online at `http://se.inf.ethz.ch/people/polikarpova/`.

Theories, Tools and Experiments, (Philadelphia, Pennsylvania, USA), January 2012

P4. Klebanov V. et al., *The 1st Verified Software Competition: Experience Report*, Proceedings of FM'11: 17th International Symposium on Formal Methods, (Limerick, Ireland), June 2011, **best paper award**

P3. Polikarpova N., Furia C., Meyer B., *Specifying Reusable Components*, Proceedings of VSTTE'10: Verified Software: Theories, Tools and Experiments, (Edinburgh, Scotland), August 2010

P2. Polikarpova N., Tochilin V., Shalyto A., *Method of Reduced Tables for Generation of Automata with a Large Number of Input Variables Based on Genetic Programming*, Journal of Computer and Systems Sciences International, 49(2):265–283, February 2010

P1. Polikarpova N., Ciupa I., Meyer B., *A comparative study of programmer-written and automatically inferred contracts*, Proceedings of ISSTA'09: International Conference on Software Testing and Analysis, (Chicago, Illinois, USA), July 2009

National conferences

N5. Polikarpova N., Tochilin V., Shalyto A., *A Library for Generating Control Automata by Means of Genetic Programming* (in Russian), Proceedings of the X International Conference on Soft Computing and Measurement, (Saint-Petersburg, Russia), June 2007

N4. Polikarpova N., Tochilin V., Shalyto A., *Applying Genetic Programming to Implementation of Systems with Complex behavior* (in Russian), Proceedings of the IV International Theoretical and Practical Conference "Integrated Models and Soft Computing in Artificial Intelligence", (Kolomna, Russia), May 2007

N3. Polikarpova N., Tochilin V., *Applying Genetic Algorithms to Generating Logics in Computational Systems* (in Russian), Proceedings of the 4th Inter-University Young Researchers Conference, (Saint-Petersburg, Russia), April 2007

N2. Polikarpova N., *Object-oriented approach to modeling and specification of entities with complex behavior*, Proceedings of SEC(R) 2006: Software Engineering Conference in Russia, (Moscow, Russia), November 2006

N1. Polikarpova N., *Inheritance Relation for Types with Complex Behavior* (in Russian), Proceedings of the 3rd Inter-University Young Researchers Conference, (Saint-Petersburg, Russia), April 2006

Books

B1. Polikarpova N., Shalyto A., *Automata-based Programming* (in Russian), Piter, 2009

Teaching activities

Teaching assistant and guest lecturer for *Introduction to Programming*, ETH Zürich, Fall 2008–2013, Prof. Bertrand Meyer

Guest lecturer for *Software Verification*, ETH Zürich, Fall 2009–2013, Prof. Bertrand Meyer, Dr. Carlo A. Furia, Dr. Sebastian Nanz

Guest lecturer for *Software Architecture*, ETH Zürich, Spring 2011, Prof. Bertrand Meyer, Dr. Carlo A. Furia, Dr. Martin Nordio

Teaching assistant for *Java and C# in depth*, ETH Zürich, Spring 2010, Prof. Bertrand Meyer, Dr. Carlo A. Furia

Guest lecturer for *Eiffel: Analysis, Design and Programming*, ETH Zürich, Fall 2009, Prof. Bertrand Meyer

Teaching assistant and guest lecturer for *Software Architecture*, ETH Zürich, Spring 2009, Prof. Bertrand Meyer

Supervised theses

Tobias Kiefer *Model-based contracts for C#*, Bachelor Thesis, ETH Zürich, October 2012

Elena Mokhon *Model-based contracts for C# collections*, Master Thesis, ETH Zürich and Tver State University (Russia), April 2011

Flaviu Roman *Improving relevancy of dynamically-inferred contracts in Eiffel*, Master Thesis, ETH Zürich and Technical University of Cluj-Napoca, June 2009

Talks

Flexible Invariants Through Semantic Collaboration FM'14, May 14 2014, Singapore

To Run What No One Has Run Before: Executing an Intermediate Verification Language RV'13, September 26 2013, Rennes, France

What Good Are Strong Specifications? ICSE'13, May 22 2013, San Francisco, California, USA

Verified calculations. VSTTE'13, May 18 2012, Atherton, California, USA

Verifying implementations of security protocols by refinement. VSTTE'12, January 28 2012, Philadelphia, Pennsylvalina, USA

EiffelBase2: strong contracts for design and verification. Workshop "Eiffel at 25", November 24, 2010, Zürich, Switzerland.

Specifying Reusable Components. VSTTE'10, August 17 2010, Edinburgh, Scotland.

Specifying reusable components with model-based contracts. IFIP WG 2.3 meeting 50. March 5 2010, Lachen, Switzerland.

A comparative study of programmer-written and automatically inferred contracts. ISSTA'09, July 21 2009, Chicago, Illinois, USA.

Applying Genetic Programming to Implementation of Systems with Complex behavior. Integrated Models and Soft Computing in Artificial Intelligence, May 28 2007, Kolomna, Russia.

Applying Genetic Algorithms to Generating Logics in Computational Systems. 4th Young Researchers Conference, April 12 2007, Saint-Petersburg, Russia.

Inheritance Relation for Types with Complex Behavior. 3rd Young Researchers Conference, April 12 2006, Saint-Petersburg, Russia.

Other events attended

Dagstuhl seminar on Evaluating Software Verification Systems: Benchmarks and Competitions. Schloss Dagstuhl, Germany. April 22–25, 2014.

LASER summer school on Innovative Languages for Software Engineering. Elba, Italy. September 2–8, 2012.

VSTTE 2012 Software Verification Competition (**bronze medal**). November 8–10, 2011.

COST Verification Competition. Turin, Italy. October 4, 2011.

LASER summer school on Tools for Practical Software Verification. Elba, Italy. September 4–10, 2011.

FOSE The Future of Software Engineering Symposium. Zürich, Switzerland. November 22–23, 2010.

LASER summer school on Empirical Software Engineering. Elba, Italy. September 5–11, 2010.

VSTTE 2010 Software Verification Competition. Edinburgh, Scotland. August, 2010

SICSA Summer School on Formal Reasoning & Representation of Complex Systems. Edinburgh, Scotland. August 14–15, 2010.

TOOLS-Europe 2009 47th International Conference on Objects, Models, Components, Patterns. Zürich, Switzerland. June 29 – July 3, 2009.

LASER summer school on Concurrency and Correctness. Elba, Italy. September 7–13, 2008.

Professional Activities

PC member, **Tutorials Chair** and **Proceedings Chair**, 14th International Conference on Runtime Verification (RV 2014)

PC member, 11th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2014)

Deputy General Chair, 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2013)

Publicity Chair, LASER Summer School (2011–2014)

Publicity Chair, 4th International Conference on Software Engineering Approaches For Offshore and Outsourced Development (SEAFOOD 2010)

Awards

ACM SIGSOFT Recognition of Services Award in appreciation for the contribution as a Deputy General Chair of ESEC/FSE 2013.

Personal data

Date of birth: 20 May 1985

Place of birth: Leningrad, USSR

Nationality: Russian

Language proficiency

Russian: native

English: fluent

German: intermediate

Italian: fair