Capturing and Exploiting Abstract Views of States in OO Verification

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op woensdag 30 november 2011 om 14.00 uur

door

Ronald Middelkoop

geboren te Utrecht

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. M.G.J. van den Brand

Copromotoren:
dr. R. Kuiper
en
dr. C. Huizing

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics)

A catalogue record is available from the Eindhoven University of Technology Library

ISBN: 978-90-386-2879-0

IPA dissertation series 2011-18

Typeset using LaTeX
Cover design by Frans Goris
Printed by Printservice Technische Universiteit Eindhoven

Preface

While briefly considering going into industry after finishing my Master's, I knew I wanted the opportunity to really sink my teeth into a subject, get a chance to get to the core of a problem. When I was offered a PhD position with much freedom of direction and a subject of practical interest, it did not take me long to decide and it is a choice I've never regretted. My past few years in industry have shown me how much this PhD has taught me about objects, their structure, and the appropriate abstract views, and that experience makes me enjoy my current work so much more.

I'd like to express my gratitude to a number of people who have given me this opportunity and helped me along the way.

First of these are the co-promoters of this thesis, Ruurd Kuiper en Kees Huizing. The combination of your depth-first and breadth-first approaches really suited my style and made it a joy to work with you. I'm grateful for freedom that you gave me, while at the same time structuring my work and never letting me get away with a 'solution' that I hadn't thought through. As a very close second, Erik Luit, for countless hours of reading through countless revisions. Erik, I know I'm still not following all your guidelines, but you've made me a far better technical writer. Third, my promotor, Mark van den Brand, for his continued support. There is no doubt that without him, this thesis would have still been in the making.

Furthermore, professor Peter O'Hearn and Cristiano Calcagno, for first sparking my interest in research during my internship at Queen Mary University. Rustan Leino, for making my stay at Microsoft Research one of the highlights of my time as a PhD student, teaching me everything from SMT solvers to step aerobics. I am very proud to have you as a member of my doctoral committee. The members of FM and SET groups at the TU/e for a nice working environment, and in particular Francien Dechesne for being such a great roommate during the first

years of my promotion. Professor Bruce Watson, for supervising the first years of my promotion. The members of my doctoral committee, for taking the time to read through a lengthy thesis.

The people at my current employer, ISAAC, for their flexibility in allowing me to complete my PhD, as well as for providing me the interesting new problems I was looking for.

My friends, among others for their continued 'support' in the form of banter. Had I ever lost my intrinsic motivation, I'm sure I would have finished this thesis just to not give you the pleasure of rubbing it in. I'm glad that two of you will be available for me to consult during my defense.

My family, for always being supportive no matter what choices I make in life. I have been very, very lucky to have all of you.

And finally, Rianne, the love of my life. Thank you for seeing this through with me, I know this wasn't always easy on you. I'm ready for a new challenge and I can't wait to start it with you.

Contents

CHAPTER 1

Introduction

In this thesis, we study several implementation, specification and verification techniques for Object-Oriented (OO) programs. Our focus is on capturing conceptual structures in OO states in abstractions, and then exploiting such an abstract view of the state in specification and implementation approaches in a way that allows for formal verification.

These days, no introductory section is needed to explain the importance of software. Software has become pervasive in both industrial development and everyday life and there is no sign of its continued increase in importance to stop in the near future.

What we do introduce in this chapter is the concept of OO implementation (Sect. 1). In particular, we highlight the importance of an abstract view of a state in the execution of an OO implementation. We also introduce two formalisms for formal specification that are studied in this thesis. These are algebraic specification (Sect. 2.1) and OO specification (Sect. 2.2). Then, we give a short introduction to the problem of verifying that an implementation satisfies its specification (Sect. 3).

In the course of these introductions we formulate several research questions that underly the work in this thesis. In Sect. 4, we outline the thesis and sketch our contributions with regards to these research questions.

```
class Point {
 int x, y;
 new Point(int xCoordinate, int yCoordinate) {
  x = xCoordinate;
  y = yCoordinate;
 }

 void moveRight(int distance) {
  x = x + distance;
 }
}
```

Example 1.1: Point class

```
class Line {
 Point first, second;
 new Line(Point firstPoint, Point secondPoint) {
  first = firstPoint;
  second = secondPoint;
 }

 void moveRight(int distance) {
  first.moveRight(distance);second.moveRight(distance);
 }
}
```

Example 1.2: Line class

1 OO Implementations

To introduce OO implementations and the need for abstract views of the state, we consider the OO implementation of a simple program for two-dimensional drawing. The core concepts of our implementation are points and lines. Consider Exmpls. 1.1 and 1.2, which show a *class* for each of these two core concepts. A class is the basis of an OO program. It defines state and behavior. State is defined by *fields* and behavior by *methods*. For example, the `Point` class has two integer fields that represent the coordinates of a `Point`. The two methods `new` and `moveRight` define the behavior that a `Point` can be created, and can move right (we have omitted other behavior for simplicity). Note that a class can define that another class is part of its state. For example, the `Line` class defines that its state consists of two `Point`s.

During the execution of an object oriented program, classes act as blueprints. More specifically, a class can be instantiated to create an *object* of that class. For example, the code snippet

```
class Square {
 Line left, top, right, bottom;
 new Square(int bottomLeftX, int bottomLeftY, int size) {
  Point bottomLeft = new Point(bottomLeftX, bottomLeftY);
  Point topLeft = new Point(bottomLeftX, bottomLeftY + size);
  left = new Line(bottomLeft, topLeft);
  Point leftTop = new Point(bottomLeftX, bottomLeftY + size);
  Point rightTop = new Point(bottomLeftX + size, bottomLeftY +
  size);
  top = new Line(leftTop, rightTop);
  Point topRight = new Point(bottomLeftX + size, bottomLeftY +
  size);
  Point bottomRight = new Point(bottomLeftX + size, bottomLeftY);
  right = new Line(topRight, bottomRight);
  Point rightBottom = new Point(bottomLeftX + size, bottomLeftY);
  Point leftBottom = new Point(bottomLeftX, bottomLeftY);
  bottom = new Line(rightBottom, leftBottom);
 }

 void moveRight(int distance) {
  left.moveRight(10);
  top.moveRight(10);
  right.moveRight(10);
  bottom.moveRight(10);
 }
}
```

Example 1.3: Square class, implementation 1

```
 x = new Point(0,0); y = new Point(2,2); z = new Line(x,y);
```

creates two `Point`s and uses them to create one `Line`. Once an object has been created, methods can be called on it. For example, we can continue the previous code with `z.moveRight(4);` to move our `Line` right by four units. What the actual behavior of a method is, is determined by the implementation of that method. For example, the implementation of the `moveRight` method of a `Line` moves both its points right by calling the `moveRight` methods of those `Point`s. Calling the `moveRight` method on a `Point` updates its `x` coordinate.

The program may also offers more advanced things to draw that are constructed from lines, like a `Square`. To create a `Square`, all the user needs to provide are the coordinates of one of the corner `Point`s and the size of the square (i.e., the length of each of its `Line`s). Likewise, when moving a `Square`, the user does not have to describe the impact on the four individual `Line`s. In other words, the abstract view of the `Square` is not as four `Line`s, but as a single, rectangular object.

```
class Square {
 Line left, top, right, bottom;
 new Square(int bottomLeftX, int bottomLeftY, int size) {
  Point bottomLeft = new Point(bottomLeftX, bottomLeftY);
  Point topLeft = new Point(bottomLeftX, bottomLeftY + size);
  left = new Line(bottomLeft, topLeft);
  Point rightTop = new Point(bottomLeftX + size, bottomLeftY +
  size);
  top = new Line(topLeft, rightTop);
  Point bottomRight = new Point(bottomLeftX + size, bottomLeftY);
  right = new Line(rightTop, bottomRight);
  bottom = new Line(bottomRight, bottomLeft);
 }

 void moveRight(int distance) {
  left.moveRight(10);right.moveRight(10);
 }
}
```

Example 1.4: Square class, implementation 2

```
class Square {
 Point bottomLeft;
 int theSize;
 new Square(int bottomLeftX, int bottomLeftY, int size) {
  Point bottomLeft = new Point(bottomLeftX, bottomLeftY);
  theSize = size;
 }

 void moveRight(int distance) {
  bottomLeft.moveRight(10);
 }
}
```

Example 1.5: Square class, implementation 3

A `Square` can then be implemented in different ways. For example, consider Exmpl. 1.3. Note that in this implementation, the conceptual action of drawing a square, which is implemented by `new`, leads to the creation of no less then thirteen objects (a `Square`, four `Lines` and eight `Points`). This implementation is conceptually simple. When we are implementing the `moveRight` method of `Square`, our abstract view of the state is that of four independent lines. To move the `Square`, all we have to understand is how to move a `Line`.

The implementation in Exmpl. 1.4 'only' needs nine objects to represent a square (a `Square`, four `Line`s and four `Point`s). In this implementation, every `Point` is referenced by two different `Line`s (the start of one line is the end of another). This is known as *aliasing*. This has a consequence on the behavior of moving right a `Line` of the `Square`, despite the implementation of the `moveRight` method of `Line` being the same as in the first implementation; As a side-effect of moving one `Line`, the start or end of another `Line` is moved as well. So, despite the `Square` having the same four `Line` fields as in the first implementation, our abstract view differs as the lines are not independent and we somehow have to take that into account when moving the `Square`.

The third implementation (Exmpl. 1.5) needs fewer objects still, but is conceptually the most complicated. When we implement the `moveRight` method, our abstract view is not of lines, but of the more complex square (although in this case, that is an abstraction we can probably manage).

Our first observation is that different implementations lead to different ways in which objects reference each other, and to different abstract views of the objects in the implementation.

Which implementation is chosen may depend on the personal preference of the implementer and the other methods that have to be implemented. It does not matter to the user of the program, who only cares that a `Square` is displayed on the drawing canvas (the part of the screen designated for drawing) as a square.

Even in these simple examples, we see that the OO state consists of many objects, and that these objects reference each other in possibly complicated ways. At the same time, we see that at any one point in the execution of the program, we can reason about what is happening using an abstract view of the state that is much less complicated.

More generally, in a well-designed OO program, most objects have a singular purpose. For example, the purpose of a `Square` object is to represent a square on the drawing canvas. Also, objects do not depend on random other parts of the object structure (these properties are often called *high cohesion* and *loose coupling* [YC78]).

Note that the objects that we have considered so far all represent concepts from the problem domain of the user. However, there are many other types of objects in a program as well. For example, some objects may direct a specific process, like drawing the created drawing objects on a canvas. Some may be structures of drawing objects, like a linked list used by the object directing the drawing process to access the top-level objects to draw. Others may be needed for communication with external systems, like saving a drawing to the disk. Finally, some may not even be relevant to any of the processes the user is interested in. For example, there may be objects that keep track of what is going on and objects that can send a report of what was going on to the implementer if something the user does goes wrong.

Our second observation is that all in all, looking at the naked OO state outside of its context, it will be difficult to make sense of. All we see is a whole bunch of objects with fields that reference each other in many complicated ways. However, given the reasoning above, all sorts of conceptual notions that group objects can be distinguished. Furthermore, it can be distinguished which process(es) an object belongs to, and within such a process, what its purpose is. If we understand this grouping and purpose, then it can be determined if the object is relevant to what is currently important to us, or if it can be abstracted away when we are trying to capture what is going on at that particular point in the program execution.

For example, before the user of our drawing program has drawn his first object, it is likely that a hundred objects have been created. From the abstract view of the user however, the state is a blank canvas.

2 Specifications

For the purpose of this thesis, we distinguish two types of specification. These are client-level and programmer-level specification. Client-level specification is considered first.

Complex OO implementations are usually not written by the implementer for the implementer. Instead, there is a client that needs a particular problem solved. For example, the client may need a simple drawing program. Often the client understands his problem quite well, but the implementer may not. Likewise, we are assuming that the client knows *what* he wants the implementation to do, but maybe not *how* the implementation can solve the problem (in fact, often the client will not even care about the 'how').

How does the client communicate his problem to the implementer? There are several options.

- The client may sit down with the implementer and have a chat about the problem. As programmers are smart people, this approach works surprisingly well if the problem is not too complex and if there is room for trial and error.

- The client may use informal or semi-formal tools to help communicate the problem. A well-know example is the Unified Modeling Language (UML) [RJB99], which provides a number of diagrams that are tailored to specific views of the program. This is a great way to make the complexity of a problem more manageable, but ambiguities in the diagrams may lead to misunderstandings between the implementer and the client.

- The client may write a formal specification. A formal specification is an unambiguous way to describe the functionality that is desired of the system. This is the problem communication style considered in this thesis (we do not consider techniques for specifying non-functional requirements, like those regarding performance. For such techniques, see e.g. [CdPL09]).

Note that writing a formal specification requires a considerable effort on the part of the client, because by its nature it requires all ambiguities to be considered and resolved. It will often require expert help in the form of a *specifier* who has in-dept understanding of the formalism and can make sure (up to a limit) that the formal specification indeed represents the client's problem. This additional effort means that formal specification becomes more suitable when the problem considered becomes more complex or has a higher cost of failure. For example, NASA, who write programs with over a million lines of code on a regular basis, has one of the most rigorous specification processes in the industry.

This does not guarantee that their software is flawless though. Despite their specification efforts, they still lost a satellite for four months, had one Mars explorer spontaneously reset itself every now and then, and another fail temporarily because unneeded files filled up its memory [Bra07].

To further improve the quality of implementations, better techniques must be developed for 1) specification of the abstract views that are used by the client and the programmer, and 2) the verification that an implementation satisfies its specification. This thesis contributes to that effort.

Two specification formalisms are considered in this thesis: Algebraic Specification and OO Specification. These formalisms differ in the basic abstract view that is used. We discuss Algebraic Specification first.

2.1 Algebraic Specification

Client-level specification. The purpose of a client-level specification is to give formal, unambiguous meaning to statements of the client about his problem domain. For example, when describing the simple drawing program, the client can say "I want to be able to draw on the canvas a square of size two that was created with its lower left corner at point (0,0) and then moved right by 4 units". In an algebraic specification (at least in the form considered in this thesis), such statements are formalized by a list of *data types*, a list of *operators* that operate on the data types, and a list of *axioms* about the operators. Roughly, we can think of the data types as the nouns in the problem domain, and of the operators as the verbs. The axioms formalize universal truths about operators, in terms of operators. So, in our simple drawing program the specification includes data types for point, line, square and canvas, and operators for move right, create and draw. Furthermore, there may be an axiom in the specification that captures that moving a square with it bottom left corner at point (x, y) right by z units, is the same as creating a new square of the same size with its bottom left corner at point $(x + z, y)$. The axiom would look something like the following.

$$\begin{array}{l}\forall s : \mathit{Square}, \mathit{size}, x, y, \mathit{distance} : \mathit{int}, p1, p2 : \mathit{Point} \bullet \\ \quad p1 = \mathit{createPoint}(x, y) \wedge p2 = \mathit{createPoint}(x + \mathit{distance}, y) \;\Rightarrow \\ \quad\quad \mathit{moveRight}(\mathit{createSquare}(p1, \mathit{size}), \mathit{distance}) = \mathit{createSquare}(p2, \mathit{size})\end{array}$$

However, by itself, this suggests but does not really specify the desired behavior of the move right operation. To make this more clear, consider the specification of a simple calculator in which rational numbers can be added. The axioms are well known, e.g.

$$\forall n_0, n_1, d : int \quad \bullet \quad d \neq 0 \Rightarrow (\frac{n_0}{d} + \frac{n_1}{d} \Leftrightarrow \frac{n_0 + n_1}{d})$$

$$\forall n_0, n_1, d_0, d_1 : int \quad \bullet \quad d_0 \neq 0 \wedge d_1 \neq 0 \Rightarrow (\frac{n_0}{d_0} = \frac{n_1}{d_1} \Leftrightarrow n_0 \times d_1 = n_1 \times d_0)$$

These axioms allow to derive the following sequence of equalities.

$$\frac{2}{4} + \frac{2}{6} = \frac{12}{24} + \frac{8}{24} = \frac{20}{24} = \frac{5}{6} = \frac{100}{120}$$

The equalities express that each of the fractional expressions above represents the same rational number. However, when the client inputs the first expression to the calculator, he expects the expression $\frac{5}{6}$ as output. This expected input/output behavior cannot be derived from the axioms. In other words, the axioms describe what is (and is not) true about the operators. The axioms do not, however, drive the computation. This observation leads us to our first research question.

Question 1 What are the syntax and semantics of a client specification based on algebraic specification, independent of the implementation used?

Programmer-level specification. A formal client specification can be used to assign blame in case an output does not match the expectation of the client. It can also be used as a starting point for a development style that allows formal reasoning about the implementation. As was argued, reasoning about OO implementations is difficult due to the complexity of a state in the execution of a OO program.

However, consider the following informal reasoning.

1. Moving a rectangle to the right by *distance* units, is the same as drawing a new rectangle *distance* units further to the right.

2. Drawing a new rectangle is the same as drawing the four lines of which it consists.

3. Moving a line to the right by *distance* units, is the same as drawing a new line *distance* units to the right.

4. Therefore, moving a rectangle to the right by *distance* units, is the same as moving each of the four lines of which it consists to the right by *distance* units.

This reasoning can be formalized in an algebraic specification, in a way similar to the formalization of addition and equality on rational numbers and the subsequent derivation of equalities given above. Note that this reasoning exactly matches the reasoning behind the implementation of the `moveRight` method of class `Square` in Exmpl. 1.3. More generally, when the specification captures the abstract view that the implementer has of the state, it can be exploited to reason formally about the implementation. This leads us to the second research question.

Question 2 When reasoning about an OO implementation, how can we use programmer-level algebraic specification to capture and exploit an abstract view of a state?

2.2 OO Specification

Specification in terms of object configurations. Note that the informal reasoning with which we concluded Sect. 2.1 applies to the implementation in Exmpl. 1.3, but not to the implementation in Exmpl. 1.4. In this implementation, moving the `Square` is not the same as moving its four `Lines` as moving one of the `Square`'s `Line`s affects another. We may be able to find other natural data types with an axiomatization that abstracts from this side-effect. However, in OO there are several design patterns that are conceptually understood in terms of cooperation between objects, i.e., where a configuration of objects together implement a common goal. A well-know example is the Observer Pattern [GHJV95]. In this pattern the intuition is that when an object designated as the `Subject` is changed, it notifies a set of objects designated as its `Observer`s, so that they can update their state as well.

OO specifications have the potential for more natural formalization of such patterns. An OO specification specifies, for every method, two properties of states known as the *precondition* and the *postcondition*. The semantics of the specification of any given method is as follows. For every state for which the precondition holds, the execution of that method terminates in a state for which the postcondition holds.

That is, an OO specification invites to think about the implementation in terms of methods that change object configurations. However, given the complexity of the OO state discussed in Sect. 1, it is vital that abstractions in the state are still captured and exploited somehow. So, the second research question is equally interesting when it is considered in the context of OO specifications rather than algebraic specifications. This leads us to the third research question.

Question 3 When reasoning about an OO implementation, how can we use programmer-level OO specification to capture and exploit an abstract view of a state?

Class invariants. One way of exploiting abstractions in OO state is of particular interest to this thesis. This is the exploitation of abstractions through the use of *class invariants*. A class invariant describes, from the perspective of the class in which it is specified, a property of an object configuration that is relevant to an object of that class. Roughly, the intuition is that the property is expected to hold, unless the conceptual value that the object represents is in the process of being changed. An example is an invariant of class `Square` from Exmpl. 1.3 that captures that the top of the `left Line` is the same point as the left of the `top Line` (and likewise for other `Line` pairs of the `Square`). Given the code in Exmpl. 1.3, if we assume that the fields shown are not assigned to from elsewhere, then the property holds unless the object is in the process of moving right. More specifically, note that the property does not hold in the state where the first line has been moved right, but the second one has not. When this property does not hold for a `Square` object, we cannot think of that `Square` as a square.

More generally, if the invariant of a class C does not hold for a certain C object, then the object cannot be thought of in terms of its abstract value. If the implementer of a class D reasons about that object in terms of its abstract value, then it must be ensured that the invariant holds if execution is in a method of class D. Capturing which classes reason about an object in terms of its abstract value (and which do not), is a common theme of several chapters of this thesis.

3 Verification

Verification is the process of establishing that an implementation satisfies its specification. Having a problem-independent *verification approach*, can greatly reduce the burden of verification compared to the use of ad-hoc problem-dependent reasoning. A verification approach consists of a fixed set of proof obligations and syntactical restrictions. It comes with a set of meta-level proofs that establish that the obligations and restrictions are sufficient to guarantee satisfaction. Ideally, the approach is supported by tools that allow for (partial) automatic verification. A verification approach has to be chosen with care as it is likely to impose restrictions that allow for problem-independence at the cost of being stronger than strictly necessary for a specific problem. This leads us to the final research question.

Question 4 Having used the techniques from research questions 2 and 3, how can we provide matching verification approaches?

4 Contributions and Outline

In this section we give an outline of the further chapters of this thesis. These chapters are presented and can be read as separate papers. We relate the chapters to the research questions that we formulated in Sect. 2. We also relate the chapters to our earlier publications.

- **Chap. 2 (questions 1,2 and 4), Algebraic specification and its class-based implementation:** In this chapter, we consider both client-level and programmer-level specifications based on algebraic specification. We contribute a novel syntax and semantics for the former, and we contribute an implementation approach for OO implementations based on the latter. We show that the implementation approach is suitable for problem-independent verification.

- **Chap. 3 (questions 3 and 4), Cooperation-based invariants for OO languages:** In this chapter, we contribute the programmer-level specification constructs *inc* and *coop*. The inc construct allows a method specification to make explicit that a certain enumeration of invariants does not have to hold when that method is executed. The coop construct allows a field specification to make explicit that a certain enumeration of invariants might be invalidated when the field is updated. This allows for the specification and verification of OO designs in which in the process of updating one object, other objects with which it together implements a common purpose must be updated as well. The work in this chapter was published as [1].

- **Chap. 4 (question 3 and 4), Invariants for non-hierarchical object structures:** In this chapter, we generalize the inc and coop constructs by removing a restriction to enumerations of invariants. For instance, this is needed in the Observer Pattern discussed in Sect. 2.2, where a `Subject` can have an arbitrary and dynamically changing number of `Observers`. A more general interpretation of invariants and accompanying proof system are provided as well. The work in this chapter was published as [2].

- **Chap. 5 (question 3 and 4), Specifying and exploiting layers in OO designs:** In this chapter, we contribute a programmer-level specification technique to capture *layers* in OO architectures, and we exploit these layers by providing a more liberal semantics of class invariants. We also provide a verification technique for the semantics. Layers are an abstraction at the architectural level in OO implementations that designate certain object structures in the design as sub-structures that are shared by other structures. An object in a higher layer is not relevant to the purpose of an object in the sub-structure. Given this intuition, an object in a higher layer is not part of the abstract view from an object in a lower layer. Therefore, the invariant of a higher layer object does not have to hold when a method of a lower-layer object is executing. The work in this chapter was published as [3] and is based on earlier work from [A]. The chapter has an accompanying technical report [B].

- **Chap. 6 (question 4), Consistency of pure methods and model fields:** In this chapter, we contribute a verification technique for *pure methods* and *model fields*, which are existing specification techniques for capturing an abstract view of the state in OO specifications. A method that is pure can be used as a function in predicates in class specifications. The function is axiomatized using the pre- and postcondition that are specified for the method. A model field abstracts part of the concrete state of an object into an abstract value. This too introduces an additional axiom in the underlying reasoning. The technique contributed establishes that such additional axioms do no introduce inconsistencies into the formal reasoning. It comes with heuristics that that make it amenable to automatic verification. The work in this chapter was published as [4].

Publications.

[1] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik Luit. Cooperation-based Invariants for OO Languages. In Zhiming Liu and Luís Barbosa, editors, *Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005)*, volume 160 of *Electronic Notes in Theoretical Computer Science*, pages 225–237. Elsevier, 2007

[2] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik J. Luit. Invariants for non-hierarchical object structures. In L. Ribeiro and A. Martins Moreira, editors, *Proceedings of the 9th Brazilian Symposium on Formal Methods (SBMF'06)*, volume 195C of *Electronic Notes in Theoretical Computer Science*, pages 211–229. Elsevier, 2008

[3] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik J. Luit. Specification and Verification of Invariants by Exploiting Layers in OO Designs. *Fundamenta Informaticae*, 85(1-4):377–398, 2008. Special issue: Concurrency, Specification and Programming (CS&P'07)

[4] K. Rustan M. Leino and Ronald Middelkoop. Proving consistency of pure methods and model fields. In Marsha Chechik and Martin Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2009

Supporting Publications.

[A] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik J. Luit. A new interpretation of invariants exploiting layers in OO designs. In *Proceedings of the Workshop on Concurrency, Specification and Programming (CS&P'07), September 27-29, 2007, Lagós, Poland*, 2007

[B] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik J. Luit. A proof system for invariants in layered OO designs. Technical Report CSR 08-01, Department of of Mathematics and Computer Science, Technische Universiteit Eindhoven, 2008

CHAPTER 2

Algebraic Specification and its Class-Based Implementation

1 Introduction

In this chapter, we study algebraic specifications and class-based implementations. Algebraic specification is a much-studied approach to specification. It is of particular interest to us because it is at the basis of many object-oriented specification and verification techniques. The work in this chapter can be used as a foundation for the study of some of the problems of such techniques, in a simpler setting.

The contributions of this chapter are the following.

- A novel syntax and semantics of client specifications that are based on algebraic specifications. The semantics matches the client's view of the implementation as a black box.
- A novel notion of satisfaction for class-based implementations.
- An implementation approach that formalizes and extends ideas from [Hoa72], Hoare's seminal paper on data abstraction. In addition to the Hoare-style notion of satisfaction, this approach accounts for the requirement to present the result to the user. Key parts of the approach are suitable for automatic, modular verification.

This chapter is structured as follows.

- In Sect. 2, we look at algebraic specifications from the perspective of the client and the specifier. In Sect. 2.1, we give a brief overview of first-order logic, on which algebraic specifications are based. In Sect. 2.2, we discuss and formalize algebraic specifications. We introduce a syntax and semantics of algebraic specifications that views the implementation as a black box, and that is independent of the implementation language.

- In Sect. 3, we switch to the perspective of the implementer and open up the black box. The difficulty with algebraic specifications is that the connection to implementations, i.e., to a notion of satisfaction, is not obvious. Our solution is to separate the implementation into a presentation layer and a business layer. This allows for a natural notion of satisfaction.

  In Sect. 3.1, we present the syntax and semantics of a simple class-based programming language for the business layer. In Sect. 3.2, we present the separation into a presentation and business layers and the resulting notion of satisfaction.

  The business layer is responsible for the actual computation. The presentation logic layer is only responsible for the translation from the input provided by the user to the input required by the computation in the business logic layer, and for displaying the output of the computation to the user as a closed application. In Sect. 3.3, we sketch generic (i.e., specification-independent) algorithms for the presentation layer. This allows refine the notion of satisfaction for an implementation, to a notion of satisfaction for the computation concern (i.e., for the business layer).

- In Sect. 4, we present an implementation approach that has a Hoare-style notion of data abstraction at its core.

  In Sect. 4.1, we connect the satisfaction notion for the computation concern, to a notion of satisfaction that is based on Hoare's notion of data abstraction. We observe that this Hoare-style satisfaction alone is not sufficient to establish satisfaction for computation as it does not account for the requirement of displaying the output of the computation in a generic way. As a solution, we add an additional step to the computation process that is responsible for the transformation of the computed object, to an object that is suitable for display.

  In Sect. 4.2, we present an implementation approach for the Hoare-style satisfaction notion. More specifically, we start with the satisfaction notion and make a series of design decisions that formalize and extend ideas from Hoare's work. The advantage of the approach is that it allows the implementer of a method to reason about method calls as if they do not have side-effects. We observe that the downside of the approach is that it does not allow for OO solutions where objects cooperate to achieve a goal, as side-effects are an integral part of such solutions.

- In Sect. 5, we switch to the perspective of the verifier of the business layer. We sketch a verification technique that allows to establish that a given implementation of the computation concern that follows the implementation approach presented in Sect. 3, satisfies the specification. The technique is suitable for automatic verification.

- In Sect. 6, we present conclusions and discuss future work.

1.1 Meta-level notation

Here, we discuss the meta-level notation that is used in this chapter.

Functions. $f : A \hookrightarrow B$ introduces a partial function from A to B. Functions are treated as single-valued relations. Given a property $P(a, b)$, we write '$f(a) = b$ if and only if (iff) $P(a, b)$' to define f as the smallest single-valued relation such that $\forall a \in A, b \in B \bullet (a\ f\ b$ iff $P(a, b)$ holds). With most of our definitions, existence of a unique smallest single-valued relation is obvious. Additional remarks are provided where this is not the case. $f : A \rightarrow B$ introduces a total function from A to B. $\mathit{Domain}(f)$ and $\mathit{Range}(f)$ are sequences that denote the domain and range of function f.
$f[a \mapsto b]$ is the function like f, but with a mapped to b. If f is a partial function, then this can be used whether or not $a \in \mathit{Domain}(f)$. $f[a \mapsto b, \ldots, c \mapsto d]$ is shorthand for $f[a \mapsto b] \ldots [c \mapsto d]$.

Aside. In the digital version of this chapter, all functions are hyperlinked to their definition. Most PDF readers support backwards and forwards navigation through hyperlinks, making it much easier to look up definitions while reading.

Sequences and sets. We use n as the typical element of the set of natural numbers $\mathbb{N}$ (which includes 0). A sequence $A_1, \ldots, A_n$ can be the empty sequence, whereas sequence $A_0, \ldots, A_n$ has at least one element. A *record* $\mathit{Record} : \mathit{f}_1 \in A_1 \times \ldots \times \mathit{f}_n \in A_n$ denotes a tuple $\mathit{Record} : A_1 \times \ldots \times A_n$, of which the elements are indexed by names $\mathit{f}_1, \ldots, \mathit{f}_n$. $\mathit{Record}.\mathit{f}_i$ denotes the value of the field with name f_i of Record. Given a sequence Σ, $\Sigma[i]$, $\Sigma[i, j]$ and $\Sigma[i..]$ denote element, consecutive subsequence and postfix. $\langle\rangle$ denotes the empty sequence. $\Sigma_0 \triangleright \Sigma_1$ denotes the concatenation of sequences Σ_0 and Σ_1. We write $\overline{x} \in \mathit{Seq}(x)$ to denote that $\overline{x}$ is a sequence of elements from X. Given a set X, we write $\mathit{xSet} \in \mathit{Set}(X)$ to denote that xSet is a set of elements from X. $|A|$ denotes the length of sequence or set A.

Equality. 'A is B' denotes that A and B are syntactically the same. $A = B$ (strong equality) denotes that A and B are both defined and have the same interpretation, i.e., that both evaluate to a value, and that these values are syntactically the same.

2 Specifier's Perspective

In this section, we look at algebraic specifications from the perspective of the client and the specifier.

(1) In Sect. 2.1, we give a brief overview of first-order logic, on which algebraic specifications are based.
(2) In Sect. 2.2, we discuss and formalize algebraic specifications. We introduce a novel syntax and semantics of algebraic specifications that views the implementation as a black box, and that is independent of the implementation language.

2.1 The Formalism: First-Order Logic

Many-sorted first-order logic is at the basis of many program specification and verification techniques. In this section, we give a brief introduction to many-sorted partial first-order logic with equality to fix the notation and terminology that we use. More thorough treatments can be found in, e.g., [CMR98, ST99]. Note that operations in the problem domain are often partial (consider e.g. the division operation i/j, which is undefined when $j = 0$). The use of partial logic allows to specify such operations more directly (which does not mean that a verification system has to use partial logic, see section Sect. 5).

2.1.1 Syntax

Here, we formalize the syntax of multi-sorted first order logic.

Sorts, variables and operators. In this paragraph, we introduce the primitive elements of (the syntax of) a first-order logic.

Definition 2.1.1. *Sort* is the set of *sorts*

Definition 2.1.2. *Var* is the set of *variables.* Each variable has associated with it a sort S. Var_S denotes the set of variables of sort S. We write v_S to denote that $v_S \in Var_S$.

Consider Def. 2.1.3. The function *VarSort* yields the sort of a variable.

Definition 2.1.3. $VarSort : Var \to Sort$
$VarSort(v_S) = S$.

Consider Def. 2.1.4, in which we define a notion of an operator. Note that for simplicity, we do not consider polymorphism in this chapter. If we did, this definition would have to be changed slightly.

Definition 2.1.4. *Op* is the set of *operators.* Every operator has associated with it a tuple of *domain sorts* $\langle S_1, \ldots, S_n \rangle$ and a *range sort* S_0, where $S_0, \ldots, S_n \in \mathit{Sort}$. We write $o : S_1 \times \ldots \times S_n \hookrightarrow S_0$ to denote an operator with domain sorts $\langle S_1, \ldots, S_n \rangle$ and range sort S_0.

Consider Defs. 2.1.5 and 2.1.6. The functions *DomainSorts* and *RangeSort* yield the domain sorts and the range sort of an operator.

Definition 2.1.5. $\mathit{DomainSorts} : \mathit{Op} \rightarrow \mathit{Seq}(\mathit{Sort})$
$\mathit{DomainSorts}(o : S_1 \times \ldots \times S_n \hookrightarrow S_0) = \langle S_1, \ldots, S_n \rangle$.

Definition 2.1.6. $\mathit{RangeSort} : \mathit{Op} \rightarrow \mathit{Sort}$
$\mathit{RangeSort}(o : S_1 \times \ldots \times S_n \hookrightarrow S_0) = S_0$.

Definition 2.1.7. A *constant* is an operator o such that $\mathit{DomainSorts}(o) = \langle\rangle$. *Constant* is the set of all constants.

Signatures. A signature fixes a set of symbols that are used to construct terms. Every signature contains three sets of special-purpose operators (that come with a predefined meaning, see Sect. 2.1.2).

We first introduce the three sets of predefined operators (Def. 2.1.8-2.1.11). Then we formally define signatures (Def. 2.1.12).

Definition 2.1.8. *BoolOp* is the set that consists of the two constants $\mathit{true} : \hookrightarrow \mathit{Bool}$ and $\mathit{false} : \hookrightarrow \mathit{Bool}$.

Definition 2.1.9. *LogConOp* is the set that consists of the usual logical connectives (e.g. $\wedge : \mathit{Bool} \times \mathit{Bool} \hookrightarrow \mathit{Bool}$). *LogConOp* also includes, for every $S \in \mathit{Sorts}$, the operator $=_S : S \times S \hookrightarrow \mathit{Bool}$.

Definition 2.1.10. *QuantOp* is the set that consists of, for every $S \in \mathit{Sorts}$, the operators $\forall_S : S \times \mathit{Bool} \hookrightarrow \mathit{Bool}$ and $\exists_S : S \times \mathit{Bool} \hookrightarrow \mathit{Bool}$.

Definition 2.1.11. *PredefinedOp* is the set $\mathit{BoolOp} \cup \mathit{LogConOp} \cup \mathit{QuantOp}$.

Consider Def. 2.1.12. A signature *sig* is a record that consists of three parts: a set of sorts *sig.sorts*, a set of operators *sig.ops*, and a subset *sig.tOps* of *sig.ops* (this is the subset of total operators, see Def. 2.1.2). *sig.tOps* contains every predefined operator with all domain sorts in *sig.sorts*. Note that this includes the set *BoolOp*. Furthermore, for every operator o in *sig.ops*, every domain and range sort of o is a sort from *sig.sorts*. Note that therefore $\mathit{Bool} \in \mathit{sig.sorts}$, as $\mathit{BoolOp} \in \mathit{sig.ops}$.

Definition 2.1.12. A *signature* is a record $\mathit{sig} : \mathit{sorts} \in \mathit{Set}(\mathit{Sort}) \times \mathit{ops} \in \mathit{Set}(\mathit{Op}) \times \mathit{tOps} \in \mathit{Set}(\mathit{Op})$ such that
  $\mathit{sig.tOps} \subseteq \mathit{sig.Ops}$, and
  $\{o \in \mathit{PredefinedOp} \mid \mathit{DomainSorts}(o) \subseteq \mathit{sig.sorts}\} \subseteq \mathit{sig.tOps}$, and
  for every $o \in \mathit{sig.ops}$,
    $\mathit{DomainSorts}(o) \subseteq \mathit{sig.sorts}$, and
    $\mathit{RangeSort}(o) \in \mathit{sig.sorts}$.
Sig is the set of all signatures.

Consider Def. 2.1.13. $\mathit{SigUnion}(sig_0, sig_1)$ returns the signature that unites sig_0 and sig_1 in the obvious way.

Definition 2.1.13. $\mathit{SigUnion} : \mathit{Sig} \times \ \mathit{Sig} \rightarrow \mathit{Sig}$
$\mathit{SigUnion}(sig_0, sig_1) = sig_2$ iff
$sig_2.sorts = sig_0.sorts \cup sig_1.sorts$, and
$sig_2.ops = sig_0.ops \cup sig_1.ops$, and
$sig_2.tOps = sig_0.tOps \cup sig_1.tOps$.

Terms. Terms are constructed from variables and operators.

Definition 2.1.14. A *term* t (and its sort) is inductively defined as one of the following:

(1) A variable v of a sort S. In this case, the sort of t is S.
(2) A tuple $\langle o, \langle t_1, \ldots, t_n\rangle\rangle$ of (A) an operator $o : S_1 \times \ldots \times S_n \hookrightarrow S_0$, and (B) a tuple of terms $\langle t_1, \ldots, t_n\rangle$ such that
for every $i \in [1, n]$, t_i has sort S_i, and
if $o \in \mathit{QuantOp}$, then t_1 is a variable.
In this case, the sort of t is S_0 (the range sort of the operator).

Term is the set of all terms.

We usually write a term $\langle o, \langle t_1, \ldots, t_n\rangle\rangle$ as $o(t_1, \ldots, t_n)$. Additionally, when $o \in \mathit{QuantOp}$, we know that n is 2 and t_1 is a variable v_S, and we usually write the term as $o\ v : S \bullet t_2$. Furthermore, we usually write $o()$ as o if it is clear from the context that o is a used as an term and not as an operator. E.g., we often write *true* instead of *true*() and 1 instead of 1(). Finally, we often write certain well-known operators using infix notation. E.g., we may write $4 + 2$ instead of $+(4, 2)$.

Consider Def. 2.1.15. An application is a term that is not a variable, and in which no predefined operators occur (the latter is for technical reasons).

Definition 2.1.15. An *application* is a term $o_0(t_1, \ldots, t_n)$ in which no $o_1 \in \mathit{PredefinedOp}$ occurs. *Application* is the set of all applications.

We conclude this section with several definitions that are used in the rest of the chapter, most of which are well-known.

Definition 2.1.16. Variable v_0 is *free* in term t iff (1) t is v_0, or (2) t is $o\ v_1 : S \bullet t$ and v_0 is not v_1 and v_0 is free in t, or (3) t is $o(t_1, \ldots, t_n)$ and $o \notin \mathit{QuantOp}$ and there is a $i \in [1, n]$ such that v_0 is free in t_i. $\mathit{Free}(t)$ denotes the set of free variables in t.

Definition 2.1.17. Term t is *closed* if it contains no free variables. *ClosedAppl* is the set of all closed applications.

Definition 2.1.18. A *predicate* is a term of sort *Bool*. *Predicate* is the set of all predicates.

Definition 2.1.19. A *sentence* s is a closed predicate. *Sentence* is the set of all sentences.

Consider Def. 2.1.20. Roughly, *Terms* maps a signature *sig* to the set of all terms that can be built from operators and sorts from *sig*.

Definition 2.1.20. $Terms : Sig \to Set(Term)$
$t \in Terms(sig)$ *iff*
  or *there are* $S \in sig.sorts,\ v \in Var_S$ *such that* t *is* v,
  or *there are* $o \in sig.ops,\ t_1, \ldots, t_n \in Terms(sig)$ *such that* t *is* $o(t_1, \ldots, t_n)$.

Definition 2.1.21. $o_0 \in Op$ *occurs in* $t_{n+1} \in Term$ iff
  there are $o_1 \in Op$, $t_1, \ldots, t_n \in Term$ such that
    t is $o_1(t_1, \ldots, t_n)$, and
    either o_0 is o_1, or there is an $i \in [1, n]$ such that o_0 occurs in t_i.

Consider Def. 2.1.22. *ClosedAppls*(sig) is the set of all closed applications (Def. 2.1.17) in *Terms*(sig).

Definition 2.1.22. $ClosedAppls : Sig \to Set(ClosedAppl)$
$ca \in ClosedAppls(sig)$ *iff* $ca \in Terms(sig)$

2.1.2 Semantics

In this section we formalize the well-known notions of an algebra and a valuation to define the semantic meaning of a term. More specifically, we define how to evaluate a term to a value (Def. 2.1.31). We then use this evaluation to formalize the notion of a model, which relates algebras and sentences.

Algebras. Consider Defs. 2.1.23 to 2.1.26, in which we define an algebra and its parts. A carrier function associates certain sorts with non-empty sets of values. An interpretation function associates certain operators with functions. Note that an interpretation function does not associate pre-defined operations with functions. An algebra has an interpretation function that associates operators with functions of which the domain and range values are from the carrier sets of the domain and range sorts of the operator. Note that the interpretation of an $o \in PredefinedOp$ does not depend on the algebra, but is always the same and is as expected. For example, $interpretation(\wedge : Bool \times Bool \to Bool)$ is the function $\mathfrak{and} : \{\mathcal{T}, \mathcal{F}\} \times \{\mathcal{T}, \mathcal{F}\} \to \{\mathcal{T}, \mathcal{F}\}$ such that $\mathfrak{and}(a, b) = \mathcal{T}$ iff $a = \mathcal{T}$ and $b = \mathcal{T}$.

Definition 2.1.23. $\mathbb{V}$ is the set of *values*.

Definition 2.1.24. A *carrier function* is a function $carrier : Sort \hookrightarrow Set(\mathbb{V})$ such that
  $carrier(Bool) = \{\mathcal{T}, \mathcal{F}\}$, and
  for every $S \in Domain(carrier)$, $|\, carrier(S)\, | > 0$.
Carrier is the set of all carrier functions.

Definition 2.1.25. An *interpretation function* is a function $\mathit{interpretation} : \mathit{Op} \hookrightarrow \mathit{Function}$ such that
 for every $o \in \mathit{PredefinedOp}$, $\mathit{interpretation}(o)$ is as usual (see e.g. the interpretation of $\wedge$ above).
Interpretation is the set of all interpretation functions.

Definition 2.1.26. An *algebra* is a record $\mathcal{A} : \mathit{carrier} \in \mathit{Carrier} \times \mathit{interpretation} \in \mathit{Interpretation}$ such that for every $o : S_1 \times \ldots \times S_n \hookrightarrow S_0 \in \mathit{Domain}(\mathcal{A}.\mathit{interpretation})$,
 $\mathit{Domain}(\mathcal{A}.\mathit{interpretation}(o)) = \langle \mathcal{A}.\mathit{carrier}(S_1), \ldots, \mathcal{A}.\mathit{carrier}(S_n) \rangle$, and
 $\mathit{Range}(\mathcal{A}.\mathit{interpretation}(o)) = \mathcal{A}.\mathit{carrier}(S_0)$.
Alg is the set of all algebras.

Consider Def. 2.1.27. For convenience, it defines a notion of inclusion on algebras in the obvious way.

Definition 2.1.27. $\subseteq : \mathit{Alg} \times \mathit{Alg} \to \mathit{Bool}$
$\mathcal{A}_0 \subseteq \mathcal{A}_1$ iff
 $\mathcal{A}_1.\mathit{carrier} \subseteq \mathcal{A}_0.\mathit{carrier}$, and
 $\mathcal{A}_1.\mathit{interpretation} \subseteq \mathcal{A}_0.\mathit{interpretation}$.

Valuations. Consider Def. 2.1.28. A valuation (sometimes called a variable assignment) maps variables to values.

Definition 2.1.28. A *valuation* is a partial function $\mathit{va} : \mathit{Var} \hookrightarrow \mathbb{V}$. *Valuation* is the set of all valuations.

Definition 2.1.29. *emptyva* is the valuation with $\mathit{Domain}(\mathit{emptyva}) = \{\}$.

Definition 2.1.30. $\mathit{va} \in \mathit{Valuation}$ is *well-sorted for* $\mathcal{A} \in \mathit{Alg}$ iff for every $v_S \in \mathit{Var}$, if $\mathit{va}(v_S) = \nu$, then $\nu \in \mathcal{A}.\mathit{carrier}(S)$. $\mathit{WellSortedVas}\mathcal{A}$ is the set of all valuations that are well-sorted for $\mathcal{A}$.

Term evaluation. The semantics of terms in total first order logic is well-known (see e.g. [GH93]). Partial logic has to in addition deal with undefined terms, where for a given algebra $\mathcal{A}$, the arguments of an application of operator o evaluate to values outside the domain of $\mathcal{A}.\mathit{interpretation}(o)$, or where a variable is not in the domain of the valuation. To deal with undefined terms, several choices can be made (see [CMR98] for an overview) For concreteness, this chapter follows the approach from [CMR98], where logical connectives and quantifiers are total operators that are false when applied to terms that do not evaluate to a value (sometimes called 'negative logic'). Other choices, however, are possible and do not really influence the problems studied in this chapter.

Consider Def. 2.1.31. Function $\mathit{Sem}(t, \mathit{va}, \mathcal{A})$ evaluates term t in algebra $\mathcal{A}$ under valuation va. The definition of $\mathit{Sem}(t, \mathit{va}, \mathcal{A})$ is straightforward.

Definition 2.1.31 (semantics of terms). $\mathit{Sem} : \mathit{Term} \times \mathit{Valuation} \times \mathit{Alg} \hookrightarrow \mathbb{V}$
$\mathit{Sem}(t_0, \mathit{va}, \mathcal{A}) = \nu$ *iff*

```
or  $t_0 \in \mathit{Var}$ and $\nu = \mathit{va}(t_0)$,
or  $t_0$ is true and $\nu = \mathcal{T}$,
or  $t_0$ is false and $\nu = \mathcal{F}$,
or  there are $o \in \mathit{Op}$, $t_1, \ldots, t_n \in \mathit{Term}$ such that
      $t_0$ is $o(t_1, \ldots, t_n)$, and
      let $\mathfrak{f}$ be $\mathcal{A}.\mathit{interpretation}(o)$ in $\nu = \mathfrak{f}(\mathit{Sem}(t_1, \mathit{va}, \mathcal{A}), \ldots, \mathit{Sem}(t_n, \mathit{va}, \mathcal{A}))$,
or  there are $v_S \in \mathit{Var}, t_1 \in \mathit{Term}$ such that
      or  $t_0$ is $\exists v : S \bullet t_1$, and
            if there is a $\nu \in \mathcal{A}.\mathit{carrier}(S)$ such that $\mathit{Sem}(t_1, \mathit{va}[v_S \mapsto \nu], \mathcal{A}) = \mathcal{T}$,
            then $\nu = \mathcal{T}$, else $\nu = \mathcal{F}$,
      or  $t_0$ is $\forall v : S \bullet t_1$, and
            if for every $\nu \in \mathcal{A}.\mathit{carrier}(S)$, $\mathit{Sem}(t_1, \mathit{va}[v_S \mapsto \nu], \mathcal{A}) = \mathcal{T}$,
            then $\nu = \mathcal{T}$, else $\nu = \mathcal{F}$,
or  there are $t_1, t_2 \in \mathit{Term}$ such that
      $t_0$ is $t_1 = t_2$, and
      if $\mathit{Sem}(t_1, \mathit{va}, \mathcal{A}) = \mathit{Sem}(t_2, \mathit{va}, \mathcal{A})$, then $\nu = \mathcal{T}$, else $\nu = \mathcal{F}$.
```

Models. Consider Def. 2.1.32. Note that the value of a sentence s does not depend on the valuation as a sentence does not contain free variables. So, if there is a $\mathit{va} \in \mathit{Valuation}$ such that $\mathit{Sem}(s, \mathit{va}, \mathcal{A}) = \mathcal{T}$, then for every $\mathit{va} \in \mathit{Valuation}$, $\mathit{Sem}(s, \mathit{va}, \mathcal{A}) = \mathcal{T}$.

Definition 2.1.32. $\mathcal{A} \in \mathit{Alg}$ is a *model* for $s \in \mathit{Sentence}$ iff for every $\mathit{va} \in \mathit{Valuation}$, $\mathit{Sem}(s, \mathit{va}, \mathcal{A}) = \mathcal{T}$.

Definition 2.1.33. $\mathcal{A} \in \mathit{Alg}$ is a *model* for $\mathit{Sen} \in \mathit{Set}(\mathit{Sentence})$ iff for every $s \in \mathit{Sen}$, $\mathcal{A}$ is a model for s.

Consider Def. 2.1.34. If Sen is a set of sentences, then $\mathit{Models}(\mathit{Sen})$ is the set of all models of Sen.

Definition 2.1.34. $\mathit{Models} : \mathit{Set}(\mathit{Sentence}) \to \mathit{Set}(\mathit{Alg})$
$\mathcal{A} \in \mathit{Models}(\mathit{Sen})$ *iff* $\mathcal{A}$ *is a model for* Sen.

2.2 Algebraic Client Specification

In this section, we discuss client specifications that are based on algebraic specifications. Our aim is to come up with a flexible, foundational notion of specification. To this end, we introduce a novel syntax (Sect. 2.2.1) that incorporates a notion of canonicity, and a novel semantics (Sect. 2.2.2) that closely matches the client's view of the implementation as a black box.

These specifications are particularly suited to situations where the client is interested only in the input/output behavior of the implementation, i.e., situations where the client has a set of possible questions that the implementation should compute the answers to. In such situations, the client expects the implementation to rewrite an input into an equivalent, most basic form. Note that the client can then use this output as the basis for another input.

In our formalism, a client specification consists of an algebraic specification and a canonicity function. An algebraic specification consists of 1) a signature that describe the sorts and operators of the client's problem domain, and 2) a set of sentences, called *axioms* in this context, that formalize the properties that the client desires of the operators in the signature. The canonicity function determines which elements of the set of possible outputs are of a most basic form.

The meaning of a client specification consisting of a signature *sig*, a set of axioms *ax* and a canonicity function *isCanonical* is as follows.

- Every model $\mathcal{A}$ of *ax* provide a notion of equality that is acceptable to the client, as *ax* formalizes the desired properties of the operators in the signature. In particular, two closed applications ca_0 and ca_1 are equal in a model $\mathcal{A}$ iff $Sem(ca_0 = ca_1, emptyva, \mathcal{A}) = \mathcal{T}$.
- Therefore, every model $\mathcal{A}$ of *ax* induces a division of $ClosedAppls(sig)$, which are the closed applications that can be formed using only the operators in the signature, into equivalence classes.
- The canonicity function *isCanonical* determines for each such set of equivalence classes, the set of class representatives (with a small caveat: our formalism does not forbid multiple representatives per equivalence class, instead the representatives of a given class are determined by the specifier through the canonicity function).
- An implementation satisfies the specification iff for one of these sets of equivalence classes, given any input *in* of any equivalence class *EqClass*, the implementation outputs a representative of *EqClass*.

For example, assume that *CS* is the specification of a simple calculator with signature *sig*. Then *sig* defines the sort $\mathbb{N}$, constants like `1,2,...` and operators like `+` and `*`. Any closed application built from the operators of the problem domain, i.e., every element of $ClosedAppls(sig)$, represents a question from the problem domain and can be used as an input to the implementation. For example, $ClosedAppls(sig)$ contains `+(3,5)` and `*(4,3)`.

For the input `+(3,5)` the client will expect the output `8`, and for `*(4,3)` the expected output is `12`. More generally, the client expects the implementation to transform an input *in* into an output *out* that is a *canonical form* of *in*. That is,

- like *in*, *out* should be expressed using the operators of the client's problem domain, i.e., *out* should be an element of $ClosedAppls(sig)$.

- *out* should be canonical, i.e., it should be of a certain basic form. A possible notion of canonicity for our calculator example is that a closed application is canonical iff it is a constant. E.g., `8` is canonical, but `+(3,5)` is not.
- *in* and *out* should be equal. Every model $\mathcal{A}$ of the axioms of the specification provide a notion of equality that is acceptable to the client (as explained above). Which of these notions of equivalence is used by the implementation is up to the implementor. So, in our example, the specifier should ensure that the axioms for `+` are such that for every model $\mathcal{A}$ of the axioms, $\mathit{Sem}(+(\mathtt{3},\mathtt{5}) = \mathtt{8}, \mathit{emptyva}, \mathcal{A}) = \mathcal{T}$ (and that `+(3,5)` is not equal to any other constant).

We formalize the above in more detail in Sections 2.2.1 and 2.2.2.

2.2.1 Syntax of Specifications

In this section, we formalize the notions of algebraic specification, canonicity function and client specification.

Consider Def. 2.2.1. An algebraic specification consists of a signature and a set of sentences, called axioms. These axioms only contains operators from the signature.

Definition 2.2.1. An *algebraic specification AS* is a record $\mathit{sig} \in \mathit{Sig} \times \mathit{ax} \in \mathit{Set}(\mathit{Sentence})$ such that
$AS.\mathit{ax} \subseteq \mathit{Terms}(AS.\mathit{sig})$
AlgSpec is the set of all algebraic specifications.

Consider Def. 2.2.2. Recall that every model of the axioms of the specification, induces a notion of equality that is acceptable to the client. The intention is that given such a model, the canonicity function of a specification determines whether a given closed application is a representative of an equivalence class and thus suitable as output of the implementation. In other words, the canonicity function determines whether a given closed application is 'most basic'. For this to be the case, there should be no equivalent 'more basic' closed application. Which closed applications are equal is determined by the supplied algebra (two closed applications ca_0 and ca_1 are equal in an algebra $\mathcal{A}$ iff $\mathit{Sem}(\mathit{ca}_0 = \mathit{ca}_1, \mathit{emptyva}, \mathcal{A}) = \mathcal{T}$).

Definition 2.2.2. A *canonicity function* is a function $\mathit{canonFunc} : \mathit{ClosedAppl} \times \mathit{Alg} \to \mathit{Bool}$.
CanonFunc is the set of all canonicity functions.

Consider Defs. 2.2.3 and 2.2.1. Roughly, a client specification consists of an algebraic specification and a canonicity function such that for every model of the axioms, every meaningful closed application has a canonical form. The latter is ensured by $\mathit{IsEachEqClassRepresented}(CS.\mathit{isCanonical}, CS.\mathit{as}) = \mathcal{T}$. More generally, $\mathit{IsEachEqClassRepresented}(\mathit{canonFunc}, AS) = \mathcal{T}$ iff for every model $\mathcal{A}$ of

$AS.sig$, for every closed application ca_0 that 1) is defined by $AS.sig$, and 2) can be evaluated to a value using $\mathcal{A}$, there is canonical representation ca_1 of ca_0. In other words, iff for any given notion of equivalence that follows from $AS.ax$, *canonFunc* is such that every equivalence class has at least one representative.

Note that these definitions reflect the intended separation of concerns. The concern of the axioms is to capture the desired properties of the operators and thus define the acceptable notions of equality. Given any such acceptable notion of equality, the concern of the canonicity function is to determine the representatives of the equivalence class of a given closed application.

Definition 2.2.3. $\mathit{IsEachEqClassRepresented} : \mathit{CanonFunc} \times \mathit{AlgSpec} \to \mathit{Bool}$
$\mathit{IsEachEqClassRepresented}(\mathit{canonFunc}, AS) = \mathcal{T}$ iff
for every $ca_0 \in \mathit{ClosedAppls}(AS.sig)$, $\mathcal{A} \in \mathit{Models}(AS.ax)$, $\nu \in \mathbb{V}$,
if $\mathit{Sem}(ca_0, \mathit{emptyva}, \mathcal{A}) = \nu$,
then there is a $ca_1 \in \mathit{ClosedAppls}(AS.sig)$ such that
$\mathit{canonFunc}(ca_1, \mathcal{A}) = \mathcal{T}$, and
$\mathit{Sem}(ca_1, \mathit{emptyva}, \mathcal{A}) = \nu$.

Definition 2.2.4. A *client specification CS* is a record $as \in \mathit{AlgSpec} \times \mathit{isCanonical} \in \mathit{CanonFunc}$ such that
$\mathit{IsEachEqClassRepresented}(CS.\mathit{isCanonical}, CS.ax) = \mathcal{T}$.
ClientSpec is the set of all client specifications.

Before we present the semantics of algebraic specifications in Sect. 2.2.2, we show three examples of the syntax (Exmpls. 2.1 to 2.3).

Example 2.1. (*Rationals, specification*)

signature and axioms

We distinguish between partial and total operators using $\hookrightarrow$ and $\to$, and use some other self-explanatory shorthand as well. For technical reasons, we write $new_{Rat}(n, d)$ instead of the more usual n/d. Also note that among the omitted operators and axioms are those that establish that every integer is a rational, thus excluding trivial models of the specification.

sorts $\mathit{Rat}, \mathit{Int}$

operators

$new_{Rat} : \mathit{Int} \times \mathit{Int} \hookrightarrow \mathit{Rat}$
$\mathit{add} : \mathit{Rat} \times \mathit{Rat} \to \mathit{Rat}$
$\mathit{equals} : \mathit{Rat} \times \mathit{Rat} \to \mathit{Bool}$

axioms

$$\begin{aligned} &\forall n_0, n_1, d_0, d_1 \in \mathit{Int} \bullet \\ &\quad \mathit{add}(new_{Rat}(n_0, d_0), new_{Rat}(n_1, d_0)) = new_{Rat}(n_0 + n_1, d_0) \\ \wedge &\quad \mathit{equals}(new_{Rat}(n_0, d_0), new_{Rat}(n_1, d_1)) = \mathit{true} \Leftrightarrow n_0 * d_1 = n_1 * d_0 \end{aligned}$$

operators and axioms for *Int*, the sort that models unbounded integers, are omitted.

canonicity

A *Rat* r is canonical iff there are $n, d \in \mathit{Int}$ such that 1) r is $\mathit{new}_{\mathit{Rat}}(n, d)$, and 2) n and d are canonical, and 3) the greatest common divider of n and d is 1. A formal definition is straightforward and is therefore omitted.

Example 2.2. (*Multiple notions of equality*)

This example shows a common pattern for the canonicity function. It also shows how the canonicity function can determine representatives in the case where several notions of equality are induced by the axioms.

Below is the definition of the signature and axioms of the algebraic specification (using some self-explanatory syntactic sugar).

sorts X

operators

$\mathit{zero} : \to X$

$\mathit{succ} : X \to X$

$\mathit{add} : X \times X \to X$

axioms

$$\begin{array}{ll} \forall x, x_0, x_1 \in X \bullet & \\ & \mathit{add}(x, \mathit{zero}()) = x \\ \wedge & \mathit{add}(x_0, \mathit{succ}(x_1)) = \mathit{succ}(\mathit{add}(x_0, x_1)) \\ \wedge & \mathit{succ}(\mathit{succ}(x)) \neq \mathit{succ}(x) \end{array}$$

Now consider the following two algebras $\mathcal{A}_0$ and $\mathcal{A}_1$:

1. in $\mathcal{A}_0$, the carrier for X is $\mathbb{N}$, $\mathit{zero}()$ is interpreted as $0 \in \mathbb{N}$, *succ* as the successor function and *add* as the $+$ (i.e., the addition function on natural numbers).
2. in $\mathcal{A}_1$, the carrier for X is *Bool*, $\mathit{zero}()$ is interpreted as $\mathcal{F}$, *succ* as the logical not and *add* as the exclusive or.

Note that both $\mathcal{A}_0$ and $\mathcal{A}_1$ are a model of the axioms.

Next, we define the canonicity function *isCanonical* of the specification. Roughly, a closed application *ca* is canonical iff only *zero* and *succ* occur in *ca* (i.e., *ca* is one of $\mathit{zero}(), \mathit{succ}(\mathit{zero}()), \mathit{succ}(\mathit{succ}(\mathit{zero}())), \ldots$), and there is no equivalent closed application with less occurrences of *succ*. This follows a common pattern for the canonicity function, where a closed application *ca* is canonical iff every operator that occurs in *ca* comes from a set of *generators*, and there is no equivalent closed application that consists of fewer applications of these generators.

The definition of the canonicity function uses a helper function *sCount* that returns the number of occurrences of operator *succ* in a closed application in which only *succ* and *zero* occur. It is defined as follows:

$sCount : ClosedAppl \hookrightarrow \mathbb{N}$
$sCount(ca_0) = n$ iff
  or ca_0 is $zero()$ and $n = 0$,
  or there is a $ca_1 \in ClosedAppl$ such that ca_0 is $succ(ca_1)$ and $n = 1 + sCount(ca_1)$.

$isCanonical(ca_0, \mathcal{A}) = \mathcal{T}$ iff
  only *zero* and *succ* occur in ca_0, and
  for every $ca_1 \in ClosedAppl$,
    if only *zero* and *succ* occur in ca_1, and
      $Sem(ca_0 = ca_1, emptyva, \mathcal{A}) = \mathcal{T}$,
    then $sCount(ca_0) \leq sCount(ca_1)$.

Assume that $ca = succ(succ(zero()))$. Note that $isCanonical(ca, \mathcal{A}_0) = \mathcal{T}$, but that $isCanonical(ca, \mathcal{A}_1) = \mathcal{F}$ as $Sem(ca = zero(), emptyva, \mathcal{A}_1) = \mathcal{T}$.

Example 2.3. (*the Stack of Int example, specification*) Here we present the classic example of an algebraic specification, that of a Stack, in the setting of our specification technique.

Note that the last 2 axioms essentially define equality on stacks.

sorts *Stack*, *Int*
operators
  $new_{Stack} : \rightarrow Stack$
  $push : Stack \times Int \rightarrow Stack$
  $pop : Stack \hookrightarrow Stack$
  $top : Stack \hookrightarrow Int$
axioms
  $\forall s, s_0, s_1 \in Stack, i, i_0, i_1 \in Int \bullet$
    $pop(push(s, i)) = s$
  $\wedge$ $top(push(s, i)) = i$
  $\wedge$ $new_{Stack}() \neq push(s, i)$
  $\wedge$ $push(s1, i1) = push(s2, i2) \Leftrightarrow s1 = s2 \wedge i1 = i2$

operators and axioms for Int, the sort that models unbounded integers, are omitted.

The canonicity function follows the common pattern shown in Exmpl. 2.2. A closed application *ca* of sort `Stack` is canonical iff *ca* only consists of the generators new_{Stack}, *pop* and integer constants, and there is no equivalent closed application that consists of fewer applications of these generators. We omit the formal definition.

Aside. We omit a division of its operators into two sets: *interface operators* and *auxiliary operators*. The intuition is that interface operators are the verbs of the problem description. Auxiliary operators only serve to axiomatize the interface operators. For example, the example from Hoare's classic paper on data abstraction revolves around sets which are axiomatized using, among others, a *size* operator. But the paper states that only the *insert*, *remove* and *has* operators occur in the abstract program. That is, only these operators are interface operators. The other operators, like *size*, are auxiliary (only used to axiomatize the *SmallIntSet*).

Aside. By interpreting the axioms of an algebraic specification as left-to-write rewrite rules, it is possible to specify the canonicity function indirectly (where a closed application is canonical is none of the rules applies to it). For example, Maude [CDE$^+$02, BJM97] is a program that, given an signature and a set of rewrite rules, allows the user to input a closed application and outputs a closed application (assuming that the rewrite rules are Church-Rosser, terminating and sort-decreasing).

2.2.2 Semantics of Specifications

In this section, we present an intuitive semantics of specifications that is independent of the choice of a programming language.

A client specification intends to capture a set of implementations that are acceptable to the client. A core assumption of the client specification technique is that the client only cares about the input/output behavior of the implementation. Roughly, an implementation is acceptable to the client if, given a meaningful input, the implementation outputs a canonical equivalent. For example, the implementation may be an executable that takes one command line parameter, which is a closed application (typed in by the user). Execution returns a closed application (the answer), and displays it as a string on the screen. Note that the client can use the output as the basis for another input.

Consider Def. 2.2.5 and Fig. 2.1. An answer function is the obvious semantics that matches this black box view of the implementation.

Definition 2.2.5. An *answer function* is a function $answer : ClosedAppl \hookrightarrow ClosedAppl$.
Answer is the set of all answer functions.

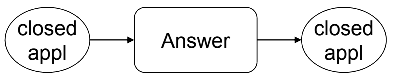


Figure 2.1: Answer function: black box model of implementation

Consider Def. 2.2.6. Note that not given an algebraic specification $CS.as$, not every answer function provides the 'right answer' for any given input ca_0. The intuition is that the semantics of CS determines the set of answer functions that do provide the 'right answer' for any give input. Roughly, the semantics of a specification is the set of those answer functions for which there exists a model of the axioms such that for every input ca_0 that evaluates to an value, there is a canonical output ca_1 that evaluates to the same value. Note that it suffices that there is *a* model, as every model induces a notion of equivalence that is acceptable to the client.

Definition 2.2.6. $\mathit{SemAS} : \mathit{ClientSpec} \rightarrow \mathit{Set}(\mathit{Answer})$

```
$\mathit{answer} \in \mathit{SemAS}(CS)$ iff
  there is an $\mathcal{A} \in \mathit{Models}(CS.as.ax)$ such that
    for every $ca_0 \in \mathit{ClosedAppls}(CS.as.sig)$, $\nu \in \mathbb{V}$
     if      $\mathit{Sem}(ca_0, \mathit{emptyva}, \mathcal{A}) = \nu$,
     then    there is an $ca_1 \in \mathit{ClosedAppls}(CS.as.sig)$ such that
               $\mathit{answer}(ca_0) = ca_1$, and
               $\mathit{Sem}(ca_1, \mathit{emptyva}, \mathcal{A}) = \nu$, and
               $CS.\mathit{isCanonical}(ca_1, \mathcal{A}) = \mathcal{T}$.
```

This semantics is intuitive, as it directly describes the set of (semantics of) black box implementations that are acceptable to the client. Another advantage is that it is independent of the choice of a programming language. Abstracting the semantics of program in a concrete programming language to an answer function can be treated as a separate concern. In sections Sections 3.1 and 3.2, we present a class-based programming language and a notion of satisfaction that connects class-based implementations to algebraic specifications.

3 Implementer's Perspective: Satisfaction for Computation

In this section, we connect class-based implementations to client specifications through a notion of satisfaction. The structure of this section is as follows.

(1) In Sect. 3.1, we formalize a simple class-based programming language.
(2) In Sect. 3.2, we connect programs in this language to client specifications through the introduction of a presentation layer, which allows to define a straightforward notion of satisfaction.
(3) In Sect. 3.3, we sketch an implementation of the presentation layer.
(4) In Sect. 3.4, we use the implementation of the presentation layer to refine the notion of satisfaction for implementations to a notion of satisfaction for class-based programs.

3.1 A Class-Based Language

In this section, we formalize an imperative, class-based language. In the next section, we make a connection between algebraic specifications and programs in this class-based language.

The problem of how to implement an algebraic specification is usually studied in the context of a functional programming language (see e.g. [EKP79, Moi82, Wan82]). Little attention has been paid to implementing algebraic specifications in imperative programming languages [Lin93]. A main advantage of using an imperative programming language is that it is well suited to implementing (some of the) operations with more efficient 'in-place' algorithms. For example, when implementing operation *push* from Exmpl. 2.3, one would like to extend the existing data structure with the new integer, rather than duplicate the entire structure and add the integer to the copy. Studying the implementation of algebraic specifications in an imperative programming language is not just an interesting exercise by itself, it is also useful because algebraic specifications underly many of the specification techniques for imperative programs, including those for object-oriented (OO) programs.

We choose a class-based implementation language because it presents several of the key challenges in OO specification and verification, while at the same time avoiding the complexity of a full-fledged OO language. Furthermore, this choice allows us to stay close to Hoare's work on data abstraction. For simplicity, we omit subclassing and subsorting. Extending the language to accommodate these is possible but this extension can be treated as a separate concern (see e.g. [Lei95]).

Roughly, we model an imperative program as follows: it takes a sequence of statements, and computes an evaluation context (which can be thought of as the state of the computation's memory at a given point in time) in the context of a set of class definitions. This is visualized in Fig. 3.1.

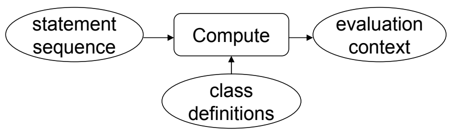


Figure 3.1: computation

The formal syntax of the class-based programming language is presented in Sect. 3.1.1. Its semantics is presented in Sect. 3.1.2.

3.1.1 Syntax

In this section, we define the syntax of our class-based programming language. To this end, we first define the syntax of statements, then that of method and class definitions. We also define a notion of an extended method body in which the return statement, which is implicit in our method definitions, is made explicit.

To define the syntax of statements, we first define the primitive sorts[1] and operators of the programming language. Then, we define the set of methods. Next, we define the different kinds of variables that can occur in a statement. We then use these definitions to define the statement grammar. Finally, we define when a statement is well-typed.

> **Aside.** Note that a practically useful language would impose more syntactical restrictions than presented in this section. However, for the purpose of this chapter, additional restrictions are not necessary and are therefore omitted.

Sorts and operators. We keep the syntax of statements in line with that of applications, e.g., we write $v_0 := o(v_1, v_2)$ instead of $v_0 = v_1.o(v_2)$. It is straightforward to add a layer of syntactic sugar to bring the syntax in line with that of well-known OO languages like Java and C#.

Consider Defs. 3.1.1 and 3.1.2. The primitive sorts and operators of the programming language are defined by the primitive signature (Def. 3.1.2). For uniformity, constants of primitive sorts are considered total parameterless operations, e.g., $1 :\rightarrow \mathbb{N} \in \mathit{PrimSig.ops}$. For simplicity, apart from constants we only consider primitive operators that have exactly two parameters, i.e., that are elements of $\mathit{Op}2$. The generalization to an arbitrary number of parameters is straightforward.

Definition 3.1.1. $\mathit{Op}2 \subset \mathit{Op}$ is the set $\{f : S_1 \times S_2 \hookrightarrow S_0 \in \mathit{Op} \mid \mathcal{T}\}$.

Definition 3.1.2. $\mathit{PrimSig} \in \mathit{Sig}$ (the *primitive signature*) is a signature such that
$\mathit{PrimSig.ops} \subset (\mathit{Op}2 \cup \mathit{Constant})$, and
$\mathit{PrimSig.ops} \cap \mathit{Constant} \subseteq \mathit{PrimSig.tOps}$.

Consider Def. 3.1.3. The intuition is that *ClassSort* is the set of sorts that can be used as a class name. This set does not include any primitive sorts.

Definition 3.1.3. *ClassSort* is a set such that *ClassSort* and $\mathit{PrimSig.sorts}$ are disjoint.

Consider Defs. 3.1.4 to 3.1.6, in which we define the set of methods. For simplicity, we only consider methods that have exactly two parameters. Again, the generalization to an arbitrary number of parameters is straightforward. There are two types of methods: constructors and non-constructor methods.

[1]At the level of programming languages, sorts are often referred to as types

Consider Def. 3.1.4. The intuition is that $\texttt{new}_C$ identifies the constructor of class C and must return an object of sort C.

Consider Def. 3.1.5. The intuition is that the first parameter of a non-constructor method is the receiver, i.e., the object on which the method is called.

Consider Def. 3.1.6. Note that *Method* and *PrimSig.ops* are disjoint, as methods have a *ClassSort* in either their domain sorts or their range sort, and primitive operators do not.

Definition 3.1.4. For every $C \in \mathit{ClassSort}, S_1, S_2 \in \mathit{Sort}$, there is a a special-purpose operator $\texttt{new}_C : S_1 \times S_2 \hookrightarrow C$ (a *constructor*). $\mathit{Constructor} \subset \mathit{Op}2$ is the set of all constructors.

Definition 3.1.5. $\mathit{NonConstrMethod} \subset \mathit{Op}2$ (the *non-constructor methods*) is a set such that for every $m \in \mathit{NonConstrMethod}$, $\mathit{DomainSorts}(m)[0] \in \mathit{ClassSort}$.

Definition 3.1.6. $\mathit{Method} \subset \mathit{Op}2$ (the *methods*) is the set $\mathit{Constructor} \cap \mathit{NonConstrMethod}$.

Consider Def. 3.1.7. $\mathit{OpSort}(o)$ determines a sort based on the signature of operator o. The intuition is if o is a method, then this is the class to which o belongs. For constructors, this is the return sort, and for other methods this is the sort of the first parameter (the receiver).

Definition 3.1.7. $\mathit{OpSort} : \mathit{Op} \to \mathit{Sort}$
$\mathit{OpSort}(o) = S$ iff
or $o \in \mathit{Constructor} \cup \mathit{Constant}$, and $S = \mathit{RangeSort}(o)$,
or $o \notin \mathit{Constructor} \cup \mathit{Constant}$, and $S = \mathit{DomainSorts}(o)[0]$.

Variables. Consider Defs. 3.1.8 to 3.1.14, which define the kinds of variables that can occur in a statement. Such a variable is either a field, or a stack variable. A stack variable is either a local variable, a formal parameter, or the special-purpose variable `this`. For simplicity, we assume that these sets are disjoint, i.e., that the kind of a variable in a statement can be determined syntactically. The following intuitions are formalized in Sect. 3.1.2. A non-constructor method has two formal parameters called `this` and `p`. Constructors have an implicitly defined variable called `this`, and two formal parameters called `q` and `p`. Constructors and void methods return (the value stored by) `this`. Other methods return (the value stored by) the special-purpose local variable `result`. Special-purpose `dummy` variables are used to simplify the treatment of statements that assign the result of a call to a field. Special-purpose $\texttt{lv}i$ variables are used to store intermediate results during translation (see Sect. 3.3.1).

Definition 3.1.8. $\mathit{ThisVar} \subset \mathit{Var}$ is the set $\{\texttt{this}_C \mid C \in \mathit{ClassSort}\}$.

Definition 3.1.9. $\mathit{FPar} \subset \mathit{Var}$ (the set of *formal parameters*) is the set $\{\texttt{p}_S \mid S \in \mathit{Sort}\} \cup \{\texttt{q}_S \mid S \in \mathit{Sort}\}$.

Definition 3.1.10. There is a set $LocVar \subset Var$ (the set of *local variables*) that is disjoint from $ThisVar$ and $FPar$. For every $i \in N, S \in Sort$, there are special-purpose variables $\mathtt{dummy}_S, \mathtt{lv}i_S \in LocVar$.

Definition 3.1.11. $ResultVar \subset LocVar$ (the set of special-purpose *result variables*) is the set $\{\mathtt{result}_S \mid S \in Sort\}$.

Definition 3.1.12. $ReturnVar$ (the set of *return variables*) is the set $ThisVar \cup ResultVar$.

Definition 3.1.13. $StackVar$ is the set $FPar \cup LocVar \cup ThisVar$.

Definition 3.1.14. There is a set $Field \subset Var$ that is disjoint from $StackVar$.

Statements. Consider Fig. 3.2, which defines the syntax of statements. Note that return statements are left implicit. For uniformity, we let a void method return the receiver (i.e., the `this` variable). This avoids the need for a separate void method call statement $m(r, r)$.

Note that reference f is essentially shorthand for $\mathtt{this}.f$ (see Sect. 3.1.2 on semantics). Having it as an explicit language construct makes the work in following sections easier at the cost of some elegance here.

$$t \in StackVar, f \in Field, po \in PrimSig.ops, m \in Method, \overline{s} \in Seq(Statement)$$

$$\begin{array}{lcl} r \in Ref & ::= & t \mid t.f \mid f \\ e \in Expr & ::= & r \mid po(r,r) \mid po() \\ rhs \in RHS & ::= & e \mid m(r,r) \\ s \in Statement & ::= & \mathtt{if}\ (r)\ \overline{s} \mid \mathtt{while}\ (r)\ \overline{s} \mid r := rhs \end{array}$$

Figure 3.2: Statement grammar.

Consider Def. 3.1.15. We only want to consider statements that are well-typed. To formalize this notion, $RefSort$ associates a sort with a reference in the obvious way.

Definition 3.1.15. $RefSort : Ref \to Sort$
$RefSort(r) = S$ iff there are $t, t_S \in StackVar, f_S \in Field$ such that
or r *is* t_S,
or r *is* $t.f_S$,
or r *is* f_S.

Consider Def. 3.1.16. Roughly, the restriction of statements to the set $Stmt$ ensures a property that is often referred to as type-safety (where the sort of the value a variable evaluates to, matches the sort of the variable). Furthermore, the restriction ensures that if- and while-guards are of sort $Bool$.

Definition 3.1.16. $Stmt \subset Statement$ (the set of *well-formed statements*) is inductively defined as follows.

$s \in \mathit{Stmt}$ iff
there are $r_0, r_1, r_2 \in \mathit{Ref}$, $\overline{s} \in \mathit{Seq}(\mathit{Stmt})$, $po : \hookrightarrow S \in \mathit{PrimSig.ops}$, $o : S_1 \times S_2 \hookrightarrow S_0 \in (\mathit{Method} \cup \mathit{PrimSig.ops})$ such that
or s is $r_0 := r_1$ and $\mathit{RefSort}(r_0) = \mathit{RefSort}(r_1)$,
or s is $r_0 := po()$ and $\mathit{RefSort}(r_0) = S$,
or s is $r_0 := o(r_1, r_2)$ and $\mathit{RefSort}(r_0) = S_0$ and $\mathit{RefSort}(r_1) = S_1$ and $\mathit{RefSort}(r_2) = S_2$.
or s is `if` (r_0) $\overline{s}$ and $\mathit{RefSort}(r_0) = \mathit{Bool}$,
or s is `while` (r_0) $\overline{s}$ and $\mathit{RefSort}(r_0) = \mathit{Bool}$.

Consider Def. 3.1.17. Informally, we say $s_0 \in \mathit{Stmt}$ is a *substatement* of $\overline{s}_1 \in \mathit{Seq}(\mathit{Stmt})$ iff (1) $s_0 \in \overline{s}_1$, or (2) $\overline{s}_1$ has an element `while` (r) $\overline{s}_2$, and s_0 is a substatement of $\overline{s}_2$, or (3) $\overline{s}_1$ has an element `if` (r) $\overline{s}_3$, and s_0 is a substatement of $\overline{s}_3$.

Definition 3.1.17. $\mathit{SubStmts}() : \mathit{Seq}(\mathit{Stmt}) \to \mathit{Set}(\mathit{Stmt})$
Given the informal definition above, the formal definition is straightforward and is omitted.

Method and class definitions. In this paragraph, we formalize method and class definitions.

Consider Def. 3.1.18. Roughly, a method definition consists of a method signature (which defines the method's name, the sorts of its parameters, and its return sort), a method body, and an indication whether the method is a void method. Note that constructors are not void methods, and that void methods return an object of the sort of their receiver (as $\mathit{OpSort}(\mathit{methDef.sig})$ returns the sort of the receiver for non-constructor methods, see Def. 3.1.7). For simplicity, we do not require to define the names of the formal parameter(s) in a method definition. Instead, we use fixed names `this` and `p` (for non-constructors) or `q` and `p` (for constructors).

Definition 3.1.18. A *method definition* is a record $\mathit{methDef} : \mathit{sig} \in \mathit{Method} \times \mathit{body} \in \mathit{Seq}(\mathit{Stmt}) \times \mathit{isVoid} \in \mathit{Bool}$ such that
if $\mathit{methDef.sig} \in \mathit{Constructor}$, then $\mathit{methDef.isVoid} = \mathcal{F}$, and
if $\mathit{methDef.isVoid} = \mathcal{T}$, then $\mathit{RangeSort}(\mathit{methDef.sig}) = \mathit{OpSort}(\mathit{methDef.sig})$.
$\mathit{MethDef}$ is the set of all method definitions.

Consider Def. 3.1.19. Roughly, it defines the following restrictions on the definition of a class C. The class definition 1) defines at least one field (for convenience), 2) does not define two methods with the same signature, 3) only defines methods that belong to the class (i.e., that have a receiver of the sort that is the name of the class, or that are a constructor of the class), and 4) if a method body of a method defined in C uses the shorthand field assignment statement $f := \mathit{rhs}$, then f is a field of C. Other natural restrictions are omitted as they are not relevant to this chapter. Note that for simplicity, there is no notion of subclassing in the language.

Definition 3.1.19. A *class definition* is a record $classDef : name \in ClassSort \times mds \in Seq(MethDef) \times fields \in Seq(Field)$ such that
$|\, classDef.fields\,| > 0$, and
for every $methDef_0, methDef_1 \in classDef.mds$,
or $methDef_0 = methDef_1$,
or $methDef_0.sig \neq methDef_1.sig$, and
for every $methDef \in classDef.mds$,
$OpSort(methDef.sig) = classDef.name$, and
for every $f \in Field$,
if $f := rhs \in SubStmts(methDef.body)$,
then $f \in classDef.fields$.
$ClassDef$ is the set of all class definitions.

Consider Def. 3.1.20. Roughly, a set of class definitions is well-formed if it does not contain two classes with the same name.

Definition 3.1.20. A set $cds \in Set(ClassDef)$ is *well-formed* iff
for every $classDef_0, classDef_1 \in cds$,
or $classDef_0 = classDef_1$,
or $classDef_0.name \neq classDef_1.name$.
$ClassDefSet$ is the set $\{cds \in Set(ClassDef) \mid cds$ is well-formed $\}$.

Consider Def. 3.1.21. Given a method signature and a set of class definitions, *GetMethDef* returns the definition of that method (if there is one). Note that a method signature $m \in Method$ uniquely defines a method definition *methDef* in a well-formed set of class definitions *cds*, as the range sort C of *sig* (i.e., the return type of the method) is the name of the class definition *classDef* that defines it (as *cds* does not define two classes with the same name), and as *classDef* does not define two methods with the same signature.

Definition 3.1.21. $GetMethDef : Method \times ClassDefSet \hookrightarrow MethDef$
$GetMethDef(m, cds) = methDef$ *iff*
there is a $classDef \in cds$ *such that*
$methDef \in classDef.mds$, *and*
$methDef.sig = m$.

Consider Defs. 3.1.22 and 3.1.23. Given a method signature m and a set of class definitions *cds*, the convenience functions $GetBody(m, cds)$ and $IsVoid(m, cds)$ use $GetMethDef(m, cds)$ to return the body of m, and whether or not m is a void method.

Definition 3.1.22. $GetBody : Method \times ClassDefSet \hookrightarrow Seq(Stmt)$
$GetBody(m, cds) = \overline{s}$ iff
there is a $methDef \in MethDef$ such that
$GetMethDef(m, cds) = methDef$, and
$methDef.body = \overline{s}$.

Definition 3.1.23. $\mathit{IsVoid} : \mathit{Method} \times \mathit{ClassDefSet} \hookrightarrow \mathit{Bool}$
$\mathit{IsVoid}(m, \mathit{cds}) = b$ iff
there is a $\mathit{methDef} \in \mathit{MethDef}$ such that
$\mathit{GetMethDef}(m, \mathit{cds}) = \mathit{methDef}$, and
$\mathit{methDef}.\mathit{isVoid} = b$.

Extended method bodies. In an extended method body, the implicit return statement of a method body is made explicit.

Consider Def. 3.1.24. An extended statement is a well-formed statement concatenated with a return statement.

Definition 3.1.24. $\mathit{ExtStmt}$ (the set of *extended statements*) is the set $\mathit{Stmt} \cup \{\texttt{return}\ v \mid v \in \mathit{ReturnVar}\}$.

Consider Defs. 3.1.25 and 3.1.26. Roughly, $\mathit{GetExtBody}(m, \mathit{cds})$ returns the method body of m as defined in cds, concatenated with the appropriate return statement. As defined by $\mathit{GetReturnVar}(m, \mathit{cds})$, this return statement returns `this` when m is a constructor or void method, and returns `result` otherwise.

Definition 3.1.25. $\mathit{GetReturnVar} : \mathit{Method} \times \mathit{ClassDefSet} \hookrightarrow \mathit{ReturnVar}$
$\mathit{GetReturnVar}(m : S_1 \times S_2 \hookrightarrow S_0, \mathit{cds}) = \mathit{rv}$ iff
if $m \in \mathit{Constructor}$ or $\mathit{IsVoid}(m, \mathit{cds}) = \mathcal{T}$,
then rv is $\texttt{this}_{S_0}$,
else rv is $\texttt{result}_{S_0}$.

Definition 3.1.26. $\mathit{GetExtBody} : \mathit{Method} \times \mathit{ClassDefSet} \hookrightarrow \mathit{Seq}(\mathit{ExtStmt})$
$\mathit{GetExtBody}(m, \mathit{cds}) = \overline{s}_0$ iff
there are $\overline{s}_1 \in \mathit{Seq}(\mathit{Stmt})$, $\mathit{rv} \in \mathit{ReturnVar}$ such that
$\mathit{GetBody}(m, \mathit{cds}) = \overline{s}_1$, and
$\mathit{GetReturnVar}(m, \mathit{cds}) = \mathit{rv}$, and
$\overline{s}_0 = \overline{s_1} \triangleright \langle \texttt{return}\ \mathit{rv} \rangle$.

Other definitions. Here we define several straightforward functions that are not needed to define the semantics, but that are useful later on.

Consider Def. 3.1.27. Recall that a non-constructor method has two formal parameters called `this` and `p`, and that a constructor has two formal parameters called `q` and `p`. *GetFormalParams* returns these formal parameters.

Definition 3.1.27. $\mathit{GetFormalParams} : \mathit{Method} \times \mathit{ClassDefSet} \hookrightarrow \mathit{Seq}(\mathit{StackVar})$
$\mathit{GetFormalParams}(m : S_1 \times S_2 \hookrightarrow S_0, \mathit{cds}) = \overline{sv}$ *iff*
$|\overline{sv}| = 2$, *and*
$\overline{sv}[1] = \texttt{p}_{S_2}$, *and*
or $m \notin \mathit{Constructor}$ *and* sv *is* $\texttt{this}_{S_1}$,
or $m \in \mathit{Constructor}$ *and* sv *is* $\texttt{q}_{S_1}$.

Consider Def. 3.1.28. Roughly, given a set of class definitions cds, $\mathit{GetFields}(C, \mathit{cds})$ returns the fields defined in class C of cds.

Definition 3.1.28. $\mathit{GetFields} : \mathit{ClassSort} \times \mathit{ClassDefSet} \hookrightarrow \mathit{Seq}(\mathit{Field})$
$\mathit{GetFields}(C, \mathit{cds}) = \overline{f}$ iff
there is a $\mathit{classDef} \in \mathit{cds}$ such that
$\mathit{classDef}.\mathit{name} = C$, and
$\mathit{classDef}.\mathit{fields} = \overline{f}$.

Consider Def. 3.1.29. Roughly, given a set of class definitions cds, if class C of cds defines fields $\langle f_1, \ldots, f_n\rangle$, then $\mathit{GetFieldSorts}(C, \mathit{cds})$ returns the sequence of the sorts of $\langle f_1, \ldots, f_n\rangle$.

Definition 3.1.29. $\mathit{GetFieldSorts} : \mathit{ClassSort} \times \mathit{ClassDefSet} \hookrightarrow \mathit{Seq}(\mathit{Sort})$
$\mathit{GetFieldSorts}(C, \mathit{cds}) = \langle S_1, \ldots, S_n\rangle$ iff
there is a $\langle f_1, \ldots, f_n\rangle \in \mathit{Seq}(\mathit{Field})$ such that
$\mathit{GetFields}(C, \mathit{cds}) = \langle f_1, \ldots, f_n\rangle$, and
for every $i \in [1, n]$,
$\mathit{VarSort}(f_i) = S_i$.

Consider Def. 3.1.30. Roughly, $\mathit{GetClassNames}(\mathit{cds})$ returns the set of names of the class definitions in cds.

Definition 3.1.30. $\mathit{GetClassNames} : \mathit{ClassDefSet} \to \mathit{Set}(\mathit{ClassSort})$
$C \in \mathit{GetClassNames}(\mathit{cds})$ iff
there is a $\mathit{classDef} \in \mathit{cds}$ such that
$\mathit{classDef}.\mathit{name} = C$.

3.1.2 Semantics

In this section we present a semantics for the syntax presented in Sect. 3.1.1. This semantics leads to a fairly standard model of computation in a class-based language formalized in Def. 3.1.55. The novelty is that we use a small-step (or natural) semantics. The motivation is that a small-step semantics is more suitable to the formalization of properties that hold in specific states of the executions, like class invariants. For an example of an OO language defined using natural semantics, see e.g. [Pie06].

The implementation algebra. First, we introduce an algebra that gives a semantics to primitive operations and defines a carrier set for every sort.

Definition 3.1.31. $\mathbb{A} \subset \mathbb{V}$ is the set of *objects* (or *addresses*).

Consider Def. 3.1.32. Roughly, the implementation algebra (1) gives a semantics to every primitive operation, (2) defines a carrier set for every sort, (3) defines a set of addresses as the carrier set for every non-primitive sort, and (4) has disjoint carrier sets (for convenience). We sometimes write $\mathbb{A}_C$ for the set $\mathcal{A}_{impl}.\mathit{carrier}(C)$.

Definition 3.1.32. $\mathcal{A}_{impl} \in Alg$ (the *implementation algebra*) is such that
$Domain(\mathcal{A}_{impl}.interpretation) = PrimSig.ops$, and
$Domain(\mathcal{A}_{impl}.carrier) = Sort$, and
for every $C \in ClassSort$, $\mathcal{A}_{impl}.carrier(C) \subseteq \mathbb{A}$, and
for every $S_0, S_1 \in Sort$,
if S_0 differs from S_1, then $\mathcal{A}_{impl}.carrier(S_0) \cap \mathcal{A}_{impl}.carrier(S_1) = \{\}$.

Language values. Consider Defs. 3.1.33 and 3.1.35. A language value is either a primitive value, an object, or the special purpose value *nil*.

Definition 3.1.33. $\mathbb{P} \subset \mathbb{V}$ (the set of *primitive values* of the programming language) is the set $\{\nu \in \mathbb{V} \mid \text{there is an } S \in PrimSig.sorts \text{ such that } \nu \in \mathcal{A}_{impl}.carrier(S)\}$.

Definition 3.1.34. There is a special-purpose value $nil \in \mathbb{V}$ such that $nil \notin (\mathbb{P} \cup \mathbb{A})$.

Definition 3.1.35. $\mathbb{L} \subset \mathbb{V}$ (the set of *language values*) is defined as $\mathbb{P} \cup \mathbb{A} \cup \{nil\}$.

Consider Def. 3.1.36. Recall that for convenience, we have required that all carrier sets of $\mathcal{A}_{impl}$ are disjoint. This requirement, while not essential, allows us to think of a language value λ as *having* sort S. $SortL(\lambda)$ returns this sort S.

Definition 3.1.36. $SortL : \mathbb{L} \hookrightarrow Sort$
$SortL(\lambda) = S$ *iff* $\lambda \in \mathcal{A}_{impl}.carrier(S)$.

States. Consider Def. 3.1.37. A stack frame is a finite mapping from variables to language values such that 1) at most one `this` variable is mapped to a value, and 2) variables of primitive sorts are mapped to a value from their carrier set, and 3) variables of non-primitive sorts are either mapped to a value from their carrier set, or mapped to *nil*. Note that $StackFrame \subset Valuation$.

Definition 3.1.37. A *stack frame* is a function $sf : Var \hookrightarrow \mathbb{L}$ such that
$Domain(sf)$ is finite, and
$|\,Domain(sf) \cap ThisVar\,| \leq 1$, and
for every $v_S \in Domain(sf)$,
if $S \in PrimSig.sorts$,
then $sf(v_S) \in \mathcal{A}_{impl}.carrier(S)$,
else $sf(v_S) \in (\{nil\} \cup \mathcal{A}_{impl}.carrier(S))$.
StackFrame is the set of all stack frames.

Consider Defs. 3.1.38 and 3.1.39. An object store is a finite mapping from locations to language values, where a location can be thought of as a field of an object. This mapping is such that 1) fields of primitive sorts are mapped to a value from their carrier set, and 2) fields of non-primitive sorts are either mapped to a value from their carrier set, or mapped to *nil*.

Definition 3.1.38. A *location* is a tuple $\mathbb{A} \times \mathit{Field}$.
Loc is the set of all locations.

Definition 3.1.39. An *object store* is a function $os : \mathit{Loc} \hookrightarrow \mathbb{L}$ such that
$\mathit{Domain}(os)$ is finite, and
for every $\langle \alpha, f_S \rangle \in \mathit{Domain}(os)$,
if $S \in \mathit{PrimSig.sorts}$,
then $os(\langle \alpha, f_S \rangle) \in \mathcal{A}_{impl}.\mathit{carrier}(S)$,
else $os(\langle \alpha, f_S \rangle) \in \{\mathit{nil}\} \cup \mathcal{A}_{impl}.\mathit{carrier}(S)$.
ObjStore is the set of all object stores.

Consider Def. 3.1.40. An evaluation context allows to evaluate a language expression to a value (as formalized in Def. 3.1.47).

Definition 3.1.40. An *evaluation context* is a record $sf \in \mathit{StackFrame} \times os \in \mathit{ObjStore}$.
EvalContext is the set of all evaluation contexts.

Definition 3.1.41. *emptysf* and *emptyos* are the stack frame and object store of which the domain is empty. *emptyec* is the evaluation context $\langle \mathit{emptysf}, \mathit{emptyos} \rangle$.

Consider Def. 3.1.42. The intuition (formalized later) is that at a method call, the current stack frame and the local variable to which the result of the method is to be assigned are pushed on the call stack, to be popped when the method call returns.

Definition 3.1.42. A *call stack* is a sequence of tuples $\mathit{StackFrame} \times \mathit{LocVar}$.
CallStack is the set of all call stacks.

Consider Def. 3.1.43, which formalizes the central notion of the semantics, that of an (execution) state.

Definition 3.1.43. A *state* is a record $ec \in \mathit{EvalContext} \times cs \in \mathit{CallStack} \times \mathit{stmtSeq} \in \mathit{Seq}(\mathit{ExtStmt})$.
State is the set of all states.

Expression Evaluation. Next, we define how to evaluate an expression to a value using an evaluation context.

Consider Def. 3.1.44. Recall from Def. 3.1.37 that a stack frame *sf* maps at most one `this` variable to a value. $\mathit{ThisObj}(sf)$ returns that value, if it exists.

Definition 3.1.44. $\mathit{ThisObj} : \mathit{StackFrame} \hookrightarrow \mathbb{A}$
$\mathit{ThisObj}(sf) = \alpha$ iff there is a $C \in \mathit{ClassSort}$ such that $sf(\mathtt{this}_C) = \alpha$.

Consider Defs. 3.1.45 and 3.1.46. If reference r is not a stack variable, then *RefToLoc* maps r to the location that holds its value. *RefToL* evaluates a reference to a language value in the obvious way.

Definition 3.1.45. $\mathit{RefToLoc} : \mathit{Ref} \times \mathit{StackFrame} \hookrightarrow \mathit{Loc}$
$\mathit{RefToLoc}(r, \mathit{sf}) = \langle \alpha, f \rangle$ iff
or there is a $t \in \mathit{StackVar}$ such that r is $t.f$, and $\mathit{sf}(t) = \alpha$,
or r is f, and $\mathit{ThisObj}(\mathit{sf}) = \alpha$.

Definition 3.1.46. $\mathit{RefToL} : \mathit{Ref} \times \mathit{EvalContext} \hookrightarrow \mathbb{L}$
$\mathit{RefToL}(r, \langle \mathit{sf}, \mathit{os} \rangle) = \lambda$ iff
or $r \in \mathit{StackVar}$, and $\mathit{sf}(r) = \lambda$,
or there is an $l \in \mathit{Loc}$ such that $\mathit{RefToLoc}(r, \mathit{sf}) = l$, and $\mathit{os}(l) = \lambda$.

Consider Def. 3.1.47. *Eval* evaluates an expression to a value in the obvious way.

Definition 3.1.47. $\mathit{Eval} : \mathit{Expr} \times \mathit{EvalContext} \hookrightarrow \mathbb{L}$
$\mathit{Eval}(e, \mathit{ec}) = \lambda_0$ iff
or $e \in \mathit{Ref}$ and $\mathit{RefToL}(e, \mathit{ec}) = \lambda_0$,
or $e \in \mathit{PrimSig.ops} \cap \mathit{Constant}$, and $\mathit{Sem}(e, \mathcal{A}_{impl}, \mathit{emptyva}) = \lambda_0$,
or there are $\mathit{po} : S_1 \times S_2 \hookrightarrow S_0 \in \mathit{PrimSig.ops}$, $r_0, r_1 \in \mathit{Ref}, \lambda_1, \lambda_2 \in \mathbb{L}$,
$v_0, v_1 \in \mathit{Var}$ such that
  e is $\mathit{po}(r_0, r_1)$, and
  $\mathit{RefToL}(r_0, \mathit{ec}) = \lambda_1$, and
  $\mathit{RefToL}(r_1, \mathit{ec}) = \lambda_2$, and
  let va be $\mathit{emptyva}[v_0 \mapsto \lambda_1, v_1 \mapsto \lambda_2]$ in $\mathit{Sem}(\mathit{po}(v_0, v_1), \mathcal{A}_{impl}, \mathit{va}) = \lambda_0$

Allocation. Consider Defs. 3.1.48 to 3.1.50. The intuition is that if $\mathit{Alloc}(C, \mathit{os}_0, \mathit{cds}) = \langle \alpha, \mathit{os}_1 \rangle$, then os_1 is the object store like os_0, except that it contains a newly allocated object α of class C (as defined in *cds*) of which all the fields have been initialized. Here, a field of sort S is initialized to $\mathit{InitialValue}(S)$. $\mathit{InitialValue}(S)$ is *nil* if $S \in \mathit{ClassSort}$, otherwise it is an element of the carrier set of S (which element this is, is not relevant here). Note that allocation is always possible (i.e., we assume an infinite amount of memory).

Definition 3.1.48. $\mathit{IsAllocated}() : \mathbb{A} \times \mathit{ObjStore} \to \mathit{Bool}$
$\mathit{IsAllocated}(\alpha, \mathit{os}) = \mathcal{T}$ iff there is an $f \in \mathit{Field}$ such that $\langle \alpha, f \rangle \in \mathit{Domain}(\mathit{os})$.

Definition 3.1.49. There is a function $\mathit{InitialValue} : \mathit{Sort} \to \mathbb{L}$ such that
for every $S \in \mathit{ClassSort}$, $\mathit{InitialValue}(S) = \mathit{nil}$, and
for every $S \notin \mathit{ClassSort}$, $\mathit{InitialValue}(S) \in \mathcal{A}_{impl}.\mathit{carrier}(S)$.

Definition 3.1.50. There is a function $\mathit{Alloc} : \mathit{ClassSort} \times \mathit{ObjStore} \times \mathit{ClassDefSet} \hookrightarrow \mathbb{A} \times \mathit{ObjStore}$ such that
if there is a $\mathit{classDef} \in \mathit{cds}$ such that $\mathit{classDef.name} = C$,
then there are $\alpha \in \mathbb{A}$, $\mathit{os}_1 \in \mathit{ObjStore}$ such that
  $\mathit{Alloc}(C, \mathit{os}_0, \mathit{cds}) = \langle \alpha, \mathit{os}_1 \rangle$, and
  $\mathit{IsAllocated}(\alpha, \mathit{os}_0) = \mathcal{F}$, and
  os_1 is the object store like os_0, except that
   for every $f_S \in \mathit{classDef.fields}$, $\mathit{os}_1(\langle \alpha, f_S \rangle) = \mathit{InitialValue}(S)$.

Computation. Consider Def. 3.1.51, which defines a transition relation on states in the context of a *cds* $\in$ *ClassDefSet*. If *cds* is clear from the context, it can be omitted. One or more applications of $\overset{c}{\rightsquigarrow}_{cds}$ are denoted by $\overset{c}{\rightsquigarrow}^{+}_{cds}$, zero or more by $\overset{c}{\rightsquigarrow}^{*}_{cds}$, and n applications by $\overset{c}{\rightsquigarrow}^{n}_{cds}$. The intuition is that this relation defines the small-step statement semantics of our programming language. The rules for if statements, while statements and sequential composition (`IF, WHILE` and `COMP`) are standard (recall that $\triangleright$ denotes sequence concatenation). In the `READ` rule, the value of the stack variable *sv* on the left-hand side of the assignment is updated with the value to which the expression on the right-hand side evaluates. In the `WRITE` rule, the location to which reference r on the left-hand side of the assignment evaluates, is updated with the value to which the expression on the right-hand side evaluates. The `SH` rule shows how we treat a statement $r_0 := m(r_1, r_2)$ as *s*hort*h*and for the statement sequence $\langle \texttt{dummy}_S := m(r_1, r_2), r_0 := \texttt{dummy}_S\rangle$, which can be dealt with by the `READ` and `WRITE` rules. Next, consider the `CALL` rule, which considers the execution of a method call statement $sv := m(r_0, r_1)$ from an evaluation context sf, os. The current stack frame sf and the stack variable sv to which the outcome of the method call is to be assigned, are pushed on the call stack. Execution continues with an stack frame in which only `this` and `p` are defined (the formal parameters of the method). The actual first parameter r_0 is the receiver, and its value is assigned to `this`. The second is a 'normal' parameter and its value is assigned to `p`. Execution continues with the execution of the extended method body. Consider the `CONSTR` rule, which considers a constructor call. The rule is similar to the `CALL` rule. Recall that like other methods, constructors have two parameters. However, unlike other methods, constructors do not have a receiver. Therefore, a constructor has two formal parameters called `q` and `p`. The value of the first actual parameter is assigned to `q`, the value of the second to `p`. As usual, the newly created object is assigned to the `this` variable of the constructor. Finally, consider the `RET` rule. In this rule, the stack frame sf_1 and stack variable sv that are on top of the call stack are popped. The current stack frame sf_0 is replaced by sf_1, and then the value of result variable rv is assigned to sv.

Definition 3.1.51. *Small-step semantics in the context of cds* $\in$ *ClassDefSet. Brackets around states are omitted to improve readability.* $r, r_0, r_1, r_2 \in Ref, ec \in EvalContext, cs \in CallStack, s_0 \in ExtStmt, \overline{s}_0, \overline{s}_1, \overline{s}_2 \in Seq(ExtStmt), sv \in StackVar, e \in Expr, f \in Field, S, S_0, S_1 \in Sort, C \in ClassSort, \alpha \in \mathbb{A}, \lambda, \lambda_0, \lambda_1 \in \mathbb{L}, rv \in ResultVar, sf, sf_0, sf_1 \in StackFrame, os, os_0, os_1 \in ObjStore, cs \in CallStack, \alpha \in \mathbb{A}.$

$$\frac{Eval(r, ec) = \mathcal{T}}{ec, cs, \langle \texttt{if}\ (r)\ \overline{s}_0\rangle \quad \overset{c}{\rightsquigarrow}_{cds} \quad ec, cs, \overline{s}_0}\ \texttt{IFt}$$

$$\frac{Eval(r, ec) = \mathcal{F}}{ec, cs, \langle \texttt{if}\ (r)\ \overline{s}_0\rangle \quad \overset{c}{\rightsquigarrow}_{cds} \quad ec, cs, \langle\rangle}\ \texttt{IFf}$$

$$\frac{\mathit{Eval}(r, ec) = \mathcal{T}}{ec, cs, \langle \texttt{while}\ (r)\ \overline{s}_0\rangle \quad \overset{c}{\leadsto}_{cds} \quad ec, cs, \overline{s}_0 \triangleright \langle \texttt{while}\ (r)\ \overline{s}_0\rangle}\texttt{WHILEt}$$

$$\frac{\mathit{Eval}(r, ec) = \mathcal{F}}{ec, cs, \langle \texttt{while}\ (r)\ \overline{s}_0\rangle \quad \overset{c}{\leadsto}_{cds} \quad ec, cs, \langle\rangle}\texttt{WHILEf}$$

$$\frac{|\,\overline{s}_1\,| > 0 \qquad ec, cs, \langle s_0\rangle \overset{c}{\leadsto}_{cds} ec', cs', \overline{s}_2}{ec, cs, \langle s_0\rangle \triangleright \overline{s}_1 \quad \overset{c}{\leadsto}_{cds} \quad ec', cs', \overline{s}_2 \triangleright \overline{s}_1}\texttt{COMP}$$

$$\frac{\mathit{Eval}(e, \langle sf, os\rangle) = \lambda}{\langle sf, os\rangle, cs, \langle sv := e\rangle \quad \overset{c}{\leadsto}_{cds} \quad \langle sf[sv \mapsto \lambda], os\rangle, cs, \langle\rangle}\texttt{READ}$$

$$\frac{\mathit{RefToLoc}(r, sf) = \langle \alpha, f\rangle \qquad \langle \alpha, f\rangle \in \mathit{Domain}(os) \qquad \mathit{Eval}(e, \langle sf, os\rangle) = \lambda}{\langle sf, os\rangle, cs, \langle r := e\rangle \quad \overset{c}{\leadsto}_{cds} \quad \langle sf, os[\langle \alpha, f\rangle \mapsto \lambda]\rangle, cs, \langle\rangle}\texttt{WRITE}$$

$$\frac{r_0 \notin \mathit{StackVar} \qquad r_0 \in \mathit{Var}_S}{ec, cs, \langle r_0 := m(r_1, r_2)\rangle \quad \overset{c}{\leadsto}_{cds} \quad ec, cs, \langle \texttt{dummy}_S := m(r_1, r_2), r_0 := \texttt{dummy}_S\rangle}\texttt{SH}$$

$$\frac{\begin{array}{c}\mathit{GetExtBody}(m, cds) = \overline{s}_0 \qquad \mathit{DomainSorts}(m) = \langle C, S\rangle \\ m \notin \mathit{Constructor} \qquad \mathit{Eval}(r_0, \langle sf, os\rangle) = \alpha \qquad \mathit{Eval}(r_1, \langle sf, os\rangle) = \lambda\end{array}}{\begin{array}{c}\langle sf, os\rangle, cs, \langle sv := m(r_0, r_1)\rangle \quad \overset{c}{\leadsto}_{cds} \\ \langle \mathit{emptysf}[\texttt{this}_C \mapsto \alpha, \texttt{p}_S \mapsto \lambda], os\rangle, \langle sf, sv\rangle \triangleright cs, \overline{s}_0\end{array}}\texttt{CALL}$$

$$\frac{\begin{array}{c}\mathit{DomainSorts}(\texttt{new}_C) = \langle S_0, S_1\rangle \\ \mathit{Eval}(r_0, \langle sf, os_0\rangle) = \lambda_0 \qquad \mathit{Eval}(r_1, \langle sf, os_0\rangle) = \lambda_1 \\ \mathit{GetExtBody}(\texttt{new}_C, cds) = \overline{s}_0 \qquad \mathit{Alloc}(C, os_0, cds) = \langle \alpha, os_1\rangle\end{array}}{\begin{array}{c}\langle sf, os_0\rangle, cs, sv := \texttt{new}_C(r_0, r_1) \quad \overset{c}{\leadsto}_{cds} \\ \langle \mathit{emptysf}[\texttt{this}_C \mapsto \alpha, \texttt{q}_{S_0} \mapsto \lambda_0, \texttt{p}_{S_1} \mapsto \lambda_1], os_1\rangle, \langle sf, sv\rangle \triangleright cs, \overline{s}_0\end{array}}\texttt{CONSTR}$$

$$\frac{sf_0(rv) = \lambda}{\langle sf_0, os\rangle, \langle sf_1, sv\rangle \triangleright cs, \texttt{return}\ rv \quad \overset{c}{\leadsto}_{cds} \quad \langle sf_1[sv \mapsto \lambda], os\rangle, cs, \langle\rangle}\texttt{RET}$$

Aside. For simplicity, we have omitted the statement $v := \texttt{null}$. Extending the methodology to allow this statement is feasible, but is a significant complication as `null` has to be treated as a constant of every sort, to which no operations can be applied successfully.

Next, we formalize several well-known notions of termination.

Consider Def. 3.1.52. The intuition is that if $\mathit{ExecTerminatesIn}(\sigma_0, cds) = \sigma_1$, then the execution of σ_0 in the context of *cds* terminates in σ_1.

Definition 3.1.52. $\mathit{ExecTerminatesIn} : \mathit{State} \times \mathit{ClassDefSet} \hookrightarrow \mathit{State}$
$\mathit{ExecTerminatesIn}(\sigma_0, cds) = \sigma_1$ iff

$\sigma_0 \rightsquigarrow^*_{cds} \sigma_1$, and
there is no $\sigma_2 \in \mathit{State}$ such that $\sigma_1 \rightsquigarrow_{cds} \sigma_2$.

Consider Def. 3.1.53. If $\mathit{IsExecTerminating}(\sigma, \mathit{cds}) = \mathcal{T}$, then the execution of σ in the context of *cds*, terminates. It is defined in the obvious way.

Definition 3.1.53. $\mathit{IsExecTerminating} : \mathit{State} \times \mathit{ClassDefSet} \rightarrow \mathit{Bool}$
$\mathit{IsExecTerminating}(\sigma_0, \mathit{cds}) = \mathcal{T}$ *iff*
there is a $\sigma_1 \in \mathit{State}$ *such that* $\mathit{ExecTerminatesIn}(\sigma_0, \mathit{cds}) = \sigma_1$.

Consider Def. 3.1.54. The intuition is that if $\mathit{ExecTerminatesNormallyIn}(\sigma_0, \mathit{cds}) = \mathit{ec}$, then the execution of σ_0 in the context of *cds* terminates normally in a state with evaluation context *ec*. We say execution terminates normally in *ec*, rather than in $\langle ec, \sigma.ec, \langle\rangle\rangle$, as the stack frame and statement sequence follow implicitly.

Definition 3.1.54. $\mathit{ExecTerminatesNormallyIn} : \mathit{State} \times \mathit{ClassDefSet} \hookrightarrow \mathit{State}$
$\mathit{ExecTerminatesNormallyIn}(\sigma, \mathit{cds}) = \mathit{ec}$ iff
the execution of σ in the context of *cds* terminates in $\langle ec, \sigma.ec, \langle\rangle\rangle$.

Consider Def. 3.1.55. A program in the class-based language consists of a set of well-formed class definitions *cds* and a sequence of well-formed statements $\overline{s}$ that can be thought of as the body of the main method. A computation takes a such a program, and returns the evaluation context in which the execution of $\overline{s}$ in the context of *cds* terminates normally (if there is any).

Definition 3.1.55. $\mathit{Compute} : \mathit{Seq}(\mathit{Stmt}) \times \mathit{ClassDefSet} \hookrightarrow \mathit{EvalContext}$
$\mathit{Compute}(\overline{s}, \mathit{cds}) = \mathit{ec}$ iff
$\mathit{ExecTerminatesNormallyIn}(\langle \mathit{emptyec}, \langle\rangle, \overline{s}\rangle, \mathit{cds}) = \mathit{ec}$.

3.2 A Notion of Satisfaction

In this section, we open up the black box, now looking from the implementer's perspective rather than that of the client. The task of the implementer is to create an implementation that meets an agreed notion of satisfaction with regard to the specification.

The problem that we face when formalizing this notion of satisfaction is the following. In the previous section, we introduced a fairly standard model of computation in a class-based language. However, this model does not match the input/out behavior desired from the answer functions in the semantics of the specification (i.e., take a closed application and compute a closed application). Therefore, we separate the implementation into a presentation logic layer and a business logic layer. This is visualized in Fig. 3.3. The presentation logic layer is responsible for the translation from the input provided by the user to the input required by the computation in the business logic layer, and for displaying the output of the computation to the user as a closed application.

Aside. In some implementations it may be more efficient to intertwine the presentation logic and the business logic. Modeling such implementations may be feasible but requires 'polluting' the class based language presented in Sect. 3.1 with concepts needed for translation and display.

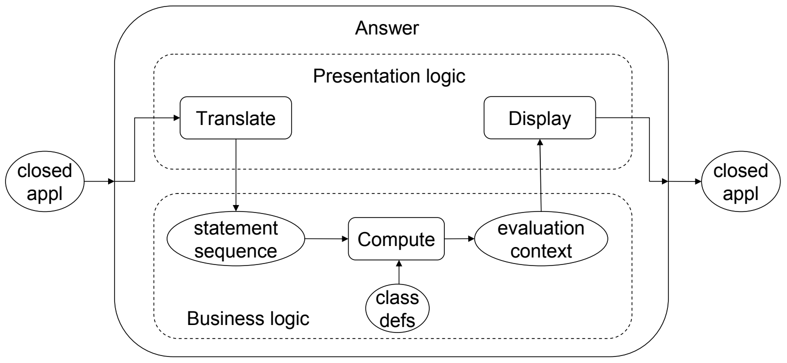


Figure 3.3: answer function flowchart

In this section, we formalize the translation and display concerns (Defs. 3.2.1 and 3.2.2). This, combined with the earlier formalization of the computation concern (Def. 3.1.55), allows to formalize a straightforward, suitable notion of satisfaction (Def. 3.2.5).

Consider Defs. 3.2.1 and 3.2.2. The translation and display concerns are both modeled as a black box. The intuition, sketched in Fig. 3.3, is as follows. The translation function takes the closed application that is provided by the user as input to the implementation, and maps this closed application to a sequence of statements that perform the actual computation (when executed in the context of a set of class definitions that are written by the implementer). If this computation terminates normally in an evaluation context *ec*, then the display function takes *ec* and maps it to closed application that is represented by *ec*. This closed application is presented as the output of the implementation to the user.

Definition 3.2.1. A *translation function* is a function $\mathit{translate} : \mathit{ClosedAppl} \hookrightarrow \mathit{Seq}(\mathit{Stmt})$.
Translate is the set of all translation functions.

Definition 3.2.2. A *display function* is a function $\mathit{display} : \mathit{EvalContext} \hookrightarrow \mathit{ClosedAppl}$.
Display is the set of all display functions.

Consider Def. 3.2.3. In our approach, an implementation consists of a translation function, a set of class definitions and a display function.

Definition 3.2.3. An *implementation* is a tuple $impl : Translate \times ClassDefSet \times Display$.
Implementation is the set of all implementations.

Consider Def. 3.2.4. The refinement of the implementation into translation, computation and display concerns allows to formalize the semantics of implementations in terms of answer functions.

Definition 3.2.4. $SemImpl : Implementation \rightarrow Answer$
$SemImpl(\langle translate, cds, display\rangle) = answer$ iff
for every $ca_0, ca_1 \in ClosedAppl$,
$answer(ca_0) = ca_1$ iff $display(Compute(translate(ca_0), cds)) = ca_1$.

Consider Def. 3.2.5. As the semantics of an implementation is an answer function, we can use the standard notion of satisfaction without making additional design decisions; An implementation *impl* satisfies a specification *AS* iff the semantics of the implementation (an answer function) is an element of the semantics of the specification (a set of answer functions).

Definition 3.2.5. $impl \in Implementation$ *satisfies* $CS \in ClientSpec$ iff $SemImpl(impl) \in SemAS(CS)$.

Given this notion of satisfaction, we now present an implementation approach for the presentation layer (Sect. 3.3). This allows to refine the notion of satisfaction to a notion of satisfaction for the computation concern (Sect. 3.4).

3.3 An Implementation Approach for Translation and Display

In this section, we sketch algorithms for the presentation layer, i.e. for the translation and display concerns. Our goal is for the presentation layer to be generic, i.e to be independent of or inferrable from the client specification and the implementation of the computation concern.

- For the translation concern, the approach is to let the input and output of the translation function be structurally very similar. This is achieved by having a method for every non-primitive operator in the client specification, and a class for every non-primitive sort.

- For the display concern, the approach is to require that the closed application *ca* that is to be displayed, is encoded in the input of the display function (an evaluation context) in a predefined and structured way. To this end, we use a special-purpose variable that refers to the root of a parse tree representing *ca*.

We discuss the translation concern in Sect. 3.3.1, and the display concern in Sect. 3.3.2. In Sect. 3.4, we use the implementation of the presentation layer to present a straightforward refinement of the satisfaction notion for implementations (Def. 3.2.5) to a satisfaction notion for the computation concern.

3.3.1 The Translation Concern

In this section we sketch a generic algorithm for the translation concern.

As stated above, this is achieved by letting the input and output of the translation function be structurally very similar. So, ideally, the translation function would translate an arbitrary closed application ca of a sort S to the single statement $\texttt{result}_S \texttt{ := } ca$. However, our simple programming language requires that the actual parameters of a call statement are variables (and not applications). For example, consider an algebraic specification AS that defines operations $f : S \times S \hookrightarrow S$, $g : \hookrightarrow S$, and $h : \hookrightarrow S$. Then $f(g(), h()) \in \mathit{ClosedIfaceAppl}(AS)$, but $\texttt{result}_\texttt{S} := \texttt{f}(\texttt{g}(), \texttt{h}()) \notin \mathit{Stmt}$. This slightly complicates the definition of the translation function. We present a translation function that treats the closed application provided as input, as a parse tree. The statements produced by the function execute this parse tree from the bottom up, storing intermediate results into variables that are used as actual parameters. We then discuss why this simple approach restricts the implementation of the computation concern, and present a simple solution. Finally, we sketch the algorithm that is the generic implementation of the translation concern.

Aside. In Sect. 2.2, we mentioned that the client may use an incremental process where the output of the implementation is the basis for another input. With the translation function described here, this is not very efficient as the answer to the original input is be computed again. Amending the implementation to retain and reuse the earlier result is certainly feasible.

Bottom-up translation. Consider Def. 3.3.1. The intuition is that if ca is a closed application of sort S, and $\mathit{TransBottomUp}(ca, n) = \overline{s}$, then

(1) $\overline{s}$ computes the value of ca and stores it in $\texttt{lv}n_S$ (the special purpose variables $\texttt{lv}i_S$ are defined in Def. 3.1.10), and
(2) for every $i < n$, for every $S \in \mathit{Sort}$, $\overline{s}$ does not assign to $\texttt{lv}i_S$ (this prevents intermediate results from being overwritten).

Note that intuition is formalized and proven correct in Lem. 4.2.

Definition 3.3.1. $\mathit{TransBottomUp} : \mathit{ClosedAppl} \times \mathbb{N} \hookrightarrow \mathit{Seq}(\mathit{Stmt})$
$\mathit{TransBottomUp}(ca_0, n) = \overline{s}_0$ *iff*

there are $f : S_1 \times \ldots \times S_i \hookrightarrow S_0 \in Op$, $ca_1, \ldots, ca_i \in ClosedAppl$, $\overline{s}_1, \ldots, \overline{s}_i \in Seq(Stmt)$ *such that*
ca_0 *is* $f(ca_1, \ldots, ca_i)$, *and*
$TransBottomUp(ca_1, n+1) = \overline{s}_1$, *and*
$\vdots$
$TransBottomUp(ca_2, n+i) = \overline{s}_i$, *and*
$\overline{s}_0 = \overline{s}_1 \triangleright \ldots \triangleright \overline{s}_i \triangleright \langle \mathtt{lv}n_{S_0} := f(\mathtt{lv}(n+1)_{S_1}, \ldots, \mathtt{lv}(n+i)_{S_i})\rangle$.

The definition is illustrated by Exmpl. 3.1 and Exmpl. 3.2.

Example 3.1. Consider a closed application $f(g(), h())$ such that operators f, g and h all have sort S. Then
$TransBottomUp(f(g(), h()), 0) =$
$\langle \mathtt{lv1}_S := g(), \mathtt{lv2}_S := h(), \mathtt{lv0}_S := f(\mathtt{lv1}_S, \mathtt{lv2}_S)\rangle$.

Example 3.2. Consider a closed application $l(f(g(), h()), i(j(), k()))$ such that operators f, g, h, i, j, k and l all have sort S.
$TransBottomUp(l(f(g(), h()), i(j(), k())), 0) =$

$$\begin{array}{lll} \langle & \mathtt{lv2}_S := g(), \mathtt{lv3}_S := h(), \mathtt{lv1}_S := f(\mathtt{lv2}_S, \mathtt{lv3}_S), & \\ & \mathtt{lv3}_S := j(), \mathtt{lv4}_S := k(), \mathtt{lv2}_S := i(\mathtt{lv3}_S, \mathtt{lv4}_S), & \\ & \mathtt{lv0}_S := l(\mathtt{lv1}_S, \mathtt{lv2}_S) & \rangle. \end{array}$$

Note that the reuse of $\mathtt{lv3}_S$ is harmless as the intermediate value it stores is no longer needed when it is reused.

Aside. The set of client specifications for which *TransBottomUp* can be used, is restricted somewhat by the desire to map every operator in the specification to either a primitive operator or a method. This is partly due to simplifications we have made (e.g., all methods have exactly two parameters) that are easily generalized. Furthermore, there may not be a primitive operator or a method with the same name due to syntactic restrictions of class-based programming languages in general (e.g., a naming scheme for constructors). In this case a slightly more involved problem-specific mapping is needed.

Unfortunately, *TransBottomUp* is not directly suitable for use as the generic translation function. We sketch the reason and present a slightly modified version that is suitable.

Problem: bottom-up translation restricts the computation implementation. Assume that as the generic translation function, we use the translation function *translate* that, given a closed application *ca*, returns *TransBottomUp*$(ca, 0)$. The problem is that this *translate* unnecessarily restricts the computation implementation. More specifically, this *translate* forces all intermediate results to encode for a canonical closed application in the predefined and structured way that is optimized for display. As illustrated by Exmpl. 3.3, this is due to the restriction that the display function places on the evaluation context yielded by the computation of a translated closed application.

Example 3.3. Consider closed application $f(g(), h())$ from Exmpl. 3.1. Assume that *ca* is the canonical form of $f(g(), h())$. Assume that computation of *TransBottomUp*$(f(g(), h()), 0)$ yields an evaluation context ec_0. Recall that the display function requires ec_0 to encode *ca* in a predefined and structured way. Then $\mathtt{lv1}_S$ must refer to an object α that encodes *ca* in a predefined and structured way (e.g. α is the root of a parse tree). Then $f(\mathtt{lv1}_S, \mathtt{lv2}_S)$ must return α.

Now consider closed application $l(f(g(), h()), i(j(), k()))$ from Exmpl. 3.2. Note that $f(g(), h())$ is contained in this application. Consider the computation of *TransBottomUp*$(l(f(g(), h()), i(j(), k())), 0)$. Assume that ec_1 is the evaluation context after the execution of $\mathtt{lv1}_S := f(\mathtt{lv2}_S, \mathtt{lv3}_S)$. Then ec_1 and ec_0 differ only on the naming of stack variables. Then $f(\mathtt{lv2}_S, \mathtt{lv3}_S)$ must return α. So, while $f(\mathtt{lv2}_S, \mathtt{lv3}_S)$ is used to compute an intermediate result, it must still return an object that encodes for a canonical closed application, in a way that is optimized for display.

Solution: bottum-up translation followed by canonical parse tree computation. As a solution to the problem with bottom up translation sketched above, we add one additional statement to the translation. This statement is responsible for converting the evaluation context computed by *TransBottomUp*, to an evaluation context that encodes for a canonical closed application in a predefined and structured way.

More specifically, if the input to translation is a closed application of sort S, then we add a statement $\mathtt{result}_{ParseTree}$ `:=` $\mathtt{makeCanon}_S(\mathtt{lv0}_S)$.

Consider Defs. 3.3.2 to 3.3.3. The intuition is that a `ParseTree` represents a closed application in a predefined and structured way. The details of a `ParseTree`, including the closed application it represents, are not relevant to the translation concern. Calling `makeCanon` on a variable, returns a `ParseTree`. For every primitive sort S, there is a primitive operation $\mathtt{makeCanon}_C$ in the language. For classes, `makeCanon` methods have to be implemented as part of the development process. Note that due to our earlier simplifications, `makeCanon` has two parameters. The second parameter of `makeCanon` is not relevant and is silently omitted. The minor technical difficulties introduced by using primitive operators with a class as their domain sort are straightforward to solve and are ignored.

Definition 3.3.2. There is a special purpose class `ParseTree` $\in$ *ClassSort*.

Definition 3.3.3. For every $C \in \mathit{ClassSort}$, there is a special purpose method $\texttt{makeCanon}_C : C \hookrightarrow \texttt{ParseTree} \in \mathit{Method}$. For every $S \in \mathit{PrimSig.sorts}$, there is a special purpose primitive operator $\texttt{makeCanon}_S : S \hookrightarrow \in \mathit{PrimSig.ops}$.

Consider Def. 3.3.4. *Trans*(ca) simply takes the outcome of *TransBottomUp*$(ca, 0)$ (in which $\texttt{lv0}_S$ stores the value of ca), and adds a statement $\texttt{result}_{\mathit{ParseTree}}$ `:=` $\texttt{makeCanon}_S\texttt{(lv0}_S\texttt{)}$. Example 3.4 illustrates the definition.

Definition 3.3.4. $\mathit{Trans} : \mathit{ClosedAppl} \hookrightarrow \mathit{Seq}(\mathit{Stmt})$
$\mathit{Trans}(ca) = \overline{s}_0$ *iff*
 there are $\overline{s}_1 \in \mathit{Seq}(\mathit{Stmt})$, $S \in \mathit{Sort}$ *such that*
  $\mathit{TransBottomUp}(ca, 0) = \overline{s}_1$, *and*
  ca has sort S, *and*
  $\overline{s}_0 = \overline{s}_1 \triangleright \langle \texttt{result}_{\mathit{ParseTree}} := \texttt{makeCanon}_S(\texttt{lv0}_S)\rangle$.

Example 3.4. Consider a closed application $f(g(), h())$ such that operators f, g and h all have sort S (like in Exmpl. 3.1).
$\mathit{Trans}(f(g(), h())) = \langle \texttt{lv1}_S := g(), \texttt{lv2}_S := h(), \texttt{lv0}_S := f(\texttt{lv1}_S, \texttt{lv2}_S),$
 $\texttt{result}_{\mathit{ParseTree}} := \texttt{makeCanon}_S(\texttt{lv0}_S)\rangle$.

The translation algorithm. Note that *Trans*, while not presented as an algorithm, is easily converted into one. The *TransBottomUp* part of this algorithm may require the signature of the algebraic specification of the client specification, to determine which operators may occur in the input.

3.3.2 The Display Concern

In this section we sketch a generic algorithm for the display concern.

As stated earlier, the simplest approach is to require that the closed application *ca* that is to be displayed, is encoded in the input of the display function (an evaluation context) in a predefined and structured way. To this end, our algorithm requires that in the evaluation context that is its input, the special-purpose $\texttt{result}_{\mathit{ParseTree}}$ variable refers to the root of a parse tree representing *ca*. Note that we have set up the translation algorithm to make sure that $\texttt{result}_{\mathit{ParseTree}}$ is defined. It is the concern of the computation implementer to ensure that the right parse tree is referred to by this variable (i.e., a parse tree that represents a canonical representation of the input of the black box).

Our main task is to define which closed application is represented by a given parse tree. Having done that, we then sketch the generic display algorithm.

The closed application represented by a parse tree. Here, we first define which closed application is represented by a given variable in a given state. We then use this definition to present a function that can infer a display function from the signature of the client specification.

Consider Def. 3.3.8. We require that `ParseTree` has fields `sort`, `o`, `p` and `q`. The intuition is that variable v represents a closed application ca_0 in an evaluation context ec, in the context of a signature sig, if and only if the following holds.

- v is of sort `ParseTree` and refers to an object α.
- Consider Def. 3.3.5. Roughly, *ViewOf* moves the fields of α from the object store to a stack frame. The intuition is that this gives the view of the evaluation context from the α object. More formally, $\mathit{ViewOf}(\alpha, os_0)$ yields an evaluation context $\langle sf, os_1\rangle$ such that 1) if location $\langle \alpha, f\rangle$ is mapped to value ν by os_0, then f is mapped to ν by sf, and 2) os_1 is like os_0, but with the locations $\langle \alpha, f\rangle$ removed from the domain.
- Let ec be the view of $\langle sf, os\rangle$ from α. Then the outermost operator of ca is determined by fields `sort` and `op` of α (which are on the stack in the view of ec from α) in the following way.
- Consider Def. 3.3.7. A stack frame sf represents an operator o in the context of a signature sig if and only if `sort` has value i and the i'th sort in sig is S, and `op` has value j and the j'th operator of sort S is o. Note that the operators of a given sort are determined using *OpsOfSort* (Def. 3.3.6), which uses *OpSort* defined in Def. 3.1.7 to determine the sort of an operator (this is the return sort for constructors and constants, and the sort of the first parameter otherwise).
- Assume that $ec.sf$ represents operator o. If o is a constant, then ca_0 is $o()$. Otherwise, recall that we have simplified the programming language to only consider non-constant methods and operators with exactly two parameters. In this case, fields `q` and `p` of α refer to parse trees that represent the closed applications that are the actual parameters of o. These closed application ca_1 and ca_2 are determined by recursive calls to obtain ca_0.
- We informally assume that for every primitive sort S, if $\texttt{makeCanon}_S$ is called on a variable v from an evaluation context ec, then it returns a parse tree that represents a closed application that evaluates to the abstract value of v in ec.

Note that the recursive definition of *CARepre* is well-formed due to the removal of elements from the domain of the object store by *ViewOf*.

Definition 3.3.5. $\mathit{ViewOf} : \mathbb{A} \times \mathit{ObjStore} \hookrightarrow \mathit{EvalContext}$
$\mathit{ViewOf}(\alpha, os_0) = \langle sf, os_1\rangle$ iff
there is $\mathit{fSet} \in \mathit{Set}(\mathit{Field})$ such that
$\mathit{fSet} = \{f \mid \langle \alpha, f\rangle \in \mathit{Domain}(os_0)\}$, and
$\mathit{Domain}(sf) = \mathit{fSet}$, and
for every $f \in \mathit{Domain}(sf)$,
$sf(f) = os_0(\langle \alpha, f\rangle)$, and
$\mathit{Domain}(os_1) = \mathit{Domain}(os_0) - \{\langle \alpha, f\rangle \mid f \in \mathit{fSet}\}$, and
for every $l \in \mathit{Domain}(os_1)$,
$os_1(l) = os_0(l)$.

Definition 3.3.6. $\mathit{OpsOfSort} : \mathit{Sort} \times \mathit{Sig} \to \mathit{Seq}(\mathit{Op})$
$\mathit{OpsOfSort}(S, sig) = \overline{o}$ iff $\overline{o}$ is $sig.ops$ restricted to operators o such that $\mathit{OpSort}(o) = S$.

Definition 3.3.7. $\mathit{RepresentedOp} : \mathit{StackFrame} \times \mathit{Sig} \hookrightarrow \mathit{Op}$
$\mathit{RepresentedOp}(sf, sig) = o$ iff
there are $i, j \in \mathbb{N}$, $S \in \mathit{Sort}$ such that
$sf(\mathtt{sort}) = i$, and
$sig.sorts[i] = S$, and
$\mathit{OpsOfSort}(S, sig) = \overline{o}$, and
$sf(\mathtt{op}) = j$, and
$\overline{o}[j] = o$, and

Definition 3.3.8. $\mathit{CARepre} : \mathit{Var} \times \mathit{EvalContext} \times \mathit{Sig} \hookrightarrow \mathit{ClosedAppl}$
$\mathit{CARepre}(v, \langle sf, os\rangle, sig) = ca_0$ iff
$\mathit{RefSort}(v) = \mathtt{ParseTree}$, and
there are $\alpha \in \mathbb{A}$, $ec \in \mathit{EvalContext}$, $o \in \mathit{Op}$ such that
$sf(v) = \alpha$, and
$\mathit{ViewOf}(\alpha, os) = ec$, and
$\mathit{RepresentedOp}(ec.sf, sig) = o$, and
or $o \in \mathit{Constant}$, and
ca_0 is $o()$,
or $o \notin \mathit{Constant}$, and
there are $ca_1, ca_2 \in \mathit{ClosedAppl}$ such that
$ca_1 = \mathit{CARepre}(\mathtt{q}, ec, AS)$, and
$ca_2 = \mathit{CARepre}(\mathtt{p}, ec, AS)$, and
$ca_0 = o(ca_1, ca_2)$.

Consider Def. 3.3.9. The intuition is that *DisplayFactory* can infer a display function from the signature of the client specification. This display function *display* maps an evaluation context *ec* to a closed application *ca* iff in *ec*, the result variable refers to a parse tree that represents *ca* in the context of *sig*.

Definition 3.3.9. $\mathit{DisplayFactory} : \mathit{Sig} \to \mathit{Display}$
$\mathit{DisplayFactory}(sig) = \mathit{display}$ *iff*
for every $ec \in \mathit{EvalContext}$, $ca \in \mathit{ClosedAppl}$,
$\mathit{display}(ec) = ca$ *iff* $\mathit{CARepre}(\mathtt{result}_{\mathit{ParseTree}}, ec, sig) = ca$.

The display algorithm. Note that *CARepre* (and therefore *DisplayFactory*), while not presented as an algorithm, is easily converted into one. Essentially, this conversion results in an implementation of the visitor pattern, which traverses the tree, constructing the closed application to display in a bottom-up fashion (i.e., the outermost operator o is determined last, and applied to two closed applications ca_1 and ca_2 which have been determined by visits to `q` and `p`.) Proof of satisfaction for display then reduces to proving that the straightforward conversion from *CARepre* into an algorithm is correct. Details are outside the scope of this chapter.

> **Aside.** *CARepre* depends on the signature of the algebraic specification provided by the client, but only for the (order of the) available of classes and operators. To make *CARepre* independent of the client specification all together, one could use e.g. a string representation instead (at the cost of more complex definitions and possibly a less efficient display implementation).

3.4 Satisfaction for Computation

Consider Def. 3.4.1. Given the generic translation and display functions, it is straightforward to refine the notion of satisfaction for implementations (Def. 3.2.5) to a notion of satisfaction for the computation concern. This notion is the main result of Sect. 3.

Definition 3.4.1. $cds \in \mathit{ClassDefSet}$ *satisfies*$_c$ $CS \in \mathit{ClientSpec}$ iff
$\mathit{SemImpl}(\langle \mathit{Trans}(), cds, \mathit{DisplayFactory}(CS.as.sig)\rangle) \in \mathit{SemAS}(CS)$.

Theorem 3.1. *For every* $CS \in \mathit{ClientSpec}$, $cds \in \mathit{ClassDefSet}$,
if cds *satisfies*$_c$ CS,
then $\langle \mathit{Trans}(), cds, \mathit{DisplayFactory}(CS.as.sig)\rangle$ *satisfies* CS.

Proof is trivial, as this theorem basically only states that *Trans*() and *DisplayFactory*(*CS.as*) are a translation function and a display function.

4 Implementer's Perspective: An Implementation Approach

In this section, we present an implementation approach that formalizes and extends ideas from [Hoa72], Hoare's seminal paper on data abstraction.

(1) In Sect. 4.1, we connect the satisfaction notion for computation that we introduced in Def. 3.4.1, to a notion of satisfaction that is based on Hoare's notion

of data abstraction. We observe that this Hoare-style satisfaction alone is not sufficient to establish satisfaction for computation as it does not account for the requirement of displaying the output of the computation in a generic way. As a solution, we add an additional step to the computation process that is responsible for the transformation of the computed object, to an object that is suitable for display.

(2) In Sect. 4.2, we formalize an implementation approach for Hoare-style satisfaction that allows a method implementer to use an abstract view of the state when reasoning about the execution of the method body.

4.1 Separation of Satisfaction Concerns for Computation

In this section, we connect the satisfaction notion for computation that we introduced in Def. 3.4.1, to a notion of satisfaction that has a Hoare-style notion of data abstraction at its core. This connection is made in the following way.

- Hoare's approach allows the user of the computation implementation (in this case the translator) to treat the execution of a method or primitive operation as the evaluation of the operator with the same name in a model of the specification. In Sect. 4.1.1, we formalize this notion of an abstract execution. The essential property for our approach is the following. Assume $\mathcal{A}$ is an arbitrary model of the specification. If closed application *ca* evaluates to a value ν in $\mathcal{A}$, then the abstract execution of the sequence of statements *TransBottomUp*(*ca*) (see Def. 3.3.1), terminates in a valuation in which `lv0` is mapped to ν.

- In Sect. 4.1.2, we formalize a Hoare-style notion of satisfaction. At the core of this notion is a mapping from evaluation contexts to valuations that follows from an implementer-defined encoding of values in objects. If evaluation context *ec* is mapped to a valuation *va*, then we call *va* the abstract view of *ec*. Hoare-style satisfaction ensures that if the abstract execution of a statement sequence $\overline{s}$ terminates in a valuation *va*, then the concrete execution of $\overline{s}$ terminates in an evaluation context of which the abstract view is *va*.

- We observe that Hoare-style satisfaction alone is not sufficient to establish satisfaction for computation as defined in Def. 3.4.1. More specifically, Hoare-style satisfaction ensures that, given a closed application *ca*, the concrete execution of the sequence of statements *TransBottomUp*(*ca*) terminates in an evaluation context that encodes the value of *ca* in the object referred to by `lv0`. However, it does not account for the concern of producing a parse tree that encodes the canonical representation of *ca*. In Sect. 4.1.3, we introduce a notion of satisfaction for the `makeCanon`() method that accounts for this concern.

- In Sect. 4.1.4, we prove that satisfaction for the computation concern can indeed be separated into Hoare-style satisfaction and satisfaction for the `makeCanon`() method.

4.1.1 Abstract Executions

In this section, we formalize the notion of an abstract execution by providing a syntax and semantics.

Syntax. Consider Def. 4.1.1, which defines the set of variables that can occur in abstract statements. Note that $AVar \subset Ref$.

Definition 4.1.1. *$AVar$ is the set $Field \cup FPar \cup LocVar$.*

Consider Fig. 4.1, which defines a grammar for abstract statements *AStatement*. Note that $AStatement \subset Statement$ (Fig. 3.2). The main restriction is that there is no abstract statement that reads a field of a stack variable other than `this`. Also note that we no longer have to distinguish between primitive operation calls and methods calls: at the abstract level, these are treated in the same way. Likewise, we don't have to distinguish between calls and variables in the right-hand side of assignments.

$$v \in AVar,\ \overline{s} \in Seq(AStatement),\ o \in Op2,\ po \in PrimSig.ops$$
$$\begin{array}{lcl} c \in ACall & ::= & po() \mid o(v,v) \\ e \in AExpr & ::= & c \mid v \\ s \in AStatement & ::= & \texttt{if}\ (v)\ \overline{s} \mid v := e \end{array}$$

Figure 4.1: abstract statement grammar.

Aside. For simplicity, we have omitted the `while` statement. It can be treated in the standard way, using a loop invariant. Alternatively, a loop could be mimicked by a call of a method m with a body that, if the loop guard holds, first executes the loop body, and then calls m again.

Aside. For simplicity, we require abstract statements to be a subset of concrete statements. In Hoare's work, there an explicit mapping from operators to methods which gives additional flexibility. For example, due to our direct mapping, the constructor $\texttt{new}_C$ of a class C (the name of which cannot be chosen by the implementer) must be an operator in the specification. The extension with an additional mapping is straightforward.

Consider Def. 4.1.2. We are only interested in the subset of abstract statements that are well-typed (Def. 3.1.16). *AStmt* is the subset of all abstract statements that are well-typed. Note that $AStmt \subset Stmt$ (see Sect. 3.1.1).

Definition 4.1.2. *$AStmt$ is the set $Stmt \cap AStatement$.*

Consider Def. 4.1.3. Although method calls and primitive operation calls are treated in the same way at the level of abstract executions, abstract method call statements still play a prominent role in the remainder of this chapter. For convenience, *AMCStmt* is defined as the set of all well-typed abstract method call statements.

Definition 4.1.3. *AMCStmt* is the set $\{v_0 := m(v_1, v_2) \mid v_0, v_1, v_2 \in \mathit{AVar}, m \in \mathit{Method}\} \cap \mathit{AStmt}$.

Semantics. Consider Def. 4.1.4. The operational small-step semantics of *AStmt* is based around abstract states, which consist of a valuation and a sequence of well-typed abstract statements.

Definition 4.1.4. An *abstract state* is a record $va \in \mathit{Valuation} \times \mathit{stmtSeq} \in \mathit{Seq}(\mathit{AStmt})$. *AState* is the set of all abstract states.

The semantics of an abstract execution is given by transition relation $\overset{a}{\rightsquigarrow}_{\mathcal{A}}$ defined in Fig. 4.2.

$$\frac{va(v_{Bool}) = \mathcal{T}}{va, \langle \mathtt{if}\ (v_{Bool})\ \overline{s}_0\rangle \overset{a}{\rightsquigarrow}_{\mathcal{A}} va, \overline{s}_0}\ \mathtt{IFt} \qquad \frac{|\,\overline{s}_1\,| > 0 \qquad va_0, \langle s_0\rangle \overset{a}{\rightsquigarrow}_{\mathcal{A}} va_1, \overline{s}_2}{va_0, \langle s_0\rangle \triangleright \overline{s}_1 \overset{a}{\rightsquigarrow}_{\mathcal{A}} va_1, \overline{s}_2 \triangleright \overline{s}_1}\ \mathtt{COMP}$$

$$\frac{va(v_{Bool}) = \mathcal{F}}{va, \langle \mathtt{if}\ (v_{Bool})\ \overline{s}_0\rangle \overset{a}{\rightsquigarrow}_{\mathcal{A}} va, \langle\rangle}\ \mathtt{IFf} \qquad \frac{\mathit{Sem}(e, va, \mathcal{A}) = \nu}{va, \langle v := e\rangle \overset{a}{\rightsquigarrow}_{\mathcal{A}} va[v \mapsto \nu], \langle\rangle}\ \mathtt{ASSIGN}$$

Figure 4.2: Small-step semantics for abstract execution of *AStmt*. Brackets around abstract states are omitted to improve readability. $va, va_0, va_1 \in \mathit{Valuation}, \mathcal{A} \in \mathit{Alg}, \nu \in \mathbb{V}, v_{Bool}, v, v_0, v_1 \in \mathit{AVar}, s_0 \in \mathit{AStmt}, \overline{s}_0, \overline{s}_1, \overline{s}_2 \in \mathit{Seq}(\mathit{AStmt}), c \in \mathit{ACall}$.

Consider Def. 4.1.5. In an abstract execution, we can 'create' fields by assigning to them. Using *FieldsAssignedTo*, we can reason about (and thus control) the set of fields that is assigned to by an abstract statement sequence.

Definition 4.1.5. $\mathit{FieldsAssignedTo} : \mathit{Seq}(\mathit{AStmt}) \to \mathit{Seq}(\mathit{Field})$
$f \in \mathit{FieldsAssignedTo}(\overline{s})$ iff
there are $s \in \mathit{SubStmts}(\overline{s})$, $e \in \mathit{AExpr}$ such that
s is $f := e$.

4.1.2 Hoare-style Satisfaction for Computation

In this section, we use the notion of abstract executions to present a notion of satisfaction for a set of class definitions that allows to reason about properties of the concrete execution at the level of the much simpler abstract execution. In particular, this allows the user of the classes, in this case the translator, to be an abstract programmer in the sense of [Hoa72].

To relate the concrete execution to the abstract execution, the computation implementer must define an *abstraction operator* for every class (called an abstraction function in [Hoa72]). This allows to map an evaluation context to a valuation. If evaluation context ec is mapped to a valuation va, then we call va the abstract view of ec.

First, we show how the computation implementer defines the abstraction operators. Then, we formalize how the abstraction operators are used to determine the abstract view of an evaluation context. Finally, we present a Hoare-style notion of satisfaction which roughly requires the following. If the abstract execution of a statement sequence $\overline{s}$ terminates in a valuation va, then the concrete execution of $\overline{s}$ terminates in an evaluation context of which the abstract view is va.

Defining abstraction operators. We show how the computation implementer defines the abstraction operators in Def. 4.1.6. A concrete example can be found in Exmpl. 4.1.

Consider Def. 4.1.6. In our approach, the computation implementer defines the abstraction operators using an algebraic specification AS_1. For every class C in the implementation cds, $AS_1.sig$ contains a special-purpose abstraction operator $\mathtt{abstr}_C$. The domain sorts of $\mathtt{abstr}_C$ are the sorts of the fields defined in class C, and the range sort of $\mathtt{abstr}_C$ is C. The intention is that the set of axioms $AS_1.ax$ define $\mathtt{abstr}_C$ in terms of the operators from the algebraic specification AS_0 of the client specification, possibly using additional auxiliary operators defined in $AS_1.sig$. $AbstrOpDefs(cds, AS_0)$ returns the set of all possible definitions of the abstraction operators.

Definition 4.1.6. $AbstrOpDefs : ClassDefSet \times AlgSpec \rightarrow Set(AlgSpec)$
$AS_1 \in AbstrOpDefs(cds, AS_0)$ iff
for every $C \in GetClassNames(cds)$,
there is an $\mathtt{abstr}_C \in AS_1.sig.ops$ such that
$DomainSorts(\mathtt{abstr}_C) = GetFieldSorts(C, cds)$, and
$RangeSort(\mathtt{abstr}_C) = C$, and
$AS_1.ax \subseteq Terms(SigUnion(AS_0.sig, AS_1.sig))$.

Example 4.1. (*Implementing Rationals*)

Recall the algebraic specification of rationals in Exmpl. 2.1. Here, we present an implementation, including an axiom that defines the abstraction operator $\mathtt{abstr}_{Rat}$. The signature of the abstraction operator definition is left implicit as it is clear from the context. Note that we use some self-explanatory syntactic sugar. The $\mathtt{makeCanon}_{Rat}()$ method is further discussed in Exmpl. 4.5.

```
class Rat {
 Int n; //nominator
 Int d; //denominator
 abstr axioms
```

$\forall n, d \in \mathit{Int} \bullet$
$d > 0 \Rightarrow \mathtt{abstr}_{Rat}(n, d) = \mathtt{new}_{Rat}(n, d)$

```
 new$_{Rat}$(Int i, Int j) {
  n := i;
  d := j;
 }

 void add(Rat r) {
  Int n1 := n * r.d;
  Int n2 := r.n * d;
  n := n1 + n2;
  d := d * r.d;
 }

 Bool equals(Rat r) {
  Int n1 := n * r.d;
  Int n2 := r.n * d;
  return (n1 == n2);
 }

 ParseTree makeCanon$_{Rat}$() {
  //make this Rat canonical by calculating and using the greatest
  common divider.
  Int gcd := n;
  Int a := d;
  while (a != 0) {
   Int remainder := gcd % a;
   gcd := a;
   a := remainder;
  }
  n := n \ gcd;
  d := d \ gcd;
  //create a parseTree from this Rat
  ParseTree result := new$_{ParseTree}$(0, 0, makeCanon$_{Int}$(n),
  makeCanon$_{Int}$(d));
  return result;
 }
}
```

Formalizing abstract views. We formalize the notion of an abstract view in Defs. 4.1.7 to 4.1.9. An concrete example can be found in Exmpl. 4.2.

Consider Def. 4.1.9. The abstract view *va* of an evaluation context *ec* maps every variable v on the stack frame of *ec* to its abstract value (if there is such a value). This abstract value is determined using an algebra $\mathcal{A}$ and an algebraic specification AS. The intuition is that $\mathcal{A}$ is a model of the algebraic specification provided by the client, and that AS is an algebraic specification of the abstraction operators, i.e., AS is an element of *AbstrOpDefs* (see Def. 4.1.6). This intuition is formalized in the Hoare-style satisfaction notion presented later (Def. 4.1.11).

Consider Def. 4.1.8. Roughly, it defines the abstract value of a variable v_S in an execution state $\langle sf, os\rangle$. If v_S is a primitive variable that is defined by sf, then its abstract value is $sf(v_S)$. The intuition is that for primitive variables, the abstract value is the same as the concrete value. If v_S is not a primitive variable but refers to an object α, then its abstract value is determined using the algebra $\mathcal{A}$ and algebraic specification AS provided from *AbstrView* (see the intuition of Def. 4.1.9 above). This is done by evaluating the application of the abstraction operator of S to the sequence $\overline{f}$ of the fields defined by class S of *cds*.

So, the abstract value of an object depends on the abstract values of its fields (there are what is often referred to as 'layers of abstraction'). Valuation *va* maps every field of α to its abstract value (if it has one). This is accomplished by letting *va* be the abstract view of $\mathit{ViewOf}(\alpha, os)$. Recall that *ViewOf* (Def. 3.3.5) moves the fields of α from the object store to a stack frame, thus giving the concrete view of the evaluation context from the α object.

Consider Def. 4.1.7. The intuition is that the evaluation of an application of the abstraction operator only yields a value if all models $\mathcal{A}_1$ of AS_1 that include $\mathcal{A}_0$ (i.e., that agree with $\mathcal{A}_0$ on the interpretation of the operators from the client specification), agree on the value of the application of the abstraction operator. Requiring this for every such $\mathcal{A}_1$, is a technical trick to deal with underspecified abstraction operators. The benefit is illustrated by Exmpl. 4.3.

Note that the mutually recursive definitions of *AbstrView* and *AbstrVal* are well-formed due to the removal of elements from the domain of the object store by *ViewOf*.

Definition 4.1.7. $\mathit{Eval} : \mathit{Term} \times \mathit{Valuation} \times \mathit{Alg} \times \mathit{AlgSpec} \hookrightarrow \mathbb{V}$
$\mathit{Eval}(t, va, \mathcal{A}_0, AS) = \nu$ iff
  for every $\mathcal{A}_1 \in \{\mathcal{A} \mid \mathcal{A}_0 \subseteq \mathcal{A} \wedge \mathcal{A} \in \mathit{Models}(AS.ax)\}$
    $\nu = \mathit{Sem}(t, va, \mathcal{A}_1)$.

Definition 4.1.8. $\mathit{AbstrVal} : \mathit{Var} \times \mathit{EvalContext} \times \mathit{Alg} \times \mathit{AlgSpec} \times \mathit{ClassDefSet} \hookrightarrow \mathbb{V}$
$\mathit{AbstrVal}(v_S, \langle sf, os\rangle, \mathcal{A}, AS, cds) = \nu$ iff

or $S \in \mathit{PrimSig.sort}$ and $\nu = \mathit{sf}(v_S)$,
or $S \in \mathit{ClassSort}$, and
there are $\alpha \in \mathbb{A}$, $\mathit{va} \in \mathit{Valuation}$, $\overline{f} \in \mathit{Seq}(\mathit{Field})$, $\mathit{ec} \in \mathit{EvalContext}$
such that
$\mathit{sf}(v_S) = \alpha$, and
$\mathit{ViewOf}(\alpha, \mathit{os}) = \mathit{ec}$, and
$\mathit{va} = \mathit{AbstrView}(\mathit{ec}, \mathcal{A}, \mathit{AS}, \mathit{cds})$, and
$\overline{f} = \mathit{GetFields}(S, \mathit{cds})$, and
$\mathit{Eval}(\texttt{abstr}_S(\overline{f}), \mathit{va}, \mathcal{A}, \mathit{AS}) = \nu$.

Definition 4.1.9. $\mathit{AbstrView} : \mathit{EvalContext} \times \mathit{Alg} \times \mathit{AlgSpec} \times \mathit{ClassDefSet} \hookrightarrow \mathit{Valuation}$
$\mathit{AbstrView}(\mathit{ec}, \mathcal{A}, \mathit{AS}, \mathit{cds}) = \mathit{va}$ iff
$\mathit{Domain}(\mathit{va}) = \mathit{Domain}(\mathit{ec.sf})$, and
for every $v \in \mathit{Domain}(\mathit{va})$,
$\mathit{va}(v) = \mathit{AbstrVal}(v, \mathit{ec}, \mathcal{A}, \mathit{AS}, \mathit{cds})$.

Example 4.2. (*Determining an abstract view*)

Consider the algebraic specification AS_0 of rationals in Exmpl. 2.1. Consider the set of class definitions *cds* that consists only of class `Rat` defined in Exmpl. 4.1. Consider the algebraic specification AS_1 that consists only of the specification of $\texttt{abstr}_{\mathit{Rat}}()$ in class `Rat`. Consider an arbitrary $\mathcal{A}_0 \in \mathit{Models}(AS_0.\mathit{ax})$.

Now extend $\mathcal{A}_0$ with an interpretation of $\texttt{abstr}_{\mathit{Rat}}$ that is in accordance with ax. That is, consider an $\mathcal{A}_1$ such that $\mathcal{A}_0 \subseteq \mathcal{A}_1 \wedge \mathcal{A}_1 \in \mathit{Models}(\mathit{ax})$.

Consider an evaluation context ec_0 that consists of a stack frame $\{v_{\mathit{Rat}} \mapsto \alpha\}$ and an object store *os* such that $\mathit{os} = \{\langle \alpha, n\rangle \mapsto 2, \langle \alpha, d\rangle \mapsto 4\}$. Then $\mathit{ViewOf}(\alpha, \mathit{os})$, the concrete view of ec_0 from α, is the evaluation context ec_1 such that $\mathit{ec}_1 = \langle \{n \mapsto 2, d \mapsto 4\}, \langle\rangle\rangle$. Then $\mathit{AbstrView}(\mathit{ec}_1, \mathcal{A}_1, AS_1, \mathit{cds})$, the abstract view of ec_1, is a valuation *va* such that $\mathit{va} = \mathit{ec}_1.\mathit{sf}$, as both fields of `Rat` are of the primitive sort `Int`.

Then $\mathit{AbstrVal}(v_{\mathit{Rat}}, \mathit{ec}_0, \mathcal{A}_0, AS_1 \mathit{cds})$, the abstract value of v_{Rat} in ec_0, is $\mathit{Sem}(\texttt{new}_{\mathit{Rat}}(2,4), \mathit{emptyva}, \mathcal{A}_0)$. The reasoning is the following. Extend $\mathcal{A}_0$ with an arbitrary interpretation of $\texttt{abstr}_{\mathit{Rat}}$ that is in accordance with AS_1. That is, consider an $\mathcal{A}_1$ such that $\mathcal{A}_0 \subseteq \mathcal{A}_1 \wedge \mathcal{A}_1 \in \mathit{Models}(AS_1.\mathit{ax})$. As $\mathcal{A}_1 \in \mathit{Models}(\mathit{ax})$, we know that $\mathit{Sem}(\texttt{abstr}_{\mathit{Rat}}(2,4), \mathit{va}, \mathcal{A}_1) = \mathit{Sem}(\texttt{new}_{\mathit{Rat}}(2,4), \mathit{emptyva}, \mathcal{A}_1)$. As $\mathcal{A}_0 \subseteq \mathcal{A}_1$, and as $\texttt{new}_{\mathit{Rat}}$ is an operator from the client specification, we know that $\mathit{Sem}(\texttt{new}_{\mathit{Rat}}(2,4), \mathit{emptyva}, \mathcal{A}_1) = \mathit{Sem}(\texttt{new}_{\mathit{Rat}}(2,4), \mathit{emptyva}, \mathcal{A}_0)$. So, the abstract value of v_{Rat} depends on the choice of the model of $AS_0.\mathit{ax}$ (i.e., on $\mathcal{A}_0$), but not on the choice of interpretation of $\texttt{abstr}_{\mathit{Rat}}$ (i.e., on how $\mathcal{A}_0$ is extended to $\mathcal{A}_1$). This is because the evaluation of $\texttt{new}_{\mathit{Rat}}(2,4)$ does not depend on the interpretation of $\texttt{abstr}_{\mathit{Rat}}$.

Then $\mathit{AbstrView}(\mathit{ec}_0, \mathcal{A}_0, AS_1, \mathit{cds}) = \{v_{\mathit{Rat}} \mapsto \nu\}$, where $\nu = \mathit{Sem}(\texttt{new}_{\mathit{Rat}}(2,4), \mathit{emptyva}, \mathcal{A}_0)$.

Now consider an evaluation context ec_2 that is like ec_0 but with location $\langle \alpha, d \rangle$ mapped to 0. Note that in this case, the interpretation of the application of $\mathtt{abstr}_{Rat}$ *does* depend on the choice of the interpretation of $\mathtt{abstr}_{Rat}$, as AS_1 does not specify $\mathtt{abstr}_{Rat}(2, 0)$ in terms of operators from the client specification. For example, there is an $\mathcal{A}_1$ such that 1) $\mathcal{A}_0 \subseteq \mathcal{A}_1 \wedge \mathcal{A}_1 \in \mathit{Models}(AS_1.ax)$, and 2) $\mathit{Sem}(\mathtt{abstr}_{Rat}(2, 0), ec_2.sf, \mathcal{A}_1)$ is undefined. Therefore, $\mathit{AbstrVal}(v_{Rat}, ec_0, \mathcal{A}_0, AS_1, cds)$ is undefined, and $\mathit{AbstrView}(ec_2, \mathcal{A}_0, AS_1, cds) = \mathit{emptyva}$.

Hoare-style satisfaction for computation. Consider Def. 4.1.11. The intuition is that Hoare-style satisfaction is a notion of satisfaction of the implementation of the computation concern *cds* with regards to the algebraic specification AS_0 that is part of the client specification. Note that it does not consider the canonicity function (for which we define a separate notion of satisfaction in Sect. 4.1.3). Hoare-style satisfaction ensures that there are a model $\mathcal{A}$ of AS_0 and an algebraic specification AS_1 of the abstraction operators, for which *csd respects*$_h$ $\langle \mathcal{A}, AS_1 \rangle$.

Consider Def. 4.1.10. Consider a sequence of abstract statements $\overline{s}$ that does not assign to any fields. Note that the statement sequences constructed by *Trans* do not assign to fields. Roughly, *csd respects*$_h$ $\langle \mathcal{A}, AS_1 \rangle$ if the following holds. If the abstract execution of $\overline{s}$ terminates in a valuation *va*, then the concrete execution of $\overline{s}$ terminates in an evaluation context of which the abstract view is *va*.

Note that there is no requirement for a statement sequence that contains an operator *o* that is not interpreted by $\mathcal{A}$ (for example because *o* does not occur in the client specification), as the abstract execution of such a statement sequence does not terminate normally.

Furthermore, note that the implementer of the computation concern must define the abstraction operators (by means of algebraic specification AS_1), but these definitions are not part of the implementation. That is, AS_1 does not have to be provided to the client.

Some of the consequences of the satisfaction notion are illustrated by Exmpls. 4.3 and 4.4.

Definition 4.1.10. $\langle cds, AS_1 \rangle \in \mathit{ClassDefSet} \times \mathit{AlgSpec}$ *respects*$_h$ $\mathcal{A} \in \mathit{Alg}$ iff
for every $\overline{s} \in \mathit{Seq}(\mathit{AStmt})$, $va \in \mathit{Valuation}$,
if $\mathit{FieldsAssignedTo}(\overline{s}) = \{\}$, and
$\langle \mathit{emptyva}, \overline{s} \rangle \overset{a}{\rightsquigarrow}{}^{+}_{\mathcal{A}} \langle va, \langle\rangle \rangle$,
then there is an $ec \in \mathit{EvalContext}$ such that
$\langle \mathit{emptyec}, \langle\rangle, \overline{s} \rangle \overset{c}{\rightsquigarrow}{}^{+}_{cds} \langle ec, \langle\rangle, \langle\rangle \rangle$, and
$\mathit{AbstrView}(ec, \mathcal{A}, AS_1, cds) = va$.

Definition 4.1.11. $cds \in \mathit{ClassDefSet}$ *satisfies*$_h$ $AS_0 \in \mathit{AlgSpec}$ iff there are $\mathcal{A} \in \mathit{Models}(AS_0.ax)$, $AS_1 \in \mathit{AbstrOpDefs}(cds, AS_0)$ such that $\langle cds, AS_1 \rangle$ $\mathit{respects}_h$ $\mathcal{A}$.

Aside. The notion of *respects*$_h$ essentially formalizes the *consequence* of Hoare's criterion of correctness of data representations at the semantical level, which is that an abstract execution of a program can be 'replaced' by a concrete execution. It uses total correctness rather than partial correctness. Note that the criterion of correctness itself is formalized at the semantical level in Sect. 4.2, when we discuss an implementation approach for Hoare-style satisfaction.

Example 4.3. (*Hoare-style satisfaction: consequences 1*)

Recall from Def. 4.1.8 : *AbstrVal* that a variable only has an abstract value when *every* model of the axiomatization of the abstraction operators agrees on the value. This example illustrates the benefit of that choice.

1. Consider the execution of $v_{Rat} := \texttt{new}_{Rat}(1, 2)$, with class `Rat` defined as in Exmpl. 4.1.

2. Roughly, satisfaction requires that there is a model $\mathcal{A}_0$ such that the abstract view of the computed evaluation context, maps v_{Rat} to a value ν such that $\nu = \mathit{Sem}(\texttt{new}_{Rat}(1, 2), \mathit{emptyva}, \mathcal{A}_0)$.

3. As in Exmpl. 4.2, consider an evaluation context ec_2 that maps v_{Rat} to a `Rat` object that has a `d` field with a value of 0.

4. Recall from Exmpl. 4.2 that the abstract view of ec_2 is *emptyva*, due to the abstract value definition.

5. Note however, that there is *a* model $\mathcal{A}_1$ of the axiomatization such that the abstract value of v_{Rat} in ec_2 is ν, i.e., such that $\mathit{AbstrVal}(v_{Rat}, ec_1, \mathcal{A}_1, cds) = \nu$. The reason is that the axiomatization of $\texttt{abstr}_{Rat}$ does not restrict the value of $\texttt{abstr}_{Rat}(\texttt{n}, \texttt{d})$ when `d` is 0.

6. Therefore, the benefit of the abstract value definition is that it prevents that a satisfying *cds* can be such that the execution terminates in ec_2.

7. In contrast, consider an evaluation context ec_1 that maps v_{Rat} to a `Rat` object that has an `n` field with a value of 1 and a `d` field with a value of 2. As desired, a satisfying *cds* can be such that the execution terminates in ec_1. The reason is that the abstract value of v_{Rat} is ν (see Exmpl. 4.2).

Example 4.4. (*Hoare-style satisfaction: consequences 2*)

This example illustrates that Hoare-style satisfaction requires that translations that produce the same result at the abstract level, may produce different results at the concrete level as long as the abstract values are the same.

Recall the implementation of rationals in Exmpl. 4.1. Consider the three statement sequences $t1, t2$ and $t3$.

$t1:$ `lv0`$_{Rat}$ `:= new`$_{Rat}$`(1,4); lv1`$_{Rat}$ `:= new`$_{Rat}$`(3,4);`
`result`$_{Rat}$ `:= add(lv0`$_{Rat}$`, lv1`$_{Rat}$`);`
$t2:$ `result`$_{Rat}$ `:= new`$_{Rat}$`(4, 4);`
$t3:$ `result`$_{Rat}$ `:= new`$_{Rat}$`(1, 1);`

Note that *TransBottomUp* (Def. 3.3.1) yields $t1$ for the input $add(\texttt{new}_{Rat}(1,4), \texttt{new}_{Rat}(3,4))$.

Also note that the abstract executions of $t2$ and $t3$ terminate in the same valuation va_0, and that the valuation va_1 in which the abstract valuation of $t1$ terminates differs only from va_0 in that va_1 additionally defines variables `lv0` and `lv1`.

Let ec_1, ec_2 and ec_3 be the evaluation contexts in which the concrete executions of $t1$, $t2$ and $t3$ terminate. Note that ec_1 and ec_2 both map `result`$_{Rat}$ to a `Rational` of which both the `n` field and the `d` field are 4. In ec_3, `result`$_{Rat}$ maps to a `Rational` of which both fields are 1. Hoare-style satisfaction requires the abstract value of `result`$_{Rat}$ to be the same in all three evaluation contexts.

Note that this is indeed the case as the difference in the evaluation contexts is made up for by the abstraction function. More specifically, $\texttt{abstr}_{Rat}(1,1)$ and $\texttt{abstr}_{Rat}(4,4)$ are equal in every model of the axiomatization of `abstr`$_{Rat}$ that agrees with any model of the axiomatization of the rationals that is part of the client specification.

We sketch a verification methodology for this notion of satisfaction in Sect. 5.

4.1.3 Satisfaction for the makeCanon Methods

In this section we define a second notion of satisfaction that complements the Hoare-style satisfaction notion (Def. 4.1.11). In section Sect. 4.1.4 we prove that together, these two notions of satisfaction imply satisfaction for the computation concern as defined in Def. 3.4.1.

A complementing notion is needed because Hoare-style satisfaction alone is not sufficient to establish satisfaction for computation. More specifically, Hoare-style satisfaction ensures that, given a closed application ca, the concrete execution of the sequence of statements $TransBottomUp(ca)$ terminates in an evaluation context that encodes the value of ca in the object referred to by `lv0`. However, it does not account for the concern of encoding a *canonical* representation, and it does not account for the concern of encoding the result as a parse tree, i.e., in a form that is suitable for display. The complementing satisfaction notion makes these concerns the responsibility of the `makeCanon` methods.

Consider Def. 4.1.12. The intuition is that $\mathit{satisfies}_{mc}$ is a notion of satisfaction of the implementation of the computation concern *cds* with regards to the client specification CS, given the algebra $\mathcal{A}$ and algebraic specification AS for which *cds* $\mathit{respects}_h$ $CS.as$. Recall that $\mathcal{A}$ defines a notion of equality that is acceptable to the client (see Sect. 2.2), and that AS defines the abstraction operators. Assume that 1) *cds* $\mathit{satisfies}_{mc}$ CS given $\langle \mathcal{A}, AS \rangle$, and 2) the execution of a statement sequence $\overline{s}$ terminates normally in an evaluation context in which the abstract value of a variable v_C (as determined using $\mathcal{A}$ and AS) is ν. Then calling the $\mathtt{makeCanon}_C$ method on v_C returns a parse tree that represents a canonical closed application *ca* that evaluates to ν in $\mathcal{A}$.

Note that there are two separate concerns that a `makeCanon` method addresses.

(1) It must take the object referred to by v_C, which is in an implementer-defined encoding (made explicit by the abstraction operator), and return an equivalent object in the encoding optimized for display (i.e. the parse tree encoding).
(2) It must ensure that the closed application that it represented by the parse tree, is canonical.

Definition 4.1.12. $\mathit{cds} \in \mathit{ClassDefSet}$ $\mathit{satisfies}_{mc}$ $CS \in \mathit{ClientSpec}$ given $\langle \mathcal{A}, AS \rangle \in \mathit{Alg} \times \mathit{AlgSpec}$ iff
for every $v_C \in \mathit{Var}$, $\mathit{ec}_0 \in \mathit{EvalContext}$, $\nu \in \mathbb{V}$, $\overline{s} \in \mathit{Seq}(\mathit{AStmt})$,
if $\langle \mathit{emptyec}, \langle\rangle, \overline{s} \rangle \overset{c}{\leadsto}^{+}_{\mathit{cds}} \langle \mathit{ec}_0, \langle\rangle, \langle\rangle \rangle$, and
$\mathit{AbstrVal}(v_C, \mathit{ec}_0, \mathcal{A}, AS, \mathit{cds}) = \nu$,
then there are $\mathit{ec}_1 \in \mathit{EvalContext}$, $\mathit{ca} \in \mathit{ClosedAppls}(CS.as.sig)$ such that
$\langle \mathit{ec}_0, \langle\rangle, \langle \mathtt{result}_{\mathit{ParseTree}} := \mathtt{makeCanon}_C(v_C) \rangle \rangle \overset{c}{\leadsto}^{+}_{\mathit{cds}}$
$\langle \mathit{ec}_1, \langle\rangle, \langle\rangle \rangle$, and
$\mathit{CARepre}(\mathtt{result}_{\mathit{ParseTree}}, \mathit{ec}_1, CS.as.sig) = \mathit{ca}$, and
$CS.\mathit{isCanonical}(\mathit{ca}, \mathcal{A}) = \mathcal{T}$, and
$\mathit{Sem}(\mathit{ca}, \mathit{emptyva}, \mathcal{A}) = \nu$.

Example 4.5. (*Satisfaction for the makeCanon method of the rationals*)

Consider the client specification CS of rationals in Exmpl. 2.1. Consider the implementation *cds* of rationals in Exmpl. 4.1. Consider the algebraic specification AS that consists only of the specification of $\mathtt{abstr}_{\mathit{Rat}}()$ in class `Rat`. Consider an arbitrary $\mathcal{A}$. We show that *cds* $\mathit{satisfies}_{mc}$ CS given $\langle \mathcal{A}, AS \rangle$.

Consider the execution of $\mathtt{result}_{\mathit{ParseTree}} := \mathtt{makeCanon}_C(v)$ from an evaluation context ec_0 in which v is mapped to a `Rat` object α of which fields `n` and `d` are n_0 and n_1. Assume $\mathit{AbstrVal}(v, \mathit{ec}_0, \mathcal{A}, AS, \mathit{cds}) = \nu$. Assume that *va* maps the fields of α to their abstract value. Then $\mathit{Sem}(\mathtt{new}_{\mathit{Rat}}(n_0, n_1) = \nu, \mathit{va}, \mathcal{A})$ (Def. 4.1.8). That is, the abstract value of v in *ec* is the value to which $\mathtt{new}_{\mathit{Rat}}(n_0, n_1)$ evaluates in va. Then $\texttt{makeCanon}_{ParseTree}(v)$ must return a parse tree that represents a canonical closed application ca that evaluates to ν given $\mathcal{A}$.

The $\texttt{makeCanon}_{ParseTree}()$ method first divides n_0 and n_1 by their greatest common divider. We refer to these updated values as n_2 and n_3. It then returns a parse tree with fields `sort`, `o`, `q`, `p` set to 0,0,$\texttt{makeCanon}_{Int}(n_2)$ and $\texttt{makeCanon}_{Int}(n_3)$, respectively. Note that the first sort in $CS.as.sig$ is `Rat`, and that the first operator in the `Rat` specification of $CS.as.sig$ is $\texttt{new}_{Rat}$. Furthermore, we have informally assumed (see the last item of the intuition of Def. 3.3.8) that $\texttt{makeCanon}_{Int}(n_2)$ and $\texttt{makeCanon}_{Int}(n_3)$ represent parse trees that evaluate to the same values as n_2 and n_3. Then ca and $\texttt{new}_{Rat}(n_2, n_3)$ evaluate to the same value. As the greatest common divider of n_2 and n_3 is 1, $\texttt{new}_{Rat}(n_2, n_3)$ is canonical. Furthermore, as the division of n_0 and n_1 by their greatest common divider does not change the evaluation of $\texttt{new}_{Rat}(n_0, n_1)$, we know that $\texttt{new}_{Rat}(n_2, n_3)$ evaluates to ν given $\mathcal{A}$. Then ca evaluates to ν.

Then *cds satisfies*$_{mc}$ *CS* given $\langle \mathcal{A}, AS \rangle$.

4.1.4 Separation Theorem

In this section, we prove that Hoare-style satisfaction and satisfaction for the `makeCanon` methods together imply satisfaction for the computation concern.

Consider Thm. 4.1. Note that it does not use *satisfies*$_h$ directly, but instead uses its intermediate notion of *respects*$_h$. This allows to connect the two notions on the algebra $\mathcal{A}$ and algebraic abstractor operator specification AS that are used. Roughly, it states that for every client specification CS, and every implementation of the computation concern *csd*, if there are a notion of equality $\mathcal{A}$ and an algebraic abstractor operator specification AS that *cds* respects and for which *cds* satisfies the `makeCanon` method concerns, then *cds* satisfies the computation concern. Recall from Thm. 3.1 that if this is the case, then $\langle Trans(), cds, DisplayFactory(CS.as.sig)\rangle$ satisfies CS.

Theorem 4.1. *For every CS* $\in$ *ClientSpec, cds* $\in$ *ClassDefSet,*
if there are $\mathcal{A} \in Models(CS.as.ax)$, $AS \in AlgSpec$ *such that*
$\langle csd, AS\rangle$ *respects*$_h$ $\mathcal{A}$ *, and*
csd satisfies$_{mc}$ *CS given* $\langle \mathcal{A}, AS\rangle$*,*
then cds satisfies$_c$ *CS.*

The proof requires an intermediate lemma for *TransBottomUp* that is presented first.

Intermediate lemma for *TransBottomUp* Consider Lemma 4.2. It formalizes the intuition behind Def. 3.3.1: *TransBottomUp* given in Sect. 3.3.1, and shows that *TransBottomUp* behaves as an abstract programmer in the sense of [Hoa72]. The lemma uses the intermediate notion *IsAgreeingOnLV*.

Consider *IsAgreeingOnLV*. Roughly, $\mathit{IsAgreeingOnLV}(n, va_0, va_1)$ expresses that two valuations va_0 and va_1 agree on a set of special purpose local variables $\mathtt{lv}m_S$, for every sort S.

Definition 4.1.13. $\mathit{IsAgreeingOnLV} : \mathbb{N} \times \mathit{Valuation} \times \mathit{Valuation} \to \mathit{Bool}$
$\mathit{IsAgreeingOnLV}(n, va_0, va_1) = \mathcal{T}$ *iff*
for every $S \in \mathit{Sort}$, $\nu \in \mathbb{V}$, $m \in \{0, \ldots, n\}$
$va_0(\mathtt{lv}m_S) = \nu$ *iff* $va_1(\mathtt{lv}m_S) = \nu$.

Lemma 4.2. *For every* $AS \in \mathit{AlgSpec}$,
for every $ca \in \mathit{ClosedAppls}(AS.sig)$, $\mathcal{A} \in \mathit{Models}(AS)$, $\nu \in \mathbb{V}$, $S \in \mathit{Sort}$, $va_0 \in \mathit{Valuation}$, $n \in \mathbb{N}$,
if $\mathit{Sem}(ca, \mathit{emptyva}, \mathcal{A}) = \nu$, *and*
ca is of sort S,
then *there is a* $va_1 \in \mathit{Valuation}$, $\overline{s} \in \mathit{Seq}(\mathit{AStmt})$ *such that*
$\mathit{TransBottomUp}(ca, n) = \overline{s}$, *and*
$\langle va_0, \overline{s}\rangle \overset{a}{\rightsquigarrow}{}^{+}_{\mathcal{A}} \langle va_1, \langle\rangle\rangle$, *and*
$va_1(\mathtt{lv}n_S) = \nu$, *and*
if $n > 0$, *then* $\mathit{IsAgreeingOnLV}(n-1, va_0, va_1) = \mathcal{T}$.

The proof outline for Lem. 4.2 is as follows.

1 Consider an arbitrary $AS \in \mathit{AlgSpec}$, $\mathcal{A} \in \mathit{Models}(AS)$, $ca \in \mathit{ClosedAppls}(AS.sig)$, $S \in \mathit{Sort}$ and $va_0 \in \mathit{Valuation}$ (premise Lem. 4.2).
2 Assume $\nu \in \mathbb{V}$ is such that $\mathit{Sem}(ca, va, \mathcal{A}) = \nu$. Assume ca is of sort S (premise Lem. 4.2).
3 Assume $f : S_1 \times \ldots \times S_i \hookrightarrow S_0 \in \mathit{Op}$, and $ca_1, \ldots, ca_i \in \mathit{ClosedAppl}$ are such that ca_0 is $f(ca_1, \ldots, ca_i)$ (Def. 2.1.17 : *ClosedAppl*).
4 Proof is by induction on the number of subapplications in ca_0.
5 Base case: 0 subapplications. Then $ca_0 = f()$.

a Then $\mathit{TransBottomUp}(ca_0, n) = \langle \mathtt{lv}n_{S_0} := f()\rangle$ (Def. 3.3.1: *TransBottomUp*).
b Then we can assume $va_1 \in \mathit{Valuation}$ such that $\langle va_0, \overline{s}\rangle \overset{a}{\rightsquigarrow}{}^{+}_{\mathcal{A}} \langle va_1, \langle\rangle\rangle$ and $va_1(\mathtt{lv}n_{S_0}) = \nu$ (Fig. 4.2 : $\overset{a}{\rightsquigarrow}$).
c Then if $n > 0$, then $\mathit{IsAgreeingOnLV}(n-1, va_0, va_1) = \mathcal{T}$ (as va_0 and va_1 differ only on $\mathtt{lv}n_{S_0}$).
d Then the conclusion of Lem. 4.2 holds, which concludes the proof of the base case.

6 Step case: number of subapplications > 0.

a the number of subapplications in every $ca_j \in \{ca_1, \ldots, ca_i\}$ is smaller than the number of subapplications in ca_0. Therefore, the induction hypothesis can be applied to every such ca_j.
b This allows to assume, for every $ca_j \in \{ca_1, \ldots, ca_i\}$, a $\overline{s}_j$ such that there is a $va_1 \in \mathit{Valuation}$ such that

1 $\mathit{TransBottomUp}(ca_j, n+j) = \overline{s}_j$, and
2 $\langle va_0, \overline{s}_j \rangle \overset{a}{\rightsquigarrow}{}^{+}_{\mathcal{A}} \langle va_1, \langle\rangle \rangle$, and
3 $va_1(\mathtt{lv}n + j_{S_1}) \ = \mathit{Sem}(ca_j, \mathit{emptyva}, \mathcal{A})$, and
4 $\mathit{IsAgreeingOnLV}(j, va_0, va_1) = \mathcal{T}$.

c For simplicity and readability, we only prove the case where $i = 2$. The generalization is straightforward.

d Informally, the induction hypothesis allows to assume that $\overline{s}_1$ computes the value of ca_1 and stores it in $\mathtt{lv}(n+1)_{S_1}$. It does not assign to any $\mathtt{lv}m$, $0 \leq m \leq n$. More formally, it allows to assume $va_1 \in \mathit{Valuation}$ such that $\langle va_0, \overline{s}_1 \rangle \overset{a}{\rightsquigarrow}{}^{+}_{\mathcal{A}} \langle va_1, \langle\rangle \rangle$, where

1 $va_1(\mathtt{lv}n + 1_{S_1}) \ = \mathit{Sem}(ca_1, \mathit{emptyva}, \mathcal{A})$, and
2 $\mathit{IsAgreeingOnLV}(n, va_0, va_1) = \mathcal{T}$.

e Informally, the induction hypothesis allows assume that $\overline{s}_2$ computes the value of ca_2 and stores it in $\mathtt{lv}n + 2_{S_2}$. It does not assign to any $\mathtt{lv}m$, $0 \leq m \leq n+1$. More formally, it allows to assume $va_2 \in \mathit{Valuation}$ such that $\langle va_1, \overline{s}_2 \rangle \overset{a}{\rightsquigarrow}{}^{+}_{\mathcal{A}} \langle va_2, \langle\rangle \rangle$, where

1 $va_2(\mathtt{lv}n + 2_{S_2}) \ = \mathit{Sem}(t_2, \mathit{emptyva}, \mathcal{A})$, and
2 $\mathit{IsAgreeingOnLV}(n+1, va_1, va_2) = \mathcal{T}$.

f As $\mathit{IsAgreeingOnLV}(n+1, va_1, va_2) = \mathcal{T}$, we know $va_2(\mathtt{lv}n + 1_{S_1}) = \mathit{Sem}(ca_1, \mathit{emptyva}, \mathcal{A})$. That is, $\mathtt{lv}n + 1_{S_1}$ stores the value of ca_1 after the execution of $\overline{s}_1 \triangleright \overline{s}_2$, because execution of $\overline{s}_1$ computes the value and stores it in $\mathtt{lv}n + 1_{S_1}$, and execution of $\overline{s}_2$ does not change the value of $\mathtt{lv}n + 1_{S_1}$.

g Then $\mathit{Sem}(f(\mathtt{lv}n{+}1_{S_1}, \mathtt{lv}n{+}2_{S_2}), va_2, \mathcal{A}) = \nu$ (as $\mathit{Sem}(f(ca_1, ca_2), va_2, \mathcal{A}) = \nu$).

h Then $\langle va_2, \langle \mathtt{lv}n_{S_0} := f(\mathtt{lv}n + 1_{S_1}, \mathtt{lv}n + 2_{S_2}) \rangle \rangle \overset{a}{\rightsquigarrow}_{\mathcal{A}} \langle va_2[\mathtt{lv}n_{S_0} \mapsto \nu], \langle\rangle \rangle$ (see Fig. 4.2 : $\overset{a}{\rightsquigarrow}$).

i Assume $\overline{s}_0 \in \mathit{Seq}(\mathit{Stmt})$ such that $\overline{s}_0 = \overline{s}_1 \triangleright \overline{s}_2 \triangleright \langle \mathtt{lv}n_{S_0} := f(\mathtt{lv}n + 1_{S_1}, \mathtt{lv}n + 2_{S_i}) \rangle$ (conclusion Lem. 4.2).

j Then $\langle va_0, \langle s_0 \rangle \rangle \overset{a}{\rightsquigarrow}_{\mathcal{A}} \langle va_2[\mathtt{lv}n_{S_0} \mapsto \nu], \langle\rangle \rangle$. That is, $\mathtt{lv}n + 1_{S_1}$ stores the value of $f(ca_1, ca_2)$ after the execution of $\overline{s}_0$.

k Also, if $n > 0$, then $\mathit{IsAgreeingOnLV}(n-1, va_0, va_2[\mathtt{lv}n_{S_0} \mapsto \nu])$ holds (as $\mathit{IsAgreeingOnLV}(n+1, va_1, va_2)$ and $\mathit{IsAgreeingOnLV}(n, va_0, va_1)$ both hold).

l That concludes the proof outline of the step case.

7 That concludes the proof outline of Lem. 4.2.

Proof outline of Thm. 4.1 The proof outline for the main theorem of this section, Thm. 4.1, is as follows.

1 Consider an arbitrary $\mathit{CS} \in \mathit{ClientSpec}$, and $\mathit{cds} \in \mathit{ClassDefSet}$ such that the premise of Thm. 4.1 holds.

2 Then we can assume $\mathcal{A} \in \mathit{Models}(\mathit{CS.as.ax})$ and $\mathit{AS} \in \mathit{AbstrOpDefs}(\mathit{cds}, \mathit{CS.as})$ such that 1) $\langle \mathit{csd}, \mathit{AS} \rangle$ $\mathit{respects}_h$ $\mathcal{A}$, and 2) csd $\mathit{satisfies}_{mc}$ CS given $\langle \mathcal{A}, \mathit{AS} \rangle$ (premise Thm. 4.1).

3 Then the goal is to prove that $\mathit{cds}\ \mathit{satisfies}_h\ \mathit{CS}$ (conclusion Thm. 4.1).
4 Then the goal is to prove that
$\mathit{SemImpl}(\langle \mathit{Trans}(), \mathit{cds}, \mathit{DisplayFactory}(\mathit{CS}.\mathit{as}.\mathit{sig})\rangle) \in \mathit{SemAS}(\mathit{CS})$ (Def. 3.4.1 : $\mathit{satisfies}_h$).
5 Consider an arbitrary $\mathit{ca}_0 \in \mathit{ClosedAppls}(\mathit{CS}.\mathit{as}.\mathit{sig})$, $\nu \in \mathbb{V}$, $S \in \mathit{Sort}$ such that $\mathit{Sem}(\mathit{ca}_0, \mathit{emptyva}, \mathcal{A}) = \nu$ and ca_0 is of sort S (premise Def. 2.2.6 : SemAS).
6 Then we can assume $\mathit{va}_0 \in \mathit{Valuation}, \overline{s} \in \mathit{Seq}(\mathit{Stmt})$ such that $\mathit{TransBottomUp}(\mathit{ca}, 0) = \overline{s}$, and $\langle \mathit{emptyva}, \overline{s}\rangle \overset{a}{\rightsquigarrow}{}^{+}_{\mathcal{A}} \langle \mathit{va}_0, \langle\rangle\rangle$, and $\mathit{va}_0(\mathtt{lv}n_S) = \nu$ (Lem. 4.2, the intermediate lemma we have just proven).
7 Then $\mathit{FieldsAssignedTo}(\overline{s}) = \{\}$ (Def. 3.3.1 : $\mathit{TransBottomUp}$ does not assign to fields).
8 Then we can assume $\mathit{ec}_0 \in \mathit{EvalContext}$ such that
1) $\langle \mathit{emptyec}, \langle\rangle, \overline{s}\rangle \overset{c}{\rightsquigarrow}{}^{+}_{\mathit{cds}} \langle \mathit{ec}, \langle\rangle, \langle\rangle\rangle$, and
2) $\mathit{AbstrView}(\mathit{ec}, \mathcal{A}, \mathit{AS}, \mathit{cds}) = \mathit{va}$ (Def. 4.1.10 : $\mathit{respects}_h$).
9 Then we can assume $\mathit{ec}_1 \in \mathit{EvalContext}$, $\mathit{ca}_1 \in \mathit{ClosedAppl}\,\mathit{CS}.\mathit{as}.\mathit{sig}$ such that
1) $\langle \mathit{ec}_0, \langle\rangle, \langle \mathtt{result}_{\mathit{ParseTree}} := \mathtt{makeCanon}_C(\mathtt{lv0}_C)\rangle\rangle \overset{c}{\rightsquigarrow}{}^{+}_{\mathit{cds}} \langle \mathit{ec}_1, \langle\rangle, \langle\rangle\rangle$, and
2) $\mathit{CARepre}(\mathtt{result}_{\mathit{ParseTree}}, \mathit{ec}_1, \mathit{CS}.\mathit{as}.\mathit{sig}) = \mathit{ca}_1$, and
3) $\mathit{CS}.\mathit{isCanonical}(\mathit{ca}_1, \mathcal{A}) = \mathcal{T}$, and
4) $\mathit{Sem}(\mathit{ca}_1, \mathit{emptyva}, \mathcal{A}) = \nu$ (Def. 4.1.12 : $\mathit{satisfies}_{mc}$).
10 Then we can assume $\mathit{display} \in \mathit{Display}$ such that 1) $\mathit{DisplayFactory}(\mathit{CS}.\mathit{as}.\mathit{sig}) = \mathit{display}$, and 2) $\mathit{display}(\mathit{ec}_1) = \mathit{ca}_1$ (Def. 3.3.9 : $\mathit{DisplayFactory}$).
11 We know $\mathit{Trans}(\mathit{ca}) = \overline{s}_0 \triangleright \langle \mathtt{result}_S := \mathtt{lv0}_S\rangle$ (Def. 3.3.4: Trans).
12 Then we can assume $\mathit{answer} \in \mathit{Answer}$ such that
1) $\mathit{SemImpl}(\langle \mathit{Trans}(), \mathit{cds}, \mathit{DisplayFactory}(\mathit{CS}.\mathit{as}.\mathit{sig})\rangle) = \mathit{answer}$, and
2) $\mathit{answer}(\mathit{ca}_0) = \mathit{ca}_1$ (Def. 3.2.4 : $\mathit{SemImpl}$).
13 That proves the earlier goal that
$\mathit{SemImpl}(\langle \mathit{Trans}(), \mathit{cds}, \mathit{DisplayFactory}(\mathit{CS}.\mathit{as}.\mathit{sig})\rangle) \in \mathit{SemAS}(\mathit{CS})$ (conclusion Def. 2.2.6 : SemAS).
14 That concludes the proof outline for Thm. 4.1.

4.2 An Implementation Approach for Hoare-Style Satisfaction

In this section, we formalize an implementation approach for Hoare-style satisfaction that allows method body implementers to be abstract programmers. In other words, it allows a method body implementer to reason about the abstract execution of the body instead of the much more complicated concrete execution.

- In Sect. 4.2.1, we extend the notion of an abstract view introduced in Sect. 4.1.2. Where the original notion is suitable only for top-level evaluation contexts, the refined notion is suitable for evaluation contexts that are part of method body executions as well. To this end, the refined notion of a *full* abstract view not only considers stack variables, but additionally maps the fields of the `this` object to their abstract value.

- In Sect. 4.2.2, we refine the notion of respect (Def. 4.1.10) that was introduced in Sect. 4.1.2 as a main part of the Hoare-style notion of satisfaction. The original notion of respect relates the pre- and post states of concrete and abstract executions of a *statement sequence*. The refined notion instead relates the pre- and post states of concrete and abstract executions of a *method call*. Essentially, the refined notion formalizes Hoare's criterion of correctness of data representations at the semantical level.

- In Sect. 4.2.3, we separate the notion of respect for method call executions, into three concerns:

  (1) Method calls must terminate.
  (2) Terminating method calls of the form $v_0 := m(v_1, v_2)$ must be result correct, i.e., must terminate normally and assign the right value to v_0.
  (3) When viewed abstractly, method calls that terminate normally must be side-effect free.

  The motivation for this separation is that these three concerns require three different proof techniques.

- In Sect. 4.2.4, we refine the notion of result correctness of a method call, to a notion of result correctness of the method body. This is a purely technical step which does not require design decisions.

- In Sect. 4.2.5, we define a notion of result correctness that considers only the abstract execution. We show that Hoare-style satisfaction is established if 1) method calls terminate and are side-effect free (when viewed abstractly), and 2) the abstract execution of method bodies is result correct. This theorem is at the heart of Sect. 4.2, as it shows under which conditions the method body implementer can be an abstract programmer.

- In Sect. 4.2.6, we show under which conditions the implementer of a method m can reason modularly, i.e., in such a way that changes to methods other than m do not affect the reasoning about m. Essentially, this represents a formalization and extension of Hoare's main proof obligation at the semantical level.

- In Sect. 4.2.7, we discuss the consequences of this implementation approach.

4.2.1 Full Abstract Views

In this section, we extend the notion of an abstract view introduced in Sect. 4.1.2. Where the original notion is suitable only for reasoning top-level evaluation contexts, the refined notion of a *full* abstract view is suitable for reasoning about evaluation contexts that are part of method body executions as well.

Consider Def. 4.2.1, which defines an intermediate notion that is used in the definition of a full abstract view. The intuition is that

$AbstrViewThis(\langle sf, os\rangle, \mathcal{A}, AS, cds)$ gives the abstract view of `this` in evaluation context $\langle sf, os\rangle$. Roughly, it maps the fields of the `this` object to their abstract value. If no `this` object is defined (as is the case in a top-level execution context), then the abstract view is the empty valuation.

Definition 4.2.1. $AbstrViewThis : EvalContext \times Alg \times AlgSpec \times ClassDefSet \hookrightarrow Valuation$
$AbstrViewThis(\langle sf, os\rangle, \mathcal{A}, AS, cds) = va$ iff
or $(Domain(sf) \cap ThisVar) = \{\}$ and $va = emptyva$,
or there are $\alpha \in \mathbb{A}$, $ec \in EvalContext$ such that
$ThisObj(sf) = \alpha$, and
$ViewOf(\alpha, os) = ec$, and
$AbstrView(ec, \mathcal{A}, AS, cds) = va$.

Consider Def. 4.2.2. The intuition is that the notion of a full abstract view can be applied to both top-level evaluation contexts and evaluations context in a method body execution. To this end, the notion combines the original notion of an abstract view with that of the abstract view of `this`. So, a full abstract view determines the abstract value of 1) all stack variables except for `this`, and 2) all fields of `this`.

Note that the domains of $AbstrView(ec, \mathcal{A}, AS, cds)$ and $AbstrViewThis(ec, \mathcal{A}, AS, cds)$ are disjoint, as the former only contains stack variables and the latter only contains fields.

Definition 4.2.2. $AbstrViewFull : EvalContext \times Alg \times AlgSpec \times ClassDefSet \hookrightarrow Valuation$
$AbstrViewFull(ec, \mathcal{A}, AS, cds) = va_0$ iff
there are $va_1, va_2 \in Valuation$ such that
$AbstrView(ec, \mathcal{A}, AS, cds) = va_1$, and
$AbstrViewThis(ec, \mathcal{A}, AS, cds) = va_2$, and
$Domain(va_0) = (Domain(va_1) - ThisVar) \cup Domain(va_2)$, and
for every $v \in Domain(va_0)$,
or $v \in Domain(va_1)$ and $va_0(v) = va_1(v)$,
or $v \in Domain(va_2)$ and $va_0(v) = va_2(v)$.

4.2.2 Divide and Conquer: Respect for Top-level Method Calls

In Sect. 4.1.2, we introduced a notion of respect (Def. 4.1.10 : $respects_h$) as a main part of the Hoare-style notion of satisfaction. This notion of respect relates the pre- and post states of concrete and abstract executions of a *statement sequence*. In this section, we refine this notion to one that relates the pre- and post states of concrete and abstract executions of a *method call*.

The refinement consist of the following steps.

(1) Consider a client specification CS, a computation implementation cds and a specification AS of the abstraction operators. We show that $\langle cds, AS\rangle$

satisfies$_h$ $CS.as$ if there is a model $\mathcal{A}$ of $CS.as$ such that $\langle cds, AS\rangle$ respects every top-level statement execution of every relevant statement sequence (where a statement sequence $\overline{s}$ is relevant to $\mathcal{A}$ only if its abstract execution terminates normally).

(2) We prove that 1) the execution of a read statement $v_0 := v_1$ is always respected, and 2) primitive operation calls are always respected as well, as long as the model $\mathcal{A}$ of $CS.as$ agrees with the implementation algebra $\mathcal{A}_{impl}$ on the interpretation of primitive operations (Lem. 4.5).

(3) We combine the previous steps to prove the refinement to relevant top-level method call executions (Thm. 4.6).

Essentially, the resulting notion of respect for top-level method call executions formalizes Hoare's criterion of correctness of data representations at the semantical level.

Refining $\textit{respects}_h$ **to respect for relevant top-level statement executions,** $\textit{respects}_{se}$**.** This first step represents a design decision that requires that individual statement executions can be reasoned about at the abstract level. While this might excludes some implementations that meet the $\textit{respects}_h$ notion (which only requires this for sequences of statements), this refined property is the basis for an implementation approach in which method bodies can be reasoned about at the level of abstract executions.

The refinement consists of the following steps.

(1) We generalize $\textit{respects}_h$ to start the concrete execution from a given evaluation context (Def. 4.2.3 : $\textit{respects}_{sse}$).

(2) We specialize the generalized notion to consider only a single statement execution (Def. 4.2.4 : $\textit{respects}_{se}$).

(3) We formalize the intermediate notions of relevant statement sequences (Def. 4.2.5) and top-level statement executions (Def. 4.2.5).

(4) We use the intermediate notions to formalize and prove the refinement to relevant top-level statement executions. More specifically, given a computation implementation *cds* and a specification AS of the abstraction operators AS, we show that if $\langle cds, AS\rangle$ respects algebra $\mathcal{A}$ for all top-level statement executions of all relevant statement sequences of $\mathcal{A}$, then $\langle cds, AS\rangle$ $\textit{respects}_h$ $\mathcal{A}$ (Lem. 4.3).

Consider Def. 4.2.3, which formalizes a notion of respect for statement sequence executions. Assume that $\langle cds, AS\rangle$ $\textit{respects}_{sse}$ $\mathcal{A}$ for $\langle ec_0, \overline{s}\rangle$. Recall from Def. 4.1.10 : $\textit{respects}_h$ that the intuition is that $\mathcal{A}$ is a model of the specification and that AS specifies the abstraction operators. Roughly, the following holds. If the abstract execution of $\overline{s}$ from the abstract view va_0 of evaluation context ec_0 terminates normally in va_1, then the concrete execution of $\overline{s}$ from ec_0 terminates normally in an evaluation context of which the abstract view is va_1.

Definition 4.2.3. $\langle cds, AS\rangle \in \textit{ClassDefSet} \times \textit{AlgSpec}$ $\textit{respects}_{sse}$ $\mathcal{A} \in \textit{Alg}$ *for* $\langle ec_0, \overline{s}\rangle \in \textit{EvalContext} \times \textit{Seq}(\textit{AStmt})$ iff

for every $va_0, va_1 \in \mathit{Valuation}$,
if $va_0 = \mathit{AbstrViewFull}(ec_0, \mathcal{A}, AS, \mathit{cds})$, and
$\langle va_0, \overline{s}\rangle \overset{a}{\rightsquigarrow}^{+}_{\mathcal{A}} \langle va_1, \langle\rangle\rangle$,
then there is an $ec_1 \in \mathit{EvalContext}$ such that
$\langle ec_0, \langle\rangle, \overline{s}\rangle \overset{c}{\rightsquigarrow}^{+}_{\mathit{cds}} \langle ec_1, \langle\rangle, \langle\rangle\rangle$, and
$va_1 = \mathit{AbstrViewFull}(ec_1, \mathcal{A}, AS, \mathit{cds})$.

Consider Def. 4.2.4. It defines a notion of respect for the execution of a single statement using the earlier definition of respect for a statement sequence execution in the obvious way.

Definition 4.2.4. $\langle \mathit{cds}, AS\rangle \in \mathit{ClassDefSet} \times \mathit{AlgSpec}$ *respects*$_{se}$ $\mathcal{A} \in \mathit{Alg}$ *for* $\langle ec_0, s\rangle \in \mathit{EvalContext} \times \mathit{AStmt}$ iff
$\langle \mathit{cds}, AS\rangle$ *respects*$_{sse}$ $\mathcal{A} \in \mathit{Alg}$ for $\langle ec_0, \langle s\rangle\rangle$

Consider Lem. 4.3. This lemma uses the intermediate notions of relevant statement sequences and top-level statement executions to relate *respects*$_{se}$ to *respects*$_h$. The intuition is that *cds* is an implementation of the computation concern, AS is a specification of the abstraction operators, and $\mathcal{A}$ is a model of the algebraic specification of the client specification. The lemma formalizes that $\langle \mathit{cds}, AS\rangle$ *respects*$_{se}$ $\mathcal{A}$ for all top-level statement executions of all relevant statement sequences of $\mathcal{A}$, then $\langle \mathit{cds}, AS\rangle$ *respects*$_h$ $\mathcal{A}$.

Consider Def. 4.2.5. The intuition is if $\mathcal{A}$ is a model of the algebraic specification of a client specification, then $\mathit{RelevantStmtSeqs}(\mathcal{A})$ returns a set of all abstract statement sequences $\overline{s}$ that are relevant to $\mathcal{A}$. Here, $\overline{s}$ is relevant to $\mathcal{A}$ only if 1) its abstract execution in the context of $\mathcal{A}$ terminates normally, and 2) it does not assign to fields. Note that this is exactly the set of abstract statement sequences for which *respects*$_h$ imposes a requirement.

Definition 4.2.5. $\mathit{RelevantStmtSeqs} : \mathit{Alg} \to \mathit{Set}(\mathit{Seq}(\mathit{AStmt}))$
$\overline{s} \in \mathit{RelevantStmtSeqs}(\mathcal{A})$ iff
there is a va such that
$\langle \mathit{emptyva}, \overline{s}\rangle \overset{a}{\rightsquigarrow}^{+}_{\mathcal{A}} \langle va, \langle\rangle\rangle$, and
$\mathit{FieldsAssignedTo}(\overline{s}) = \{\}$.

Consider Def. 4.2.6. Roughly, a $\langle ec, s\rangle$ is a top-level statement execution in the execution of a statement sequence $\overline{s}$ if s is executed from ec during the top level of the execution of $\overline{s}$ (where an execution state is top-level if it is not part of a method call execution).

Note that the lemma allows to establish Hoare-style satisfaction through *respects*$_{se}$, rather than through *respects*$_h$. Recall from Def. 3.4.1 that *cds satisfies*$_h$ an algebraic specification if it has a model $\mathcal{A}$ such that there is a specification AS of the abstraction operators such that $\langle \mathit{cds}, AS\rangle$ *respects*$_h$ $\mathcal{A}$.

Definition 4.2.6. $\mathit{TopLevelStmtExecs} : \mathit{Seq}(\mathit{AStmt}) \times \mathit{EvalContext} \times \mathit{ClassDefSet} \to \mathit{Set}(\mathit{EvalContext} \times \mathit{AStmt})$

$\langle ec_1, s\rangle \in \mathit{TopLevelStmtExecs}(\overline{s_0}, ec_0, cds)$ iff
  there is a $\overline{s}_1 \in \mathit{Seq}(\mathit{AStmt})$ such that
    $\langle ec_0, \langle\rangle, \overline{s}_0\rangle \overset{c}{\rightsquigarrow}{}^*_{cds} \langle ec_1, \langle\rangle, \langle s\rangle \triangleright \overline{s}_1\rangle$.

Lemma 4.3. *For every* $cds \in \mathit{ClassDefSet}$, $\mathcal{A} \in \mathit{Alg}$, $AS \in \mathit{AlgSpec}$,
if *for every* $\overline{s} \in \mathit{RelevantStmtSeqs}(\mathcal{A})$, $va \in \mathit{Valuation}$,
  for every $\langle ec, s\rangle \in \mathit{TopLevelStmtExecs}(\overline{s}, \mathit{emptyec}, cds)$,
    cds *respects*$_{se}$ $\langle\mathcal{A}, AS\rangle$ *for* $\langle ec, s\rangle$,
then cds *respects*$_h$ $\langle\mathcal{A}, AS\rangle$.

The proof of Lem. 4.3 depends on the following intermediate lemma.

Lemma 4.4. *For every* $cds \in \mathit{ClassDefSet}$, $\mathcal{A} \in \mathit{Alg}$, $AS \in \mathit{AlgSpec}$, $\overline{s} \in \mathit{Seq}(\mathit{AStmt})$, $ec_0 \in \mathit{EvalContext}$,
if *for every* $\langle ec_1, s\rangle \in \mathit{TopLevelStmtExecs}(\overline{s}, ec_0, cds)$,
  cds *respects*$_{se}$ $\langle\mathcal{A}, AS\rangle$ *for* $\langle ec_1, s\rangle$,
then cds *respects*$_{sse}$ $\langle\mathcal{A}, AS\rangle$ *for* $\langle ec_0, \overline{s}\rangle$.

The proof outline of Lem. 4.4 is as follows.

1 Consider arbitrary $cds \in \mathit{ClassDefSet}$, $\mathcal{A} \in \mathit{Alg}$, $AS \in \mathit{AlgSpec}$, $\overline{s} \in \mathit{Seq}(\mathit{AStmt})$, $ec_0 \in \mathit{EvalContext}$ such that the premise of Lem. 4.4 holds.
2 Assume $va_0, va_1 \in \mathit{Valuation}$ such that
$\mathit{AbstrViewFull}(ec_0, \mathcal{A}, AS, cds) = va_0$, and
$\langle va_0, \overline{s}_0\rangle \overset{a}{\rightsquigarrow}{}^*_{\mathcal{A}} \langle va_1, \langle\rangle\rangle$ (premise Def. 4.2.3 : *respects*$_{sse}$).
3 Then the goal is to prove that there is an $ec_1 \in \mathit{EvalContext}$ such that
$\langle ec_0, \langle\rangle, \overline{s}_0\rangle \overset{c}{\rightsquigarrow}{}^* \langle ec_1, \langle\rangle, \langle\rangle\rangle$, and
$\mathit{AbstrViewFull}(ec_1, \mathcal{A}, AS, cds) = va_1$ (conclusion Def. 4.2.3 : *respects*$_{sse}$). Proof is by induction on the length of $\overline{s}_0$.
4 Base case: Assume $|\,\overline{s}_0\,| = 0$. Then $\overline{s}_0 = \langle\rangle$. Then 1) $va_1 = va_0$, and 2) the concrete execution of $\overline{s}_0$ from ec_0 terminates normally in ec_0. That concludes the proof of the base case.
5 Step case: Assume $|\,\overline{s}_0\,| > 0$.

  a Then we can assume $s \in \mathit{AStmt}, \overline{s}_1 \in \mathit{Seq}(\mathit{AStmt})$ such that $\overline{s}_0 = \langle s\rangle \triangleright \overline{s}_1$.
  b Then $\langle ec_0, s\rangle \in \mathit{TopLevelStmtExecs}(\overline{s}_0, \mathit{emptyec}, cds)$ (Def. 4.2.6 : *TopLevelStmtExecs*).
  c Then $\langle cds, AS\rangle$ *respects*$_{se}$ $\mathcal{A}$ for $\langle ec_0, s\rangle$ (premise Lem. 4.3).
  d Then we can assume $va_2 \in \mathit{Valuation}$, $ec_2 \in \mathit{EvalContext}$ such that
  va_2 is the abstract view of ec_2, and
  the abstract execution of s from va_0 terminates normally in va_2, and
  the concrete execution of s from va_0 terminates normally in ec_2 (Def. 4.2.4 : *respects*$_{se}$, as we know that the abstract execution of $\overline{s}_0$ terminates normally).
  e We know that the abstract execution of $\overline{s}_1$ from va_2 terminates normally in va_1 (as the abstract execution of $\overline{s}_0$ terminates normally in va_0).
  f As $|\,\overline{s}_1\,| < |\,\overline{s}_0\,|$, we can use the induction hypothesis. Then we can assume $ec_1 \in \mathit{EvalContext}$ such that

the concrete execution of $\overline{s}_1$ from ec_2 terminates normally in ec_1, and va_1 is the abstract view of ec_1.

g Then the concrete execution of $\overline{s}_0$ from ec_0 terminates normally in ec_1.

h That concludes the proof of the step case.

That concludes the proof of Lem. 4.4.

The proof outline of Lem. 4.3 is as follows.

1 Consider an arbitrary $cds \in ClassDefSet$, $\mathcal{A} \in Alg$, $AS \in AlgSpec$ such that the premise of Lem. 4.3 holds.

2 Assume $\overline{s}_0 \in Seq(AStmt)$, $va \in Valuation$ such that $FieldsAssignedTo(\overline{s}) = \{\}$, and $\langle emptyva, \overline{s}_0\rangle \overset{a}{\rightsquigarrow}{}^{+}_{\mathcal{A}} \langle va, \langle\rangle\rangle$ (premise Def. 4.1.10 : $respects_h$).

3 Then $\overline{s}_0 \in RelevantStmtSeqs(\mathcal{A})$ (Def. 4.2.6: *RelevantStmtSeqs*).

4 Then $cds\ respects_{sse}\ \langle \mathcal{A}, AS\rangle$ for $\langle emptyec, \overline{s}_0\rangle$ (Lem. 4.4).

5 Then we can assume (conclusion Def. 4.2.3 : $respects_{sse}$) $ec \in EvalContext$ such that
$\langle emptyec, \langle\rangle, \overline{s}_0\rangle \overset{c}{\rightsquigarrow}{}^{*} \langle ec, \langle\rangle, \langle\rangle\rangle$, and
$AbstrViewFull(ec, \mathcal{A}, AS, cds) = va$.

6 We know *va* does not does not define any fields (as *emptyva* does not define any fields and as $FieldsAssignedTo(\overline{s}) = \{\}$). Furthermore, *va* does not define a `this` variable (as `this` is not defined in *emptyva* and as `this` does not occur in abstract statements).

7 Then $AbstrViewFull(ec, \mathcal{A}, AS, cds) = AbstrView(ec, \mathcal{A}, AS, cds)$ (Def. 4.1.9 : *AbstrView*, Def. 4.2.2 : *AbstrViewFull*).

8 Then $AbstrView(ec, \mathcal{A}, AS, cds) = va$ (as $AbstrViewFull(ec, \mathcal{A}, AS, cds) = va$).

9 Then the conclusion of Def. 4.1.10 : $respects_h$ holds.

That concludes the proof of Lem. 4.3.

Respecting primitive operation calls and read statement executions. In this second step, we show that a minor restriction on the model $\mathcal{A}$ of the algebraic specification in the client specification suffices to establish that for any computation implementation *cds* and any specification *AS* of the abstraction operators, $\langle cds, AS\rangle\ respects_{se}\ \mathcal{A}$ for any primitive operation call and every read statement execution.

Consider Def. 4.2.7. Roughly, $IsFieldsAssignedToDefined(\overline{s}, ec) = \mathcal{T}$ only if every field that is assigned to in $\overline{s}$, is defined as a field of the object referred to by `this` in the prestate. Note that if this property holds, then a concrete execution does not terminate abnormally because of an assignment to an undefined field (as `this` is not assigned to by a sequence of abstract statements and fields cannot be 'undefined').

Definition 4.2.7. $IsFieldsAssignedToDefined : Seq(AStmt) \times EvalContext \rightarrow Bool$
$IsFieldsAssignedToDefined(\overline{s}, ec) = \mathcal{T}$ iff

for every $f \in \mathit{FieldsAssignedTo}(\overline{s})$,
there is an $\alpha \in \mathcal{A}$ such that
$\mathit{ThisObj}(ec) = \alpha$, and
$\langle \alpha, f \rangle \in \mathit{Domain}(ec.os)$.

Consider Lem. 4.5, which is defined using the intermediate notion of respectable algebras. Roughly, it states that given a respectable algebra $\mathcal{A}$, any pair $\langle \mathit{cds}, \mathit{AS} \rangle$ of a computation implementation and a specification of the abstraction operators $\mathit{respects}_{se}$ $\mathcal{A}$ for any primitive operation call or read statement execution (as long as it does not assign to a field that does not exist).

Consider Def. 4.2.8. Roughly, a respectable algebra agrees with the implementation algebra (Def. 3.1.32) on the interpretation of primitive operations.

Definition 4.2.8. $\mathcal{A} \in \mathit{Alg}$ is *respectable* iff
for every $\mathit{op} \in \mathit{PrimSig.ops} \cap \mathit{Domain}(\mathcal{A}.\mathit{interpretation})$,
$\mathcal{A}.\mathit{interpretation}(\mathit{op}) = \mathcal{A}_{\mathit{impl}}.\mathit{interpretation}(\mathit{op})$.
RespectableAlg is the set of all respectable algebras.

Lemma 4.5. *For every $\mathcal{A} \in \mathit{RespectableAlg}$, $ec \in \mathit{EvalContext}$, $v \in \mathit{AVar}$, $e \in \mathit{Expr} \cap \mathit{AExpr}$,*
if $\mathit{IsFieldsAssignedToDefined}(\langle v := e \rangle, ec) = \mathcal{T}$,
then *for every $\mathit{cds} \in \mathit{ClassDefSet}$, $\mathit{AS} \in \mathit{AlgSpec}$,*
$\langle \mathit{cds}, \mathit{AS} \rangle$ *$\mathit{respects}_{se}$ $\mathcal{A}$ for $\langle ec, v := e \rangle$.*

Aside. In practice, the restriction to respectable algebras should not be much of a burden. The specifier of the client specification CS can expect the specifier of the programming language to provide an algebraic specification $\mathit{AS}_{\mathit{impl}}$ of the primitive signature (Def. 3.1.2) such that $\mathcal{A}_{\mathit{impl}}$ is a model of $\mathit{AS}_{\mathit{impl}}$. Every model of $\mathit{CS.as}$ is a respectable if the specifier of $\mathit{CS.as}$ includes $\mathit{AS}_{\mathit{impl}}$ in $\mathit{CS.as}$ and does not define any axioms that restrict the primitive operations.Note that this restriction is implicit in Hoare's work.

The proof outline of Lem. 4.5 is as follows.

1 Consider $\mathcal{A} \in \mathit{Alg}$, $\mathit{cds} \in \mathit{ClassDefSet}$, $\langle \mathit{sf}, \mathit{os} \rangle \in \mathit{EvalContext}$, $v \in \mathit{AVar}$, $e \in \mathit{Expr}$ such that the premise of Lem. 4.5 holds.
2 Assume $\mathit{va}_0, \mathit{va}_1 \in \mathit{Valuation}$ such that that
$\mathit{AbstrViewFull}(\langle \mathit{sf}, \mathit{os} \rangle, \mathcal{A}, \mathit{AS}, \mathit{cds}) = \mathit{va}_0$, and
$\langle \mathit{va}_0, \langle v := e \rangle \rangle \overset{a}{\rightsquigarrow}{}^{+}_{\mathcal{A}} \langle \mathit{va}_1, \langle \rangle \rangle$ (premise Def. 4.2.4: $\mathit{respects}_{se}$).
3 Then we can assume $\nu \in \mathbb{V}$ such that
$\mathit{Sem}(e, \mathit{va}_0, \mathcal{A}) = \nu$, and
$\mathit{va}_1 = \mathit{va}_0[v \mapsto \nu]$ (Fig. 4.2: $\overset{a}{\rightsquigarrow}$).
4 We know e is either a formal parameter, a local variable, a primitive operation call, or a field (as $e \in \mathit{Expr} \cap \mathit{AExpr}$).
5 If e is a primitive operation call, then $\mathit{Eval}(e, \langle \mathit{sf}, \mathit{os} \rangle) = \nu$, as

a Then $\mathcal{A}$ and $\mathcal{A}_{\mathit{impl}}$ agree on the primitive operator interpretation (Def. 4.2.8 : *RespectableAlg*).

b Furthermore, as all domain sorts of a primitive operator are primitive sorts and as as $\mathit{Sem}(e, \mathit{va}_0, \mathcal{A}) = \nu$, va_0 and sf agree on all actual parameters of the call.

c Then $\mathit{Eval}(e, \langle \mathit{sf}, \mathit{os}\rangle) = \nu$ (Def. 3.1.47 : *Eval*).

6 If e is a formal parameter or a local variable, then $\mathit{AbstrView}(\langle \mathit{sf}, \mathit{os}\rangle)$ defines e and $\mathit{Eval}(e, \langle \mathit{sf}, \mathit{os}\rangle) = \mathit{sf}(e)$, as

a Then $\mathit{AbstrView}(\langle \mathit{sf}, \mathit{os}\rangle)$ defines e (Def. 4.2.2 : *AbstrViewFull*, as $\mathit{va}_0(e)$ is defined and as $\mathit{AbstrViewThis}()$ only defines fields).

b Then $\mathit{sf}(e)$ is defined (Def. 4.1.9 : *AbstrView*).

c Then $\mathit{Eval}(e, \langle \mathit{sf}, \mathit{os}\rangle)$ is $\mathit{sf}(e)$ (Def. 3.1.47 : *Eval*).

7 If e is a field, then $\mathit{Eval}(e, \langle \mathit{sf}, \mathit{os}\rangle) = \mathit{os}(\langle \mathit{ThisObj}(\langle \mathit{sf}, \mathit{os}\rangle), e\rangle)$, due to the following.

a Then e must be defined by $\mathit{AbstrViewThis}(\langle \mathit{sf}, \mathit{os}\rangle, \mathcal{A}, \mathit{AS}, \mathit{cds})$ (as va_0 defines e and as *AbstrView* does not define fields).

b Then e is on the stack frame of $\mathit{ViewOf}(\mathit{ThisObj}(\langle \mathit{sf}, \mathit{os}\rangle), \mathit{os})$ (Def. 4.2.1 : *AbstrViewThis*).

c Then $\mathit{os}(\langle \mathit{ThisObj}(\langle \mathit{sf}, \mathit{os}\rangle), e\rangle)$ is defined (Def. 3.3.5 : *ViewOf*).

d Then $\mathit{Eval}(e, \langle \mathit{sf}, \mathit{os}\rangle)$ is $\mathit{os}(\langle \mathit{ThisObj}(\langle \mathit{sf}, \mathit{os}\rangle), e\rangle)$ (Def. 3.1.47 : *Eval*).

8 Then if e is a formal parameter, a local variable or a field, and e is of a primitive sort, then $\mathit{Eval}(e, \langle \mathit{sf}, \mathit{os}\rangle) = \nu$ (Def. 4.1.9 : *AbstrView*, as the abstract value of a primitive variable is its stack frame value).

9 We can distinguish the following cases.

10 Case: $v \notin \mathit{Field}$ and e is a formal parameter, a local variable, a field or a primitive operation call, and e is of a primitive sort.

a Then $\langle\langle \mathit{sf}, \mathit{os}\rangle, \langle\rangle, \langle v := e\rangle\rangle \rightsquigarrow_{\mathit{cds}} \langle\langle \mathit{sf}[v \mapsto \nu], \mathit{os}\rangle, \langle\rangle, \langle\rangle\rangle$ (the `READ` rule from Def. 3.1.51 : $\rightsquigarrow$, as we have shown that $\mathit{Eval}(e, \langle \mathit{sf}, \mathit{os}\rangle) = \nu$ in this case).

b Then $\mathit{AbstrViewFull}(\langle \mathit{sf}[v \mapsto \nu], \mathit{os}\rangle, \mathcal{A}, \mathit{AS}, \mathit{cds}) = \mathit{va}_1$ (Def. 4.2.2 : *AbstrViewFull*, as $\mathit{va}_1 = \mathit{va}_0[v \mapsto \nu]$).

11 Case: $v \notin \mathit{Field}$ and e is a formal parameter, a local variable or a field, and e is of a class sort.

a Then we can assume $\alpha \in \mathcal{A}$ such that $\langle\langle \mathit{sf}, \mathit{os}\rangle, \langle\rangle, \langle v := e\rangle\rangle \rightsquigarrow_{\mathit{cds}} \langle\langle \mathit{sf}[v \mapsto \alpha], \mathit{os}\rangle, \langle\rangle, \langle\rangle\rangle$ (the `READ` rule from Def. 3.1.51 : $\rightsquigarrow$, as we have shown that $\mathit{Eval}(e, \langle \mathit{sf}, \mathit{os}\rangle)$ is defined in this case).

b By definition, $\mathit{AbstrViewFull}(\langle \mathit{sf}[v \mapsto \nu], \mathit{os}\rangle, \mathcal{A}, \mathit{AS}, \mathit{cds})$ can only differ from va_0 on v (Def. 4.2.2 : *AbstrViewFull*).

c We know $\mathit{AbstrViewFull}(\langle \mathit{sf}[v \mapsto \nu], \mathit{os}\rangle, \mathcal{A}, \mathit{AS}, \mathit{cds})$ maps v to ν, as va_0 maps e to ν and the object store has not changed.

d Then $\mathit{AbstrViewFull}(\langle \mathit{sf}[v \mapsto \nu], \mathit{os}\rangle, \mathcal{A}, \mathit{AS}, \mathit{cds}) = \mathit{va}_1$ (as $\mathit{va}_1 = \mathit{va}_0[v \mapsto \nu]$).

12 Case: $v \in \mathit{Field}$.

a Then we can assume $\alpha \in \mathcal{A}$ such that

$\mathit{ThisObj}(\langle \mathit{sf}, \mathit{os}\rangle) = \alpha$, and
$\langle \alpha, v_0\rangle \in \mathit{Domain}(\mathit{os})$ (Def. 4.2.7 : *IsFieldsAssignedToDefined*).

b Then the proof follows the two cases above, except that it uses the `WRITE` rule from Def. 3.1.51 : $\overset{c}{\rightsquigarrow}$, and the fact that *AbstrViewFull* maps the fields of this to their abstract value, using *AbstrViewThis*.

13 Then in every case, $\mathit{AbstrViewFull}(\langle \mathit{sf}[v \mapsto \nu], \mathit{os}\rangle, \mathcal{A}, \mathit{AS}, \mathit{cds}) = \mathit{va}_1$.
14 Then $\langle \mathit{cds}, \mathit{AS}\rangle$ $\mathit{respects}_{se}$ $\mathcal{A}$ for $\langle\langle \mathit{sf}, \mathit{os}\rangle, s\rangle$ (Def. 4.2.4 : $\mathit{respects}_{se}$).
15 Then the conclusion of Lem. 4.5 holds.

That concludes the proof outline of Lem. 4.5.

Respect for relevant top-level method calls implies Hoare-style satisfaction. In this third and last step we combine the previous two steps to produce the main result of Sect. 4.2.2, which is Thm. 4.6. Note that the proof of the theorem uses a lemma (Lem. 4.4) with a premise that essentially formalizes Hoare's criterion of correctness of data representations at the semantical level.

Consider Def. 4.2.9. Recall that $\mathit{TopLevelStmtExecs}(\overline{s}, \mathit{ec}, \mathit{cds})$ yields the set of top-level statement executions in the execution of $\overline{s}$ from ec. $\mathit{TopLevelMCExecs}(\overline{s}, \mathit{ec}, \mathit{cds})$ simply restricts these top-level statement executions to method call executions (see Def. 4.1.3 : *AMCStmt*).

Definition 4.2.9. $\mathit{TopLevelMCExecs} : \mathit{Seq}(\mathit{AStmt}) \times \mathit{EvalContext} \times \mathit{ClassDefSet} \to \mathit{Set}(\mathit{EvalContext} \times \mathit{AStmt})$
$\langle \mathit{ec}, s\rangle \in \mathit{TopLevelMCExecs}(\overline{s}, \mathit{ec}, \mathit{cds})$ iff
 $\langle \mathit{ec}, s\rangle \in \mathit{TopLevelStmtExecs}(\overline{s}, \mathit{ec}, \mathit{cds})$, and
 $s \in \mathit{AMCStmt}$.

Consider Def. 4.2.10. For convenience, it combines the notions of a model of a specification and a respectable algebra, into a notion of a respectable model of a specification.

Definition 4.2.10. $\mathit{RespectableModels} : \mathit{AlgSpec} \to \mathit{Set}(\mathit{Alg})$
$\mathcal{A} \in \mathit{RespectableModels}(\mathit{AS})$ iff
 $\mathcal{A} \in \mathit{Models}(\mathit{AS}.\mathit{ax}) \cap \mathit{RespectableAlg}$.

Consider Thm. 4.6. a computation implementation cds $\mathit{satisfies}_h$ the algebraic specification AS of a client specification if there are a respectable model $\mathcal{A}$ of AS_0 and a specification AS_1 of the abstraction operators such that $\langle \mathit{cds}, \mathit{AS}_1\rangle$ $\mathit{respects}_{se}$ every top-level method call of every relevant statement sequence.

Theorem 4.6. *For every* $\mathit{cds} \in \mathit{ClassDefSet}$, $\mathit{AS}_0 \in \mathit{AlgSpec}$,
if *there are* $\mathcal{A} \in \mathit{RespectableModels}(\mathit{AS}_0)$, $\mathit{AS}_1 \in \mathit{AbstrOpDefs}(\mathit{cds}, \mathit{AS}_0)$ *such that*
 for every $\overline{s} \in \mathit{RelevantStmtSeqs}(\mathcal{A})$, $\mathit{va} \in \mathit{Valuation}$,
  for every $\langle \mathit{ec}, s\rangle \in \mathit{TopLevelMCExecs}(\overline{s}, \mathit{emptyec}, \mathit{cds})$,
   $\langle \mathit{cds}, \mathit{AS}_1\rangle$ $\mathit{respects}_{se}$ $\mathcal{A}$ *for* $\langle \mathit{ec}, s\rangle$,
then cds $\mathit{satisfies}_h$ AS_0.

The proof of Thm. 4.6 depends the following intermediate lemma (which is reused when we reason about the method body).

Lemma 4.7. *For every $cds \in ClassDefSet$, $\mathcal{A} \in RespectableAlg$, $AS \in AlgSpec$, $\overline{s} \in Seq(AStmt)$, $ec_0 \in EvalContext$,*
if $IsFieldsAssignedToDefined(ec_0, \overline{s}) = \mathcal{T}$, *and*
for every $\langle ec_1, s\rangle \in TopLevelMCExecs(\overline{s}, ec_0, cds)$,
cds *respects*$_{se}$ $\langle \mathcal{A}, AS\rangle$ *for* $\langle ec_1, s\rangle$,
then cds *respects*$_{sse}$ $\langle \mathcal{A}, AS\rangle$ *for* $\langle ec_0, \overline{s}\rangle$.

Aside. Essentially, the premise of this lemma formalizes Hoare's criterion of correctness of data representations at the semantical level, and its conclusion formalizes the consequence of the criterion that abstract executions can be 'validly replaced' by concrete executions.

The proof outline of Lem. 4.7 follows that of Lem. 4.4. We sketch the proof, highlighting only the differences. First, we assume that the premise of the lemma holds for a $\overline{s}_0$. Then we assume that the abstract execution of $\overline{s}_0$ terminates normally in va_1 (premise Def. 4.2.3 : *respects*$_{sse}$). Then we prove, by induction on the length of $\overline{s}_0$, that the concrete execution terminates normally in an evaluation ec_1 of which va_1 is the abstract view. The base case is trivial. In the step case, we assume that $s_0 = s \triangleright s_1$. We want to conclude that $\langle cds, abstrViewEC\rangle$ *respects*$_{se}$ $\mathcal{A}$ for $\langle ec_0, s\rangle$. Unlike in step 5c of the proof of Lem. 4.4, this does not follow directly from the premise. Instead, it follows from the following case distinction on s.

1 Case: $s \in AMCStmt$ (i.e., s is a method call statement).
Then $\langle ec_0, s\rangle \in TopLevelMCExecs(\overline{s}_0, emptyec, cds)$ (Def. 4.2.9 : *TopLevelMCExecs*). Then it follows from the premise of Thm. 4.6 that $\langle cds, AS_1\rangle$ *respects*$_{se}$ $\mathcal{A}$ for $\langle ec_0, s\rangle$.

2 Case: there are $v \in Field$, $e \in Expr \cap AExpr$, such that s is $v := e$ (i.e., s is the assignment of a field, variable or primitive operation to a field or variable).

  a Then $IsFieldsAssignedToDefined(ec_0, \overline{s}) = \mathcal{T}$ (as $IsFieldsAssignedToDefined(ec_0, \overline{s}) = \mathcal{T}$, Def. 4.2.7 : *IsFieldsAssignedToDefined*).

  b Then $\langle cds, AS_1\rangle$ *respects*$_{se}$ $\mathcal{A}$ for $\langle ec_0, s\rangle$ (Lem. 4.5).

3 Case: assume $v \in AVar, \overline{s}_4 \in Seq(AStmt)$ such that s is `if` (v) $\overline{s}_4$. Then it follows from Def. 4.1.9 : *AbstrView* that $va_0(v) = ec.sf(v)$, as va_0 is the abstract view of ec_0, and v is of primitive sort *Bool*. Then two cases can be distinguished:

  a Case: $va_0(v) = \mathcal{F}$. Then the abstract execution of s from va_0 terminates normally in va_0, and the concrete execution of s from ec_0 terminates normally in ec_0 (Fig. 4.2 : $\overset{a}{\rightsquigarrow}$ and Def. 3.1.51 : $\overset{c}{\rightsquigarrow}$). Then $\langle cds, AS_1\rangle$ *respects*$_{se}$ $\mathcal{A}$ for $\langle ec_0, s\rangle$, as va_0 is the abstract view of ec_0.

  b Case: $va_0(v) = \mathcal{T}$. Then
  1) $\langle va_0, s\rangle \overset{a}{\rightsquigarrow}_{\mathcal{A}} \langle va_0, \overline{s}_4\rangle$ (Fig. 4.2 : $\overset{a}{\rightsquigarrow}$), and

2) $\langle ec_0, s\rangle \rightsquigarrow_{cds} \langle ec_0, \overline{s}_4\rangle$ (Def. 3.1.51 : $\rightsquigarrow$). As $|\overline{s}_4| < |\overline{s}_0|$, it follows from the induction hypothesis that $\langle cds, AS_1\rangle$ *respects*$_{se}$ $\mathcal{A}$ for $\langle ec, \overline{s}_4\rangle$. Then it is straightforward to conclude that $\langle cds, AS_1\rangle$ *respects*$_{se}$ $\mathcal{A}$ for $\langle ec_0, s\rangle$.

Then the concrete execution of s from ec_0 terminates normally in an evaluation context ec_2. We want to use the induction hypothesis for the remaining execution of $\overline{s}_1$ from ec_2 (step 5f in the proof of Lem. 4.4). This additionally requires us to prove that $\mathit{IsFieldsAssignedToDefined}(ec_1, \overline{s}_1) = \mathcal{T}$. This holds, as s does not assign to `this` (no abstract statement does), and as fields cannot be 'undefined'. That concludes the proof outline of Lem. 4.7.

The proof outline of Thm. 4.6 follows that of Lem. 4.3, but uses Lem. 4.7 instead of Lem. 4.4.

4.2.3 Separation of Concerns: Termination, Result Correctness and Side-effect Freeness

Consider a client specification CS, a computation implementation cds and a specification AS of the abstraction operators. In Sect. 4.2.2, we have shown that $\langle cds, AS\rangle$ *satisfies*$_c$ $CS.as$ if there is a respectable model $\mathcal{A}$ of $CS.as$ such that $\langle cds, AS\rangle$ *respects*$_{se}$ $\mathcal{A}$ for every top-level method call execution of every relevant statement sequence.

Here we separate that notion of respect for method call executions, into three concerns:

(1) Method calls must terminate.
(2) Terminating method calls of the form $v_0 := m(v_1, v_2)$ must be result correct, i.e., must terminate normally and assign the right value to v_0.
(3) When viewed abstractly, method calls that terminate normally must be side-effect free.

The motivation for this separation is that only result correctness can be reasoned about at the level of the abstract execution of the method body (as a consequence, these three concerns require three different proof techniques).

To separate into the three concerns, we take the following steps.

(1) We briefly revisit the notion of termination.
(2) We define a notion of correctness of the result of a method call execution (Def. 4.2.11).
(3) We define a notion of side-effect freeness (Def. 4.2.12).
(4) We show that *respects*$_{se}$ can be separated into the three defined concerns (Lem. 4.8).
(5) We show that $\langle cds, AS\rangle$ *satisfies*$_c$ $CS.as$ if there is a respectable model $\mathcal{A}$ of $CS.as$ such that the three concerns are met for every top-level method call of relevant every statement sequence (Thm. 4.9).

Concern (1): termination. The first concern is captured by the existing notion of termination of an execution. That is, execution of method call $v_0 := m(v_1, v_2)$ from evaluation context ec terminates if $\mathit{IsExecTerminating}(\langle ec, \langle\rangle, \langle v_0 := m(v_1, v_2)\rangle\rangle, \mathit{cds}) = \mathcal{T}$.

Concern (2): result correctness. The second concern is captured by the notion of result correctness of the result of a method call execution.

Consider Def. 4.2.11. Roughly, $\mathit{IsResultCorrect}(ec_0, \mathit{mcs}, \mathcal{A}, AS, \mathit{cds}) = \mathcal{T}$ only when the following holds. If 1) the abstract execution of method call statement $v_0 := m(v_1, v_2)$ from the abstract view va_0 of evaluation context ec_0 terminates normally in va_1, and 2) the concrete execution of $v_0 := m(v_1, v_2)$ from ec_0 terminates, then 1) the concrete execution terminates normally in an evaluation context with abstract view va_2, and 2) va_2 and va_1 map v_0 to the same value.

Definition 4.2.11. *IsResultCorrect* : $\mathit{EvalContext} \times \mathit{AMCStmt} \times \mathit{Alg} \times \mathit{AlgSpec} \times \mathit{ClassDefSet} \to \mathit{Bool}$
$\mathit{IsResultCorrect}(ec_0, v_0 := m(v_1, v_2), \mathcal{A}, AS, \mathit{cds}) = \mathcal{T}$ *iff*
for every $va_0, va_1 \in \mathit{Valuation}$,
if $\mathit{IsExecTerminating}(\langle ec_0, \langle\rangle, \langle v_0 := m(v_1, v_2)\rangle\rangle, \mathit{cds}) = \mathcal{T}$, *and*
$\mathit{AbstrView}(ec_0, \mathcal{A}, AS, \mathit{cds}) = va_0$, *and*
$\langle va_0, \langle v_0 := m(v_1, v_2)\rangle\rangle \overset{a}{\rightsquigarrow}_{\mathcal{A}} \langle va_1, \langle\rangle\rangle$,
then *there are* $ec_1 \in \mathit{EvalContext}, va_2 \in \mathit{Valuation}$ *such that*
$\langle ec_0, \langle\rangle, \langle v_0 := m(v_1, v_2)\rangle\rangle \overset{c}{\rightsquigarrow}^{+}_{\mathit{cds}} \langle ec_1, \langle\rangle, \langle\rangle\rangle$, *and*
$\mathit{AbstrView}(ec_1, \mathcal{A}, AS, \mathit{cds}) = va_2$, *and*
$va_2(v_0) = va_1(v_0)$.

Concern (3): side-effect freeness. The third concern is captured by the notion of side-effect freeness of a method call execution.

Consider Def. 4.2.12. Roughly, $\mathit{IsSideEffectFree}(ec_0, v_0 := f(v_1, v_2), \mathcal{A}, AS, \mathit{cds}) = \mathcal{T}$ only when the following holds. If 1) the abstract execution of method call statement $v_0 := f(v_1, v_2)$ from the abstract view va_0 of evaluation context ec_0 terminates normally, and 2) the concrete execution of $v_0 := f(v_1, v_2)$ from ec_0 terminates normally in an evaluation context with abstract view va_2, then va_0 and va_2 differ only on v_0.

Definition 4.2.12. *IsSideEffectFree* : $\mathit{EvalContext} \times \mathit{AMCStmt} \times \mathit{Alg} \times \mathit{AlgSpec} \times \mathit{ClassDefSet} \to \mathit{Bool}$
$\mathit{IsSideEffectFree}(ec_0, v_0 := m(v_1, v_2), \mathcal{A}, AS, \mathit{cds}) = \mathcal{T}$ iff
for every $va_0, va_1 \in \mathit{Valuation}, ec_1 \in \mathit{EvalContext}$,
if $\mathit{AbstrView}(ec_0, \mathcal{A}, AS, \mathit{cds}) = va_0$, and
$\langle va_0, \langle v_0 := m(v_1, v_2)\rangle\rangle \overset{a}{\rightsquigarrow}_{\mathcal{A}} \langle va_1, \langle\rangle\rangle$, and
$\langle ec_0, \langle\rangle, \langle v_0 := m(v_1, v_2)\rangle\rangle \overset{c}{\rightsquigarrow}^{+}_{\mathit{cds}} \langle ec_1, \langle\rangle, \langle\rangle\rangle$,
then there is a $va_2 \in \mathit{Valuation}$ such that
$\mathit{AbstrView}(ec_1, \mathcal{A}, AS, \mathit{cds}) = va_2$, and
$va_2 = va_0[v_0 \mapsto va_2(v_0)]$.

Separating respect for method calls into the three concerns. What remains is to show that Hoare-style satisfaction is established if the three concerns are met for all top-level method call executions of all relevant statement sequences (Thm. 4.9).

Consider Lem. 4.8. To establish Thm. 4.9, we first show that if the three concerns are met for a method call execution, then that method call execution is respected (Lem. 4.8).

Lemma 4.8. *For every $ec \in \mathit{EvalContext}$, $mcs \in \mathit{AMCStmt}$, $cds \in \mathit{ClassDefSet}$, $\mathcal{A} \in \mathit{Alg}$, $AS \in \mathit{AlgSpec}$,*
if $\mathit{IsExecTerminating}(\langle ec, \langle\rangle, \langle mcs\rangle\rangle, cds) = \mathcal{T}$, *and*
$\mathit{IsResultCorrect}(ec, mcs, \mathcal{A}, AS, cds) = \mathcal{T}$, *and*
$\mathit{IsSideEffectFree}(ec, mcs, \mathcal{A}, AS, cds) = \mathcal{T}$,
then $\langle cds, AS\rangle$ *respects*$_{se}$ $\mathcal{A}$ *for* $\langle ec, mcs\rangle$.

The proof outline of Lem. 4.8 is as follows.

1 Assume $ec_0 \in \mathit{EvalContext}$, $v_0 := m(v_1, v_2) \in \mathit{AMCStmt}$, $cds \in \mathit{ClassDefSet}$, $\mathcal{A} \in \mathit{Alg}$, $AS \in \mathit{AlgSpec}$ such that the premise of Lem. 4.8 holds.
2 Assume $va_0, va_1 \in \mathit{Valuation}$ such that
$va_0 = \mathit{AbstrView}(ec_0, \mathcal{A}, AS, cds)$, and
$\langle va_0, \langle v_0 := m(v_1, v_2)\rangle\rangle \overset{a}{\rightsquigarrow}^{+}_{\mathcal{A}} \langle va_1, \langle\rangle\rangle$ (premise Def. 4.2.4 : *respects*$_{se}$).
3 Then we can assume $ec_1 \in \mathit{EvalContext}$, $va_2 \in \mathit{Valuation}$ such that
$\langle ec_0, \langle\rangle, \langle v_0 := m(v_1, v_2)\rangle\rangle \overset{c}{\rightsquigarrow}^{+}_{cds} \langle ec_1, \langle\rangle, \langle\rangle\rangle$, and
$\mathit{AbstrView}(ec_1, \mathcal{A}, AS, cds) = va_2$, and
$va_2(v_0) = va_1(v_0)$ (conclusion Def. 4.2.11 : *IsResultCorrect*).
4 Then $va_2 = va_0[v_0 \mapsto va_2(v_0)]$ (conclusion Def. 4.2.12: *IsSideEffectFree*).
5 Then $va_2 = \mathit{AbstrView}(ec_1, \mathcal{A}, AS, cds)$.
6 Then $\langle cds, \mathit{abstrViewEC}\rangle$ *respects*$_{se}$ $\mathcal{A}$ for $\langle ec, mcs\rangle$ (conclusion Def. 4.2.4 : *respects*$_{se}$).

That concludes the proof outline of Lem. 4.8.

Consider Thm. 4.9. Roughly, it states that an implementation of the computation concern *cds* satisfies$_c$ a specification AS_0 if there are a respectable model $\mathcal{A}$ of AS_0 and a specification AS_1 of the abstraction operators such that the three concerns are met for every top-level method call of every relevant statement sequence.

Theorem 4.9. *For every $cds \in \mathit{ClassDefSet}$, $AS_0 \in \mathit{AlgSpec}$,*
if *there are* $\mathcal{A} \in \mathit{RespectableModels}(AS_0)$, $AS_1 \in \mathit{AbstrOpDefs}(cds, AS_0)$ *such that*
for every $\overline{s} \in \mathit{RelevantStmtSeqs}(\mathcal{A})$, $va \in \mathit{Valuation}$,
for every $\langle ec, s\rangle \in \mathit{TopLevelMCExecs}(\overline{s}, emptyec, cds)$,
$\mathit{IsExecTerminating}(\langle ec, \langle\rangle, \langle s\rangle\rangle, cds) = \mathcal{T}$, *and*
$\mathit{IsResultCorrect}(ec, s, \mathcal{A}, AS_1, cds) = \mathcal{T}$, *and*
$\mathit{IsSideEffectFree}(ec, s, \mathcal{A}, AS_1, cds) = \mathcal{T}$,
then cds *satisfies*$_h$ AS_0.

Theorem 4.9 follows almost directly from Thm. 4.6 and Lem. 4.8.

4.2.4 Refining Result Correctness from Method Call to Method Body Level

In this section, we refine the notion of result correctness of a method call, to a notion of result correctness of the method body. This is a purely technical refinement which does not require design decisions.

The refinement consists of the following steps.

(1) Consider the execution of a method call $v_0 := m(v_1, v_2)$ from an evaluation context ec_0. Assume that the body of m is executed from an evaluation context ec_1. We relate the evaluation of the application of a method m to its *actual* parameters in the *full* abstract view of ec_1, to the evaluation of the application of m to its *formal* parameters in the abstract view of ec_1 (Lem. 4.10).
(2) Assume that the execution of the body terminates normally in ec_2, and that the execution of the call terminates normally in ec_3. We relate the value of the return variable in abstract view of ec_2, to the value of v_0 in the *full* abstract view of ec_3 (Lem. 4.11).
(3) We relate the value of a variable in an abstract view, to a value of a term in the full abstract view (Lem. 4.12).
(4) We present the body-level result-correctness notion (Def. 4.2.14) and prove that it is indeed a refinement (Lem. 4.13).

Steps 1-2: relating method call executions to method body executions. In this step we present two lemmas that relate call executions to body executions.

Consider Lem. 4.10. This lemma relates the full abstract view of the evaluation context from which a method call $\langle v_0 := m(v_1, v_2)\rangle$ is executed, to the abstract view of the evaluation context of the state σ from which its method body is executed. Recall from Def. 3.1.27 that $\mathit{GetFormalParams}(m, \mathit{cds})$ returns the formal parameters of method m, which are `this` and `p` variables if $m \notin \mathit{Constructor}$ and `q` and `p` variables otherwise. Note that in step 2 of this section, we relate the value of $m(\overline{sv})$ in the abstract view of $\sigma.ec$, to its value in the full abstract view of $\sigma.ec$.

Lemma 4.10. *For every $ec \in \mathit{EvalContext}$, $cs \in \mathit{CallStack}$, $v_0, v_1, v_2 \in \mathit{Var}$, $m \in \mathit{Method}$, $\sigma \in \mathit{State}$, $\mathcal{A} \in \mathit{Alg}$, $AS \in \mathit{AlgSpec}$, $\mathit{cds} \in \mathit{ClassDefSet}$, $va_0, va_1 \in \mathit{Valuation}$, $\nu \in \mathbb{V}$, $sv \in \mathit{StackVar}$,*

if $\langle ec, cs, \langle v_0 := m(v_1, v_2)\rangle\rangle \overset{c}{\rightsquigarrow}_{\mathit{cds}} \sigma$, *and*
$\mathit{AbstrViewFull}(ec, \mathcal{A}, AS, \mathit{cds}) = va_0$, *and*
$\mathit{AbstrView}(\sigma.ec, \mathcal{A}, AS, \mathit{cds}) = va_1$, *and*
$\mathit{Sem}(m(v_1, v_2), va_0, \mathcal{A}) = \nu$, *and*
$\mathit{GetFormalParams}(m, \mathit{cds}) = \overline{sv}$,
then $\mathit{Sem}(m(\overline{sv}), va_1, \mathcal{A}) = \nu$.

The proof outline of Lem. 4.10 is as follows.

1 Consider the execution of a method call $\langle v_0 := m(v_1, v_2)\rangle$ from an evaluation context ec.

2 Then the first step of the execution is a context switch to a state σ such that the body of the method is executed from $\sigma.ec$.
3 Assume $\mathit{GetFormalParams}(m, \mathit{cds})$ is $\overline{sv}$.
4 Then the context switch assigns the value of v_1 to $\overline{sv}[0]$ and the value of v_2 to $\overline{sv}[1]$. It does not change the object store (Def. 3.1.51: `CALL` and `CONSTR` rules).
5 Then the abstract values of $\overline{sv}$ in $\sigma.ec$, are the same as the abstract values of v_1 and v_2 in ec (Def. 4.1.9 : *AbstrView*).
6 Therefore, given $\mathcal{A}$ to interpret m, if $m(v_1, v_2)$ evaluates to ν in the abstract view of ec, then $m(\overline{sv})$ evaluates to ν in the abstract view of $\sigma.ec$.

That concludes the proof outline of Lem. 4.10.

Consider Lem. 4.11. Roughly, this lemma shows that the abstract value that is returned by a method body execution, is assigned to the variable on top of the stack. Note that this is the left-hand side variable of the corresponding method call execution (see Def. 3.1.51: $\rightsquigarrow$).

Lemma 4.11. *For every $ec \in \mathit{EvalContext}$, $sf \in \mathit{StackFrame}$, $v \in \mathit{Var}$, $cs \in \mathit{CallStack}$, $rv \in \mathit{ReturnVar}$, $\mathit{cds} \in \mathit{ClassDefSet}$, $\sigma \in \mathit{State}$, $\mathcal{A} \in \mathit{Alg}$, $AS \in \mathit{AlgSpec}$, $va_0, va_1 \in \mathit{Valuation}$, $\nu \in \mathbb{V}$,*

$$\begin{array}{ll} \textit{if} & \langle ec, \langle sf, v\rangle \rhd cs, \langle \texttt{return } rv\rangle\rangle \rightsquigarrow_{cds} \sigma, \textit{ and} \\ & \mathit{AbstrView}(ec, \mathcal{A}, AS, \mathit{cds}) = va_0, \textit{ and} \\ & \mathit{AbstrViewFull}(\sigma.ec, \mathcal{A}, AS, \mathit{cds}) = va_1, \textit{ and} \\ & va_0(rv) = \nu, \\ \textit{then} & va_1(v) = \nu. \end{array}$$

Proof of Lem. 4.11 is straightforward:

1 If $ec.sf$ maps rv to λ, then $\sigma.ec.sf$ maps v to λ (`RET rule` of Def. 3.1.51).
2 Furthermore, $ec.os$ and $\sigma.ec.os$ are the same (again, `RET rule`).
3 Then the abstract value of v in $\sigma.ec$ is the same as the abstract value of rv in ec (Def. 4.1.8 : *AbstrVal*).

That concludes the proof of Lem. 4.11.

Step 3: relating abstract views and full abstract views. Consider Def. 4.2.13 and Lem. 4.12. The intuition is that *ReplaceThis* can be used to relate a variable from the abstract view of an evaluation to a term in the full abstract view. More specifically, if the first formal parameter of a method is v and $\mathit{ReplaceThis}(v, \mathit{cds}) = t_0$, then the value of v in the abstract view of ec is the same as the value of t_0 in the full abstract view of ec. Likewise, if $\mathit{GetReturnVar}(m, \mathit{cds}) = rv$ and $\mathit{ReplaceThis}(rv, \mathit{cds}) = t_1$, then the value of rv in the abstract view of ec is the same as the value of t_1 in the full abstract view of ec.

Definition 4.2.13. $\mathit{ReplaceThis} : \mathit{Var} \times \mathit{ClassDefSet} \to \mathit{Term}$
$\mathit{ReplaceThis}(v, \mathit{cds}) = t$ *iff*

or $v \notin \mathit{ThisVar}$ *and* t *is* v,
or *there are* $C \in \mathit{ClassSort}$, $\overline{f} \in \mathit{Seq}(\mathit{Field})$ *such that*
v *is* $\mathtt{this}_C$, *and*
$\mathit{GetFields}(C, \mathit{cds}) = \overline{f}$, *and*
t *is* $\mathtt{abstr}_C(\overline{f})$.

Lemma 4.12. *For every* $ec \in \mathit{EvalContext}$, $\mathcal{A} \in \mathit{Alg}$, $AS \in \mathit{AlgSpec}$, $\mathit{cds} \in \mathit{ClassDefSet}$, $va_0, va_1 \in \mathit{Valuation}$, $v \in \mathit{Var}$, $t \in \mathit{Term}$, $\nu \in \mathbb{V}$,
if $\mathit{AbstrView}(ec, \mathcal{A}, AS, \mathit{cds}) = va_0$, *and*
$\mathit{AbstrViewFull}(ec, \mathcal{A}, AS, \mathit{cds}) = va_1$, *and*
$\mathit{ReplaceThis}(v) = t$,
then *or* $va_0(v)$ *undefined and* $\mathit{Eval}(t, va_1, \mathcal{A}, AS)$ *is undefined,*
or $va_0(v) = \mathit{Eval}(t, va_1, \mathcal{A}, AS)$.

The proof of Lem. 4.12 is fairly given the definitions of *AbstrView* and *AbstrViewFull*. The proof outline is as follows.

1 Assume $ec \in \mathit{EvalContext}$, $\mathcal{A} \in \mathit{Alg}$, $AS \in \mathit{AlgSpec}$, $\mathit{cds} \in \mathit{ClassDefSet}$, $va_0, va_1 \in \mathit{Valuation}$, $v \in \mathit{Var}$, $t \in \mathit{Term}$, $\nu \in \mathbb{V}$ such that the premise of Lem. 4.12 holds. Two cases can be distinguished.
2 Case: $v \notin \mathit{ThisVar}$. Then Def. 4.2.2 : *AbstrViewFull* defines the abstract value of v using Def. 4.1.9 : *AbstrView* and the conclusion holds almost trivially.
3 Case: $v \in \mathit{ThisVar}$.
  a Then we can assume $C \in \mathit{ClassSort}$, $\overline{f} \in \mathit{Seq}(\mathit{Field})$ such that v is $\mathtt{this}_C$, and $\mathit{GetFields}(C, \mathit{cds}) = \overline{f}$, and t is $\mathtt{abstr}_C(\overline{f})$ (Def. 4.2.13).
  b Assume $\mathcal{A}_1 \in \{\mathcal{A} \mid \mathcal{A}_0 \subseteq \mathcal{A} \wedge \mathcal{A} \in \mathit{Models}(AS.ax)\}$ (Def. 4.1.7: *Eval*).
  c Two cases can be distinguished.
  d Case: $va_0(v) = \nu$.
    1 Then va_0 maps v to the evaluation of t in a valuation va_2 that maps the fields $\overline{f}$ to their abstract values, using $\mathcal{A}$ to interpret $\mathtt{abstr}_C()$ (Def. 4.1.9).
    2 Then va_1 maps the fields $\overline{f}$ to the same abstract values as va_2 (Def. 4.2.2 : *AbstrViewFull*).
    3 Then $\mathit{Sem}(t, va_1, \mathcal{A}_1) = \nu$.
  e Case: $va_0(v)$ is undefined. Then by reversing the reasoning in the case above, we can conclude that $\mathit{Sem}(t, va_1, \mathcal{A}_1)$ is also undefined.

That concludes the proof outline of lemma Lem. 4.12.

Step 4: a body-level notion of result correctness. Consider Def. 4.2.14. The intuition is that *IsResCorBody* defines the body-level refinement of method-level result correctness notion (Def. 4.2.11: *IsResultCorrect*). Roughly, it states that if 1) the body of a method m is execution from an evaluation context ec_0 with abstract view va_0, and 2) the application of m to its formal parameters evaluates to ν, then the execution terminates normally in an evaluation context with an abstract view va_1 that maps the variable rv that holds the return value to ν.

Definition 4.2.14. *IsResCorBody* : *EvalContext* × *Method* × *Alg* × *AlgSpec* × *ClassDefSet* → *Bool*
$IsResCorBody(ec_0, m, \mathcal{A}, AS, cds) = \mathcal{T}$ *iff*
for every va_0, *Valuation*, $t_0 \in Term$,
if $IsExecTerminating(\langle ec_0, \langle\rangle, GetBody(m, cds)\rangle, cds) = \mathcal{T}$, *and*
$AbstrViewFull(ec_0, \mathcal{A}, AS, cds) = va_0$, *and*
$GetFormalParams(m, cds) = \langle sv_0, sv_1\rangle$, *and*
$ReplaceThis(rv, cds) = t_0$, *and*
$Eval(m(t_0, sv_1), va_0, \mathcal{A}, AS) = \nu$,
then *there are* $ec_1 \in EvalContext$, $va_1 \in Valuation$, $rv \in ReturnVar$
such that
$\langle ec_0, \langle\rangle, GetBody(m, cds)\rangle \overset{c}{\rightsquigarrow}{}^{+}_{cds} \langle ec_1, \langle\rangle, \langle\rangle\rangle$, *and*
$AbstrViewFull(ec_1, \mathcal{A}, AS, cds) = va_1$, *and*
$GetReturnVar(m, cds) = rv$, *and*
$ReplaceThis(rv, cds) = t_1$, *and*
$Eval(t_1, va_0, \mathcal{A}, AS) = \nu$.

Consider Lem. 4.13. This lemma formalizes that the body-level result correctness notion is indeed a refinement of the method-call-level notion.

Lemma 4.13. *For every* $ec \in EvalContext$, $v_0, v_1, v_2 \in AVar$, $m \in Method$, $\sigma \in State$, $\mathcal{A} \in Alg$, $AS \in AlgSpec$, $cds \in ClassDefSet$,
if $\langle ec, \langle\rangle, \langle v_0 := m(v_1, v_2)\rangle\rangle \overset{c}{\rightsquigarrow}_{cds} \sigma$, *and*
$IsResCorBody(\sigma.ec, m, \mathcal{A}, AS, cds) = \mathcal{T}$,
then $IsResultCorrect(ec, \langle v_0 := m(v_1, v_2)\rangle, \mathcal{A}, AS, cds) = \mathcal{T}$.

The proof is fairly straightforward and therefore omitted. It uses Def. 4.2.14 : *IsResCorBody* and Def. 4.2.11 : *IsResultCorrect*, and Lems. 4.10 to 4.12 above, and must take into account that if $v_0 \in Field$, then the SH (ShortHand) rule from Def. 3.1.51: $\overset{c}{\rightsquigarrow}$ introduces an additional step in the concrete execution that assigns the outcome of the method call to dummy variable which is then assigned to v_0.

4.2.5 Abstract Reasoning About Method Body Executions

In this section, we show under which conditions the method body implementer can be an abstract programmer. This is the main result of Sect. 4.2.

We take the following steps.

(1) We define a notion of result correctness that considers only the abstract execution (Def. 4.2.15). This notion essentially formalizes Hoare's main proof obligation at the semantical level.
(2) We formalize and prove that Hoare-style satisfaction is established if 1) executions of method calls terminate and are side-effect free when viewed abstractly, and 2) abstract executions of method bodies are result correct (Thm. 4.9).

The consequences of this style of reasoning are discussed in Sect. 4.2.7.

Step 1: body-level result correctness of the abstract execution In this first step we define a notion of result correctness that considers only the abstract execution.

Consider Def. 4.2.15. Abstract body-level result correctness is very similar to the original notion of body-level result correctness (Def. 4.2.14: *IsResCorBody*), but requires that the right result is returned by the abstract execution, rather than by the abstract view of the concrete execution. Note that we need the *full* abstract view to reason about the abstract execution of the method body, as the body may use the fields of the this object as abstract variables.

Definition 4.2.15. $\mathit{IsAbstrResCorBody} : \mathit{Valuation} \times \mathit{Method} \times \mathit{Alg} \times \mathit{AlgSpec} \times \mathit{ClassDefSet} \hookrightarrow \mathit{Bool}$
$\mathit{IsAbstrResCorBody}(va_0, m : S_1 \times S_2 \hookrightarrow S_0, \mathcal{A}, AS, cds) = \mathcal{T}$ iff
for every $sv_0, sv_1 \in \mathit{StackVar},\ t_0 \in \mathit{Term},\ \nu \in \mathbb{V}$,
if $\mathit{GetFormalParams}(m, cds) = \langle sv_0, sv_1 \rangle$, and
$\mathit{ReplaceThis}(sv_0, cds) = t_0$, and
$\mathit{Eval}(m(t_0, sv_1), va_0, \mathcal{A}, AS) = \nu$,
then there are $va_1 \in \mathit{Valuation}, rv \in \mathit{ReturnVar},\ t_1 \in \mathit{Term}$ such that
$\langle va_0, \mathit{GetBody}(m, cds) \rangle \rightsquigarrow^*_{\mathcal{A}} \langle va_1, \langle\rangle \rangle$, and
$\mathit{GetReturnVar}(m, cds) = rv$, and
$\mathit{ReplaceThis}(rv, cds) = t_1$, and
$\mathit{Eval}(t_1, va_1, \mathcal{A}, AS) = \nu$.

Aside. This property essentially formalizes Hoare's main proof obligation at the semantical level, except that it does not consider modular reasoning (see Sect. 4.2.6).

Step 2: refined separation of concerns for Hoare-Style satisfaction. In this second step, we show under which conditions the method body implementer can be an abstract programmer.

Consider Thm. 4.14. Recall from Thm. 4.9 that Hoare-style satisfaction is established if, for every relevant statement sequence, all *top-level* executions of method calls terminate, are result correct, and side-effect free when viewed abstractly. Here, we refine Thm. 4.9 and show that Hoare-style satisfaction is established if, for every relevant statement sequence, 1) all executions of method calls terminate and are side-effect free when viewed abstractly, and 2) all *abstract* executions of method *bodies* are result correct. Intermediate notions are used to capture all method call executions, and to relate a method call execution to the corresponding abstract method body execution.

Consider Def. 4.2.16. Roughly, $\mathit{MethCallExecs}(\overline{s_0}, ec_0, cds)$ returns the set of all method call executions from the execution of $\overline{s}$ from ec_0, including those that are part of method body executions.

Consider Def. 4.2.17. Consider the execution of a method call statement *mcs* from an evaluation context *ec*. This execution consists of a context switch, a method body execution, and another context switch. Roughly, if $\mathit{AbstrMethBodyExec}(mcs, ec, \mathcal{A}, AS, cds) = \langle va, m \rangle$, then the method called is m, and the full abstract view of the evaluation context from which the method body is executed is va.

Definition 4.2.16. $\mathit{MethCallExecs} : \mathit{Seq}(\mathit{AStmt}) \times \mathit{EvalContext} \times \mathit{ClassDefSet} \to \mathit{Set}(\mathit{EvalContext} \times \mathit{AStmt})$
$\langle ec_1, s \rangle \in \mathit{MethCallExecs}(\overline{s_0}, ec_0, cds)$ iff
  there is are $cs \in \mathit{CallStack}$, $\overline{s}_1 \in \mathit{Seq}(\mathit{AStmt})$ such that
    $\langle ec_0, \langle\rangle, \overline{s}_0 \rangle \rightsquigarrow^*_{cds} \langle ec_1, cs, \langle s \rangle \triangleright \overline{s}_1 \rangle$, and
    $s \in \mathit{AMCStmt}$.

Definition 4.2.17. $\mathit{AbstrMethBodyExec} : \mathit{EvalContext} \times \mathit{AMCStmt} \times \mathit{Alg} \times \mathit{AlgSpec} \times \mathit{ClassDefSet} \to \mathit{Valuation} \times \mathit{Method}$
$\mathit{AbstrMethBodyExec}(mcs, ec, \mathcal{A}, AS, cds) = \langle va, m \rangle$ iff
  there are $v_0, v_1, v_2 \in \mathit{AVar}$, $\sigma \in \mathit{State}$ such that
    mcs is $v_0 := m(v_1, v_2)$, and $\langle ec, \langle\rangle, \langle mcs \rangle \rangle \rightsquigarrow \sigma$, and
    $\mathit{AbstrViewFull}(\sigma.ec, \mathcal{A}, AS, cds) = va$.

Theorem 4.14. *For every* $cds \in \mathit{ClassDefSet}$, $AS_0 \in \mathit{AlgSpec}$,
if   *there are* $\mathcal{A} \in \mathit{RespectableModels}(AS_0)$, $AS_1 \in \mathit{AbstrOpDefs}(cds, AS_0)$ *such that*
  for every $\overline{s} \in \mathit{RelevantStmtSeqs}(\mathcal{A})$,
    for every $\langle ec, mcs \rangle \in \mathit{MethCallExecs}(\overline{s}, \mathit{emptyec}, cds)$,
      $\mathit{IsExecTerminating}(\langle ec, \langle\rangle, \langle mcs \rangle \rangle, cds) = \mathcal{T}$, *and*
      $\mathit{IsSideEffectFree}(ec, mcs, \mathcal{A}, AS_1, cds) = \mathcal{T}$, *and*
      there are $va \in \mathit{Valuation}$, $m \in \mathit{Method}$ *such that*
        $\mathit{AbstrMethBodyExec}(mcs, ec, \mathcal{A}, AS_1, cds) = \langle va, m \rangle$, *and*
        $\mathit{IsAbstrResCorBody}(va, m, \mathcal{A}, AS_1, cds) = \mathcal{T}$,
then cds *satisfies*$_h$ AS_0.

Proof of Thm. 4.14 depends on two intermediate lemmas. These are presented below, followed by the proof of the theorem.

Proof of step 2: intermediate lemma 1. Consider Lem. 4.15. Roughly, *IsAbstrResCorBody* ensures that the abstract execution of the body of m returns the right result, and that $\mathit{respects}_{sse}$ relates the concrete execution of the body to its abstract execution. This first intermediate lemma is used in the second intermediate lemma.

Lemma 4.15. *for every* $ec \in \mathit{EvalContext}$, $m \in \mathit{Method}$, $cds \in \mathit{ClassDefSet}$, $\mathcal{A} \in \mathit{RespectableAlg}$, $AS \in \mathit{AlgSpec}$, $va \in \mathit{Valuation}$,
if $\mathit{AbstrViewFull}(ec, \mathcal{A}, AS, cds) = va$, *and*
  $\mathit{IsAbstrResCorBody}(va, m, \mathcal{A}, AS, cds) = \mathcal{T}$, *and*
  cds $\mathit{respects}_{sse}$ $\mathcal{A}$ *for* $\langle ec, \mathit{GetBody}(m, cds) \rangle$, *and*
then $\mathit{IsResCorBody}(ec, m, \mathcal{A}, AS, cds) = \mathcal{T}$.

The proof outline of Lem. 4.15 is as follows.

1 Consider $ec_0 \in \mathit{EvalContext}$, $m \in \mathit{Method}$, $cds \in \mathit{ClassDefSet}$, $\mathcal{A}_0 \in \mathit{Alg}$, $AS \in \mathit{AlgSpec}$, $va_0 \in \mathit{Valuation}$ such that the premise of Lem. 4.15 holds.
2 Assume (Def. 4.2.3 : $\mathit{respects}_{sse}$) $ec_1 \in \mathit{EvalContext}$, $va_1 \in \mathit{Valuation}$ such that $\langle ec_0, \mathit{GetBody}(m, cds)\rangle \overset{c}{\rightsquigarrow}^*_{csd} \langle ec_1, \langle\rangle, \langle\rangle\rangle$, and
$\mathit{AbstrView}(ec_1, \mathcal{A}, AS, cds) = va_1$.
3 Assume (premise Lem. 4.2.14 : $\mathit{IsResCorBody}$) $va_2 \in \mathit{Valuation}$, $sv_0, sv_1 \in \mathit{StackVar}$, $t_0 \in \mathit{Term}$, $\nu \in \mathbb{V}$ such that
$\mathit{AbstrView}(ec_0, \mathcal{A}, AS, cds) = va_2$, and
$\mathit{GetFormalParams}(m, cds) = \langle sv_0, sv_1\rangle$, and
$\mathit{ReplaceThis}(sv_0, sv_1) = t_0$, and
$\mathit{Eval}(m(t_0, sv_1), va_2, \mathcal{A}_0, AS) = \nu$.
4 Then we can assume (conclusion Def. 4.2.15 : $\mathit{IsAbstrResCorBody}$) $va_3 \in \mathit{Valuation}, rv \in \mathit{ReturnVar}$, $t_1 \in \mathit{Term}$ such that
$\langle va_0, \mathit{GetBody}(m, cds)\rangle \overset{a}{\rightsquigarrow}^*_{\mathcal{A}} \langle va_3, \langle\rangle\rangle$, and
$\mathit{GetReturnVar}(m, cds) = rv$, and
$\mathit{ReplaceThis}(rv, cds) = t_1$, and
$\mathit{Eval}(t_1, va_2, \mathcal{A}, AS) = \nu$.
5 We know that $va_2 = \mathit{AbstrViewFull}(ec_1, \mathcal{A}, AS, cds)$ (Def. 4.2.3: $\mathit{respects}_{sse}$).
6 Then $\mathit{IsResCorBody}(ec_0, m, \mathcal{A}_0, AS, cds) = \mathcal{T}$.

That concludes the proof outline of Lem. 4.15.

Proof of step 2: intermediate lemma 2. Consider Lem. 4.16. Note that its premise is very similar to that of Thm. 4.14. The proof of this lemma brings together all the main lemmas that we have proven in this section. The intuition is the following. Consider an arbitrary method call execution in the execution of $\overline{s}$. This call execution consists of a context switch, a body execution, and another context switch. It follows by induction that every call execution in the body execution is respected. Therefore, the body execution is respected (Lem. 4.7). Then, as the abstract execution of the body is result correct by assumption, its concrete execution is result correct as well (Lem. 4.15). Then, as the method call execution is terminating and side-effect free by assumption, it is respected (Lem. 4.8).

Lemma 4.16. *for every $\overline{s} \in \mathit{Seq}(\mathit{AStmt})$, $ec_0 \in \mathit{EvalContext}$, $cds \in \mathit{ClassDefSet}$, $\mathcal{A} \in \mathit{RespectableAlg}$, $AS \in \mathit{AlgSpec}$,*
if *for every $\langle ec_1, mcs\rangle \in \mathit{MethCallExecs}(\overline{s}, ec_0, cds)$,*
$\mathit{IsExecTerminating}(\langle ec_1, \langle\rangle, \langle mcs\rangle\rangle, cds) = \mathcal{T}$, and
$\mathit{IsSideEffectFree}(ec_1, mcs, \mathcal{A}, AS, cds) = \mathcal{T}$, and
there are $va \in \mathit{Valuation}$, $m \in \mathit{Method}$ such that
$\mathit{AbstrMethBodyExec}(mcs, ec_1, \mathcal{A}, AS, csd) = \langle va, m\rangle$, and
$\mathit{IsAbstrResCorBody}(va, m, \mathcal{A}, AS, cds) = \mathcal{T}$,
then *for every $\langle ec_1, mcs\rangle \in \mathit{MethCallExecs}(\overline{s}, ec_0, cds)$,*
$\langle cds, AS\rangle$ $\mathit{respects}_{se}$ $\mathcal{A}$ for $\langle ec_1, mcs\rangle$.

The proof outline of Lem. 4.16 is as follows.

1 Assume $\overline{s}_0 \in \mathit{Seq}(\mathit{AStmt})$, $\mathit{cds} \in \mathit{ClassDefSet}$, $\mathcal{A} \in \mathit{Alg}$, $\mathit{AS} \in \mathit{AlgSpec}$ such that the premise of Lem. 4.16 holds.

2 Proof is by induction on the size of $\mathit{MethCallExecs}(\overline{s}_0, \mathit{cds})$.

3 Base Case: $\mid \mathit{MethCallExecs}(\overline{s}_0, \mathit{cds}) \mid = 0$. In this case, $\mathit{MethCallExecs}(\overline{s}_0, \mathit{ec}_0, \mathit{cds}) = \{\}$ and the conclusion holds trivially.

4 Step Case: Assume $n > 0$ such that $\mid \mathit{MethCallExecs}(\overline{s}_0, \mathit{cds}) \mid = n$. The induction hypothesis concludes respect for all method call executions in all sequences with fewer than n method call executions.

5 Consider an arbitrary method call execution $\langle \mathit{ec}_1, v_0 := m(v_1, v_2)\rangle \in \mathit{TopLevelMCExecs}(\overline{s}_0, \mathit{ec}_0, \mathit{cds})$.

6 Then we can assume (premise Lem. 4.16) $\sigma \in \mathit{State}$, $\mathit{va}_0 \in \mathit{Valuation}$ such that $\langle \mathit{ec}_1, \langle\rangle, \langle v_0 := m(v_1, v_2)\rangle\rangle \rightsquigarrow \sigma$, and $\mathit{AbstrViewFull}(\sigma.\mathit{ec}, \mathcal{A}, \mathit{AS}, \mathit{cds}) = \mathit{va}_0$.

7 Then $\mathit{MethCallExecs}(\mathit{GetBody}(m, \mathit{cds}), \sigma.\mathit{ec}, \mathit{cds}) \subset \mathit{MethCallExecs}(\overline{s}, \mathit{ec}_0, \mathit{cds})$ (Def. 3.1.51: $\rightsquigarrow$).

8 Then for every $\langle \mathit{ec}_2, \mathit{mcs}\rangle \in \mathit{MethCallExecs}(\mathit{GetBody}(m, \mathit{cds}), \sigma.\mathit{ec}, \mathit{cds})$, $\langle \mathit{cds}, \mathit{AS}\rangle$ $\mathit{respects}_{se}$ $\mathcal{A}$ for $\langle \mathit{ec}_2, \mathit{mcs}\rangle$ (induction hypothesis).

9 In other words, if an arbitrary method call $v_0 := m(v_1, v_2)$ is executed from top-level evaluation context ec_1, and as a result, the body of m is executed from an evaluation context $\sigma.\mathit{ec}$, then every method call execution in this method body execution is respected. We now prove that $\langle \mathit{ec}_0, v_0 := m(v_1, v_2)\rangle$ itself is respected as well, using Lem. 4.8.

  a To this end, we first prove that $\mathit{IsResCorBody}(\sigma.\mathit{ec}, m, \mathcal{A}, \mathit{AS}, \mathit{cds}) = \mathcal{T}$ using Lem. 4.15.

    1 To this end, we first prove $\langle \mathit{cds}, \mathit{AS}\rangle$ $\mathit{respects}_{sse}$ $\mathcal{A}$ for $\langle \sigma.\mathit{ec}, \mathit{GetBody}(m, \mathit{cds})\rangle$ using Lem. 4.7.

      a We know $\mathit{TopLevelMCExecs}(\mathit{GetBody}(m, \mathit{cds}), \sigma.\mathit{ec}, \mathit{cds}) \subseteq \mathit{MethCallExecs}(\overline{s}, \mathit{ec}_0, \mathit{cds})$ (as the initial call stack does not affect an execution).

      b Then for every $\langle \mathit{ec}_1, s\rangle \in \mathit{TopLevelMCExecs}(\mathit{GetBody}(m, \mathit{cds}), \sigma.\mathit{ec}, \mathit{cds})$, cds $\mathit{respects}_{se}$ $\langle \mathcal{A}, \mathit{AS}\rangle$ for $\langle \mathit{GetBody}(m, \mathit{cds}), \sigma.\mathit{ec}\rangle$ (already proven for every method call execution in the method body execution).

      c We also know $\mathit{IsFieldsAssignedToDefined}(\sigma.\mathit{ec}, \mathit{GetBody}(m, \mathit{cds})) = \mathcal{T}$, as 1) a method body only assigns to fields that are defined in its class (Def. 3.1.19 : $\mathit{ClassDef}$), and 2) all fields of the class of an object are defined when the object is allocated (Def. 3.1.50), and 3) fields cannot be 'undefined' (Def. 3.1.51 : $\rightsquigarrow$).

      d Then $\langle \mathit{cds}, \mathit{AS}\rangle$ $\mathit{respects}_{sse}$ $\mathcal{A}$ for $\langle \sigma.\mathit{ec}, \mathit{GetBody}(m, \mathit{cds})\rangle$ (Lem. 4.7).

    2 Furthermore, we can assume (premise Lem. 4.16) $\mathit{AbstrViewFull}(\sigma.\mathit{ec}, \mathcal{A}, \mathit{AS}, \mathit{cds}) = \mathit{va}$, and $\mathit{IsAbstrResCorBody}(\mathit{va}_0, \mathit{mcs}, \mathcal{A}, \mathit{AS}, \mathit{cds}) = \mathcal{T}$.

    3 Then $\mathit{IsResCorBody}(\sigma.\mathit{ec}, m, \mathcal{A}, \mathit{AS}, \mathit{cds}) = \mathcal{T}$ (Lem. 4.15).

  b Then $\mathit{IsResultCorrect}(\mathit{ec}_1, \langle v_0 := m(v_1, v_2)\rangle, \mathcal{A}, \mathit{AS}, \mathit{cds}) = \mathcal{T}$ (Lem. 4.13).

c Furthermore, we can assume (premise Lem. 4.16) $\mathit{IsExecTerminating}(\langle ec_1, \langle\rangle, \langle v_0 := m(v_1, v_2)\rangle\rangle, \mathit{cds}) = \mathcal{T}$, and $\mathit{IsSideEffectFree}(ec_1, v_0 := m(v_1, v_2), \mathcal{A}, AS, \mathit{cds}) = \mathcal{T}$.
d Then $\langle \mathit{cds}, AS\rangle$ $\mathit{respects}_{se}$ $\mathcal{A}$ for $\langle ec_1, v_0 := m(v_1, v_2)\rangle$ (Lem. 4.8).

10 So, for an arbitrary top-level method call execution, both the method call execution itself and all the method call executions it contains are respected.
11 Then the conclusion of Lem. 4.16 holds, i.e., all method call executions in the execution of $\overline{s}$ from ec_0 are respected.
12 That concludes the proof of the step case.

That concludes the proof outline of Lem. 4.16.

Proof of step 2: proof of Thm. 4.14. Proof of Thm. 4.14 is straightforward given Lem. 4.16 (which has a premise that is implied by the premise of Thm. 4.14, and which concludes that all method call executions are respected) and Thm. 4.6 (which concludes that Hoare-style satisfaction is established when all top-level method call executions are respected).

4.2.6 Modular Reasoning

In Sect. 4.2.5, we presented a reasoning approach to establish Hoare-style satisfaction. This approach requires properties to be established for every method call execution that can occur in any execution of a relevant statement execution in the context of *cds*. The downside of the approach is that any change to *cds* can invalidate this reasoning. In this section, we show under which conditions the implementer of a method m can reason modularly, i.e., in such a way that changes to methods other than m do not affect the reasoning about m.

Containment Consider Def. 4.2.18. The intuition is that $\mathit{AreMethCallExecsContained}(\mathit{mceSet}, \mathcal{A}, \mathit{cds})$ holds only if the set of method call executions *mceSet* contains all method call executions of all executions of statement sequences that are relevant to $\mathcal{A}$. Recall from Def. 4.2.5 that $\overline{s}$ is relevant only if its abstract execution terminates normally. Note that this is the set of method call executions for which the premise of Thm. 4.14 requires termination and side-effect freeness.

Definition 4.2.18. $\mathit{AreMethCallExecsContained} : \mathit{Set}(\mathit{EvalContext} \times \mathit{AMCStmt}) \times \mathit{Alg} \times \mathit{ClassDefSet} \to \mathit{Bool}$
$\mathit{AreMethCallExecsContained}(\mathit{mceSet}, \mathcal{A}, \mathit{cds}) = \mathcal{T}$ iff
for every $\overline{s} \in \mathit{RelevantStmtSeqs}(\mathcal{A})$,
$\mathit{MethCallExecs}(\overline{s}, \mathit{emptyec}, \mathit{cds}) \subseteq \mathit{mceSet}$.

Consider Def. 4.2.19. Recall from Def. 4.2.17 that *AbstrMethBodyExec* relates a method call execution to the corresponding abstract method body execution. The intuition is that $\mathit{AreAbstrMethBodyExecsContained}(\mathit{ambeSet}, \mathit{mcsSet}, \mathcal{A}, AS, \mathit{cds})$ holds only if for every method call execution $\langle ec, mcs\rangle \in \mathit{mceSet}$, the corresponding abstract method body execution is contained in the set *ambeSet*. Note that if $\mathit{AreMethCallExecsContained}(\mathit{mceSet}, \mathcal{A}, \mathit{cds})$ and $\mathit{AreAbstrMethBodyExecsContained}(\mathit{ambeSet}, \mathit{mcsSet}, \mathcal{A}, AS, \mathit{cds})$ both hold, then *ambeSet* contains every abstract method body execution for which the premise of Thm. 4.14 requires abstract result correctness.

Definition 4.2.19. $\mathit{AreAbstrMethBodyExecsContained} : \mathit{Set}(\mathit{Valuation} \times \mathit{Method}) \times \mathit{Set}(\mathit{EvalContext} \times \mathit{AMCStmt}) \times \mathit{Alg} \times \mathit{AlgSpec} \times \mathit{ClassDefSet} \to \mathit{Bool}$
$\mathit{AreAbstrMethBodyExecsContained}(\mathit{ambeSet}, \mathit{mcsSet}, \mathcal{A}, AS, \mathit{cds}) = \mathcal{T}$ iff
for every $\langle ec, mcs\rangle \in \mathit{mceSet}$,
$\mathit{AbstrMethBodyExec}(mcs, ec, \mathcal{A}, AS, \mathit{cds}) \in \mathit{ambeSet}$

Modular reasoning Consider Thm. 4.17.

The intuition is that the implementer must establish termination and side effect freeness only for an arbitrary method call execution *mce* for which he assumes a specific set of properties to hold. In the theorem, *mceSet* represents the set of all method call executions with these properties. Let *relevantMceSet* be the set of all method call executions of all executions of statement sequences that are relevant to $\mathcal{A}$. Given the premise of Thm. 4.14, *mceSet* needs to contain *relevantMceSet*. Note that this is the case when $\mathit{AreMethCallExecsContained}(\mathit{mceSet}, \mathcal{A}, \mathit{cds}) = \mathcal{T}$. This is typically established by 1) requiring that the method body implementer ensures that a certain set of syntactical and/or semantical properties holds in *mce*, and 2) providing a meta-level proof (i.e., a proof that is not problem-specific, proven by the provider of the methodology) that these properties constrain *relevantMceSet* enough to enure that *mceSet* contains *relevantMceSet*.

Similarly, the implementer must establish result correctness only for an arbitrary abstract method body execution *mcbe* for which he assumes a specific set of properties to hold. In the theorem, *ambeSet* represents the set of all abstract method body execution with these properties. Given the premise of Thm. 4.14, *ambeSet* must be such that $\mathit{AreAbstrMethBodyExecsContained}(\mathit{ambeSet}, \mathit{mcsSet}, \mathcal{A}, AS_1, \mathit{cds})$ holds. Again, this is typically established by requiring a set of syntactical and/or semantical properties holds in *mce*, and providing a meta-level proof. In Exmpl. 4.6, we discuss a well-known property that the implementer can assume to hold in the abstract method body execution: the class invariant.

Theorem 4.17. *For every* $\mathit{cds} \in \mathit{ClassDefSet}$, $AS_0 \in \mathit{AlgSpec}$,

if   *there are* $\mathcal{A} \in \mathit{RespectableModels}(AS_0)$, $AS_1 \in \mathit{AbstrOpDefs}(cds, AS_0)$,
$\mathit{mceSet} \in \mathit{Set}(\mathit{EvalContext} \times \mathit{AMCStmt})$,
$\mathit{ambeSet} \in \mathit{Set}(\mathit{Valuation} \times \mathit{Method})$ *such that*
$\mathit{AreMethCallExecsContained}(\mathit{mceSet}, \mathcal{A}, cds) = \mathcal{T}$, *and*
$\mathit{AreAbstrMethBodyExecsContained}(\mathit{ambeSet}, \mathit{mceSet}, \mathcal{A}, AS_1, cds) = \mathcal{T}$,
and for every $\langle ec, mcs\rangle \in \mathit{mceSet}$,
$\mathit{IsExecTerminating}(\langle ec, \langle\rangle, \langle mcs\rangle\rangle, cds) = \mathcal{T}$, *and*
$\mathit{IsSideEffectFree}(ec, mcs, \mathcal{A}, AS, cds) = \mathcal{T}$, *and*
for every $\langle va, m\rangle \in \mathit{ambeSet}$,
$\mathit{IsAbstrResCorBody}(va, m, \mathcal{A}, AS_1, cds) = \mathcal{T}$,
then   *cds satisfies*$_h$ AS_0.

Proof of Thm. 4.17 is straightforward given Thm. 4.14, Def. 4.2.18 and Def. 4.2.19.

Example 4.6. (*Class Invariants*)

In this context, a class invariant is a relation between the (abstract values of) fields of an object. For example, a class invariant for class `Rat` from Exmpl. 4.1 could be that the greatest common divider of fields `n` and `d` is 1.

Note that this property cannot be established by the caller of a `Rat` method. For example, in the abstract view of the caller, $\texttt{new}_{Rat}(1, 2)$ and $\texttt{new}_{Rat}(2, 4)$ are the same value. We discuss an informal specification and implementation approach for invariants.

A class invariant of a class C can be specified in the same way as an abstraction operator: introduce a special purpose operator `inv` that has the sorts of the fields of C as its domain sorts and Bool as its range sort, and define one or more axioms that relate `inv` to operators of the specification.

We say that the invariant of $\alpha \in \mathcal{A}$ holds in $ec_0 \in \mathit{EvalContext}$ when 1) α is of a class C, and 2) $\mathit{ViewOf}(\alpha, ec_0.os) = ec_1$ (recall from Def. 3.3.5 that *ViewOf* moves the fields of α from the object store to the stack frame of ec_1), and 3) the full abstract view of ec_1 is $va \in \mathit{Valuation}$, and 4) C has fields $\overline{f}$, and 5) $\texttt{inv}(\overline{f})$ evaluates to $\mathcal{T}$ in va.

Consider an arbitrary $\mathit{mceSet} \in \mathit{Set}(\mathit{EvalContext} \times \mathit{AMCStmt})$, $\mathcal{A} \in \mathit{Alg}$, $cds \in \mathit{ClassDefSet}$ such that 1) $\mathit{AreMethCallExecsContained}(\mathit{mceSet}, \mathcal{A}, cds) = \mathcal{T}$, and 2) the body of every method in *cds* is a sequence of abstract statements. Consider an arbitrary $\mathit{ambeSet} \in \mathit{Set}(\mathit{Valuation} \times \mathit{Method})$, $C \in \mathit{ClassSort}$ such that for every $\langle va, m\rangle \in \mathit{ambeSet}$, if $m \notin \mathit{Constructor}$ and $\mathit{OpSort}(m) = C$, then the class invariant of C holds in va. It can be proven that $\mathit{AreAbstrMethBodyExecsContained}(\mathit{ambeSet}, \mathit{mcsSet}, \mathcal{A}, AS, cds) = \mathcal{T}$ if 1) for every $\langle va_0, m\rangle \in \mathit{ambeSet}$, m is of class C, and the abstract execution of the body of m terminates normally in valuation va_1, then the invariant of C holds in va_1, and 2) no object referred to by a field of an object, is aliased.

A key part of this meta-level proof uses the assumption that all method bodies are abstract statements to show that

1) an invariant of object α is not invalidated by a statement execution when no method that has α as the receiver is active, i.e., when there is no stack frame on the stack in which `this` refers to α.
2) there are no callbacks, i.e., if the `this` object of the stack frame of a state refers to α, then there is no stack frame on the stack in which `this` refers to α.

Consider a non-constructor method m of a class C with fields $\overline{f}$. Given the meta-level proof, when reasoning about abstract result correctness of the execution of the body of m from a valuation va, the implementer can assume that $\texttt{inv}(\overline{f})$ evaluates to $\mathcal{T}$ in va.

4.2.7 Consequences of the Implementation Approach

In this section, we discuss the main advantages and limitations of the implementation approach.

One main advantage of the approach is that it allows the implementer of a method body to reason about the abstract execution of the body, rather than the much more complicated concrete execution. Another is that the termination, side-effect freeness and result correctness properties are suitable for verification. We sketch verification techniques in Sect. 5.

One inherent and severe limitation of the approach is that certain well-established OO design patterns that are based on cooperation between objects that are conceptually at the same level of abstraction, cannot be used. An example is the Observer Pattern [GHJV95]. The reason is that these patterns use methods that are not side-effect free when viewed abstractly. Use of such patterns requires a different abstract view of an evaluation context that considers the state as a configuration of objects, where a single statement execution can change the entire configuration. This different style of reasoning requires a different style of specification as well (one that is studied in the rest of this thesis). Note that this could still be combined with a client specification that is based on algebraic specifications, although we do not know of any research in this direction.

Another inherent limitation is that methods are required to be deterministic when viewed abstractly (i.e., always produce the same result for the same input parameters). This is intuitive as our abstract view of methods is functional. More specifically, consider a method call statement $v_0 := m(v_0, v_1)$. If abstract views va_0 and va_1 agree on the values of v_0 and v_1, we expect the result of the abstract execution of the call from va_0 and va_1 to be the same. As an example, consider an algebraic specification AS that defines an underspecified operator `choose: C` $\rightarrow$ `Bool` that does not occur in any of the axioms of AS. Consider an implementation with a class `C` that has a field `switch` of sort `Bool`, and a `choose` method with a body `switch := !switch; return switch;`. Then Hoare-style satisfaction cannot be established using the implementation approach, as we have to choose a single model $\mathcal{A}$ of AS, in which `choose` evaluates either to $\mathcal{T}$ or to $\mathcal{F}$. More specifically, consider two abstract view va_0 and va_1 that do not agree on the value of `switch`. Then the abstract execution of the body is result correct for only one of these views, as `switch` must evaluate to the same value as `choose` in the valuation in which the execution terminates. So, again, patterns that rely on non-deterministic methods require a different style of reasoning and specification.

Finally, a minor limitation of the approach as we've presented it, is that every method in the implementation must be an operator in the client specification. This is fairly simple to remedy by having a second, implementer-defined algebraic specification that contains the operators that are not relevant to the client.

This ends the treatment from the implementer's perspective. We have defined an implementation approach where the implementer can reason about the method bodies at the abstract level. The next step is to define syntactical restrictions and proof obligations that establish satisfaction for computation, i.e., to switch to the verifier's perspective.

5 Verifier's Perspective

In this section we briefly sketch a proof system for the business layer, i.e., for the computation concern.

Consider a computation implementation $cds \in ClassDefSet$ and a client specification CS. Our goal is to establish that cds $satisfies_c$ CS, i.e., that the implementation satisfies the computation concern. Recall that we have proven in Sect. 3.4 that in that case, the implementation that consists of cds and the problem-independent presentation layer, satisfies CS.

In Sect. 4.1.4, we have proven that satisfaction for the computation concern can be separated into the following concerns.

(1) Hoare-style satisfaction, and
(2) Satisfaction for the `makeCanon` methods.

In section Sect. 4.17, we have proven that Hoare-style satisfaction can be separated into three different concerns. Roughly, these are the following.

(1a) Method call executions must terminate, and
(1b) Method call executions must be side-effect free when viewed abstractly, and
(1c) Abstract method body executions must be result correct.

So, there are four concerns that the verification approach has to establish. We briefly discuss proof systems for each of them.

5.1 Termination

Establishing this concern is much simplified by the requirement that method bodies are finite sequences of abstract statements, due to simple syntax of such statements. The verification of termination is treated in detail in e.g. [MP74, CS02].

5.2 Side-Effect Freeness

Reasoning about side-effect freeness requires reasoning about the *footprint* of an abstract variable: the set of locations that determine its abstract value. Reasoning about the footprint can be done indirectly, using a notion of ownership [Cla01, Mül02].

To ensure side-effect freeness, it needs to be ensured that

(1) the footprints of any two abstract variables are disjoint, and
(2) if an object α_0 inside the footprint is exposed to an object α_1 outside it, then α_1 does not call modifying methods on α_0, does not store α_0, and does not expose α_0 to objects that modify or store it.

These two properties either require intricate reasoning within the ownership system, or fairly severe syntactic restrictions. For example, read statements $v_0 := v_1$ must be disallowed or severely restricted, methods must not return objects referred to by their fields, and modifying methods must not be called on formal parameters. The severeness of the required restrictions reflects the severeness of the inherent absence of cooperation between object in the methodology.

5.3 Abstract Method Body Execution Result Correctness

Recall from Def. 4.2.15 that *IsAbstrResCorBody* formalizes the concern of result correctness of an abstract method body execution. Given a computation implementation $cds \in ClassDefSet$ and a method m, *IsAbstrResCorBody* is separated into three concerns. Recall from Def. 3.1.25 that $GetReturnVar(m, cds)$ returns a `this` variable for void methods and constructors, and returns a `result` variable otherwise. Recall from Def. 4.2.13 that *ReplaceThis* replaces a $\mathtt{this}_C$ variable by an application of the abstraction operator of C to the fields of C. The concerns are as follows.

(1) The abstract execution terminates.
(2) If the abstract execution terminates, then the abstract execution terminates normally.
(3) If the execution terminates normally in a valuation va, then $ReplaceThis(GetReturnVar(m, cds))$ evaluates to the right value.

The second of these concerns can be further separated.

(2a) Variables are defined when they are evaluated.

(2b) Applications are defined when they are evaluated.

Concern (1): termination of the abstract execution. Given the simple syntax and semantics of abstract executions, it is straightforward to prove that every abstract execution terminates.

Concern (2a): variables are defined when evaluated. Concern 2a is established using definite assignment flow analysis. Definite assignment is the property that a variable is assigned a value before it is read. Definite assignment ensures that a variable is defined when it is read, as it is defined by the assignment and cannot be undefined.

Flow analysis for definite assignment is studied in detail for C# in [Fru04] and for Java in [SSB01]. The analysis needed in our setting is much simpler, due to the simple syntax and semantics of abstract executions. Essentially, it needs to be ensured that 1) in every method, every local variable is definitely assigned, 2) in constructors, fields are definitely assigned, and 3) in constructors, all fields of non-primitive sorts are assigned (which ensures that these fields are assigned in the prestate of a non-constructor execution). Note that definite assignment (or a property that is sufficiently similar) can also be established in a more liberal way through the use of a proof system, at the cost of additional complexity.

The Boogie verification system. Concerns 2b and 3 can be established using a automated theorem prover.

As a concrete example, we consider the Boogie verification system [Lei08, BCD$^+$06]. Essentially, the Boogie verification system provides a partial boolean function *Boogie* that has two arguments.

(1) A sequence of abstract statements, annotated with assumptions and assertions.
(2) An algebraic specification.

Roughly, the verification system allows to assume following *verification axiom.* If $\mathit{Boogie}(\overline{s}, AS)$ returns true (i.e., verification succeeds), then for every $\mathcal{A} \in AS$, for every $va \in \mathit{Valuation}$, if $\overline{s}$ is executed from va in the context of $\mathcal{A}$ and all assumptions hold, then all assertions hold as well.

There is one complication. The Boogie verification system uses two-valued logic, rather than the three-valued logic used in this chapter (as two-valued logic is more suitable for verification, the underlying reasoning is based on [GS95]). The impact on the axiom that we sketched above is the following.

(1) Partial operations in AS are interpreted as total operations (which affects the set of models for which the property holds).
(2) The property only holds for a $va \in \mathit{Valuation}$ if va defines all variables that occur in $\overline{s}$.

Concern (2b): applications are defined when evaluated. The main problem in establishing this concern is that the verification system cannot establish properties about undefinedness directly, as the *Boogie* function interprets all operators in the algebraic specification that is provided to it, as total operators.

A solution is to require the specifier to make definedness explicit in the algebraic specification AS of the client specification. To this end, we require AS to be such that for every method $m : S_1 \times S_2 \hookrightarrow S_0$ in the signature, the following holds.

(1) AS contains a special-purpose operator $\texttt{pre}_m : S_1 \times S_2 \rightarrow \mathit{Bool}$
(2) AS contains a *definedness axiom* $\forall s_1 : S_1, s_2 : S_2 \bullet \texttt{pre}_m(s_1, s_2) \Rightarrow m(s_1, s_2) = m(s_1, s_2)$. Roughly, this states that if the precondition holds, then m is defined
(3) Every axiom in AS in which m occurs is prefixed with the precondition.

It is assumed that the primitive operators are defined in this way as well. Then, for every $o(v_0, v_1) \in \mathit{ACall}$ (i.e., for every application of a method or primitive operator), for every method body, before every substatement $v := o(v_0, v_1)$ in the body, assert that $\texttt{pre}_o(v_0, v_1) = \mathit{true}$.

Given this restriction and the verification axiom of the verification system, it is straightforward to deduce that if verification succeeds, then every application that is evaluated during an abstract execution, is defined.

As an example of a precondition, consider the Set example in Hoare's paper on data abstraction, [Hoa72]. Hoare informally states that the size of the set is not greater than 100, and leaves the algebraic specification implicit. In our approach, this requirement is made explicit by a precondition for the insert method (which takes a `Set` and an integer). This precondition states that if the second parameter is not an element of the first, then first actual parameter is a set with a size of at most 99.

Concern (3): return the right result. One way of establishing this property is by using the proof obligation proposed by Hoare in [Hoa72] (slightly adapted to deal with non-void methods). Given a method m in a computation implementation cds, require the following.

(1) If $\mathit{GetFormalParams}(m, \mathit{cds}) = \langle sv_0, sv_1 \rangle$, and $\mathit{ReplaceThis}(sv_0, \mathit{cds}) = t_0$, then assume that $\texttt{x} = m(t, sv_1)$ holds before the execution of the first statement of the body of m (where `x` is a special purpose variable that is not assigned to in the body).
(2) If $\mathit{ReplaceThis}(\mathit{GetReturnVar}(m, \mathit{cds})) = t_1$, then assert that $t_1 = \texttt{x}$ holds in the evaluation context in which the abstract execution terminates.

The downside of this technique is that it cannot be used to prove the implementation of underspecified operators like the operator $\texttt{choose} : C \rightarrow \mathit{Bool}$ that we discussed in Sect. 4.2.7, for which no axioms are defined. A more involved solution that does allow this, requires to prove the following.

(1) For every axiom ax in the specification, for every occurrence of m

in ax, if ax' is like ax, but with the occurrence of m replaced by $\mathit{ReplaceThis}(\mathit{GetReturnVar}(m, \mathit{cds}))$, then ax' holds in the valuation in which the abstract execution of the body terminates.

(2) The body is deterministic.

Details are outside the scope of this chapter.

Note that these approaches only work if we establish that the union of the algebraic specification of the client specification and the algebraic specification of the abstraction operators, is consistent. The reason is that the verification system establishes the desired property for all models of this union of specifications, but it does not establish that there is such a model. Establishing consistency is outside the scope of this chapter. It is studied in more detail in Sect. 6.

5.4 Satisfaction for `makeCanon` methods

Satisfaction for a `makeCanon` method (Def. 4.1.12) can be established by reasoning about the abstract execution of the body of that method. Therefore, it can be established using the Boogie verification system introduced in Sect. 5.3.

This requires that the verification system can deduce properties of `ParseTree`s. To this end, when verifying the body of a `makeCanon` method, the algebraic specification is extended with a sort `ParseTree` and an underspecified operator $\texttt{new}_{\mathit{ParseTree}} : \mathit{Int} \times \mathit{Int} \times \texttt{ParseTree} \times \texttt{ParseTree}$.

There are two main difficulties in the formulation of proof obligations (i.e., asserts).

(1) The abstract value of the `this` object in the evaluation context from which the method body is executed, must be related to the evaluation of the closed application represented by the parse tree that is returned by the execution.
(2) It must be possible to reason axiomatically about the canonicity of a closed application.

The key to a solution for the first difficulty is to extend the algebraic specification AS that is provided to the *Boogie* function with, for every $S \in \mathit{GetClassNames}(\mathit{cds}) \cup \mathit{PrimSig.sorts}$,

(1) An operator $\mathit{toClosedAppl}_S$ that takes a `ParseTree` and returns an S that the parse tree represents, and with axioms that essentially follow the display algorithm. This ensures that if a parse tree p represents a closed explication ca in a valuation va, then $\mathit{toClosedAppl}_S(p) = ca$ in va.
(2) An operator $\texttt{makeCanon}_S$ that takes an S and returns a `ParseTree`, with an axiom $\forall s \in S \bullet \mathit{toClosedAppl}_S(\texttt{makeCanon}_S(s)) = s$. This captures the intended behavior of the $\texttt{makeCanon}_S$ method and is allows to reason about calls to that method from `makeCanon` methods of classes other than C.

Then for an arbitrary $\texttt{makeCanon}_C$ method, the statement sequence that needs to be verified is the body of the method, annotated with an assumption that $\texttt{abstr}_C(\overline{s}) = X$ holds before the abstract execution of the body, and an assertion that $toClosedAppl_C(result) = X$ holds after the execution.

The key to a solution for the second difficulty is to require the canonicity function to be defined in terms of parse trees using an algebraic specification. To this end, the algebraic specification of the client specification must define, for every relevant sort S, a special-purpose partial operator $isCanon_S$ that takes a `ParseTree` and that returns a $Bool$. For example, consider the specification of rationals from Exmpl. 2.1. Note that `Rat` is sort 0 in the specification, and that $\texttt{new}_{Rat}$ is operator 0 of class `Rat`. Then the axiom is the following (where it is assumed that gcd is algebraically specified as the greatest common divider operator).

$$\begin{aligned} &\forall q, p \in \texttt{ParseTree} \bullet \\ &\qquad isCanon_{Rat}(\texttt{new}_{ParseTree}(0, 0, q, p)) = true \\ &\Leftrightarrow \quad gcd(toClosedAppl_{Int}(q), toClosedAppl_{Int}(p)) = 1 \end{aligned}$$

Then for an arbitrary $\texttt{makeCanon}_C$ method, the statement sequence that needs to be verified is the body of the method, annotated with an assertion that $isCanon_C(\texttt{result}_{\texttt{ParseTree}}) = true$ holds after the execution. Note that the this verification can be combined with the previous one, i.e., the body does not have to be verified twice with different annotations, but can be verified once with the combination of the annotations instead.

6 Conclusions

In this chapter we have presented a novel syntax and semantics of client specifications that are based on algebraic specifications (Sect. 2). These client specifications are suited to situations where the client is interested only in the input/output behavior of the implementation. A client specification has two components.

(1) An algebraic specification that consist of a signature that describes the sorts and operators of the client's problem domain, and a set of axioms. Every model of the axioms provides a notion of equality between terms from that signature that is acceptable to the client. Note that if there are no underspecified operators, then these models are isomorphic.
(2) A notion of canonicity that defines which closed applications are acceptable as output.

The semantics of specifications defines a set of black boxes that take a closed application and return an equivalent canonical representation, using one of the notions of equivalence defined by the axioms. We claim that this semantics is intuitive, as it allows the client to think about the implementation as a black box that answers any problem in the specified problem domain by rewriting the problem into its canonical form.

We have also studied class-based implementations of client specifications. We have defined when a set of classes satisfies a client specification (Sect. 3). This is done through the introduction of a generic presentation layer that 1) translates the input of the black box to a statement sequence $\overline{s}$, and 2) displays the result of the execution of $\overline{s}$ as the output of the black box. This presentation layer requires that for every non-primitive sort in the specification, the set of classes contains a class, and for every operator of that sort, the class contains a method. It also requires that the result of a computation is in a format that is suitable for display.

We have presented an implementation approach that formalizes and extends ideas from Hoare's seminal paper on data abstractions ([Hoa72]) at the semantical level (Sect. 4). Effectively, this approach allows the implementer of a method body to use an abstract view of the state when reasoning about the execution of the method body, and requires that methods behave as the corresponding operator from the language. The approach allows for modular reasoning. We conclude that the approach allows to implement methods using efficient in-place algorithms, although it is inherently unsuitable for OO implementations that rely on cooperation between objects that are conceptually at the same level of abstraction.

Finally, we have sketched a proof system that allows to formally verify that an implementation developed using the approach from Sect. 4 satisfies its specification (Sect. 5).

Interesting future work is to formalize the proof system sketched in Sect. 5, and to study the implementation of client specifications that are based on algebraic specifications, while reasoning about (parts of) the implementation using OO specifications.

CHAPTER 3

Cooperation-Based Invariants for OO Languages

This chapter contains the following paper, with minor editorial changes: *Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik Luit. Cooperation-based Invariants for OO Languages. In Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005), volume 160 of ENTCS, pages 225-237. Elsevier, 2007.* [MHKL07a] It is available online.

abstract In general, invariants may depend on the state of other objects. The approach introduced in this chapter allows this for objects of mutually visible classes, in a way that supports modular verification. To this end, dependencies are made explicit by cooperation. In particular, invariants expressing non-hierarchical object relations are supported. Furthermore, an inc-set allows a method to specify explicitly that it does not depend on the validity of a certain invariant. This way, it can be called even when that invariant is violated.

1 Introduction

We present an approach that allows the specification and verification of powerful invariants and supports the modular style of Object-Oriented (OO) development. Such modular development is essential for the component-based paradigm.

A class invariant describes the consistent states of objects instantiated from that class. In general, such an invariant can relate the state of several objects. For instance, in the well-known Observer Pattern [GHJV95], an observer is consistent when its state matches that of its subject. Traditionally, one expects invariants to hold in the pre- and post-states of method executions. Two problems related to such invariants are dealt with by our approach.

The first problem is that invariants that relate arbitrary objects can not be modularly verified. This means we have to restrict these relations. We introduce the concept of cooperation to explicitly express relations in a way that minimizes verification effort.

The second problem is that method calls from inconsistent states are sometimes unavoidable. In the case of the Observer Pattern, when the subject's state has been updated, it calls the inconsistent observers to notify them of the update. In our approach, invariants hold in all pre- and post-states of method executions unless explicitly specified otherwise. To this end, our approach introduces the novel specification construct **inc** (for inconsistent). This construct allows a method to specify explicitly that it does not depend on the validity of a certain invariant.

We discuss the concepts introduced above in more detail in section 1.1, and give an overview of the rest of the chapter in section 1.2.

1.1 Concepts

In this chapter we consider Java-like OO languages. Specifications of OO programs are often based on two fundamental specification constructs, namely on pre- and post-conditions for methods and on class invariants. Class invariants can simplify proofs and specifications as the invariant predicates can be assumed to hold in specific program states. Furthermore, by capturing a desired state relation, an invariant can guide the design of methods that have access to that state.

The *power* of invariants is determined by their *expressiveness*, i.e. by the relations they can describe in a program state, and by their *semantic strength*, which determines in which program states such a relation holds. However, for invariants to be a useful ingredient of specifications, their power has to be balanced against their *manageability.* Manageability is determined by the required *specification effort* (the ease of specification of desired relations) and *verification effort* (the number and complexity of the proof obligations associated with invariants).

Finally, verification should support the modular style of OO development [Mül02]. OO programs have an explicit structure in which classes are grouped into modules (think of components or Java packages). *Modular verification* means a class C is verified using only specifications of classes *visible* to C, where class D is visible to class C when C and D are in the same module or when C's module imports D's module. Furthermore, such verification requires proof that C is *well-behaved.* That is, there are proof obligations not induced by the specification of C itself. This is needed to guarantee that a class meets its specification in any well-behaved context, as the behavior of a class can be affected by classes not visible to it. For instance, consider overriding of methods, which requires a form of behavioral subtyping [LW94, HK01].

1.2 Overview

The next section introduces some basic terminology used in the chapter. Our approach is presented in sections 3, 4 and 5. Section 3 discusses cooperation, section 4 deals with method calls from inconsistent states and section 5 presents proof obligations for the modular verification of invariants. We discuss existing approaches in section 6 and future work in section 7. The last section presents our conclusions.

2 Terminology

This section introduces terminology used in relation with invariants.

C and D identify classes (that is, C and D are typical elements of the set of class names). f identifies a field (also known as instance variable). To simplify the presentation, a subclass is not allowed to define a fieldname that has already been defined in a superclass (known as field shadowing). Extending our approach to allow field shadowing is straightforward.

α identifies an object, i.e. the instantiation of a class (think of α as an address). A *location* $\alpha.f_C$ stores the value of object α's field f defined in class C. Class C is often omitted as it can be inferred from the type of α. g denotes a field access of the form $.f$. For $i \geq 1$, define $\alpha_1\, g_1 \ldots g_i$ inductively in the following way: $\alpha_1\, g_1 \ldots g_i = \alpha_2\, g_i$ when $\alpha_1\, g_1 \ldots g_{i-1} = \alpha_2$. For instance, when $\alpha_1.f_1 = \alpha_2$ and $\alpha_2.f_2 = \alpha_3$, $\alpha_1.f_1.f_2 = \alpha_3$.

In Java-like languages, objects and their contents are accessed by reference expressions. For simplicity, we only consider *references* that consist of a *scope variable* and zero or more field accesses. A scope variable s is either the keyword variable *this*, a method parameter p, a local variable v or a logical variable X. A reference r is an expression of the form $s\, g_1 \ldots g_i$, $i \geq 0$. A *this-reference* t is a reference in which the scope variable is *this*. Scope variable *this* is often omitted in Java-like programs. When $r = s\, g_1 \ldots g_i$ and $1 \leq j \leq i$, reference $s\, g_1 \ldots g_j$ is called a *subreference* of r (note that r is a subreference of r but, for technical reasons, s is not). While method selection is dynamic in Java-like languages, field selection is static. $\mathit{statType}(r)$ yields the static type of reference r. All references in this chapter are assumed to be type-correct (and thus to have a static type). $r \triangleright t$, the concatenation of reference r and this-reference t, is defined by $r \triangleright (\mathit{this}\ g_1 \ldots g_i) \overset{\text{def}}{=} r\, g_1 \ldots g_i$. For instance, $\mathit{this}.f_1.f_2 \triangleright \mathit{this}.f_3.f_4 = \mathit{this}.f_1.f_2.f_3.f_4$. When $\alpha_1\, g_1 \ldots g_i = \alpha_2$, we say this-reference $\mathit{this}\ g_1 \ldots g_i$ refers from α_1 to α_2.

We call the subset of predicates that are allowed as an invariant *invariant predicates.* We use R as typical element of such predicates. In this chapter, we do not consider invariants that quantify over objects (see section 7). Effectively, this means every reference in an invariant is a this-reference. When this-reference t occurs in invariant predicate R, we call every subreference of t a *supplier reference* of R. *sup*(R) yields the set of supplier references of invariant predicate R. When $t.f \in$ *sup*(R) and *statType*$(t) = C$, we say that R *depends* on field f of class C.

A program state is called a *visible state* if it is a pre- or post-state of a method execution [MPHL06]. The traditional semantics of invariants, in which all invariants hold in all visible states, is referred to as the *visible state semantics*.

The OO syntax in the examples is assumed self-explanatory. In the examples we ignore the orthogonal issue of how to specify what a method leaves untouched [Lei95, Rey02]. This problem is alleviated, but not solved by invariants. Furthermore, all fields are considered publicly available at the specification level. Hiding fields at this level [Par72, Mül02] is a separate concern. See for instance [LBR06, Lei95, Mül02] for specification language support for information hiding.

3 Cooperation

This section introduces the concept of cooperation. Cooperation entails that a field specifies explicitly, through the specification construct **coop**, which invariants might be invalidated when the field is assigned to. Cooperation restricts dependencies to those that are mutually visible (that is, when an invariant in class C depends on a field of a class D, D is visible to C and vice versa). This restriction enables modular verification of invariants given the visible state semantics. Furthermore, this explicit specification greatly reduces the total verification effort required.

Of course, the most expressive invariants are those that can depend on arbitrary fields. However, in that case any assignment can possibly invalidate such an invariant. That means that, given the visible state semantics, any pair of an invariant and a method has to be verified. In [HK00], whole-program analysis is used to verify these pairs. Unfortunately, such an approach does not support modular verification (section 1.1). When modularly verifying a class that defines an invariant, it can not be proven that methods of other classes preserve the invariant as their implementation is not available. Furthermore, when modularly verifying whether a method is well-behaved, it can not be proven that it preserves all invariants of all classes, as not all classes are visible. Therefore, a restriction of dependencies is unavoidable.

In our approach, a class can define multiple, named invariants. An invariant is defined in a class in the following way:

inv I **def** R

That is, an invariant has a *name* I and a *definition* R (which is an invariant predicate). I_C identifies the invariant with name I defined in class C. *def*(I_C) yields the invariant predicate R that is the definition of invariant I_C. To simplify the presentation, we do not allow a subclass to define an invariant name that has already been defined in a superclass (which we call invariant shadowing). Extending our approach to allow invariant shadowing is straightforward.

```
class Node {
  Node next coop I(this), J(next);
  Node prev coop J(this), I(prev);
  inv I def $this.next = null \lor this = this.next.prev$;
  inv J def $this.prev = null \lor this = this.prev.next$;
  Node() {this.prev := null; this.next := null;}
  void insertAfter(Node n) {
  pre:   $this.next = null \land this.prev = null \land n \neq null \land n.next = X$
  post:   $n.next = this \land this.next = X$
    this.prev := n; this.next := n.next; n.next := this;
    if (this.next ≠ null) { this.next.prev := this; }
  }
}
```

Example 3.1: Doubly Linked Nodes

An invariant defined in a class must hold for every instantiation of that class. To differentiate between these instantiations, we introduce *instantiated invariants*. Instantiated invariant $I_C(\alpha)$ identifies the instantiation of invariant I_C on object α. To identify instantiated invariants in the specification language, *reference invariants* are used. A reference invariant $I_C(r)$ identifies instantiated invariant $I_C(\alpha)$ when reference r refers to object α. We call r the *dependent reference* of $I_C(r)$. Classname C is often omitted in instantiated or reference invariants as it can be inferred from α's type or r's static type.

When, in a given state, a change of the value of location $\alpha.f$ can invalidate instantiated invariant $I(\alpha)$, we call $I(\alpha)$ *vulnerable* to $\alpha.f$ in that state. For instance, in example 3.1, when α is a *Node* object, $J(\alpha)$ is vulnerable to $\alpha.prev$ in a state in which $\alpha.prev$ stores null and $\alpha.prev.next$ doesn't store α.

With every location, our approach associates a set of instantiated invariants that may be vulnerable to that location. While this requires some additional specification effort, it greatly reduces overall verification effort. In a well-behaved method, the instantiated invariants associated with locations the method assigns to are reproven, but nothing has to be proven for invariants not in this set (see section 5). Note that this benefits from having multiple, named invariants. Furthermore, this means that the more accurate the set is, the less verification effort is required. Properties of locations are specified by properties of fields. In our approach, a field is specified in the following way:

modifier T f **coop** *coop-set*

The access modifier *modifier*, type T and name f of the field are all standard. The *coop-set* is a set of reference invariants. We say the field *cooperates* with these reference invariants. For instance, in example 3.1, field *next* of class *Node* cooperates with *I(this)* and *J(next)*. When class D defines a field f and C is a subclass of D, $coop(f, C)$ yields the coop-set of field f of class D.

Now consider a location $\alpha_1.f_C$. When this-reference t refers from α_1 to α_2 in a given state, and $I(t) \in coop(f, C)$, we say $\alpha_1.f_C$ cooperates with $I(\alpha_2)$ in that state.

The *cooperation obligation* below ensures that an instantiated invariant is only vulnerable to locations that cooperate with it. Only invariants that meet this obligation are admissible. An invariant meets this obligation when it can be written as a disjunction of invariant predicates $R_1 \vee \ldots \vee R_i$[1], where $dco(R_j, I_C)$ holds for every disjunct R_j. $dco(R_j, I_C)$ holds (*d*isjunct R_j *co*operates with I_C) when, for every supplier reference $t_1.f$ of R_j, there is a this-reference t_2 such that field f on which the invariant depends cooperates with $I(t_2)$, and such that $R_j \Rightarrow this = t_1 \triangleright t_2$. This implication guarantees cooperation with the appropriate instantiated invariant when invariant predicate R_j holds.

Definition 3.1 (cooperation obligation)**.** *There exists a set of invariant predicates R_1 to R_i such that*
$def(I_C) \Leftrightarrow (R_1 \vee \ldots \vee R_i)$*, and*
$\forall j : 1 \leq j \leq i : dco(R_j, I_C)$

, where $dco(R, I_C) \overset{def}{=} \forall t_1, f : t_1.f \in sup(R) : (\exists t_2 :: I(t_2) \in coop(f, statType(t_1))$ *and* $R \Rightarrow this = t_1 \triangleright t_2)$

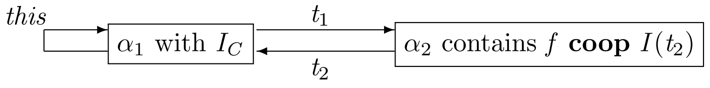
Node's invariant J, for instance, meets the cooperation obligation. When R_1 is $this.prev = null$ and R_2 is $this = this.prev.next$, $def(J_{Node}) \Leftrightarrow R_1 \vee R_2$. R_1 has a single supplier reference, *this.prev*, which cooperates as the field *prev* has *J(this)* in its coop-set and as $this = this \triangleright this$ is trivially true. R_2 has supplier references *this.prev* and *this.prev.next*. The cooperation argument for *this.prev* is the same as above. Consider *this.prev.next*. The static type of *this.prev* is *Node*. Field *next* of class *Node* cooperates with *J(this.next)*, as *J(this.next)* $\in coop($*next*, *Node*$)$. Since $this = this.prev.next \Rightarrow this = this.prev \triangleright this.next$, the cooperation obligation is met.

Treating individual disjuncts in the cooperation obligation allows the coop-set to be more accurate than when the entire invariant is treated at once. Also, it allows for a weaker obligation, which means that more invariants are admissible. The static set of supplier references of an invariant defines a dynamic set of *suppliers* to an instantiated invariant: location $\alpha_1.f$ is a supplier to instantiated invariant $I_C(\alpha_2)$ in a given state when I_C has a supplier reference $t.f$ and t refers from α_1 to α_2 in that state. In any state, the set of locations to which $I_C(\alpha)$ is vulnerable is a subset of its set of suppliers. The obligation ensures that suppliers to a *valid* disjunct of the invariant cooperate. The invariant is not vulnerable to a supplier that only occurs in invalid disjuncts, as assignment to such a supplier might re-validate the disjunct, but can not invalidate it. Why the obligation ensures cooperation with the right object is illustrated by the picture and text below.

By definition, when $t_1.f \in sup(def(I_C))$ and t_1 refers from α_1 to α_2, $\alpha_2.f$ is a

[1] $\vee$, $\Rightarrow$ and $\Leftrightarrow$ are symbols from the underlying predicate logic, $\exists$ and $\forall$ are not

supplier to $I_C(\alpha_1)$. $\alpha_2.f$ cooperates with $I_C(\alpha_1)$ when there is a this-reference t_2 such that $I_C(t_2) \in \mathit{coop}(f, \mathit{statType}(t_1))$ and such that t_2 refers from α_2 to α_1. The first is guaranteed explicitly by the obligation. The second is guaranteed by $R \Rightarrow \mathit{this} = t_1 \triangleright t_2$. As *this* refers from α_1 to α_1 by definition, $\mathit{this} = t_1 \triangleright t_2$ guarantees $t_1 \triangleright t_2$ also refers from α_1 to α_1. When t_1 refers from α_1 to α_2 and $t_1 \triangleright t_2$ refers from α_1 to α_1, t_2 must refer from α_2 to α_1.



Besides mutual visibility, cooperation requires the existence of dependent references (t_2 in the example above). That is, the invariant's class must be reachable from the supplier's class. However, no expressive power is lost due to the additional requirement, as auxiliary state (i.e., state only used for the purpose of specification and verification) can be used when needed.

A formalization of the obligations needed to ensure that invariants hold when they should is postponed until section 5.

4 Calls from Inconsistent States

By means of the novel specification construct **inc** that is introduced in this section, methods can make explicit that they will *not* rely on certain invariants of certain objects. It is allowed to call these methods from *inconsistent* states where these invariants do not necessarily hold. That is, **inc** allows the specifier to pinpoint visible states in which the visible state semantics is too strong and weakens it for those specific states only.

Sometimes, initialization or update of an invariant is impossible without a method call as it requires access to a set of fields that can not be accessed by any single method. Consider example 4.1. Invariant *I* of class *Left* relates *Left*'s field *rVal* to class *Right*'s field *val*. *Right*'s method *setVal* assigns to field *val*. As shown by the specification of *val*, this might invalidate invariant *I* of the *Left*-object referred to by *Right*'s field *l*. However, *setVal* can not assign to *Left*'s *rVal* to restore the invariant. Instead, it has to call *Left*'s method *sync* from an inconsistent state. The challenge is to allow such programs without having to weaken the invariant.

As a solution, we propose a weakening of the visible state semantics based on the idea that the most intuitive invariants will *almost* always hold. Therefore, we treat the cases where they do not as the exceptions that require additional effort and again rely on a form of cooperation. To this end, method specifications can be extended by means of the specification construct **inc**:

inc: *inc-set*

The *inc-set* is a set of reference invariants the method will not rely upon to hold in its precondition. *Left*'s method *sync*, for instance, makes explicit it does not rely on *I(this)*. A method inherits the inc-set of a method it overrides and can extend

```
class Left {
 protected Right r coop I(this),
                        J(r);
 protected int val coop J(r);
 protected int rVal coop I(this);
 inv I def
  this.rVal = this.r.val ∧ this.r.l = this;
 Left() {
  Right ri := new Right(this);
 }
 void setRight(Right ri) {
 inc: I(this), J(ri)
 pre: this = ri.l ∧ this.r = null  ∧
 this.val = X ∧ this.rVal = ri.val
 post: I(this) ∧ this.r = ri ∧ this.val = X
  this.r := ri;
 }
 void sync() {
 inc: I(this)
 pre: this.r.l = this
 post: I(this)
  int i := this.r.getVal();
  this.rVal := i;
 }
}
```

```
class Right {
 protected Left l coop I(l),
                       J(this);
 protected int val coop I(l);
 protected int lVal coop J(this);
 inv J def
  this.lVal = this.l.val ∧ this.l.r = this;
 Right(Left le) {
 inc: I(le)
 pre: le.r = null ∧ le.val = 0 ∧ le.rVal = 0
 post: I(le)
  this.l := le;
  this.l.setRight(this);
 }
 void setVal(int newVal) {
 post: this.val = newVal
  this.val := newVal;
  this.l.sync();
 }
 int getVal() {
 inc: I(this.l)
 post: return = this.val
  return := this.val;
 }
}
```

Example 4.1: Left/Right (int fields initialize to 0, reference fields to null)

it if needed. Our semantics of invariants is captured by the following definition.

Definition 4.1 (invariant property). *A program has the* invariant property *iff for every execution sequence of the program the following holds:*

- *in the prestate of a method execution, the set of invalid instantiated invariants is a subset of the set of instantiated invariants identified by the reference invariants in that method's inc-set.*
- *In the poststate of a method execution, the set of invalid instantiated invariants is a subset of the set of invalid instantiated invariants in the prestate of that method execution.*

Note that this semantics stays close to the visible state semantics. All invariants hold in the pre- and post-state of a method with an empty inc-set. Only methods that are involved in the initialization or update of certain invariants might need a non-empty inc-set. For these methods, an important monotonicity-property is maintained: every invariant that holds in the prestate of a method execution, holds in the poststate of that method execution (even if it is in the method's inc-set).

inc can be used when the invariant's class is visible to the class whose method is to be called. When this is not the case, we have to rely on the more traditional technique of weakening the invariant using a *flag*. A flag is a boolean condition b (for instance an auxiliary boolean field of the class) that signals whether or not the object is consistent. When the desired invariant predicate is R, define the invariant as $b \Rightarrow R$ instead. Then, when b is false, the invariant might be vulnerable to the flag, but is not vulnerable to any other location. However, a consequence of this technique is that the relation between the invariant and object consistency is reversed. Instead of the object being consistent when the invariant holds, the invariant holds when the object is consistent.

While very flexible, using a flag means verification or specification effort is required whenever the invariant is to be relied on. In particular, when specifying a method it has to be decided whether or not it needs the invariant. In contrast, **inc** requires deciding which methods are involved in initialization or update of certain invariants. This leaves more implementation freedom. That is, the inc-set is typically empty, which means every invariant may be relied upon. **inc** also works more naturally with subclassing. An overriding method in a subclass can rely on invariants the superclass method does not rely on, for instance those added by the subclass. Such an overriding method can also be used in the update or initialization of additional invariants as extending the inc-set in the subclass method does not affect the superclass method or any of its users.

5 Proof Obligations

This section presents proof obligations suitable for the modular verification of invariants. These proof obligations utilize the **coop** and **inc** constructs introduced in the previous sections.

In the formulation of the proof obligations, it is assumed every method is fully annotated, i.e. that every statement x has a precondition identified by P_x and a postcondition identified by Q_x. The following theorem is established (an unpublished proof exists for a sequential Java-like language):

Theorem 5.1. *When a program is correctly annotated and when the proof obligations presented in this section are met, the program has the invariant property.*

In section 3, we have defined when a field cooperates with a reference invariant. Reference invariants are added to the proposition language to describe *how* fields cooperate. A reference invariant holds when there is no referenced object, or when the referenced object's instantiated invariant holds.

$$I_C(r) \stackrel{\text{def}}{=} \mathit{noObj}(r) \vee \mathit{def}(I_C)[r/\mathit{this}]$$

$P[r/\mathit{this}]$ is the predicate P with all occurrences of *this* replaced by r. $\mathit{noObj}(r)$ is defined as:

$$noObj(s\ g_1 \ldots g_i) \overset{\text{def}}{=} \exists j : 0 \leq j \leq i : s\ g_1 \ldots g_j = null$$

Method calls whose return value is assigned to a field with a non-empty coop-set are disallowed (i.e., should be broken up into two statements). This avoids the complication of invariants that are invalidated by a method return context switch. The extension is straightforward.

Simply put, the *dependency obligation* ensures that when an assignment invalidates an invariant: 1. it is re-proven in a later state, and 2. until it is re-proven, no method that relies on the invariant is called. This simple notion is complicated by two issues: 1. one needs to keep track of the instantiated invariant that might be invalid, and 2. state(ment) ordering is complicated by branching and looping. To simplify the presentation, the latter complication is avoided by disallowing method calls in branches and loops. Allowing such calls is a fairly straightforward extension. Then, the body of a method M is a sequence *body*(M) of method calls and local code blocks (*lcb*s) of statements that are not method calls. We write $x < y$ when x occurs before y in *body*(M). *calls*(M) and *lcbs*(M) yield the sequence of method calls and lcbs in *body*(M), respectively. The pre- and postcondition of an lcb are the precondition of the first, and the postcondition of the last statement of the lcb, respectively.

The dependency obligation is given below. It uses a logical variable X to 'freeze' a reference to the object whose invariant might be invalidated by the assignment. The invariant must be re-proven in a postcondition Q_z after the assignment. No method called between the assignment and the postcondition Q_z may rely on the invariant, which is guaranteed by *inInc* defined below. To improve readability of the obligations in this section, implies binds weakest, and all free variables on the left-hand side of an implies are considered universally quantified over the implication.

Definition 5.1 (dependency obligation)**.**

```
if      $x \in lcbs(M)$ contains an assignment $s$ to $r.f$, and
        $I(t) \in coop(f, statType(r))$,
then    $\exists X ::$ $P_s \Rightarrow X = r \triangleright t$, and
              $\exists z : z \in body(M)$ and $x \leq z$ :
                $Q_z \Rightarrow I(X)$, and
              $\forall y : y \in calls(M)$ and $x < y \leq z : inInc(y, I, X)$
```

inInc requires several other definitions. *inc*(M) yields the inc-set of method M. e identifies a side-effect free expression. When statement y is a method call on $r.m(e_1, \ldots, e_i)$, *callee*(y) yields the fully qualified name M of the method determined by m and *statType*(r). When *callee*(y) has formal parameters p_1 to p_i, actualization *act*(r', y) equals $r'[r, e_1, \ldots, e_i / this, p_1, \ldots, p_i]$, i.e. *act*$(r', y)$ refers to the same value in the prestate of method call y as r' refers to in the prestate of called method M. *inInc*(y, I, r_1) holds when there is a reference invariant in the inc-set of *callee*(y) that identifies the same instantiated invariant as identified by $I(r_1)$ in the prestate of the call.

$$\mathit{inInc}(y, I, r_1) \overset{\text{def}}{=} \exists r_2 : r_2 \in \{r \mid I(r) \in \mathit{inc}(\mathit{callee}(y))\} : P_y \Rightarrow r_1 = \mathit{act}(r_2, y)$$

We illustrate the use of the dependency obligation. Consider method *setVal* in example 4.1. It consists of a lcb and a method call. The lcb contains (consists of) an assignment to *this.val*. As *I(this.l)* $\in$ *coop*(*val*, *Right*), the left-hand side of the dependency obligation is met. That means the following is required of the annotation of the method. There must be a logical variable X such that $X = \mathit{this} \triangleright \mathit{this.l}$ holds in the precondition of the assignment. Furthermore, $I(X)$ must hold in the postcondition of either the assignment or that of the method call. Looking at the example, the first will not be the case but the second will. In that case, $\mathit{inInc}(\mathit{this.l.sync}(), I, X)$ must hold as well. As the inc-set of *Left*'s method *sync* contains *I(this)* and $\mathit{act}(\mathit{this}, \mathit{this.l.sync}())$ equals *this.l*, *inInc* requires that $X = \mathit{this.l}$ holds in the prestate of the call.

An invariant in a method M's inc-set may not be assumed to hold. The *inconsistency obligation* ensures that no method called by M relies on such an invariant unless it has been re-proven before the call. P_M identifies the precondition of the first statement in $\mathit{body}(M)$.

Definition 5.2 (inconsistency obligation)**.**

$$\begin{array}{ll} \textit{If} & I(r) \in \mathit{inc}(M), \\ \textit{then} & \exists X :: \\ & \quad P_M \Rightarrow X = r, \textit{ and} \\ & \quad \forall x : \quad x \in \mathit{calls}(M) : \\ & \qquad\qquad \mathit{inInc}(x, I, X), \textit{ or} \\ & \qquad\qquad \exists y : y \in \mathit{body}(M) \textit{ and } y \leq x : P_y \Rightarrow I(X) \end{array}$$

This only leaves the issue of initialization. In general, an invariant I defined by a class C will not hold in the prestate of a constructor of the class. In Java-like languages, the first (possibly implicit) statement in a constructor is a call to a superclass constructor. In the Java semantics, the dynamic type of the this-object in the prestate of this call is either C or a subclass of C. Due to dynamic method binding, a method call in a superclass constructor might execute a method of C. Due to the semantics of invariants, this method assumes all objects are consistent while in fact this is not the case. There is no modular way to prevent this scenario without restricting either invariants (to hold by default) or the programming language.

Such a restriction is avoided by assuming constructor behavior more akin to that of C++. We assume that in the prestate of a constructor of class C, the dynamic type of the this-object is Object. After the (possibly implicit) superclass constructor call, there is an implicit statement that changes the dynamic type of the this-object to type C. Note that, when D is the superclass of C, the type of the this-object is D in the poststate of the superclass constructor call.

first_M identifies the first statement of method M. The *construction obligation* ensures invariants are initialized by constructors and are not relied upon before initialization:

Definition 5.3 (construction obligation).

If method M *is a constructor of class* C *and* C *defines an invariant* I,
then $\exists y : \; y \in \mathit{body}(M)$ *and* $\mathit{first}_M < y$:
$Q_y \Rightarrow I(\mathit{this})$, *and*
$\forall x : x \in \mathit{calls}(M)$ *and* $\mathit{first}_M < x \leq y : \mathit{inInc}(x, I, \mathit{this})$

6 Related work

As the previous sections have shown, one solution to the problem of vulnerability (introduced in section 3) is to ensure that all invariants vulnerable to a location are visible when this location is updated. The drawback is that this requires dependencies to be mutually visible. Without this restriction, invariants that have been invalidated by a method can not always be expected to be restored before the end of that method. That means that in the prestate of a method execution, an unknown set of invariants will not hold. Existing approaches without the mutual visibility restriction use a notion of *ownership* to be able to express which invariants *do* hold. Ownership means an object has control over updates of the objects it owns.

The ownership approach presented in [MPHL06] relies on an ownership type system [Mül02, CPN98]. In such a type system, every object has a context of owned objects which are reachable only through their owner. The approach allows invariants to depend on fields that are (transitively) owned. The semantics of invariants is such that invariants of objects outside the context can not be assumed to hold in pre- or post-conditions of methods. Methods are in general not allowed to call methods on objects outside their context to prevent that an invariant is assumed to hold when it does not.

The Boogie approach [BDF$^+$04, LM04] uses a dynamic notion of ownership. The main advantage is that this allows ownership transfer. Boogie equips every invariant with a flag (see section 4). This flag can only be updated by special-purpose statements that guarantee the invariant holds when the object is made consistent. Furthermore, objects make explicit that an invariant of a consistent object is vulnerable to their state. Updates of such objects are forbidden. In the Boogie approach, the semantics of invariants is such that in every state in which an object is consistent (i.e., in which the flag holds), its invariant holds.

Ownership is a concept that is natural to OO development. However, ownership relations are non-cyclic by nature, and control over updates of the locations to which an invariant is vulnerable is not always possible (or desired). Therefore, it is not suitable for non-hierarchical situations like the Observer Pattern or the examples in this chapter.

The visibility approach in [MPHL06], which generalizes work in [LM04], has mutual visibility as the only requirement. However, as argued in section 3, overall verification effort is greatly reduced when it is made explicit which instantiated invariants might be vulnerable to a location. Furthermore, it is argued that no expressive power is lost in the process.

The most closely related work is that of the friendship approach [BN04]. The friendship approach *requires* auxiliary state to relate locations to vulnerable invariants and uses special-purpose statements to prevent unwanted updates. This chapter shows how this additional specification layer can be avoided at the cost of some additional verification effort. The main difference between the two approaches, however, is in the semantics of invariants. The friendship approach has been developed to complement the Boogie approach, and uses the same semantics. This means that in cases where a flag is unavoidable, their solution is elegant. However, the disadvantages of a flag-based solution as discussed in section 4 apply.

[PCdB04] introduces an extension of the friendship approach that supports static invariants that quantify over objects and discusses uses for such invariants.

7 Future work

We see cooperation-based and ownership-based approaches as complementary. The friendship approach [BN04] shows the benefits of such a combination. Complementing our cooperation-based approach with a notion of ownership is considered a priority.

More complex forms of cooperation can be achieved by supporting coop- and inc-sets like $\{I(X) \mid P\}$, where P is a predicate on X (that is, when $P[r/X]$ holds in a given state, $I(r)$ is in the set in that state). Perhaps quantification over objects in invariants can also be supported this way.

Some invariants should not be publicly accessible as they expose hidden information. In such cases, the definition of the invariant could be made private to the class that defines it. As an invariant's name does not expose information, it can still be used in coop-sets. An interesting side-effect is that this can achieve that field f has public read-access, but private write-access. For instance, consider a publicly accessible field f whose coop-set contains $I(\mathit{this})$, where invariant I is defined by **inv** I **private def** *true*.

Finally, consider the Observer Pattern again. As the concrete observer is not visible to the concrete subject (which is exactly the purpose of the abstract classes in the pattern), our approach does not allow the observer's invariant to depend on the subject's state. However, implementations of the pattern that do not use abstract classes [BN04] are supported. Specification of the full pattern requires a notion of an abstraction of an invariant. Perhaps the abstract predicates of [PB05] or dynamic contracts of [HK03] can provide such a notion.

8 Conclusions

Given a strong semantics of invariants, modular verification is not possible when invariants can arbitrarily depend on fields. The approach presented in this chapter allows dependencies that are mutually visible. In particular, this allows invariants over non-hierarchical object structures. The approach allows for the separation of two concerns that are often entwined, namely that of vulnerability and that of method calls from inconsistent states. The dynamic vulnerability relations are made explicit with the cooperation construct **coop**, which reduces verification effort. The semantics of invariants is such that every object is consistent in every visible state unless explicitly specified otherwise by means of the novel construct **inc**. This semantics is flexible, yet captures the intuitive notion of invariants. Finally, the proof obligations that we have presented enable the modular verification of invariants.

Acknowledgements. We would like to thank Mike Barnett and Erik Poll for useful comments on earlier versions of this chapter.

CHAPTER 4

Invariants for Non-Hierarchical Object Structures

This chapter contains the following paper, with minor editorial changes: *Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper and Erik J. Luit. Invariants for Non-Hierarchical Object Structures. In Proceedings of the 9th Brazilian Symposium on Formal Methods (SBMF'06), volume 195C of ENTCS, pages 211–229. Elsevier, 2008* [MHKL08a]. It is available online. Thanks to Rob Verhoeven for pointing out a typographical mistake in the published version.

abstract We present a Hoare-style specification and verification approach for invariants in sequential OO programs. It allows invariants over non-hierarchical object structures, in which update patterns that span several objects and methods occur frequently. This gives rise to invalidating and subsequent re-establishing of invariants in a way that compromises standard data induction, which assumes invariants hold when a method is called. We provide specification constructs (inc and coop) that identify objects and methods involved in such patterns, allowing a refined form of data induction. The approach now handles practical designs, as illustrated by a specification of the Observer Pattern.

1 Introduction

Traditionally, an invariant is a consistency property of the data of a single object, enabling reduced specification effort. Data induction is, essentially, the observation that if an object is only approached through its methods, a property is invariant if it is established by the constructors and preserved by the methods [LW94]. But in practice, invariants may range over more than one object. Furthermore, an invariant is sometimes invalid at a method call, in particular, when this method is called to re-establish the invariant. Obviously this method can not rely on the invariant. Therefore, data induction must be refined. Some approaches successfully exploit the dependency hierarchy between objects [Mül02, LM04]. However, there are natural OO designs that are inherently non-hierarchical. A case in point is the Observer Pattern [GHJV95]. The approach in [BN04] allows for non-hierarchical invariants, but drops data induction. We present an alternative that retains it.

First, we introduce the specification construct **inc** that makes explicit that a method preserves, but does not rely on, certain invariants of certain objects. We extend results from [MHKL07a]: instead of the previously used fixed set of object references, predicates are introduced to describe a set of objects involved. We argue that the additional flexibility offered by **inc** is essential in the use of invariants over non-hierarchical object structures. Second, we introduce the **coop** construct that specifies which invariants might be invalidated when a field is assigned to. This enables verification of invariants even when their definition is not visible. In particular, this supports modular development. We extend previous results with predicates to describe the set of objects involved. Third, we remove a limitation on method calls in while and if statements. Finally, the consequences of these extensions are incorporated in a proof system. More invariants are admissible and more implementations can be verified than before. In fact, whereas the approach previously could only be used for somewhat tailor-made examples, the extensions enable to specify the inspiration for the approach: the Observer Pattern.

Following this introduction, section 2 introduces invariants. Section 3 introduces the **inc** construct, section 4 introduces the **coop** construct and section 5 contains the formalization. Section 6 describes related and future work. Section 7 concludes the chapter.

2 Invariants in OO development

OO programs are structured by a decomposition into classes, which group related data and methods operating on this data. A method of one class can use (objects of) another class in its implementation. A *proper user* of a class C is a method that does not contain references to fields defined in C, but only interacts with objects of class C via C's methods. Note that our proof technique does not require restriction to proper use.

We say a method M *preserves* a property if, when the property holds when M is called, it also holds when M terminates. An *invariant property* of an object is established by the object's constructor and preserved by all the object's methods. For every `Book` object in the (Java-like) example in Figure 2.1, "`title` is not null" is an invariant property. Proper users of `Book` do not invalidate the invariant property of a `Book` object. For every `Book` object, once the invariant property is established by the constructor, it always holds. The program capitalizes on this.

```
class Book {
  String title;
  Book() { this.title := "unknown"; }
  boolean hasTitle(String t) {
    return this.title.equals(t);
  }
  int getTitle() { return this.title; }
  void setTitle(String newT) {
    if (newT != null) { this.title := newT; };
  }
}
```

```
class UI {
  void showHasTitle(Book b, String t) {
    if (b != null) {
      boolean ht := b.hasTitle(t);
      //show boolean ht on the screen
  }
  void showTitle(Book b) {
    if (b != null) {
      String s := b.getTitle();
      String s := s.concat(" is the title"))};
      //show String s on the screen
  }
}
```

Figure 2.1: Book/UI, invariant properties and their use

Book's `hasTitle` doesn't check if the `title` is non-null. `UI`'s method `showHasTitle` appends to the result of `getTitle` without checking if it is null.

A Hoare-style method specification contains a pre- and a post-condition in terms of the data of its class. For each method M, it should be verified that it terminates normally and that its postcondition holds (we do not consider exception handling). A verifier can assume that the precondition holds when M is called. When verification of M fails without the assumption that a certain property holds when M is called we say (the verification of) M *relies on* that property. In Figure 2.1, `Book`'s method `hasTitle` and `UI`'s method `showTitle` rely on the invariant property of `Books` `this` and `b`. If a property is specified as a precondition, a user (like method `showHasTitle`) must prove that the property holds before a call. Thus any such user relies on the property as well. Specification of a property in the precondition of these users means *their* users must prove the property holds before a call, and so on. The property propagates throughout the program's specification. An invariant property can be specified with an *invariant.* Consider the following aim:

Aim: *The verifier of a method M that relies on an invariant I can*

1. *assume that I holds when M is called, and*
2. *deduce if a method called by M preserves I.*

When the aim is met, propagation of invariant properties is prevented, significantly reducing specification overhead. Furthermore, the code is more flexible, as a re-implementation of a method can rely on a different set of invariants without affecting users. Besides these advantages, invariants allow the specification of data consistency properties and behavioral properties to be separate concerns. This makes communication of such properties much easier [Mey97]. Finally, they support the specification of a class in terms of an abstraction of its data [Hoa72] (see section 4).

The Book/UI example suggests that the assumptions in the aim above are sound when an invariant is 1) established by the constructor of an object and 2) preserved

```
class C {
  int i,j;
  inv this.i < this.j;
  int m() { this.i := this.calcVal(); this.j := this.calcVal(); } //constructor and calcVal omitted
}
```

Figure 2.2: the call-back problem

by every method in the program. Due to what is known as the *call-back problem*, this is not the case. The call-back problem is illustrated in Figure 2.2. As the example shows, an invariant is specified as a predicate on the logical variable **this**, that represents the object the invariants applies to. Assume that method `calcVal` always returns a value that is greater than `this.i`. Assume that the constructor of a C object establishes its invariant and that every other method in the program (including `calcVal` and `m`) preserves it. Then the invariant of `C` object `this` still might not hold when the second call to `calcVal` in `m` is made (as the assignment to `this.i` might invalidate it).

More generally, a method M may temporarily invalidate an invariant. When M calls another method before the invariant is re-established, a method that relies on the invariant might be called while the invariant does not hold. The most straightforward solution to the call-back problem is to require that any invariant that is invalidated by a method is re-established before a method call is made. These observations lead to the following theorem, whose conclusion clearly meets the aim.

Theorem 2.1 (data induction).
If, for any invariant I of any object,

1) the constructor of that object establishes I, and

2) all methods in the program preserve I, and

3) no method is called while I is invalid

Then, for any method M, for any invariant I of any object,

1) unless M is the constructor of that object, I holds when M is called, and

2) I holds when a method called by M terminates

Proof. Proof (by induction on the length of execution sequences) is straightforward □

Execution of a program that has been proven correct with the *classical technique* (described and proven sound in [MPHL06]) meets the premises of Theorem 2.1. In the classical technique only *local* invariants are admissible. An invariant is *local* if it only depends on the fields of the object it applies to (i.e., the predicate that defines the invariant only contains references of the form **this**.f).

```
class Member {
  Book loaned;
  inv MI def this.loaned ≠ null ⇒
             this.loaned.loanedTo = this;
  void loan(Book b) {
    pre:  this.loaned = null ∧
          b.loanedTo = null;
    post: this.loaned = b;
    impl: this.loaned := b; b.loanTo(this);
  } //other fields and methods omitted
}
```

```
class Book {
  Member loanedTo;
  inv BI def this.loanedTo ≠ null ⇒
             this.loanedTo.loaned = this;
  void loanTo(Member m) {
    inc:  MI(m);
    pre:  this.loanedTo = null ∧ m.loaned = this;
    post: this.loanedTo = m;
    impl: this.loanedTo := m;
  } //other fields and methods omitted
}
```

Figure 3.1: Example of a non-local invariant

3 Non-local Invariants

3.1 The specification construct inc

We call a specification *feasible* when there is an implementation of this specification that can be verified. This section shows that many natural OO designs that include non-local invariants are infeasible due to the third premise of Theorem 2.1. The specification construct **inc** is presented as a solution to this problem. It allows a method M to specify that M preserves, but does not rely on certain invariants of certain objects. It is also argued that many non-hierarchical designs are *only* feasible in a specification language that includes a construct like **inc**. There is a hierarchy between two objects if, when a method M is called on one, no method can be called (or field accessed) on the other until M terminates.

Non-local invariants are natural in many OO designs. This is illustrated by Figure 3.1, which could be part of a library management system. The invariants are named to allow one to distinguish between different invariants of a class. We ignore the orthogonal issue of how to specify what a method leaves untouched [Lei95, Mül02, Rey02]. This problem is alleviated, but not solved by invariants. We assume that relevant changes are reflected in the method's postcondition. Due to the third premise of Theorem 2.1, the design in Figure 3.1 is infeasible (given proper use). The assignment to `loaned` invalidates the invariant of `Member this`. To re-establish the invariant, field `loanedTo` of `Book b` needs to be updated. This is not possible without a method call. `Book` provides method `loanTo` for this purpose. However, the invariant needs to be re-established before the method call that re-establishes it is allowed! Updating `loanedTo` before `loaned` is similarly impossible. The essence of the problem is that no single method can update all relevant fields when re-establishing an invariant.

The specification construct **inc** (for inconsistent), first introduced in [MHKL07a], offers flexibility at little cost. In this particular example, the specification of method `loanTo` includes **inc:** `MI(m)`. This makes explicit that (the verification of) `loanTo` does not rely on invariant `MI` of parameter `m`. This allows `loanTo` to be called by method `loan` after the assignment to `this.b`. Method `loanTo` does not have to re-establish the invariant (but from its postcondition it can be deduced that it does).

More generally, with every method M, a so called *inc-set* is associated. The inc-set is specified by the **inc** construct. By default, the set is empty. In the approach introduced in [MHKL07a] the inc-set can only be specified as a fixed set of *reference invariants* $I(r)$. This approach is generalized here. The inc-set of a method M is a set of elements (C, I, P), with C a classname, I the name of an invariant specified in class C and P a predicate. References in P start with either a method parameter (for instance, `this`) or the logical variable **inc**. The meaning is that for any object **inc** of class C such that P holds when M is called, $I(\mathbf{inc})$ is preserved, but not relied upon by method M. An element $I(r)$ is shorthand for the element $(C, I, \mathbf{inc} = r)$, where C is the class that defines invariant I referred to by r. Theorem 3.1 reflects the addition of the **inc** construct.

Theorem 3.1 (data induction with inc)**.**
If, for any invariant I of any object,

1) the constructor of that object establishes I, and

2) all methods in the program preserve I, and

3) while I is invalid, any method that is called specifies that it does not rely on I

Then, for any method M, for any invariant I of any object,

1) unless M is the constructor of that object or specifies that it does not rely on I, I holds when M is called and

2) unless I is invalid when a method M' is called by M, I holds when M' terminates

Proof. Proof (by induction on the length of execution sequences) is straightforward □

The premise of Theorem 3.1 is weaker than that of Theorem 2.1. For any method M, for any invariant I, the conclusion is weaker only in two cases: 1) M calls a method while I is invalid. However, given such a call, premise 3 of Theorem 2.1 is not met and Theorem 2.1 cannot be applied. 2) M specifies that it does not rely on I. In that case, I cannot be assumed to hold when M is called. However, the choice to include an invariant in the inc-set of M is made by M's developer. In Figure 3.2, the **inc** construct is applied to a more complex program. It shows how non-local invariants can be verified in a setting without information hiding. This example is derived from the Observer Pattern [GHJV95]. Users of a `CSubject` object can set a value, here **int** `d`, with method `setD`. A `CObserver` object has a field `cs` that refers to a `CSubject` object. We say it *observes* that `CSubject`. Users

```
class CSubject {
  int d;
  CObserver co;
  public void setD(int newD) {
    post: $this.d = newD$;
    impl: this.d := newD;
          if (this.co != null)
            { this.co.update(); }
  }
  public int getD() {
    inc:  I(o);
    post: result = $this.d$;
    impl: return this.d;
  }
  public void attach(CObserver o) {
    inc:  I(o), J(o);
    pre:  $this.co = null \land o.cs = this$;
    post: $I(o) \land J(o)$;
    impl: this.co := o; o.update(this.d);
  }
}
```

```
class CObserver {
  CSubject cs;
  int i;
  inv I def this.$i = f(this.cs.d)$;
  inv J def this = this.$cs.co$;
  public CObserver(CSubject toObs) {
    pre:  $toObs.co = null$;
    post: $this.cs = toObs$;
    impl: this.cs := toObs; toObs.attach(this);
  }
  public int getVal() {
    post: result = $f$(this.$cs.d$);
    impl: return this.i;
  }
  void update() {
    inc:  I(this);
    post: $I(this)$;
    impl: this.i := $f$(this.cs.getD());
  }
}
```

Figure 3.2: Observer Pattern, single observer: example of inc-sets

of a `CObserver` can retrieve a value derived from the observed `CObserver`'s field `d` by calling method `getVal`. This value is represented by $f(\mathit{this.cs.d})$ (that is, $f(\mathit{this.cs.d})$ is a placeholder for an integer expression that depends on $\mathit{this.cs.d}$). The most straightforward way to implement `getVal` is to calculate $f(\mathit{this.cs.d})$ every time `getVal` is called. However, assume that retrievals of $f(\mathit{this.cs.d})$ are more frequent than changes to `d`, and assume that it is relatively expensive to calculate $f(\mathit{this.cs.d})$. Then it is more efficient to store $f(\mathit{this.cs.d})$ in a variable that is updated when `d` is updated. Also, this variable is returned when $f(\mathit{this.cs.d})$ is requested. This implementation is specified in Figure 3.2. Note that a reference invariant $I(r)$, where I is the name of an invariant defined in (a supertype of) the static type of reference r, may occur in a predicate. Reference invariant $I(r)$ holds iff $r = \mathit{null}$ holds or $P[r/\mathbf{this}]$ holds, where P is the predicate identified by $I(r)$ and $P[r/\mathbf{this}]$ is the capture-avoiding substitution of **this** by r in predicate P.

To apply Theorem 3.1, every method must preserve all invariants. Consider an arbitrary statement in a method M. Given non-local invariants without restrictions, this statement can invalidate an arbitrary invariant of an arbitrary object. Preservation (of all invariants) is guaranteed when the verifier of M proves, for any class C, for any invariant I defined in C, for an arbitrary object of class C, that invariant I of that object 1) cannot be invalidated by the statement or 2) is re-established before the end of M. Perhaps surprisingly, such a proof is often straightforward. For most invariants, it can be determined statically that they cannot be invalidated by the statement (for instance, an assignment to a field that is not involved in the invariant). Otherwise, a more elaborate proof is needed. For example, method `setD` in Figure 3.2 contains an assignment to field `d` of class

`CSubject`. A reference to such a field also occurs in invariant `I` of class `CObserver`. To deduce that only `CObserver` `this.co` can be invalid, the proof has to use *flanking* invariant `J` (we call `J` a flanking invariant because no method relies on `J` when invariant `I` is removed). To re-establish the invariant, `this.co.update()` is called. This call is allowed as `update` specifies that it does not rely on invariant `I` of the object on which it is called. Note that the call from `update` to `getD` is allowed due to the inc-set of `getD` and the validity of flanking invariant `J` in the prestate of `update`.

We argue that a construct that specifies that a method does not rely on certain invariants is essential. On the one hand, we have the examples in Figures 3.1 and 3.2. These show natural OO designs that cannot be implemented without a call to a method that has a reference to an object with an invalid invariant. On the other hand, we have Figure 2.1, which shows an equally natural design in which methods implicitly rely on invariants of objects they have a reference to. When this is disallowed, there is an unwanted propagation of properties throughout the specification. *Only* given a construct that makes explicit that a method does not rely on certain invariants are both designs possible.

3.2 Generalized inc-sets

The generalization of inc-sets allows additional restrictions on the conditions under which an invariant is not relied upon to be specified conveniently. For instance, **inc:** $(C, I, \mathbf{inc} = \mathit{this}.s \wedge (\mathbf{inc}.a = 4 \vee \mathit{this}.a = 4))$ specifies that a method does not rely on invariant I of object `this.s` when either field `a` of `this.s` of or field `a` of object `this` has value 4.

More important, however, is that the generalized notation does not limit the inc-set to a fixed set of reference invariants. For instance, **inc:** (`CObserver, I,` *true*) specifies that a method does not rely on invariant `I` of *any* object **inc** of class `CObserver`. The example in Figure 3.3 capitalizes on this increased expressivity. This design is not feasible without generalized inc-sets. In this example, invariant `J` contains a reference of the form $r.f^i$, where $i \geq 0$. Such a reference $r.f^i$ represents reference r followed by i applications of field access $.f$. Invariant `J` specifies that a `CObserver` occurs in the list of `CObserver`s maintained by the `CSubject` it observes. The call to the `update` method in `setD` is allowed as `update` specifies that it does not rely on invariant `I` of any object **inc** of class `CObserver` that observes the same `CSubject` as the object that `update` is called on. This set of invariants cannot be specified as a fixed set of reference invariants. Note that `update` preserves all invariants, even those specified in its inc-set. This allows the verifier of `setD` to conclude that invariants of objects that have been `update`d already are preserved by an `update` call. `setD` is not a proper user (see Sect. 2) of `ONode`. This could be modified.

```
class CSubject {
  int d;
  ONode on;

  public void setD(int newD) {
    post: $this.d = newD$;
    impl: this.d := newD; ONode iter := on;
          while (iter != null) {
                iter.obs.update(newD);
                iter := iter.next; }
  }
  public int getD() {
    inc:  (CObserver, I, inc.$cs = this$);
    post: result = $this.d$;
    impl: return this.d;
  }
  public void attach(CObserver o) {
    inc:  I(o), J(o);
    pre:  $o.cs = this$;
    post: $I(o) \wedge J(o)$;
    impl: ONode n := new ONode(o, on);
          on := n; o.update(this.d);
  }
}
class ONode {
  CObserver obs;
  ONode next;

  inv I def this.$obs \neq null$;

  public ONode(CObserver o, ONode n) {
    inc:  I(o), J(o)
    post: $this.obs = o \wedge this.next = n$;
    impl: this.obs := o; this.next := n;
  }
}
```

```
class CObserver {
  CSubject cs;
  int i;

  inv I def this.$i = f($this.$cs.d)$;
  inv J def $\exists i \bullet$ this = this.$cs.on.next^i.obs$;

  public CObserver(CSubject toObs) {
    pre:  $toObs.co = null$;
    post: $this.cs = toObs$;
    impl: this.cs := toObs; toObs.attach(this);
  }
  public int getVal() {
    post: result = $this.i$;
    impl: return this.i;
  }
  void update() {
    inc:  (CObserver, I, inc.$cs = this.cs$);
    post: $I(this)$;
    impl: this.i := $f$(this.cs.getD());
  }
}
```

Figure 3.3: Observer Pattern, multiple observers: example of generalized inc-sets

4 Information Hiding

4.1 The specification construct coop

This section presents the specification construct *coop*, first introduced in [MHKL07a]. This construct specifies which invariants might be invalidated when a field is assigned to. This allows the verification of designs that include non-local invariants, even when these invariants can be hidden from a class.

So far, an important concept in OO verification has been ignored, namely that of *information hiding.* Information hiding [Par72] is an important OO design principle. Design decisions that are not relevant to a user should be hidden from that user (by means of abstraction). These decisions can then be changed without affecting that user. With *modular development* [Mül02] of a class C we mean that all except a finite and explicit set of classes are hidden from the developer of C. The benefit of modular development is that changes to hidden classes, or the addition of new classes, do not affect the verification of C.

Invariants need to be restricted to allow for data induction (Theorem 3.1) in the context of modular development. The reason is the second premise of Theorem 3.1. If invariants are not restricted, the verifier of a method M in a class C must prove for every class D, for every invariant I in D, that M preserves I. As D can be hidden from C, this comes down to proving for an arbitrary invariant of an arbitrary object of an arbitrary class, that the invariant 1) cannot be invalidated by the body of M or 2) is re-established before the end of M. This cannot be proven if the body of M is non-trivial.

Every execution of a program that has been proven correct with the technique that we introduce in section 5 meets the premises of Theorem 3.1, even when invariants can be hidden from a class. To this end, the technique 1) restricts invariants so that they can only be invalidated by assignment statements 2) has an admissibility obligation on invariants that uses the specification construct **coop**. Figure 4.1 illustrates the intuition behind this construct. An assignment to field `i` of a `CObserver` object can invalidate invariant `I` of that object. As field `i` is specified as **int** `i` **coop** `I(`**this**`)`, a verifier can assume that that invariant is the *only* invariant that might be invalidated.

More generally, a so called *coop-set* specified by the **coop** construct is associated with every field f. By default, the coop-set is empty. We generalize the approach in [MHKL07a]. A coop-set associated with a field f is a set of elements (C, I, P), with C a classname, I the name of an invariant specified in class C and P a predicate in which all references consist of one of the keyword logical variables **this** or **dep**, followed by zero or more field accesses. We say field f *cooperates with* invariant I of any object **dep** of class C for which P holds at the time field f is assigned to. The admissibility obligation guarantees the following property. When a field f of an object is assigned to, any invariant *not* cooperated with by f is *not* invalidated by the assignment. An element $I(r)$ is shorthand for the element $(C, I, \mathbf{dep} = r)$, where C is the class that defines invariant I referred to by r (i.e, it specifies that the field cooperates with invariant I of object r, when such an object exists).

Figure 4.1 shows the potential of this solution. One goal of the Observer Pattern is 'loose coupling' between `CSubject` and `CObserver` [GHJV95]: class `CObserver` should be hidden from class `CSubject`. Then, changing `CObserver` does not affect `CSubject`. In particular, this allows objects of different classes to observe a `CSubject`. The Observer pattern uses abstraction to allow this hiding. Class `Subject` is an abstraction of all classes that can be observed. Likewise, class `Observer` is an abstraction of all classes that can observe a `Subject`. Class `CObserver` is hidden from classes `CSubject` and `Subject`, and class `CSubject` is hidden from class `Observer`. The main problem is that different implementations of `Observer` have different invariants, which are hidden from class `CSubject`. We borrow an abstraction technique from [Mül02] (but omit some of the associ-

```
class Subject {
  Observer o coop I(this.o), J(this.o);
  public void attach(Observer o) {
    inc:  I(o), J(o);
    pre:  $this.o = null \wedge o.s = this$;
    post: $I(o) \wedge J(o)$;
    impl: this.o := o; o.update(this.d);
  }
}

class CSubject extends Subject {
  int d coop I(this.o);
  public void setD(int newD) {
    post: this.$d = newD$;
    impl: this.d := newD;
          if (this.o != null) {
            this.o.update(newD);
          }
  }
  public int getD() {
    inc:  I(o);
    post: result = $this.d$;
    impl: return this.d;
  }
}
```

```
interface Observer {
  abstract Subject s coop J(this);
  abstract inv I;
  inv J def this = this.$s.o$;
  void update(int d) {
    inc:  $I(this)$;
    post: $I(this)$;
  }
}

class CObserver implements Observer{
  int i coop I(this);
  CSubject cs coop I(this);
  def s by this.cs;
  def I by this.$i = f($this.$cs.d) \wedge J($this$)$;
  public CObserver(CSubject toObs) {
    pre:  $toObs.o = null$;
    post: $this.cs = toObs$;
    impl: this.cs := toObs; toObs.attach(this);
  }
  void update(int d) {
    impl: this.i := $f$(this.cs.getD());
  }
}
```

Figure 4.1: Observer Pattern with information hiding and coop-sets

ated details). Abstract fields and invariants can be introduced by the keyword **abstract**. Such fields and invariants can be implemented differently by different subclasses. The specification of a method is inherited by an overriding method. The need to include inherited flanking invariant `J` as a conjunct of `I` is a technical detail that is due to the admissibility obligation (see section 5).

4.2 Generalized coop-sets

We argue that the generalization of the **coop** construct ensures that it is expressive enough to be applied in all natural OO designs. Assume that field f is specified in class C. Assume that class D defines an invariant with name I. Consider the case where D is hidden from C. It follows from the discussion in Sect. 4.1 that any solution that allows for data induction (Theorem 3.1) in the context of modular development, must forbid that an invariant of an object of a class that is hidden from C can be invalidated by the a body of a method of C. Therefore, f does not have to cooperate with invariant I. Now consider the case where D is not hidden from C. Then invariant I of an arbitrary object of class D is cooperated with by f when f's coop-set includes $(D, I, true)$. Additional conditions on this arbitrary object can be specified by a predicate other than $true$. For instance, the specification of field `d` of class `CSubject` can be changed to **int** `d` **coop** `(CObserver, I,` $\exists i \bullet \textbf{dep} = \textbf{this}.on.next^i.obs$). This specifies that only

`CObserver`s in the list `on` maintained by `CSubject` **this** can be invalidated.

5 Formalization of the Proof Technique

The proof technique introduced in this section can complement any proof system that proves correctness of statement annotations (that has some standard properties for logical variables). Given this complemented proof system, any execution of a program that is proven correct meets the premises of Theorem 3.1. Hence, the conclusion of the theorem allows to assume the validity of (most) invariants when a method is called or terminates.

5.1 Terminology

We first introduce basic terminology. A program consists of a set of classes. C and D identify classes as before. A class defines a set of fields, methods and invariants. All are inherited when a class is extended by a subclass. f identifies a field. $coop(f, C)$ yields the coop-set of field f defined in class C (section 4.2). For simplicity, defining a subclass field with the same name as a superclass field (field shadowing) is disallowed (removing this restriction results in a number of additional typecasts). M_C identifies method M defined in class C. $inc(M_C)$ yields the inc-set of method M_C (see section 3). When M_C overrides M_D, $inc(M_C)$ must be a superset of $inc(M_D)$. α identifies an object, i.e. the instantiation of a class (think of α as an address). A *location* $\alpha.f$ stores the value of object α's field f. In Java-like languages, objects and their contents are accessed only by *references*. A reference r consists of a *scope variable* and zero or more field accesses of the form $.f$. A scope variable s_C is a *local variable* l_C, a *method parameter* p_C or a *logical variable* X_C (for convenience, the static type C of a variable is made explicit). Every method of a class C implicitly defines a parameter $\mathtt{this}_C$. $(C)r$ denotes the typecast of reference r to class C. P identifies a predicate. When $\bar{r}$ and $\bar{r}'$ are vectors of references, $P[\bar{r}/\bar{r}']$ denotes the simultaneous, capture-avoiding substitution of $\bar{r}'$ by $\bar{r}$ in predicate P. An *invariant* is a tuple (I, P), with I a name that identifies the invariant and P a predicate in which only **this** occurs as scope variable. The restriction on P guarantees that an invariant can only be invalidated by an assignment statement (in particular, not by object creation). Reference $r.f$ is called a *supplier reference* of an invariant (I, P) when $r.f$ either occurs in P or is a subreference of a reference that occurs in P. For simplicity and due to lack of space, we omit our treatment of supplier references of the shape $r.f^i$ (section 3.2) and only allow references of the shape $r.f$ in P. The names of the invariants of a class C are distinct. For convenience, these names are also assumed distinct from the names of invariants defined in superclasses of C. An *object invariant* $I(\alpha)$ denotes invariant I of object α. A reference invariant $I(r)$, where I is the name of an invariant defined in (a supertype of) the static type of reference r, may occur in a predicate (section 3.1).

5.2 Representing Control Flow

The presentation of the proof obligations is orthogonal to the rest of the proof system, i.e., it is assumed that every method body is fully and correctly annotated. When, during method execution, control is at a certain program location, the corresponding predicate holds in that state. The proof obligations force the annotation to be of a certain shape. To formulate the proof obligations, a high-level abstraction of the grammar of (fully) annotated statements suffices. We use the following grammar:

$$\begin{array}{rcl} S & ::= & \{P\} \mid \{P\}\mathit{Stat};\ S \\ \mathit{Stat} & ::= & \mathit{Basic} \mid \mathbf{if}(S, S') \mid \mathbf{while}(S) \\ \mathit{Basic} & ::= & \mathbf{assign}(r) \mid \mathbf{mc}(r, M_C) \mid \mathbf{new}(l_C, D) \mid \mathbf{dtc} \end{array}$$

The statement $\mathbf{assign}(r)$ is an assignment to reference r, where the right-hand side is a reference or a primitive value. $\mathbf{mc}(r, M_C)$ is a method call to method M_C with reference r as receiver (the call might be dynamically dispatched to a subclass-method M_D). For simplicity, method parameters (other than `this`) are not allowed. Removal of this restriction is straightforward. The use of object creation statement $\mathbf{new}(l_C, D)$ and constructor-only statement $\mathbf{dtc}$ is explained in section 5.3.

In different executions of a method, control can flow through the method in different ways. Consider fully annotated method `m` (using square brackets for annotation):

```
m(){
 [$P_0$]
 this.f := 1; [$P_1$]
 if (a == b) { [$P_2$]
  this.g.m(); [$P_3$]
 }[$P_4$]
 this.h := 0; [$P_5$]
}
```

Suppose `this.f := 1` can invalidate an invariant (field `f` has a non-empty coop-set). Then this invariant must either be implied by P_1 or P_2 (i.e., re-established before the call) or be in the inc-set of $\mathit{this.g.m}()$. It must also be implied by either P_1, P_4 or P_5 (i.e., re-established before the end of the method). We associate a graph with a method to express which methods may be called between two statements. Control flow through the method is represented by a path in the graph.

$\mathit{body}(M)$ yields M's body, the annotated statement S. In each state of a method execution, control is at a specific program location in this method. Each program

$$\begin{array}{lcl}
\mathit{graph}(\{P\}) & \overset{\text{def}}{=} & \text{Let } N = \{(P, \mathbf{end})\} \text{ in } (N, \{\}) \\
\mathit{graph}(\{P\}\mathit{Basic}; S) & \overset{\text{def}}{=} & \text{Let } G = \mathit{graph}(S) \text{ and } n = (P, \mathit{Basic}) \text{ in} \\
 & & \quad G \cup \{\{n\}, \{(n, \mathit{start}(G))\}\} \\
\mathit{graph}(\{P\}\mathbf{if}(S_1, S_2); S_0) & \overset{\text{def}}{=} & \text{Let } G_0 = \mathit{graph}(S_0) \text{ and } G_1 = \mathit{graph}(S_1) \text{ and } G_2 = \mathit{graph}(S_2) \\
 & & \text{and } n_0 = (P, \mathbf{if}(S_1, S_2)) \text{ and } n_1 = \mathit{start}(G_0) \text{ and} \\
 & & E = \{\ (n_0, \mathit{start}(G_1)), (n_0, \mathit{start}(G_2)), \\
 & & \qquad (\mathit{end}(G_1), n_1), (\mathit{end}(G_2), n_1)\} \mathit{in} \\
 & & \quad G_0 \cup G_1 \cup G_2 \cup \{\{n_0\}, E\} \\
\mathit{graph}(\{P\}\mathbf{while}(S_1); S_0) & \overset{\text{def}}{=} & \text{Let } G_0 = \mathit{graph}(S_0) \text{ and } G_1 = \mathit{graph}(S_1) \\
 & & \text{and } n_0 = (P, \mathbf{while}(S_1)) \text{ and } n_1 = \mathit{start}(G_0) \text{ and} \\
 & & n_2 = \mathit{start}(G_1) \text{ and } n_3 = \mathit{end}(G_1) \text{ in} \\
 & & \quad G_0 \cup G_1 \cup \{\{n_0\}, \{(n_0, n_1), (n_0, n_2), (n_3, n_1), (n_3, n_2)\}\}
\end{array}$$

Figure 5.1: graph construction algorithm

location identifies exactly one annotated statement S, which is represented as a node in a graph. A node n is a tuple (P, eStat), where eStat identifies an element of the set $\{\mathit{Stat}, \mathbf{end}\}$. A program location that identifies an annotated statement $\{P\}$ is represented by a node $(P, \mathbf{end})$. A program location that identifies an annotated statement $\{P\}\mathit{Stat}; S$ is represented by a node (P, Stat). Two functions on a node are defined.

$$\begin{array}{rcl}
\mathit{pre}((P, \mathit{eStat})) & \overset{\text{def}}{=} & P \\
\mathit{stat}((P, \mathit{eStat})) & \overset{\text{def}}{=} & \mathit{eStat}
\end{array}$$

An *edge* e is a tuple (n, n'). When there is an edge (n, n') in the graph, the program location n' can be reached from program location n in a single execution step. When the program counter identifies n in a particular execution state, in the next execution state it identifies a node n' such that (n, n') is an edge in the graph. A graph G is a tuple of a set of nodes N and a set of edges E. A union on graphs is defined: $(N, E) \cup (N', E') \overset{\text{def}}{=} (N \cup N', E \cup E')$. The graph G of an annotated statement contains exactly one node without incoming, and one node without outgoing edge. $\mathit{start}(G)$ and $\mathit{end}(G)$ yield these nodes. $\mathit{graph}(S)$, defined in Figure 5.1, yields the graph of annotated statement S.

$$\begin{array}{ll}
\mathit{start}((N, E)) & = n \text{ iff } n \in N \text{ and } \forall n' \bullet (n', n) \notin E \\
\mathit{end}((N, E)) & = n \text{ iff } n \in N \text{ and } \forall n' \bullet (n, n') \notin E
\end{array}$$

Seq identifies a sequence. $|\,\mathit{Seq}\,|$ yields the length of sequence Seq. $\mathit{Seq}[i]$ yields the i'th element of Seq. $\mathit{Seq}[i, j]$ yields the subsequence of Seq of elements i up to and including j. $\mathit{Seq}[i, j)$ yields the subsequence of Seq of elements i up to but not including j. $\mathit{Seq}[i..)$ yields the postfix of Seq that starts at element i. A sequence of nodes nSeq is *a path in* graph (N, E) when the nodes in the sequence are (pair-wise) adjacent, i.e., when $\mathit{nSeq}[0] \in N$ and $\forall i \bullet (0 < i < |\,\mathit{nSeq}\,| \Rightarrow (\mathit{nSeq}[i-1], \mathit{nSeq}[i]) \in E)$. When nSeq is a path in graph G, it is *a path from* node n when $\mathit{nSeq}[0] = n$

and it is *a path to* node n when $nSeq[|nSeq| -1] = n$. $nSeq$ is *cycle-free* when all its elements are distinct, i.e, $\forall i, j \bullet (0 \leq i < j <|nSeq| \Rightarrow nSeq[i] \neq nSeq[j])$.

An *execution sequence* Θ is a sequence of *states*. A state is a tuple (τ, n, Θ'), where τ consists of a heap and a stack. $nodes(\Theta)$ is the sequence of nodes $nSeq$ such that when $\Theta[i] = (\tau, n, \Theta')$, $nSeq[i] = n$. Let $graph(body(M_C)) = G$. Then Θ represents an execution of M_C when 1) $nodes(\Theta)$ is a path from $start(G)$ in G that is either infinite or is a path to $end(G)$ and 2) when $\Theta[i] = (\tau, n, \Theta')$, Θ' is the empty sequence unless $stat(n)$ is of the form $\mathbf{mc}(r, M'_D)$ (then Θ' represents an execution of method M'_D). So, when $graph(body(M_C)) = G$, any possible way control can flow in a (terminating) execution of method M_C is represented by a path from $start(G)$ to $end(G)$ in G.

Recall that s_C is a scope variable of static type C. A node $(P, eStat)$

establishes $I(s_C)$	iff	$P \Rightarrow I(s_C)$
respects $I(s_C)$	iff	if $eStat$ is of the form $\mathbf{mc}(r, M_D)$, then there is a P' such that $(C, I, P') \in inc(M_D)$, and $P \Rightarrow P'[(s_C, (D)r)/(\mathbf{inc}, \mathtt{this}_D)]$

A sequence of nodes $nSeq$

respects $I(s_C)$	iff	$\forall i \in nSeq \bullet nSeq[i]$ respects $I(s_C))$
establishes $I(s_C)$	iff	$\exists i \in nSeq \bullet ($ $nSeq[i]$ establishes $I(s_C)$ and $nSeq[o..i)$ respects $I(s_C)$ $)$
is safe for $I(s_C)$	iff	$nSeq$ respects or establishes $I(s_C)$

In an execution represented by a path that respects $I(s_C)$, no method is called that relies on the object invariant represented by $I(s_C)$. In an execution represented by a path that establishes $I(s_C)$, there is an execution state in which the object invariant represented by $I(s_C)$ holds and no method that relies on that object invariant is called before that state.

5.3 Constructors

The semantics of constructors is treacherous due to dynamic binding. The first (possibly implicit) statement in a constructor of class C is a call to the constructor of C's superclass. When the superclass constructor calls a method, and it is dynamically dispatched to an overriding method in a subclass, this method might rely on an invariant yet to be established. We do not consider this good OO design, but it is possible in Java. Rather than (slightly) changing Theorem 3.1 and introducing restrictions on programs to prevent such implementations, we use a different (more logical) semantics for constructors [MHKL07a]. The body of a constructor of class C is of the shape $\{P_0\}\mathbf{mc}(\mathtt{this}, M_D); \{P_1\}\mathbf{dtc}; S$, where $\mathbf{mc}(\mathtt{this}, M_D)$ is a call to the constructor of C's superclass. $\mathbf{new}(l_C, D)$ creates an object of class `Object` and calls the constructor of class D to initialize the object. Statement **dtc** (dynamic type change) in a constructor of class C changes the dynamic type of the object that is initialized to class C. No invariants of the object that is initialized are invalid when the superclass constructor is called (class

`Object` does not define any). This semantics of constructors is similar to that of C++.

5.4 Proof Obligations

Finally, the proof technique for invariants can be presented. The *admissibility obligation* below guarantees that, when a location $\alpha.f$ is assigned to, any object invariant *not* cooperated with by f is *not* invalidated by the assignment. A proof relies on the definition of coop in section 4.1 and a context switch.

Definition 5.1 (admissibility obligation).
For every invariant (I, P) defined in a class C,
  for every supplier reference $r.f$ of (I, P),
    if *$r.f$ refers to field f specified in class D,*
    then *there is a $(C, I, P') \in coop(f, D)$ such that*
      $P \Rightarrow P'[(\mathbf{this}, (D)r)/(\mathbf{dep}, \mathbf{this})]$

In a setting where information hiding for invariants (or more generally, modular development) is not required, this obligation and the **coop** construct are not needed. To still apply the proof technique, the default coop-set can be changed to include (C, I, true) for every class C in the program, for every invariant I defined by class C. This trivializes the admissibility obligation. For simplicity, we omit a weaker version of the obligation. For instance, one can capitalize on the fact that, in a state where P doesn't hold, an invariant $(P \vee P')$ cannot be invalidated by an assignment to a field that only occurs in P.

Before the remaining proof obligations are presented, a theorem is formulated that is essential for both the soundness of the approach and the intuition behind it.

Theorem 5.1.
If *every cycle-free path from n' in G is safe for $I(X_C)$, and*
  every cycle-free path from n' to $end(G)$ in G establishes $I(X_C)$,
then *every path from n' in G is safe for $I(X_C)$, and*
  every path from n' to $end(G)$ in G establishes $I(X_C)$

Proof. Straightforward (by induction on the number of cycles in an arbitrary path) □

As this theorem shows, only cycle-free paths have to be considered by the proof obligations.In the remainder of this section, three more proof obligations are introduced.

Definition 5.2 (constructor obligation).
For every constructor M of a class C, for every invariant (I, P) defined in C,
  if *$graph(body(M)) = G = (N, E)$ and $\{(start(G), n), (n, n')\} \subseteq E$,*
  then *every cycle-free path from n' in G is safe for $I(\mathtt{this}_C)$, and*
    every cycle-free path from n' to $end(G)$ in G establishes $I(\mathtt{this}_C)$

Here, node n' is the program location directly after the **dtc** statement (section 5.3). This proof obligation establishes that, no matter how control flows through a constructor, the invariant of the newly constructed object is established, and no method that relies on it is called before this is done, which is needed for the third premise of Theorem 3.1. One way to establish that this proof obligation is met is to use a breadth-first algorithm that searches for a node that establishes the invariant (and stops after a full cycle is traversed). An example of such an algorithm is *apv*, defined below. The intuition is that if $apv(n, G, I(s_C))$, then 1) every cycle-free path from n in G is safe for $I(s_C)$, and 2) every cycle-free path from n to $end(G)$ in G establishes $I(s_C)$. A proof by induction on the length of cycle-free paths is straightforward.

```
$apv(n, (N, E), I(s_C)) \stackrel{\text{def}}{=}$
  or   $n \notin N$,
  or   $n$ establishes $I(s_C)$,
  or   $n \neq end((N, E))$, and
       $n$ respects $I(s_C)$, and
       for every edge $(n, n') \in E$, $apv(n', (N - \{n\}, E), I(s_C))$ holds.
```

The two other proof obligations use logical variables that keep track of which invariants might be invalid.

Definition 5.3 (inc obligation)**.**
For every method M of a class C, for every $(C, I, P) \in inc(M)$, if $graph(body(M)) = G$, then there exist a predicate P' and a logical variable X_C such that
X_C does not occur free in P', and
$pre(in(G)) \Leftrightarrow (P' \wedge (X_C = null \vee P[X_C/\mathbf{inc}]))$, and
every cycle-free path from $in(G)$ in G is safe for $I(X_C)$

Let the coop-set of method M include (C, I, P). Then, for an arbitrary object X_C such that $I(X_C) \wedge P[X_C/\mathbf{inc}]$[1] holds in the prestate of an execution of M the following property is established. No matter how control flows through method M, no method that relies on $I(X_C)$ is called unless $I(X_C)$ has been established to hold. Note that $X_C = null \vee (I(X_C) \wedge P[X_C/\mathbf{inc}])$ can always be assumed to hold (it says, either there is an arbitrary object X_C such that $(I(X_C) \wedge P[X_C/\mathbf{inc}])$ holds, or there is not). A small example: Let $inc(M)$ include $(C, I, false)$. Of course, this is a meaningless inclusion in an inc-set, as this says that M will not rely on the I-invariant of any C-object for which *false* holds (and there cannot be such objects). However, the proof obligation is likewise trivial: Note that $(X_C = null \vee (I(X_C) \wedge P[X_C/\mathbf{inc}])) \Rightarrow X_C = null$. As $I(null)$ is trivially true, $in(G)$ establishes $I(X_C)$ and the proof obligation is met.

Definition 5.4 (cooperation obligation)**.**
For every method M of a class C,

[1]Many thanks to Stephanie Balzer for pointing out that there should be no obligation to re-establish invariants that were already invalid at the time of the assignment, which is achieved by the conjunct $I(X_C)$

if $graph(body(M)) = G = (N, E)$, *and* $(n, n') \in E$, *and* $stat(n) = \mathbf{assign}(r.f))$,
and $r.f$ *refers to field* f *of class* D, *and* $(C, I, P) \in coop(f, D)$,
then there exist a predicate P' *and a logical variable* X_C *such that*
X_C *does not occur free in* P', *and*
$pre(n) \Leftrightarrow (P' \wedge (X_C = null \vee (I(X_C) \wedge P[(X_C, (D)r)/(\mathbf{dep}, \mathbf{this})])))$, *and*
every cycle-free path from n' *in* G *is safe for* $I(X_C)$, *and*
every cycle-free path from n' *to* $end(G)$ *in* G *establishes* $I(X_C)$

This obligation combines the techniques of the previous two to deal with assignment statements. Note that, when n is an assignment node, there is exactly one outgoing edge (n, n'). An invariant that does not hold in the precondition of a method is trivially preserved. For simplicity, this is not capitalized on by the proof obligation.

5.5 Soundness

A full soundness proof (i.e., a full formal proof that the premises of Theorem 3.1 are met given the proof obligations) is beyond the scope of this thesis. Instead, we sketch a proof of the following central property of the proof; If an assignment in a method execution invalidates an invariant $I(\alpha_1)$, then 1) it is re-established before the method terminates, and 2) any method called before $I(\alpha_1)$ is re-established specifies that it does not rely on $I(\alpha_1)$. In a similar way, one can prove that, when an invariant is invalid when a method M_C is called, any method that is called by M_C while the invariant is invalid specifies that it does not rely on it, and that constructors establish their invariants.

A *logical environment* ω maps logical variables to values. $\tau, \omega \models P$ iff P holds given the mappings in τ and ω. $\omega[X_C \to \alpha]$ is the state like ω, but with X_C mapped to α. Let Θ_0 represent an arbitrary method execution. Let $\Theta_0[i] = (\tau_0, n, [])$. Let $n = \mathbf{assign}(r_0.f)$, where f is defined in class D. Let τ_0 map r_0 to α_0. Assume invariant $I(\alpha_1)$ is invalidated by the assignment. Assume this invariant is defined in class C as (I, P_0). Then $I(\alpha_1)$ holds in the prestate, i.e., $\forall \omega \bullet \tau_0, \omega[X_C \to \alpha_1] \models P_0[X_C/\mathbf{this}]$ (as **this** is the only scope variable in P_0). As only $\alpha_0.f$ is changed by the assignment, P_0 must have a supplier reference $r_1.f$, with f defined in C, that refers to $\alpha_0.f$. Then, due to the admissibility obligation, there is a $(C, I, P_1) \in coop(f, D)$ such that $P_0 \Rightarrow P_1[(\mathbf{this}, (D)r_1)/(\mathbf{dep}, \mathbf{this})]$. Then (using two context switches) $\forall \omega \bullet \tau_0, \omega[X_C \to \alpha_1] \models P_1[(X_C, (D)r_0)/(\mathbf{dep}, \mathbf{this})]$ (as **dep** and **this** are the only scope variables in P_1). Let $nodes(\Theta_0) = nSeq$. Assume the method execution terminates. From Theorem 5.1 and the cooperation obligation it follows that there is a $j > i$ such that $nSeq[j]$ establishes $I(X_C)$ and $nSeq(i..j]$ is safe for $I(X_C)$. The *main theorem* relied on for soundness is the following: Let $\Theta[i] = (\tau, n, \Theta')$ and $j > i$ and $\Theta[j] = (\tau', n', \Theta'')$. Then $\forall \omega \bullet (\tau, \omega \models pre(n)$ implies $\tau', \omega \models pre(n'))$. A proof relies on the correctness of the annotation, the soundness of the proof system and some (fairly standard) assumptions about logical variables. From this theorem and the above it follows that, when $\Theta_0[j] = (\tau_1, n_1, \Theta_1)$, then $\forall \omega \bullet \tau_1, \omega[X_C \to \alpha_1] \models I(X_C)$, which means $I(\alpha_1)$ is valid in $\Theta[j]$. To prove that methods called do not rely on $I(\alpha_1)$, assume that there is a k, $i < k < j$, such that $\Theta_0[k] = (\tau_2, n_2, \Theta_2)$ and $stat(n_2) = mc(r_2, M_D)$ (for some class D) and r_2 refers to α_3 in τ_2. Then, due to the cooperation obligation and Theorem 5.1, n_2 respects $I(X_C)$ and therefore, there is a P_2 such that $(C, I, P_2) \in inc(M_D)$ and $pre(n_2) \Rightarrow P_2[(X_C, (D)r_2/(\mathbf{inc}, \mathtt{this}_D))]$. Due to the main theorem above, $\forall\omega \bullet \tau_2, \omega[X_C \to \alpha_1] \models pre(n_2)$. Therefore, $\forall\omega \bullet \tau_2, \omega[X_C \to \alpha_1] \models P_2[(X_C, (D)r_2/(\mathbf{inc}, \mathbf{this}))]$. Thus, when $\Theta_2 = (\tau_3, n_3, \Theta_2)$ (as **inc** and `this` are the only scope variables in P_2), $\forall\omega \bullet \tau_3, \omega[\mathbf{inc} \to \alpha_1] \models P_2$. As any method overriding M_D also includes (C, I, P_2) in its inc-set, this proves that the method execution specifies that it does not rely on $I(\alpha_1)$. Finally, when method execution Θ_0 does not terminate, it follows that $nSeq[i..)$ is infinite and safe for $I(X_C)$, and the proof is similar to the one above.

This concludes our formal treatment. Note that proof rules that allow invariants to be assumed at certain points in the proof are omitted but these are straightforward.

6 Related and Future work

Ownership-based approaches and our approach are complementary (an object owns another when it has some form of control over access to the other's data), and combining them should be fairly straightforward. Several ownership mechanisms have been proposed (e.g., [PNCB05, LM04, Mül02]). Given modular development, a complementary approach is needed as invariants of which the name is hidden from a class C 1) cannot be in the inc-set of a method of C and 2) cannot always be expected to be preserved by methods of C when the structure is hierarchical. Our *coop* construct is similar to the explicit dependencies in [Mül02], which generalizes earlier work in [Lei95]. However, these do not allow a specification as in Figure 4.1. A liberal, but semantical admissibility obligation on invariants is used in [Mül02]. In the Boogie ([LM04]) and the friendship ([BN04]) approaches (both extending [BDF$^+$04]), as well as in [PB05], flexible abstraction mechanisms are provided that allow a method to specify that it relies on a hidden property, and that allow a user to track the validity of such a hidden property. While very useful for information hiding, these do not prevent the propagation of (abstractions of) properties. We expect the friendship approach can achieve specifications of similar strength, without propagation, when one adds either 1) an inc-like construct to specify which inv-bits do not hold in a precondition and the program invariant that all other inv-bits hold or 2) a default precondition that all inv-bits hold and the possibility to override this default. Compared to cooperation-based invariants, this gives a less intuitive semantics for invariants but a more intuitive proof technique. However, it has the additional overhead of both pack/unpack and attach/detach. The work in [PCdB05] shows uses of invariants that quantify over (unreachable) objects, and how they can be allowed. The premises of theorem 3.1 do not disallow such invariants. An extension that allows such invariants is considered future work.

7 Conclusion

Data induction allows a method to rely on an invariant without it being specified in pre- and postconditions. We have introduced an approach that allows this for invariants over object structures. The **inc** construct specifies that a method does not rely on certain invariants. We argue that this is essential for the specification of many natural, non-hierarchical designs. The **coop** construct specifies which invariants can be invalidated by an assignment. This allows the verification of invariants even when their definition is hidden. In particular, this makes the approach suitable to modular development. We introduce proof obligations that guarantee data induction is allowed.

CHAPTER 5

Specifying and Exploiting Layers in OO Designs

This chapter contains the following paper, with minor editorial changes: *Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik J. Luit. Specification and Verification of Invariants by Exploiting Layers in OO Designs. Fundamenta Informaticae, 85(1-4):377-398, 2008. Special issue: Concurrency, Specification and Programming (CS&P'07).*[MHKL08c]. It is available online. The chapter has an accompanying technical report "A Proof System for Invariants in Layered OO Designs" [MHKL08b].

abstract The layering that is present in many OO designs is not accounted for in current interpretations of invariants. We propose to make layers explicit in specifications and introduce a new interpretation of invariants that exploits these layers. Furthermore, we present a sound, modular technique to statically verify that programs satisfy the new interpretation.

1 Introduction

Object-oriented (OO) designs often contain classes and methods that provide shared functionality (i.e., are used by several otherwise unrelated objects). Examples include a canvas in graphics applications, and static method `sqrt` from the Java API. Current specification and verification techniques either disallow invocation of such shared functionality, or do not guarantee that invariants needed by the shared functionality hold when it is invoked. We introduce an approach that resolves these issues.

Together with pre- and postconditions, *class invariants* (invariants for short) have a central role in most model-based OO specification techniques. A class invariant is a boolean expression that is associated with a class. Conceptually, an invariant of a class C captures a consistency property of objects of C that clients can expect to hold [WK99]. The use of invariants can significantly reduce specification overhead [Mey97]. Invariants can also be used for abstraction [Hoa72]. They allow the specifier to keep parts of desired input/output relations implicit, thus hiding information [Par72] and increasing modularity of the design [Mül02].[1] The interpretation[2] of class invariants in OO designs is still under discussion. A key issue is that commonly (i.e., in many designs), an object's invariant is temporarily violated while the state it depends on is updated. This is unproblematic as the invariant is irrelevant to the methods called during the update: these method executions do not *rely* on the invariant. Re-establishing an invariant may require the invocation of shared functionality. For instance, it may require the calculation of a square root, which in Java can be done by an invocation of method `sqrt` (which, like most Java API methods, does not rely on any user-defined invariants). We elaborate an example in section 3. The interpretation of invariants has to account for such unproblematic scenarios.

Our observation is that to achieve loose coupling [Par72], OO designs are often *layered*, where a layer uses functionality of lower layers, but in principle not of higher layers. In particular, lower layer methods do not rely on invariants of higher layer objects. Commonly, shared functionality is in a lower layer of the design (for instance a library) than the classes that use it. For example, a class that implements the Singleton Pattern [GHJV95] provides shared functionality through global access to its instance. This class is often in a lower layer of the design than its clients. The contributions of this chapter are the following:

1. We analyze a state-of-the-art interpretation of invariants that exploits whole/part relations in OO designs and show that it is not suitable for scenarios that involve shared functionality (section 3).
2. We show how layers in a design can be made explicit with little specification overhead (section 4).
3. We refine the state-of-the-art interpretation to exploit layers in OO designs (section 4).
4. We present a modular verification technique to establishes that programs satisfy the refined interpretation of invariants. This is done in three steps.
   a. We present a semantic decomposition of the technique into separate concerns (section 5).
   b. We extend an existing technique for reasoning about whole/part relations using simple syntactic checks (static reasoning). We capture additional relations in the whole/part hierarchy, and refine the technique to reason about layer relations as well (section 6).[3]

[1] Our approach accounts for information hiding. To simplify the technical treatment, it is omitted from the language.

[2] We use *interpretation* instead of the term *semantics* which is used in, e.g., [MPHL06] as the latter sometimes leads to confusion.

[3]Another existing technique (dynamic reasoning) is extended in an accompanying technical

c. We introduce a proof technique based on static reasoning for each of the concerns (section 7).

This three-step approach gives insight into the role of the syntactic restrictions, proof obligations and captured relations. It also allows changing one technique without affecting the others.

2 Programming and Specification Language

We consider the specification of invariants in the context of a single-threaded subset of Java or C#, with minor changes made for presentational purposes. A program in this language consists of a main method and a set of classes. Every class C (except `Object`) has one direct superclass denoted $\mathit{super}(C)$. $D \subset C$ denotes that D is a proper subclass of C. Figure 2.1 shows the statement grammar. Note that method parameters are treated as local variables, that method calls only occur in statements $v{=}r.m(\vec{e})$, and that expressions SimpleE are side-effect free. Every method has exactly one `return` statement, as the last statement. Methods with return type `void` return `null`, which is assigned to a dummy variable. We relegate type casts, constructors, static fields and static methods to [MHKL08b], and do not consider exception handling, object de-allocation, inner classes, or field shadowing (where a subclass field redeclares a superclass field [Pie06]). Our example uses self-explanatory shorthand notations.

Invariants are most naturally formalized given a trace semantics for programs. Such a semantics is presented in the appendix, which shows that the statements in the language have the same interpretation as their Java counterparts. The most relevant details are discussed below. The semantics of a program is a set of program executions (i.e., traces). A *program execution* is a sequence of (execution) states. A *state* σ contains, among other things, a stack frame σ_{sf} and an object store σ_{os}. A *stack frame* is a partial function that takes a *stack variable* (`this` or a local variable) and returns a *value* (an object, *null*, a boolean, or a number). An *object store* is a partial function that takes a *location* (a tuple ⟨object,field⟩, written $x.f$) and returns a value. The semantics of simple expressions is defined by a partial function *eval* that takes a simple expression, a stack frame and an object store, and returns a value. A suitable Java-like definition is straightforward (see e.g. [Pie06]) and is omitted. Simple expression s *evaluates to* value v in σ if $\mathit{eval}(s, \sigma_{sf}, \sigma_{os}) = v$. $\mathit{type}(x)$ returns the dynamic type of object x [Lei95]. Intuitively, $\mathit{as}(\sigma)$ (see appendix) returns the statement that will be executed from σ (the 'active' statement). For a sequence Σ, $\Sigma[i]$, $\Sigma[i..j]$ and $\Sigma[i..]$ denote element, consecutive subsequence and postfix. $[A \mapsto B]$ denotes the partial function that maps A to B.

As usual, the grammar of boolean expressions in the specification language resembles first order logic over program variables. Details are not relevant and are left

report [MHKL08b]. This technique uses an encoding into an underlying proof system, which offers more flexibility at the cost of more verification overhead.

C ∈ *classnames*, *T* ∈ *types*, *m* ∈ *methodnames*, *v* ∈ *local variablenames*, *f* ∈ *fieldnames*

Stat	::=	`if` (BoolE) Stat `else` Stat | `while` (BoolE) Stat | Stat; Stat | $T\ v$ | r = SimpleE | v = r.$m(\vec{e})$ | `return` SimpleE;
SimpleE	::=	r | `null` | BoolE | NumE
r	::=	`this` | v | r.f

Figure 2.1: Statement grammar (where $\vec{e}$ denotes a sequence of simple expressions).

implicit: suitable definitions can be found in e.g. [Pie06, LBR06]. For simplicity, the boolean expressions in the programming and in the specification language are not distinguished (in the interest of computability, the former should be restricted).

A *class invariant* is a boolean expression BoolE that is associated with a class. A class invariant does not contain stack variables other than `this` (a modular verification technique needs to impose additional restrictions, see sections 5 and 7.2). Every class C has one invariant, denoted inv_C. In our examples, inv_C is defined by specification construct `inv` BoolE in C's specification (if the construct is omitted, inv_C is *true*, and multiple `inv` constructs are conjoined into a single BoolE). Several interpretations of class invariants are discussed in sections 3 and 4. These interpretations make explicit which object invariants hold in which visible states of a program execution: an *object invariant* is a tuple of a class invariant and an object, written $inv_C(x)$. $inv_C(x)$ *holds* in state σ if $type(x) \subseteq C$ and $eval(inv_C, [\mathtt{this} \mapsto x], \sigma_{os}) = true$. *The invariants of* x denotes set of object invariants $\{inv_C(x) \mid type(x) \subseteq C\}$. In program execution Σ, $\Sigma[i]$ is *visible* if it is a pre- or a poststate. $\Sigma[i]$ is a *prestate* if either $as(\Sigma[i-1])$ is a method call, or $i = 0$. $\Sigma[i]$ is a *poststate* if $as(\Sigma[i])$ is a return statement.

Matching pre- and poststates mark the first and last states of a method execution: prestate $\Sigma[i]$ and poststate $\Sigma[j]$ *match* in Σ if $i < j$ and every prestate $\Sigma[k]$, $i < k < j$, is matched by a poststate $\Sigma[l]$, $k < l < j$. A subsequence Σ' of Σ is a *method execution* if there is a prestate $\Sigma[i]$ such that either (1) $\Sigma' = \Sigma[i..j]$ and $\Sigma[i]$ and $\Sigma[j]$ match in Σ, or (2) $\Sigma' = \Sigma[i..]$ and $\Sigma[i]$ is unmatched in Σ.

States σ and σ' *differ* on x if there is a field f such that $\sigma_{os}(x.f) \neq \sigma'_{os}(x.f)$. *Control is with* object x in state σ if $\sigma_{sf}(\mathtt{this}) = x$. *Control is in* class C in σ if the static type of `this` in σ is C. *Control flows to* x in $\Sigma[i]$ if control is with x in $\Sigma[i]$, and either $\Sigma[i]$ is a prestate, or $\Sigma[i-1]$ is a poststate.

3 Problem Analysis

This chapter builds on work from [MPHL06]. Sections 3.1 and 3.2 reiterate the parts relevant to the problem analysis. The first section discusses the classical invariant interpretation and its limitations, the second an existing, state-of-the-art interpretation that addresses some of these limitations. Section 3.3 argues that this interpretation can be improved and sketches how this can be done: the idea behind the chapter.

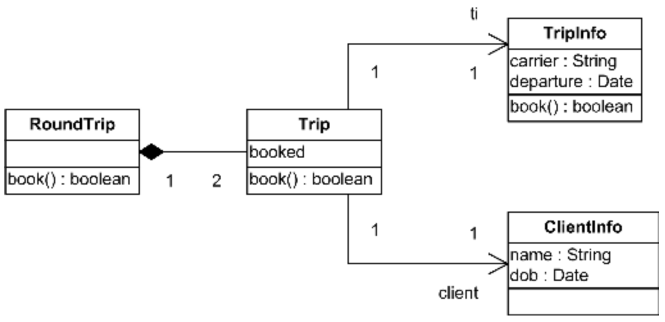


Figure 3.1: Conceptual Design of a Travel Agent Administration System

3.1 Classical Invariant Interpretation

The *Classical Invariant Interpretation* (CII) is based on visible states.

Definition 3.1 (CII)**.** *Program execution Σ satisfies the CII if, in every visible state in Σ, for every object x*[4]*, the invariants of x hold.*

The CII is a suitable interpretation for *local* invariants, i.e., invariants that only refer to the state of a single object. However, an invariant an object x often depends on the state of other objects, for instance when x is a composition of other objects. Executions of designs that include non-local invariants often do not satisfy the CII. As an example, consider the design in figure 3.1, which represents a simplified travel agent administration system. A `ClientInfo` object contains the personal information of a specific client of the travel agent. A `TripInfo` object contains the information of a specific trip that is offered by the travel agent. `TripInfo`'s method `book(ClientInfo c)` contacts the carrier's booking system to reserve a seat for client `c`. For simplicity, this always succeeds. A `Trip` object contains the information of a trip selected by a client. A `Trip` object stores whether the trip has been booked. A `RoundTrip` consists of two `Trip`s, one `outbound` and one `inbound` (i.e., there is a *whole/part* relation [Boo94, WK99] between `RoundTrip` and `Trip`). Figure 3.2 shows an implementation of `Trip` and `RoundTrip`. A non-local invariant has been specified for `RoundTrip`, which relates the state of `RoundTrip`'s parts. It is intended to ensure that a client is not faced with a `RoundTrip` of which only one leg is `booked`.

Program executions in which a `RoundTrip` R is booked do not satisfy the CII. Consider an execution of R`.book()`, which invokes R`.outbound.book()`. In the poststate of R`.outbound.book()`, the invariant of R does not hold. This means it does not hold in the prestate of R`.inbound.book()` either. Booking the `inbound` `Trip` re-establishes the invariant: in the poststate of R`.book()`, the invariants of R hold.

This is an example of a common scenario: an invariant of an object is temporarily

[4]In this chapter, quantifications range over *allocated* objects

```
class Trip {
  root TripInfo ti; root ClientInfo
  client;
  boolean booked = false;
  void book() {
    ti.book(this.client); booked =
    true;
  }  //Other methods omitted
}
```

```
class RoundTrip {
  rep Trip outbound; rep Trip inbound;
  inv outbound.booked == inbound.booked;
  void book() {
    outbound.book(); inbound.book();}
  } //Other methods omitted
}
```

Figure 3.2: Code annotated with an invariant and ownership information (`rep` and `root`, see section 6.1).

violated while its parts are updated. This is unproblematic as the design accounts for the violation. The CII is too strong to allow such scenarios. It must be refined based on observations of common scenarios. Here, the violation is accounted for by a *subordinate relation* between a `RoundTrip` and its parts:

Definition 3.2 (subordinate)**.** *Object y is a* subordinate *of object x if (1) the invariants of x is are not relied on when control flows to y, and (2) the invariants of y hold when control flows to x.*

Observation 3.1. *Commonly, if object y is a part of object x, then y is a subordinate of x.*

3.2 Ownership and the Relevant Invariant Interpretation

Ownership (see e.g. [Cla01, Mül02]) can make whole/part relations in the conceptual design explicit in the formal specification. The idea is that an object owns its subordinate parts. Special purpose owner **root** owns objects that are not a subordinate part of any object (a slight deviation from the definition in [MPHL06]).

Definition 3.1 (ownership)**.** *The set of* owners *consists of the set of objects and the special purpose owner* **root**. *In any given state, every object is* directly owned *by exactly one owner. This relation is acyclic. The* owned *relation is its transitive closure. Finally, owner(σ) = O if control is with an object that is directly owned by owner O in state σ. The owner of an object is determined by the statement that creates it.*

As figure 3.1 shows, the `inbound` and `outbound Trip` of a `RoundTrip` R are part of R. The `TripInfo ti` and `ClientInfo client` of a `Trip` are not part of any object. Ownership reflects this. A `RoundTrip` directly owns its `inbound` and `outbound Trip`, and the `TripInfo ti` and `ClientInfo client` of a `Trip` are directly owned by **root** (made explicit in figure 3.2 by keywords `rep` and `root`, see section 6.1). Figure 3.3 shows a possible configuration, in which a client $C1$ has selected a `RoundTrip` $R1$, and a client $C2$ has selected a `Trip` $L3$. $L3$ and $L1$ share a `TripInfo`: $C1$ and $C2$ travel together on $C1$'s `outbound Trip`. Scenarios where an invariant of an object is temporarily violated while its subordinate parts are updated are allowed if the CII is replaced with the weaker *Relevant Invariant Interpretation* (RII).

Definition 3.2 (RII)**.** *Program execution Σ satisfies the RII if, in every visible state $\sigma \in \Sigma$, for every object x owned by $owner(\sigma)$, the invariants of x hold.*

Consider an execution of $R1$`.book()`. Unlike the CII, the RII allows for the violation of the invariants of $R1$ in the poststate of $R1$`.outbound.book()` and in the prestate of $R1$`.inbound.book()` (as $R1$ is the owner of these states).

3.3 Class level subordinate relations

Unfortunately, subordinate relations do not just occur between a whole and its parts. Therefore, the RII sometimes requires certain object invariants hold when they are intended to be violated, and sometimes does not guarantee that certain object invariants hold when they are intended to hold. In this section, we identify a category of subordinate relations at the class level. We show that the RII is not suitable for designs like the travel agent example because it does not account for these subordinate relations.

To achieve loose coupling [Par72], there typically is a layering of the classes used in a conceptual design. The intuition is that a layer uses functionality of its own layer and lower layers, but has little or no dependency on higher layers [Boo94, RBL+90]. More specifically, we make the following observation.

Observation 3.2. *Commonly, there are classes C and D in a design such that any object of C is a subordinate of any object of D. In particular, when layers are present in the design and class C is in a lower layer than class D, this property almost always holds.*

Shared functionality is commonly in a lower layer of the design than the classes that use it. Ownership imposes a partial ordering on the object structure. We say an object x provides *shared functionality* if different objects that are not ordered by ownership invoke methods on x. A practical example is the use of a canvas in graphics applications. In the travel agent example, `TripInfo` and `ClientInfo`, which offer shared functionality, are in a lower layer of the design than `Trip` and `RoundTrip`, which use this functionality. Not accounting for the layering in the interpretation of invariants complicates both specification and verification. The RII is well suited when re-establishing the invariants of an object x only requires invocations of methods on objects owned by x, but not when lower-layer shared functionality is used. This is illustrated by the observations below.

Observation 3.3. *Commonly, to re-establish an invariant of an object x, control must flow to a lower-layer object y with a direct owner that owns x. Given observation 3.2 (and the definition of subordinate), the invariants of x are not relied on when control flows to y. The RII, however, is not satisfied: it requires the invariants of x to hold when control flows to an object with a direct owner that owns x.*

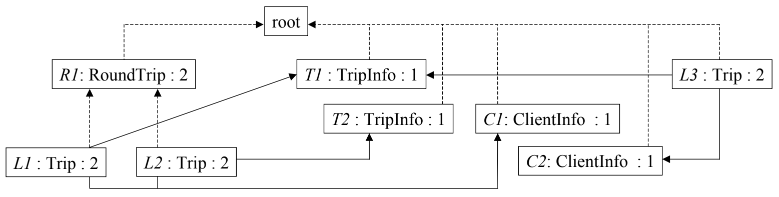


Figure 3.3: Possible Travel Agent object configuration. Objects refer to their direct owner with a dashed arrow (see section 3.2). Solid arrows refer to objects that provide shared functionality. Objects have a name, a class and a layer (see section 4).

For instance, consider figure 3.3. To book `RoundTrip` $R1$, $R1$`.inbound.book()` invokes $T2$`.book()` when an invariant of $R1$ is violated. The RII is too strong to allow this violation.

Observation 3.4. *Commonly, when control flows to an object x with direct owner O, an invariant of a lower layer object y that is not owned by O is relied on. Given observation 3.2, the invariants of y hold when control flows to x. The RII, however, does not guarantee this as O does not own y.*

Again consider figure 3.3. An execution of $L2$`.book()` invokes $T2$`.book()`, which is likely to rely on invariants of $T2$ (among others). Due to the layering of the design, we can expect that the invariants of $T2$ hold in the prestate of methods executed on $L2$. The RII is to weak to guarantee this.

The next section presents a refined interpretation of invariants that accounts for these observations.

4 Layers and the Layered Relevant Invariant Interpretation

In this section, it is shown how the layers in a conceptual design can be made explicit in the formal specification. Then, this layering is used to present a refined interpretation of invariants.

Definition 4.1 (layers)**.** *There is a function layer that takes a class and returns a rational number. layer(C), the layer of C, is determined by the specification of C. Let l be a rational number. If the specification of C contains* `layer` *l, then $layer(C) = l$ (if the* `layer` *construct is omitted, $layer(C)$ is a default value, see below). The layer of an object x, $layer(x)$, is $layer(type(x))$. If control is with x in state σ, then the layer of σ, $layer(\sigma)$, is $layer(x)$.*

The additional structure provided by layers is used to replace the RII by a more flexible interpretation that reflects the observations of section 3.3. This *Layered* Relevant Invariant Interpretation (LRII) is a conservative extension of the RII in the following sense: when every class in a program P has the same layer, any execution of P that satisfies the RII also satisfies the LRII, and vice versa. Roughly, the LRII states the following. If σ is a visible state in which control is with an object that has layer l and direct owner O, then the invariants of all lower layer objects and of all same-layer objects owned by O, hold in σ (item 1 below). In addition, if σ is the poststate of a method execution Σ', then any invariant of a higher layer object with direct owner O that holds in Σ' prestate, also holds in σ (item 2).

Definition 4.2 (LRII)**.** *Program execution Σ satisfies the LRII iff*

```
for every visible state $\sigma \in \Sigma$,
  for every object $x$,
    (1) if      or  $layer(x) < layer(\sigma)$,
                or  $(owner(\sigma)$ owns $x$ and $layer(x) = layer(\sigma))$,
        then    the invariants of $x$ hold, and
    (2) if      $\sigma$ is a poststate, and $owner(\sigma)$ owns $x$, and $layer(x) > layer(\sigma)$,
        then    for every object invariant $inv_C(x)$,
                  if     $inv_C(x)$ holds in the prestate matching $\sigma$,
                  then   $inv_C(x)$ holds in $\sigma$.
```

Observation 3.2 is reflected in two differences between the RII and the LRII. Consider an $inv_C(x)$ of an x with layer l. The LRII is stronger than the RII in the sense that with the LRII, $inv_C(x)$ is guaranteed to hold in a visible state σ if $l < layer(\sigma)$. The LRII is weaker than the RII in the sense that with the LRII, $inv_C(x)$ is not required to hold in a prestate σ if $owner(\sigma)$ owns x but $l > layer(\sigma)$.

Likely, the specifier of a class C is not be able (or willing) to change the layer specifications of existing classes. An advantage of mapping to rational numbers is that if needed, the specifier can always find a layer in between the layers of two classes D and E as long as $layer(D) \neq layer(E)$. Furthermore, using rational numbers requires very little specification overhead, in particular given default class layers.[5]

Definition 4.3 (default class layers)**.**

```
If     class $C$ does not specify its layer,
then   or  $C$ is Object and $layer(C) = -1$,
       or  $C$ is a Java API class and $super(C)$ is Object and $layer(C) = 1$,
       or  $C$ is a user-defined class and $super(C)$ is Object and $layer(C) = 2$
       or  $super(C)$ is not Object and $layer(C) = layer(super(C))$.
```

The intuition behind these default values is that Java API classes do not rely on user-defined invariants. A user-defined class that is intended to provide shared functionality (section 3.3) can explicitly specify `layer 1`. A class that does not specify its `layer` has its default layer. Therefore, user-defined classes `Trip` and

[5]If the specification language does not have rationals, floating point numbers have the same benefits for all practical purposes.

`RoundTrip` (figure 3.2) have layer 2. `TripInfo` and `ClientInfo` (not shown) specify `layer` 1, which means they have a lower layer than `Trip` and `RoundTrip`.

The differences between the RII and the LRII give the extra flexibility needed for designs like that of the travel agent example. Again consider figure 3.3. Given the LRII, the specification expresses that the invariants of $T1$ and $C1$ hold in the prestate of an execution of `book` on one of $R1$'s `Trips`. Furthermore, the invariants of $R1$ are not required to hold in a prestate of $T1$`.book()`. Note that item 2 of the LRII guarantees that any invariant of $R1$ that *does* hold in a prestate of $T1$`.book()`, also holds in the matching poststate. More generally, consider two objects x and y that have the same owner. Assume $\mathit{layer}(x) < \mathit{layer}(y)$. The additional flexibility of the LRII allows x to be called while an $\mathit{inv}_C(y)$ does not hold. As a consequence, the LRII can not require the invariants of y to hold in the poststate. However, item 2 ensures that any $\mathit{inv}_C(y)$ that did hold when x was called, is preserved by the call.

Finally, note that the layer of an object will normally not be lower than the layer of objects it owns. If object x owns object y, the intention is that y is a subordinate part of x. That means the invariants of y should hold when control flows to x. This is not guaranteed by the LRII when $\mathit{layer}(x) < \mathit{layer}(y)$.

This concludes the first part of this chapter. In section 3, we determined the interpretation of invariants to be the origin of the problems to verify the running example. At the semantical level, the problems have been solved by the LRII. The second part discusses how to establish the LRII.

5 Establishing the Layered Relevant Invariant Interpretation

The verification technique is presented in three steps. This section identifies four properties. A program execution that satisfies these, satisfies the LRII. Section 6 enables reasoning about layer and ownership relations. Section 7 treats the separate concern of how to statically verify that every execution of a program satisfies the properties identified here. These properties are similar to those underlying model-based Abstract Data Type (ADT) specifications, where invariants range over the type's state, which is encapsulated from clients of the type [GH93]. In turn, this allows a form of reasoning that is similar to the data type induction used for ADTs. That is, establishing the LRII reduces to a *local* property, i.e., a property that only considers the object that has control.

A modular verification technique needs to restrict the invariants that are considered [MHKL08a]. To this end, the following property of an invariant is defined.

Definition 5.1 (ownership based)**.** *Invariant* inv_C *is* ownership based *in program P if, for every execution Σ of P, for every two consecutive states σ and σ' in Σ, for every object x, if $\mathit{inv}_C(x)$ holds in σ but not in σ', then σ and σ' differ either on x, or on an object y that is owned by x in σ', where $\mathit{layer}(x) \geq \mathit{layer}(y)$.*

When all invariants are ownership based, a change of the state of an object x can (only) invalidate invariants of x and its owners. A method can change the state of an object x either directly, by an assignment to a field of x, or indirectly, by a method call. Assignment to a field of x is ensured to only occur when control is with x by the second property.

Definition 5.2 (classical encapsulation)**.** *Location $x.f$ is* classically encapsulated *in a program execution Σ if, for every two consecutive states σ and σ' in Σ, if $\sigma(x.f) \neq \sigma'(x.f)$, then control is with x in σ'. Program execution Σ satisfies* classical encapsulation *if every location is classically encapsulated in Σ.*

The third property ensures that method calls are such that (1) if control is with x in state $\Sigma[i]$, then control does not flow to y in $\Sigma[i+1]$ unless $layer(x) \geq layer(y)$, and (2) if x directly owns y, then a method on y can only be invoked by a method on x or on an object directly owned by x.

Definition 5.3 (ownership encapsulation)**.** *Program execution Σ satisfies* ownership encapsulation *if, for every two consecutive states σ and σ' in Σ, if σ' is a prestate, then*
or $layer(\sigma) \geq layer(\sigma')$ and $owner(\sigma) = owner(\sigma')$,
or $layer(\sigma) \geq layer(\sigma')$ and control is with $owner(\sigma')$ in σ,
or $layer(\sigma) > layer(\sigma')$ and $owner(\sigma') =$ **root***.*

With these three properties, establishing the LRII is reduced to establishing a local property that only considers the object that has control. It is formulated below, using the notion of a horizontal call state.

In program execution Σ, $\Sigma[i]$ is a *horizontal call state* if $\Sigma[i+1]$ is a prestate and $owner(\Sigma[i]) = owner(\Sigma[i+1])$ and $layer(\Sigma[i]) = layer(\Sigma[i+1])$.

Definition 5.4 (local consistency)**.** *Program execution Σ satisfies* local consistency *if, for every horizontal call state or poststate $\sigma \in \Sigma$, if control is with x in σ, then the invariants of x hold in σ.*

Theorem 5.1. *Consider an execution Σ of a program in which every invariant is ownership based. If Σ satisfies classical and ownership encapsulation as well as local consistency, then Σ satisfies the LRII.*

A detailed proof can be found in [MHKL08b]. The outline relies on intermediate properties P1 and P2.

P1. If $\Sigma[i]$ is a program execution state in which control is with object x, then for every object y, either (1) the invariants of y hold in $\Sigma[i]$, or (2) $y = x$, or (3) y owns x, or (4) $\Sigma[i]$ is a prestate, or (5) an invariant of y does not hold in the last unmatched prestate in $\Sigma[0..i-1]$.

Proof of P1 is by induction on the length of Σ. Consider a state $\Sigma[i+1]$ in which control is with an x and in which cases 2,3,4 and 5 do not hold for a y.

If $\Sigma[i]$ is not a poststate, then control is with x in $\Sigma[i]$ and the last unmatched prestate in $\Sigma[0..i]$ and $\Sigma[0..i-1]$ is the same. Then the invariants of y hold in $\Sigma[i]$ whether $\Sigma[i]$ is a prestate (case 5 doesn't hold), or not (induction hypothesis). If $\Sigma[i]$ and $\Sigma[i+1]$ differ on an object z, then $z = x$ (classical encapsulation). Therefore the invariants of y hold in $\Sigma[i+1]$ (all invariants are ownership based).

If $\Sigma[i]$ is a poststate, assume that $\Sigma[j]$ is the last unmatched prestate in $\Sigma[0..i-1]$. Then the last unmatched prestate in $\Sigma[0..j-1]$ and $\Sigma[0..i]$ is the same. Then the invariants of y hold in $\Sigma[j-1]$ whether $\Sigma[j-1]$ is a prestate (case 5 doesn't hold), or not (induction hypothesis). Then the invariants of y hold in $\Sigma[j]$ (invariants are not invalidated by a context switch). Let control in $\Sigma[j]$ be with an object z. Then (A) the invariants of y hold in $\Sigma[i-1]$, or (B) $y = z$, or (C) y owns z (induction hypothesis). In case B, the invariants of y hold in $\Sigma[i-1]$ (local consistency). In case C, either (CA) $\mathit{layer}(x) \geq \mathit{layer}(z)$ and x and z have the same direct owner, or (CB) $\mathit{layer}(x) \geq \mathit{layer}(z)$ and x directly owns z (control in $\Sigma[j-1]$ is with x, ownership encapsulation, y owns z). Both CA and CB contradict that cases 2 and 3 do not hold for y in $\Sigma[i+1]$. Then the invariants of y hold in $\Sigma[i+1]$ (invariants are not invalidated by a context switch).

P2. If control is with x in $\Sigma[i]$, then no owner of x has a lower layer than x. Proof by induction on the length of Σ, using ownership encapsulation, is straightforward.

Induction hypothesis for the main proof is that $\Sigma[0..i]$ satisfies the LRII. Let control in $\Sigma[i+1]$ be with x, and let the LRII require $\mathit{inv}_C(y)$ to hold in $\Sigma[i+1]$. Then $\Sigma[i+1]$ is a visible state. If $\Sigma[i+1]$ is a poststate, then $\mathit{inv}_C(y)$ holds in the matching prestate (by definition of LRII or by the induction hypothesis). Then $\mathit{inv}_C(y)$ holds in $\Sigma[i+1]$, or $y = x$ (P1, P2). Then $\mathit{inv}_C(y)$ holds in $\Sigma[i+1]$ (local consistency). If $\Sigma[i+1]$ is a prestate, then $\mathit{inv}_C(y)$ holds in the last unmatched prestate in $\Sigma[0..i]$ (induction hypothesis and ownership encapsulation). Let control in $\Sigma[i]$ be with z. Then $\mathit{inv}_C(y)$ holds in $\Sigma[i]$, or $y = z$ (P1, P2). Then $\mathit{inv}_C(y)$ holds in $\Sigma[i]$ (local consistency). Then $\mathit{inv}_C(y)$ holds in $\Sigma[i+1]$ (invariants are not invalidated by a context switch). Then $\Sigma[0..i+1]$ satisfies the LRII.

6 Static Reasoning

In section 6.1, it is shown how ownership relations can be captured using *static reasoning*: simple syntactic restrictions like those of a type system suffice to establish the desired properties. Section 6.2 introduces syntactic restrictions that allow layer information to be captured as well.

6.1 Capturing ownership relations

The function *ownmod* (defined below) associates an *ownership modifier* `owned`, `rep`, `peer`, `root` or `any` with every reference r that is not of a primitive type. It is ensured that, if control is with object x in state σ, and r evaluates to object y in σ, then $osafe(x, y, ownmod(r))$ holds in σ, where

$osafe(x, y, \texttt{owned})$	holds	if x owns y,
$osafe(x, y, \texttt{rep})$	holds	if x directly owns y,
$osafe(x, y, \texttt{peer})$	holds	if x and y have the same direct owner,
$osafe(x, y, \texttt{root})$	holds	if **root** directly owns y,
$osafe(x, y, \texttt{any})$	holds	always.

The `owned` modifier is useful to determine which references are allowed to occur in ownership-based invariants. The `peer, rep` and `root` modifiers allow one to determine if a method call satisfies one of the three cases of ownership encapsulation. This section is based on work on the Universe Type System [Mül02, MPHL06, DM05]. We introduce the additional `owned` and `root` modifiers.

First, the property above is formalized by the notion of ownership safety. Then, it is shown how to determine the ownership modifier of an expression that can occur in an assignment statement. Finally, two syntactic restrictions are introduced that ensure that assignments respect the property.

If a field or local variable is not of a primitive type, then must be declared with an ownership modifier. For example, in figure 3.2 the `inbound` and `outbound` fields of `RoundTrip` are declared with a `rep` modifier. These modifiers allow the formalization of the property at the level of stack variables and locations.

Definition 6.1 (ownership safety)**.** *Location $x.f$ is* ownership safe *in state σ if: if f has ownership modifier o, and $\sigma_{os}(x.f)$ is object y, then $osafe(x, y, o)$ holds in σ. If control is with object x in state σ, then stack variable s is* ownership safe *in σ if: if s has ownership modifier o, and $\sigma_{sf}(s)$ is object y, then $osafe(x, y, o)$ holds in σ. Execution state σ is* ownership safe *if every location and stack variable is ownership safe in σ. Method execution Σ is* ownership safe *if every $\sigma \in \Sigma$ is ownership safe.*

The value of a location or local variable can only be changed to another object by an assignment. The right-hand side of such assignment is either a reference, a method call, or an object creation. The latter is included for completeness as the direct owner of an object is determined at creation. An object creation statement has the shape v `= new` o $C(\ldots)$, where ownership modifier o is either `rep, peer`, or `root`. If control is with x when the statement is executed, then the direct owner of the newly created object is either x, the direct owner of x, or `root`, respectively.

If a method is not of a primitive return type, then it must be declared with an ownership modifier. The function $ownmod(E)$ uses the ownership modifiers of fields, local variables, and methods to associate an ownership modifier with each expression E that can occur in an assignment. In the definition below, A represents either a field f or a method call $m(\ldots)$ (which has the ownership modifier of m).

shape of `E`	$ownmod$(`E`)
`new` o $C(\ldots)$	o
`this`	`peer`
local variable v	ownership modifier of v
`this.`A	ownership modifier of A
`r.`A, `r` different from `this`	$ownmod(r) \oplus$ ownership modifier of A (see below)

	$omr \oplus omA$	peer	root	any	rep	owned	omA
omr	peer	peer	root	any	any	any	
	root	root	root	any	any	any	
	any	any	root	any	any	any	
	rep	rep	root	any	owned	owned	
	owned	owned	root	any	owned	owned	

The rationale behind the definition of ownership combinator $\oplus$ is as follows. Consider a reference $r.f$ in a state σ in which control is with x. Assume that r evaluates to y, and that $\sigma_{os}(y.f) = z$. Let $ownmod(r) = omr$, and let f have type C and modifier omA. If omA is `peer`, then y and z have the same direct owner and the ownership relation between x and z is the same as that between x and y. Therefore, $ownmod(r.f) = ownmod(r)$. If omA is `root`, then z is directly owned by **root**, independent of omr, so $ownmod(r.f) =$ `root`. If omA is `any`, then any ownership relation between y and z is possible. Then the same is true for x and z. Therefore, $ownmod(r.f) =$ `any`. If omA is `rep` or `owned`, then y owns z, so x owns z if x owns y (i.e., $ownmod(r.f) =$ `owned`). If x does not own y, the relation can only be expressed by `any`.

For the purpose of the two syntactic restrictions that ensure ownership safety (and those in section 6.2), a method call $r.m(s_0, \ldots, s_n)$ is treated as a series of assignments $r.p_0{=}s_0, \ldots, r.p_n{=}s_n$, where $p_0, \ldots, p_n$ are the formal parameters of m. Furthermore, a statement `return` s in a method m is treated as an assignment `result =` s, where the ownership modifier of `result` is the ownership modifier of m.

Syntactic Restriction 6.1: *If the left-hand side and right-hand side of an assignment have ownership modifiers o and o', respectively, then either o is* `any`*, or o is o', or o is* `owned` *and o' is* `rep`*.*

Syntactic Restriction 6.2: *Every assignment $r.f$* `= SimpleE` *is such that either r is* `this`*, or the ownership modifier of f is* `any`*, or $ownmod(r.f) \notin \{$`any`, `owned`$\}$.*

Roughly, restriction SR6.1 ensures that $osafe(x, y, o)$ holds after the execution of an assignment $r = E$ from a state in which control is with x, $ownmod(E)$ is o and r evaluates to y. However, as the ownership modifier of a field is relative to the object the field belongs to, one has to be careful when assigning to fields of objects other than the `this` object. For instance, consider a class `Node` with a field `rep Node next`. Then $ownmod$(`this.next`) = `rep` and $ownmod$(`this.next.next`) = `owned`. Although SR6.1 is met, assignment `this.next.next = this.next` should not be allowed as `this.next.next` must refer to an object that is directly owned by `this.next`. Such assignments that do not preserve ownership safety are prevented by SR6.2.

The above leads to the following lemma (proof is omitted given similarities with [DM05] and [Mül02]).

Lemma 6.1. *If program P meets SR6.1 and SR6.2, then every execution of P is ownership safe.*

6.2 Capturing layer relations

This section shows how to use the layer of the static type of a reference to determine the layer of the object it refers to. The following property is ensured: if a reference r with static type C refers to an object of class D, then $layer(C) = layer(D)$, unless r is `this`, or $ownmod(r)$ is `any`. Furthermore, if r is `this`, then $layer(D) \geq layer(C)$. This gives a lower bound on the layer of a visible state, without which no invariants can be assumed to hold given the LRII. As with reasoning statically about ownership, `any` is used to deal with atypical cases. The property is first formalized in terms of stack variables and locations. Then, the syntactic restrictions that establish the property are introduced.

Definition 6.1 (layer safety)**.** `this` *is* layer safe *in state σ if: if control in σ is in class C and with an object of class D, then $layer(D) \geq layer(C)$. Location or local variable $LorLV$ is* layer safe *in state σ if: if $LorLV$ has declared type C, and $\sigma(LorLV)$ is object y, then either $ownmod(LorLV)$ is* `any`*, or $layer(y) = layer(C)$. Execution state σ is* layer safe *if every location and every stack variable is layer safe in σ. Method execution Σ is* layer safe *if every $\sigma \in \Sigma$ is layer safe.*

Note that a location or local variable that is not mapped to an object is layer safe. Restrictions SR6.3, SR6.4 and SR6.5 ensure that all program executions are layer safe, resulting in lemma 6.2.

> **Syntactic Restriction 6.3:** *If $C = super(D)$, and class D contains specification* `layer` *l, then $l \geq layer(C)$.*
>
> **Syntactic Restriction 6.4:** *Every assignment of an expression with static type D to a reference r with static type C is such that either $layer(C) = layer(D)$, or $ownmod(r)$ is* `any`.
>
> **Syntactic Restriction 6.5:** *Every assignment r* `= this` *is such that $ownmod(r) =$* `any`.

Lemma 6.2. *If program P meets SR6.1-SR6.5, then every execution of P is layer safe.*

The proof outline of this lemma is as follows. Let control in state σ be in class C and with an object of class D. Then $D \subseteq C$. Due to SR6.3, $layer(D) \geq layer(C)$, which ensures layer safety of `this`. Now consider locations and local variables.

These can only be changed by assignments and method calls. As in section 6.1, a method call is treated as a series of assignments. Consider a local variable v or field f with ownership modifier o and declared type C. Let reference r be either v, or $r'.f$. Assume assignment $r = E$ is executed from a layer-safe state σ. If o is `any`, then layer safety is preserved by definition. If o differs from `any`, then SR6.2 ensures that $\mathit{ownmod}(r) \neq$ `any`. Then SR6.1 ensures that $\mathit{ownmod}(E)$ differs from `any`. Assume that E has static type D and is mapped to an object that has layer l. Two cases can be distinguished: E is either (1) a method call or a reference other than `this`, or (2) `this`. In case (1), layer safety ensures that $\mathit{layer}(D) = l$ (if E is a reference, proof by structural induction on references is straightforward, and if E is a method call, proof is only slightly more involved as a method call returns a reference `result`). Then SR6.4 ensures that $\mathit{layer}(C) = \mathit{layer}(D)$. Therefore, $\mathit{layer}(C) = l$ and the assignment preserves layer safety. Case (2) has to be restricted as the exact layer of `this` can not be reasoned about statically. SR6.5 ensures the assignment preserves layer safety, as an `any` reference can refer to an object with any layer. This restriction is weakened using dynamic reasoning in [MHKL08b].

The use of static reasoning is illustrated by a short example. Consider a program execution Σ that is ownership safe and layer safe. Consider an state σ in Σ such that control is with an object x and in a class C. Assume that the active statement $\mathit{as}(\sigma)$ is a statement v = $r.m(\ldots)$, and that σ maps r to an object y. If a syntactic check reveals that $\mathit{ownmod}(r)$ is `rep`, that r has static type D, and that $\mathit{layer}(C) \geq \mathit{layer}(D)$, then x directly owns y, and $\mathit{layer}(x) \geq \mathit{layer}(y)$.

7 Proof Techniques

This section first introduces the proof system and the LRII's role in it. Then, a proof technique based on static reasoning is introduced for each property in section 5. A proof outline is presented with each technique. Detailed proofs can be found in [MHKL08b]. The techniques are based on the *ownership technique* from [MPHL06].

7.1 Proof system

The goal of verification of a program P is to guarantee that every execution of P satisfies a set *Props* of desired properties. We only consider safety properties: if $\mathit{prop} \in \mathit{Props}$ holds for a program execution, then *prop* holds for each prefix of that execution. The LRII is one such property. The statements `assert BoolE` and `assume BoolE` (where `BoolE` is side-effect free) are added to the programming language as a convenient way to make explicit where a predicate `BoolE` must be proven to hold, or is guaranteed to hold. Either statement causes the program execution to abort when executed from a state in which `BoolE` does not hold (it assert-aborts or assume-aborts). Neither has an effect otherwise.

For every property $prop \in Props$, a proof technique PT_{prop} is devised. A proof technique consists of a set SR_{prop} of syntactic restrictions, and a set PO_{prop} of proof obligations. Proof obligations dictate where assert statements should occur in a program (i.e., proof obligations are syntactic restrictions that involve assert statements). Let P' be a program like P, but with zero or more assert statements inserted. A meta-level soundness proof must ensure that if P' meets PT_{prop}, then every execution of P' satisfies *prop*. This establishes that if no execution of P' aborts, then every execution of P satisfies *prop* (there is no execution of P' in which an assert statement has an effect).

The verifier's task is simplified by assume statements. With every $prop \in Props$, a set $Assump_{prop}$ of assumptions is associated. Assumptions allow assume statements to occur at certain program locations. A meta-level soundness proof must ensure that if every execution of a program P satisfies *prop*, and $Assump_{prop}$ allows assume statement S, then no execution of P aborts due to an execution of S.

Let PTs be the union of all PT_{prop}, and let $Assumps$ be the union of all $Assump_{prop}$ ($prop \in Props$). Verification of a program P proceeds as follows. First, insert assert statements into P, obtaining a program P'. Show that P' meets PTs (note that this is a syntactic check). Next, insert assume statements into P', obtaining a program P''. Show that every assume statement in P'' is allowed by $Assumps$. Finally, prove that no execution of P'' assert-aborts.

The crucial meta-level property is the following: If no execution of P'' assert-aborts, then every execution of P satisfies every $prop \in Props$. Proof is as follows. As P' meets PTs, P' meets every $PT_{prop} \in PTs$. Then every execution of P' satisfies every $prop \in Props$ (soundness of the proof techniques). Then every execution of P'' satisfies every $prop \in Props$ (*prop* is a safety property, an execution of an `assume` either has no effect or aborts the execution). Then no execution of P'' assume-aborts (soundness of the $Assumps$). Then no execution of P'' aborts (no execution of P'' assert-aborts). Then every execution of P satisfies *prop* (there is no execution of P' in which an `assert` or `assume` statement has an effect).

Note that the verifier underlying [BLS05] can automatically verify whether a program that contains assert and assume statements might abort. Alternatively, straightforward `assume` and `assert` rules can be added to a Hoare logic like the one in [PHM99].

Sections 7.2-7.5 give, and prove sound, proof techniques for the four properties introduced in section 5. PT_{LRII} is the union of these proof techniques (soundness of PT_{LRII} then follows directly from theorem 5.1). As the language of boolean expressions was left implicit, $Assump_{LRII}$ is not made fully explicit. Let boolean expression b be such that if, for every object x, items 1 and 2 defined in the LRII hold in state σ, then b holds in σ. $Assump_{LRII}$ allows `assume` b as the first statement of every method, and allows `assume` b to precede any `return` statement (soundness of $Assump_{LRII}$ is trivial). Note that the latter allows b to be an implicit (i.e., trivially proven) part of every postcondition.

7.2 Establishing ownership-based invariants

To ensure that invariants are ownership based, a syntactic restriction is imposed on invariants. An invariant BoolE is *ownership admissible* if BoolE is composed of (quantifications over) primitive values, the usual unary and binary operators (see e.g. [Mül02]), and references $\texttt{this}.f_1 \ldots .f_i$ ($i \geq 0$) such that if $i > 1$, then $\mathit{ownmod}(\texttt{this}.f_1 \ldots .f_{i-1})$ is either `rep` or `owned`.

Syntactic Restriction 7.1: *Every invariant is ownership admissible.*

`RoundTrip`'s invariant (figure 3.2) is ownership admissible as its two references have a `rep` modifier.

Lemma 7.1. *If program P meets SR7.1, and every execution of P is ownership safe, then every invariant is ownership based in P.*

The reasoning is as follows. An object invariant $\mathit{inv}_C(x)$ can only be invalidated by changing the value of a reference $\texttt{this}.f_1 \ldots f_i$ that occurs in inv_C. This requires an update of a location $y.f_{j+1}$, where $\texttt{this}.f_1 \ldots f_j$ ($j < i$), refers from x to y. If $j = 0$, then $y = x$. Otherwise, $\mathit{ownmod}(\texttt{this}.f_1 \ldots f_{i-1})$ is either `rep` or `owned` (inv_C is ownership admissible). Then $\mathit{ownmod}(\texttt{this}.f_1 \ldots f_j)$ is either `rep` or `owned` (section 6.1). Then x owns y (ownership safety).

7.3 Establishing classical encapsulation

If two consecutive states σ and σ' in a program execution are such that $\sigma(x.f) \neq \sigma'(x.f)$, then active statement $as(\sigma)$ is a field assignment to a reference $r.f$ such that r refers to x in σ. Therefore, classical encapsulation can be established by imposing a syntactic restriction on assignment `r = SimpleE`.

Syntactic Restriction 7.2: *Every assignment* `r.`f `= SimpleE` *is such that* `r` *is* `this`.

Note that the code in figure 3.2 meets SR7.2: only fields of `this` are assigned to. Given the reasoning above, proving the following lemma is straightforward.

Lemma 7.2. *If program P meets SR7.2, then every execution of P satisfies classical encapsulation.*

7.4 Establishing ownership encapsulation

With static reasoning (section 6), ownership encapsulation can typically be established by simple syntactic checks. More specifically, the notion of statically meeting encapsulation is introduced. A method call that statically meets encapsulation does not violate ownership encapsulation.

Definition 7.1 (statically meeting encapsulation)**.** *If reference r has static type D, then statement v = $r.m(\ldots)$ in a method of class C* statically meets encapsulation *if*

either $layer(C) \geq layer(D)$ *and* $ownmod(r)$ *is* `peer`*,*
or $layer(C) \geq layer(D)$ *and* $ownmod(r)$ *is* `rep`*,*
or $layer(C) > layer(D)$ *and* $ownmod(r)$ *is* `root`*.*

Syntactic Restriction 7.3: *Every method call statement statically meets encapsulation.*

Note that every method call in figure 3.2 statically meets encapsulation. Class `Trip` calls methods on `TripInfo this.ti`, where $ownmod($`this.ti`$) =$ `root`, and $layer($`Trip`$) > layer($`TripInfo`$)$. Class `RoundTrip` calls methods on its inbound and outbound `Trips`, which have a `rep` modifier, and $layer($`RoundTrip`$) = layer($`Trip`$)$. Static reasoning is only possible in programs that are layer and ownership safe, leading to lemma 7.3.

Lemma 7.3. *If program P meets SR7.3, then every execution of P that is ownership safe and layer safe, satisfies ownership encapsulation.*

The reasoning is the following. The three cases of the definition of ownership encapsulation can be recognized in the definition of statically meeting encapsulation. If control is with an object x and in a class C, and a reference r of static type D refers to an object y, and $ownmod(r)$ is either `peer`, `rep`, or `root`, then $layer(y) = layer(D)$ and $layer(x) \geq layer(C)$ (layer safety). As $ownmod(r)$ correctly represents the ownership relation between x and y (ownership safety), the lemma holds.

7.5 Establishing local consistency

Let σ be a horizontal call state or poststate in which control is in class C and with object x. To satisfy local consistency, the invariants of x must hold in σ, i.e., every object invariant $inv_D(x)$, $type(x) \subseteq D$, must hold in σ. Establishing this is complicated by the fact that not all subclasses are available at superclass verification time. Therefore, an inductive argument must guarantee that every object invariant $inv_D(x)$, $type(x) \subseteq D \subset C$ holds in σ. That every object invariant $inv_B(x)$, $C \subseteq B$, holds in σ can be established directly. Let $upinv_C$ denote the conjunction of the invariants declared in class C and C's superclasses. Then every $inv_B(x)$, $C \subseteq B$, holds in σ if $upinv_C$ holds in σ. If σ is a poststate, then proof obligation PO7.1 ensures every $inv_B(x)$, $C \subseteq B$, holds in σ.

proof obligation 7.1: *Every return statement in class C is preceded by* `assert` *$upinv_C$.*

If σ is a horizontal call state, then PO7.2 ensures x is consistent for $[C,$`Object`$]$ in σ. PO7.2 relies on the fact that a method call does not lead to a horizontal call state if it is not statically relevant.

Definition 7.1 (statically relevant)**.** *If reference r has static type D, then statement v* `=` *$r.m(\ldots)$ in class C is* statically relevant *if*

or $\quad$ *ownmod*(r) *is* `any`,
or $\quad$ *ownmod*(r) *is* `peer` *and layer*$(C) =$ *layer*(D),
or $\quad$ *ownmod*(r) *is* `root` *and layer*$(C) =$ *layer*(D).

proof obligation 7.2: *Every statically relevant method call in class C is preceded by* `assert` *upinv*$_C$.

Note that no method call in the code in figure 3.2 is statically relevant. To ensure that every $inv_D(x)$, $type(x) \subseteq D \subset C$ holds in σ, it is ensured that for *any* state σ', if control is with x and in C, then $inv_D(x)$ holds. If SR7.1-SR7.3 are met, then an inductive proof is straightforward if (1) an assignment cannot invalidate a subclass invariant, and (2) if a subclass invariant holds in the prestate of an invoked method execution Σ, then it also holds in Σ's poststate. Note that a subclass invariant may be temporarily violated by Σ, as long as it has been re-established in Σ's poststate.

Given SR7.2, a superclass method cannot assign to a subclass field. So, to establish (1), it only has to be ensured that a subclass invariant does not depend on a superclass field. This is ensured by SR7.4.

Syntactic Restriction 7.4: *If invariant inv_C contains reference* `this.`$f_0 \ldots .f_i$ *($i \geq 0$), then f_0 is declared in class C.*

Ownership admissibility and ownership safety ensure that if object invariant $inv_C(x)$ contains a reference `this.`$f_0 \ldots .f_i$, and $i \geq 1$, then $x.f_0 \ldots .f_j$ $(0 \leq j < i)$ refers to an object x owns. So only $x.f_0$ is a field of x. SR7.4 ensures that f_0 is not a field of a superclass of C.

(2) is established by the use of a form of encapsulation that goes beyond that provided by ownership. Consider an object x of class D. The objects owned by x are divided into *class frames* [LM04]. There is a frame for every class C such that $D \subseteq C$ (called the C-frame of x). It is ensured that invariants are *frame based*: if $inv_C(x)$ holds in $\Sigma[i]$, but not in $\Sigma[i+1]$, then $\Sigma[i]$ and $\Sigma[i+1]$ differ either on a field of x declared in class C, or on an object in the C-frame of x. Furthermore, it is ensured that every program execution Σ satisfies *frame encapsulation*: if $\Sigma[i]$ is a prestate in which control is with an object in the C-frame of x, then in $\Sigma[i-1]$ control was either with an object in the C-frame of x, or with x and in C.

Frames can be reasoned about statically. If field f_0 (local variable v, method m) is declared in class C with a `rep` or `owned` modifier, then the frame of expression `this.`$f_0 \ldots .f_i$ ($v.f_1 \ldots .f_i$, `this.`$m(\ldots)$) is C. It is ensured that if control is with x in σ, and reference r evaluates to y and has a C-frame, then y is in the C-frame of x. Formulated differently, (P1) if $\sigma_{os}(x.f) = y$, and f is declared in a class C with a `rep` or `owned` modifier, then y is in the C-frame of x, and (P2) if y is in the C-frame of x, and $\sigma_{os}(y.f) = z$, and f has `rep`, `owned` or `peer` modifier, then z is in the C-frame of x. For the purpose of these definitions, if control is with x and in C, then local variables are treated as fields of x declared in C. P1 and P2 are established by two restrictions that ensure that an expression with a C-frame is not assigned to a reference with another frame.

Syntactic Restriction 7.5: *Every field declared with a* `rep` *or* `owned` *modifier is private.*

Syntactic Restriction 7.6: *Every method or method parameter declared with a* `rep` *or* `owned` *modifier is private.*

Roughly, the proof is the following. When control is with x and in C, SR7.5 prevents x from reaching owned objects outside the C-frame via rep or owned fields. SR7.6 prevents the exposure of owned objects outside the C-frame to x via a method return. When control is not with x or not in C, SR7.5 prevents the update of owned and rep fields of x declared in C. SR7.6 prevents the exposure of owned objects outside the C-frame to x via (the parameters of) a C-method call on x.

That invariants are frame based follows almost directly from SR7.1 (ownership admissibility), SR7.4, P1 and P2. Proof of frame encapsulation is more involved: If $\Sigma[i]$ is a prestate in which control is with y in the C-frame of x, then $as(\Sigma[i-1])$ is a method call $v = r.m(\ldots)$. As y is owned by x, $owmmod(r)$ differs from `root` (ownership safety). Then $ownmod(r)$ is either `peer`, or `rep` (as the method call statically meets encapsulation). Let control be with z in $\Sigma[i-1]$. Then, due to ownership safety, either (A) z is owned by x, or (B) $z = x$. In case (A), z can reach y via r. In case (B), r has a subreference v or $\texttt{this}.f$ that evaluates to an object that can reach y (as r evaluates to y). In either case, P2 ensures that z is in the C-frame. Then P1 and SR7.5 ensure that control is with C in $\Sigma[i-1]$.

Now we can present the outline for case (2). Let control in $\Sigma[i-1]$ be with x and in class C. Let $\Sigma[i..j]$ be a method execution. Let $inv_D(x)$ ($D \subset C$) hold in $\Sigma[i]$. If $inv_D(x)$ holds in $\Sigma[k] \in \Sigma[i..j-1]$, but not in $\Sigma[k+1]$, then $\Sigma[k]$ and $\Sigma[k+1]$ differ either (A) on a field of x declared in class D, or (B) on an object y in the D-frame of x (inv_D is frame based). In either case, there is a state $\sigma \in \Sigma[i..k]$ in which control is with x and in a class E, $E \subseteq D$: in case (A), control in $\Sigma[k]$ is with x and in a class E, $E \subseteq D$ (classical encapsulation, fields of subclasses cannot be assigned to). In case (B), control is with y in $\Sigma[k]$ (classical encapsulation). Then in the state $\Sigma[l]$ from which y was called, control was either (C) with x and in C, or (D) with another object in the C-frame of x (frame encapsulation). In either case, $i \geq l < k$. In case (D), to find σ, the reasoning of case (B) can be applied again to $\Sigma[l]$. Given this state σ, we know there must be a poststate $\sigma' \in \Sigma[k..j]$ in which control is with x and in a class E, $E \subseteq D$. Due to PO7.1, $inv_D(x)$ is re-established in σ'. Given this reasoning, it can be concluded that $inv_D(x)$ holds in $\Sigma[j]$. Combining all the above, we formulate the following lemma.

Lemma 7.4. *If program P meets SR6.1-SR6.5, SR7.1-SR7.6, PO7.1 and PO7.2, then every execution of P satisfies local consistency.*

Proof is fairly straightforward: If control in poststate or horizontal call state σ is with x and in C, PO7.1, PO7.2 ensure every $inv_B(x)$, $C \subseteq B$, holds in σ. Furthermore, it has been ensured that if control is with x and in class C in state σ, then every $inv_D(x)$, $type(x) \subseteq D \subset C$, holds. Then the invariants of x hold, which concludes the proof.

8 Future and Related Work

Non-modular verification techniques for the CII are given in [HK00, PH97]. That the CII is not suitable for non-local invariants is also observed in [BHKW01], where it is proposed to make components explicit at the level of OCL designs. A component invariant is interpreted to hold when control is outside the component. The notions of component and ownership seem closely related. A problem might be that components cannot easily capture class level subordinate relations.

Considering programs with executions where the LRII does not capture the intention of the specifier and where refactoring is either impossible or undesirable, two cases can be distinguished.

(A) An invariant is not intended to hold where the LRII guarantees it (the LRII is too strong). Observations 3.1 and 3.2 suggest that such an execution represents an uncommon scenario. One can use specification constructs that make explicit that a certain invariant does *not* have to hold in a specific pre- or postcondition (note that this constitutes a refinement of the interpretation). In [MHKL08a], this approach is used to deal with the callbacks in the Observer Pattern [GHJV95]. Alternatively, a construct could allow methods to be placed in a different layer than the defining class.

(B) An invariant is intended to hold where the LRII does not require it (the LRII is too weak). If the execution represents a common scenario that can be identified, a further refinement of the interpretation of invariants might be needed. Otherwise, the invariant property can be made explicit in the pre- or postcondition where the property is expected to hold. The additional specification overhead is not an issue as the scenario is uncommon. If the definition of an invariant is intended to be hidden, *predicate abstraction* techniques [BDF$^+$04, Kas06, LM04, MHKL08a, Par05] can be used, which allow to reason about a predicate without exposing its definition. These techniques are orthogonal to the discussion on the interpretation of invariants.

Note that [BDF$^+$04, LM04] use the Boogie methodology for invariants, in which invariants *only* provide predicate abstraction. This methodology is very flexible, as no additional property is guaranteed. However, specifying which invariants hold in the pre- or poststates of a method is done ad-hoc and requires additional specification overhead. Also, the specifier of a class C cannot specify that an invariant of C holds in visible states of methods of other classes. A detailed investigation of this trade-off is future work. Extensions of this methodology treat multi-threaded programs [JSPS07], as well as invariants that depend on quantifications over objects [PCdB05], and/or static fields [LM05]. Finally, [BN04] provides a modular verification technique for invariants that are not ownership based. Other such techniques (not based on the Boogie methodology) are discussed in [MHKL08a, MPHL06]. These are largely orthogonal to our approach.

If needed, the layer ordering can be replaced by a partial ordering. Several way to construct such a partial ordering can be found in the literature. For instance,

[LM05] constructs an ordering where class C is preceded by class D if C imports D or if D extends C (and uses it to express subordination between static class invariants). Alternatively, [JSPS07] constructs a partial order of *locklevels* (and uses it to avoid deadlock), where a new locklevel is defined using a *between* construct that takes lower and higher locklevels. An advantage of using such a partial order is that if a class C is reimplemented, C can define additional classes it precedes without requiring client reverification (whereas increasing the layer of a class does require client reverification). However, establishing that this does not introduce a cycle in the ordering might require the specifications of all classes in the program. Furthermore, note that a subclass that precedes additional classes cannot be used where a superclass is expected. Our coarse-grained way to define the ordering mitigates this problem. It also requires less verification overhead.

The Spec# programming system [BLS05] combines the Boogie methodology with a RII-like interpretation. Implementing our approach in Spec# is feasible and is considered future work. This would ease the study of larger examples to confirm the practicality of the approach.

9 Conclusion

The topic of this chapter is the formal interpretation and verification of invariants in OO designs. Common scenarios are identified in which current interpretations do not allow for (easy) specification. The reason is that current interpretations do not account for the layering that is present in many OO designs. In particular, these interpretations do not exploit the subordinate relations at the class level that result from the layering. The chapter shows how these layers can be made explicit in formal specifications with very little specification overhead. Furthermore, it introduces a refined interpretation of invariants that exploits these explicit layers. Together, this allows for easy specification of class level subordinate relations. The chapter presents a sound and modular verification technique to establish that programs satisfy the refined interpretation.

Acknowledgements We are grateful for the helpful in-depth comments from the anonymous referees.

A Appendix

This appendix presents a trace semantics for the Java-like language introduced in section 2. This is done in two steps. First, a small-step semantics is introduced using the transition relation $\rightsquigarrow$ on states. Then, the trace semantics is defined using this relation. Note that several of the terms used are defined in section 2.

The central notion is that of an (execution) state. A state σ is a 4-tuple $(\sigma_{sf}, \sigma_{os}, \sigma_{stack}, \sigma_{stat})$ of a stack frame σ_{sf}, an object store σ_{os}, a call stack σ_{stack} and an extended statement σ_{stat}. A *call stack* is a sequence of tuples (sf, v), where sf is a stack frame and v is a local variable (the intuition is that on the call stack, we store the stack frame at the point of a method call as well as the variable the method's return value should be assigned to). An extended statement is either a statement or the symbol $\_$ (the 'empty statement').

as maps a state σ to an extended statement (the active statement). $as(\sigma) = nosemicolon(\sigma_{stat})$. If σ_{stat} is a statement $S_1;S_2$, then $nosemicolon(\sigma_{stat}) = nosemicolon(S_1)$. Otherwise, $nosemicolon(\sigma_{stat}) = \sigma_{stat}$. $dom(f)$ yields the domain of (partial) function f. $f[A \mapsto B]$ is the function like f, but with A mapped to B. This can be used whether or not $A \in dom(f)$. $[]$ is the empty function. $arbitr$ takes a type T and returns a non-deterministically selected value of type T. $\triangleright$ denotes prefixing a sequence with an element. $alloc$ takes a class C and an object store os and returns a tuple of (1) a non-deterministically selected object x of class C such that $x \notin dom(os)$, and (2) the heap like os, but which also, for every class $D \supseteq C$, for every field f declared in D, maps $x.f$ to the default initial value for f's type. It is assumed class `Object` declares at least one field. To simplify the presentation, it is assumed that every method has exactly one parameter named `p`, and that there is no *overloading*: a class does not declare two methods with the same name (there can be *overriding*: class C and class D can both declare a method m). Removing these restrictions is straightforward. If statement S is the body of method m declared in class C, then $body(C, m) = S$. If C is not `Object` and does not declare a method m, then $body(C, m) = body(super(C), m)$. Constructors are included for completeness. If S is the body of the constructor of C, then $cbody(C) = S$. In the reduction rules below, $b \in \mathsf{BoolE}, e \in \mathsf{SimpleE}, \{S_1, S_2, S_3\} \subset \mathsf{Stat}$, $\nu \in values, x \in objects, sf \in stack\ frames, os \in object\ stores$, and $CS \in call\ stacks$. To improve readability, brackets around states are omitted.

$$\frac{sf, os, CS, S_1 \rightsquigarrow sf', os', CS', S_3}{sf, os, CS, S_1;\ S_2 \rightsquigarrow sf', os', CS', S_3;\ S_2} \qquad \frac{sf, os, CS, S_1 \rightsquigarrow sf', os', CS', \_}{sf, os, CS, S_1;\ S_2 \rightsquigarrow sf', os', CS', S_2}$$

$$\frac{eval(b, sf, os) = true}{sf, os, CS, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \rightsquigarrow sf, os, CS, S_1}$$

$$\frac{eval(b, sf, os) = false}{sf, os, CS, \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \rightsquigarrow sf, os, CS, S_2}$$

$$\frac{eval(b, sf, os) = true}{sf, os, CS, \texttt{while } b \texttt{ do } S_1 \rightsquigarrow sf, os, CS, S_1; \texttt{ while } b \texttt{ do } S_1}$$

$$\frac{v \in dom(sf) \qquad eval(e, sf, os) = \nu}{sf, os, CS, v = e \rightsquigarrow sf[v \mapsto \nu], os, CS, \_}$$

$$\frac{eval(b, sf, os) = false}{sf, os, CS, \texttt{while } b \texttt{ do } S_1 \rightsquigarrow sf, os, CS, \_}$$

$$\frac{eval(e, sf, os) = \nu \qquad eval(r, sf, os) = x \qquad x.f \in dom(os)}{sf, os, CS, r.f = e \rightsquigarrow sf, os[x.f \mapsto \nu], CS, \_}$$

$$\frac{v \notin dom(sf)}{sf, os, CS, T\ v \rightsquigarrow sf[v \mapsto arbitr(T)], os, CS, \_}$$

$$\frac{eval(e, sf, os) = \nu}{sf, os, (sf', v) \triangleright CS, \texttt{return } e \rightsquigarrow sf'[v \mapsto \nu], os, CS, \_}$$

$$\frac{\begin{array}{c} v \in dom(sf) \\ eval(r, sf, os) = x \qquad eval(e, sf, os) = \nu \qquad body(type(x), m) = S \end{array}}{sf, os, CS, v = r.m(e) \rightsquigarrow [\texttt{this} \mapsto x, \texttt{p} \mapsto \nu], os, (sf, v) CS, S}$$

$$\frac{v \in dom(sf) \qquad eval(e, sf, os) = \nu \qquad cbody(C) = S \qquad alloc(C, os) = (os', x)}{sf, os, CS, v = \texttt{new } C(e) \rightsquigarrow [\texttt{this} \mapsto x, \texttt{p} \mapsto \nu], os', (sf, v) \triangleright CS, S}$$

If program P has a main method with a body S and a parameter of type T, then the semantics of P is the following set of traces:

$\{\Sigma \mid isReduction(\Sigma)$ and $hasProperFirstElem(\Sigma, S, T)$ and $noReducableLastElem(\Sigma)\}$, where

$isReduction(\Sigma)$ holds if for every two consecutive states σ and σ' in Σ, $\sigma \rightsquigarrow \sigma'$, and
$hasProperFirstElem(\Sigma, S, T)$ holds if there is a $\nu \in T$ such that $\Sigma[0] = ([\texttt{p} \mapsto \nu], [], ([], \texttt{result}), S)$, and
$noReducableLastElem(\Sigma)$ holds if either Σ is infinite, or has a last element σ such that $\sigma \not\rightsquigarrow$.

A program execution Σ *terminates normally* if it has a last element σ and $\sigma_{stat} = \_$. In that case, σ_{sf} maps `result` to the value returned by the main method.

CHAPTER 6

Proving Consistency of Pure Methods and Model Fields

This chapter contains the following paper, with minor editorial changes: *K. Rustan M. Leino and Ronald Middelkoop. Proving Consistency of Pure Methods and Model Fields. In Fundamental Approaches to Software Engineering (FASE 2009), volume 5503 of LNCS, pages 231–245. Springer, 2009* [LM09]. It is available online.

abstract Pure methods and model fields are useful and common specification constructs that can be interpreted by the introduction of axioms in a program verifier's underlying proof system. Care has to be taken that these axioms do not introduce an inconsistency into the proof system. This chapter describes and proves sound an approach that ensures no inconsistencies are introduced. Unlike some previous syntax-based approaches, this approach is based on semantics, which lets it admit some natural but previously problematical specifications. The semantic conditions are discharged by the program verifier using an SMT solver, and the chapter describes heuristics that help avoid common problems in finding witnesses with trigger-based SMT solvers. The chapter reports on the positive experience with using this approach in Spec# for over a year.

1 Introduction

Pure methods and model fields [Cok05, CLSE05] are useful and common specification constructs. By marking a method as pure, the specifier indicates that it can be treated as a function of the state. It can then be called in specifications.

```
pure T m(T' p)
  requires P;
  ensures Q;
```

Figure 1.1: Template

```
pure int bad()
  ensures false;
  { return 4; }
```

Figure 1.2: Inconsistency

```
pure int n(int i)
  ensures result = this.p(i);
pure int p(int i)
  ensures result = this.n(i)+1;
```

Figure 1.3: Harmful indirect recursion

Model fields provide a way to abstract from an object's concrete data. A problem with either technique is that it can introduce an inconsistency into the underlying proof system. In this chapter, we discuss how to prove (automatically) that no such inconsistency is introduced while allowing a rich set of specifications.

Starting from a review of the setting, the problem, and previous solutions, this section leads up to an overview of our contributions.

Pure Method Specifications. Figure 1.1 shows the template for a pure method specification (for simplicity, we show only a single formal parameter, named p). As usual, **requires** declares the method's precondition P, **ensures** declares the method's postcondition Q, and **result** denotes the method's return value. The only free variables allowed in P are this and p. In Q, **result** is allowed as well.

A Deduction System. Marking method m as pure adds an uninterpreted total function $\#m : C \times T' \rightarrow T$ (a *method function* [DL07]) to the specification language. In predicates in the specification, the expression $E_0.m(E_1)$ is treated as syntactic sugar for $\#m(E_0, E_1)$. Furthermore, method function $\#m$ is axiomatized in the underlying deduction system for first-order logic by the following axiom:[1]

$$\forall \sigma \in \Sigma \bullet \; [\![\forall \texttt{this}\colon C,\ \texttt{p}\colon T' \bullet \quad P \Rightarrow Q[\#m(\texttt{this},\texttt{p})/\textbf{result}] \;]\!]\sigma \tag{6.1}$$

Here, Σ denotes the set of well-formed program states. Partial function $[\![E]\!]\sigma$ evaluates expression E to its value in state σ. $[\![\#m(E_0, E_1)]\!]\sigma$ is defined as $\#m(\,[\![E_0]\!]\sigma,\ [\![E_1]\!]\sigma)$. Other details of this evaluation are unimportant here. $P[E/v]$ denotes the predicate like P, but with capture-avoiding substitution of variable v by E. For instance, pure method has from Fig. 1.4 introduces uninterpreted total function $\#\texttt{has} : \texttt{Node} \times \texttt{Object} \rightarrow \textbf{bool}$, and axiom $\forall \sigma \in \Sigma \bullet [\![\forall \texttt{this}\colon \texttt{Node},\ \texttt{obj}\colon \textbf{bool} \bullet \#\texttt{has}(\texttt{this},\texttt{obj}) = \#\texttt{count}(\texttt{this},\texttt{obj}) > 0]\!]\sigma$.

Consistency of Deduction System. If one is not careful, pure methods can introduce an inconsistency into the deduction system. As an obvious example, consider Fig. 1.2. This definition introduces *false* as an axiom into the deduction system (more precisely, it introduces $\forall \sigma \in \Sigma \bullet\ [\![\forall \texttt{this}\colon C \bullet \mathit{false}]\!]\sigma$). So, it has to

[1]The axiomatization differs slightly in the presence of class invariants. To simplify the presentation, invariants are not considered.

```
class Node {
  Object val;
  rep Node next;
  pure int count(Object obj)
    ensures result = (obj = this.val ? 1 : 0) +
                     (this.next = null ? 0 : this.next.count(obj));
  pure bool has(Object obj)
    ensures result = this.count(obj) > 0;
} //rest of class omitted
```

Figure 1.4: Singly linked list (see Sect. 4.1 for **rep**)

be ensured that for all possible values of the arguments of method function $\#m$, there is a value that the function can take. Insuring this by requiring a proof of total correctness of the implementation of m before adding the axiom is highly impractical. If $\#m$ is constrained only by the axiom introduced by m, then it suffices to prove property (6.2):

$$\forall \sigma \in \Sigma \bullet \; [\![\forall \texttt{this}\colon C,\ \texttt{p}\colon T' \bullet \exists x\colon T \bullet \quad P \Rightarrow Q[x/\textbf{result}] \;]\!] \sigma \tag{6.2}$$

If other axioms can also constrain $\#m$, as is the case in the presence of mutual recursion, then property (6.2) needs to simultaneously mention all methods involved. We aim for sound *modular verification*, which means being able to verify a program's modules separately, just like a compiler performs separate compilation of modules. If the mutual recursion can span module boundaries, then there may be no verification scope that has information about all the methods that need to be simultaneously mentioned. Therefore, the consistency of mutual recursion among pure methods is usually stated in a form different from (6.2).

Previous Solutions. Darvas and Müller [DM06] prove that inconsistency is prevented if the following two measures are taken: (A) the axiom that is introduced into the deduction system for a method function $\#m$ is not proposition (6.1), but (6.2) $\Rightarrow$ (6.1), and (B) recursion in the pure method axioms is disallowed unless it is direct and well-founded. For example, measure A prevents the pure methods in Fig. 1.2 from introducing an inconsistency, and measure B forbids the specifications in Fig. 1.3, whose axioms would otherwise introduce an inconsistency.

Darvas and Leino [DL07] discuss a problem with measure A, namely that an axiom of the form (6.2) $\Rightarrow$ (6.1) is unsuitable for automatic reasoning using today's trigger-based SMT solvers like Simplify and Z3 [DNS05, dMB08]. More specifically, these solvers are unable to come up with a witness for the existential quantification in (6.2) even in simple cases. This means that property (6.1) is 'silently ignored', which renders the pure method useless (and possibly confuses the user).

To circumvent the practical problem with measure A, Darvas and Leino introduce a simple syntactic check that allows one to conclude that (6.2) holds once and for all [DL07]. Thus, (6.1) can be introduced as an axiom into the deduction system

```
pure int findInsertionPosition(int N)
  requires 0 ≤ N;
  ensures 0 ≤ result & result ≤ N;
```

```
pure int max(int x, int y)
  ensures (x ≤ y ⇒ result = y) &
          (y ≤ x ⇒ result = x);
```

Figure 1.5: Previous syntactic checks forbid these methods; our semantic checks allow them.

```
pure bool isEven(int n)
  requires 0 ≤ n;
  ensures
    result = (n = 0 ? true : this.isOdd(n-1));
  measuredBy 2n;
```

```
pure bool isOdd(int m)
  requires 0 ≤ m;
  ensures result ≠ this.isEven(m);
  measuredBy 2m+1;
```

Figure 1.6: Odd and even (see Sect. 4.3 for **measuredBy**)

without fear of inconsistencies. However, the syntactic check is restrictive and prevents a number of natural and useful specifications, including the two in Fig. 1.5. Syntactic checks cannot guarantee the consistency of `findInsertionPosition`, because its result value is constrained by two inequalities, or of `max`, because its result-value constraints are guarded by antecedents.

Measure B is a Draconian way of dealing with mutual recursion. The syntactic check of Darvas and Leino [DL07] improves on this situation. However, this check is still restrictive; for instance, it does not permit the example in Fig. 1.6.

A Glimpse of Our Semantic Solution. In our solution, we use heuristics to guess candidate witness expressions for (2). Then we verify that in every program state allowed by the pure method's precondition, one of these candidates establishes the postcondition. For example, for pure method `max` in Fig. 1.5, we generate three candidate witnesses 1, x, and y, and construct a program snippet of the form:

$$\begin{array}{l} r := 1;\ \ \textbf{if}\ ((x \le y \Rightarrow r = y)\ \ \&\ \ (y \le x \Rightarrow r = x))\ \{\ \textbf{return}\ r;\ \} \\ r := x;\ \ \textbf{if}\ ((x \le y \Rightarrow r = y)\ \ \&\ \ (y \le x \Rightarrow r = x))\ \{\ \textbf{return}\ r;\ \} \\ r := y;\ \ \textbf{return}\ r; \end{array}$$

and then attempt to verify, using our program verifier's machinery, that this program snippet establishes the postcondition of the pure method.

Model Fields. Model fields introduce similar problems. A model field gives a way to hide details of an object's concrete state. Figure 1.7 gives an example (taken from [LM06]) of the use of model fields: by updating the satisfies clauses, e.g., to `this.width = this.w` and `this.heigth = this.h`, `Rectangle` can be re-implemented with two `int`s `w` and `h`, without affecting the verification of other classes. For every model field **model** $T\ f$ **satisfies** Q in a class C, a total function $\#f : C \to T$ (an *abstraction function*) is added to the specification language. In predicates in the specification, the expression $E.f$ is treated as syntactic sugar for $\#f(E)$. Abstraction function $\#f$ is axiomatized in the deduction system by

```
class Rectangle {
  int x1,y1,x2,y2; //lower left and upper right corner
  model int width satisfies this.width = this.x2-this.x1;
  model int height satisfies this.height = this.y2-this.y1;
  void scaleH(int factor)
    requires 0 ≤ factor;
    ensures this.width = old(this.width) ∗ factor/100;
    { this.x2 := (this.x2 - this.x1 ) ∗ factor/100 + this.x1; }
} //rest of class omitted
```

Figure 1.7: Model fields

an axiom $\forall \sigma \in \Sigma \bullet\ [\![\forall \texttt{this} : C \bullet Q]\!]\sigma$.[2] This axiom is not visible outside of C's module. The axiomatization problems we have described for method functions apply to abstraction functions as well: for the purpose of this chapter, a model field f that satisfies predicate Q can be treated as a parameterless pure method with postcondition Q, with **result** for $\texttt{this}.f$.

Contributions. The contributions of this chapter are the following:

1. We formalize and strengthen an implicit claim from [DL07]: No inconsistency is introduced by axioms of the form (6.2) ⇒ (6.1) if every method function call in a pure method m's specification lies below m in a partial order $\prec$ (Sect. 2).
2. We present a much improved scheme that leverages the power of the theorem prover to prove (6.2) once-and-for-all (Sect. 3).
3. We introduce a permissive definition for $\prec$ that improves on the one in [DL07] and allows a greater degree of (mutual) recursion than before (Sect. 4).

We report on our experience and discuss related work in Sect. 5.

2 Avoiding Inconsistency

In this section, we identify proof obligations that allow axioms of form (6.1) to be added to the deduction system without introducing inconsistencies.

Let there be $N + 1$ pure methods in the program fragment that is to be verified, labeled $m_0, \ldots m_N$. For simplicity, assume that there are no static pure methods and that every pure method m_i has exactly one formal parameter $\texttt{p}_i$ of type T'_i (extending to an arbitrary number of parameters is straightforward). Let T_i be the return type of pure method m_i. Let C_i be the class that defines m_i. Let predicates Pre_i and $Post_i$ be the pre- and postconditions of m_i. $PureAx$, defined below, represents the axioms introduced by pure methods (reformulated into a single proposition). We use $\equiv$ to define syntactical shorthands.

[2]More axioms might be added depending on the methodology, see Sect. 5.

Definition 2.1 (*PureAx*).

$$\begin{array}{rcl} \mathit{Spec}_i & \equiv & \mathit{Pre}_i \Rightarrow \mathit{Post}_i \\ \mathit{MSpec}_i & \equiv & \forall \mathtt{this}\colon C_i,\ \mathtt{p}_i\colon T'_i \bullet \mathit{Spec}_i[\#m_i(\mathtt{this}, \mathtt{p}_i)/\mathbf{result}] \\ \mathit{PureAx} & \equiv & \forall \sigma \in \Sigma \bullet\ [\![\mathit{MSpec}_0 \wedge \ldots \wedge \mathit{MSpec}_N]\!]\sigma \end{array}$$

Let *Prelude* be the conjunction of all axioms in the deduction system that are not introduced by a pure method. The goal is to find proof obligations *POs* such that if *Prelude* is consistent and *POs* hold, then adding the axioms for pure methods does not introduce inconsistencies. Theorem 2.1 formalizes this goal:

Theorem 2.1. *Prelude* $\Rightarrow$ (*POs* $\Rightarrow$ *PureAx*)

The remainder of this section discusses the proof obligations *POs* that we use to ensure that Thm. 2.1 holds. The theorem itself is proven to hold in Ap. A. If there is no recursion in pure method specifications, then Thm. 2.1 can be shown to hold using $POs \equiv PO1$ (see [DM06]):

Definition 2.2 ($PO1$).

$$\begin{array}{rcl} PO1_i & \equiv & \forall \sigma \in \Sigma \bullet\ [\![\forall \mathtt{this}\colon C_i,\ \mathtt{p}_i\colon T'_i \bullet \exists \mathbf{result}\colon T_i \bullet \mathit{Spec}_i]\!]\sigma \\ PO1 & \equiv & PO1_0 \wedge \ldots \wedge PO1_N \end{array}$$

Note that $PO1_i$ is equivalent to proposition (6.2) from the introduction.

When there is (mutual) recursion, the crucial property that is in jeopardy is *functional consistency*: if the same function is called twice from the same state and the parameters of the two calls evaluate to the same values, then the two calls evaluate to the same value. For instance, consider the methods in Fig. 1.3. If pure methods add propositions of the form (6.1) to the deduction system, then these method definitions allow one to deduce that $\#\mathtt{n}(\mathtt{this}, \mathtt{i}) = \#\mathtt{n}(\mathtt{this}, \mathtt{i}) + 1$, which contradicts functional consistency of $\#\mathtt{n}$. More formally, since $[\![\#m_i(E_0, E_1)]\!]\sigma = \#m_i(\ [\![E_0]\!]\sigma,\ [\![E_1]\!]\sigma)$ (see Sect. 1), it follows immediately that $\forall \sigma \in \Sigma,\ i \in [0, N] \bullet [\![\forall c_0, c_1\colon C_i,\ p_0, p_1\colon T'_i \bullet c_0 = c_1 \wedge p_0 = p_1 \Rightarrow \#m_i(c_0, p_0) = \#m_i(c_1, p_1)]\!]\sigma$. The proof obligations must ensure that the axioms introduced by pure methods do not contradict functional consistency.

For convenience, we define the equivalence relation $\sim$:

Definition 2.3 ($\sim$).

$$[\![\#m_i(E_0, E_1) \sim \#m_j(E_2, E_3)]\!]\sigma \ \overset{def}{=}\ i = j \wedge\ [\![E_0 = E_2 \wedge E_1 = E_3]\!]\sigma$$

Then $\forall \sigma \in \Sigma \bullet [\![\#m_i(E_0, E_1) \sim \#m_j(E_2, E_3) \Rightarrow \#m_i(E_0, E_1) = \#m_j(E_2, E_3)]\!]\sigma$.

To ensure that recursive specifications do not lead to an axiomatization that contradicts functional consistency, we require the verifier to ensure that a function call in the axiomatization of $\#m_i(o, x)$ does not (indirectly) depend on the value of $\#m_i(o, x)$. To this end, we introduce the strict partial order $\prec$ on method function calls (i.e., $\prec$ is an irreflexive and transitive binary relation on expressions of the shape $\#m_i(E_0, E_1)$). The definition of $\prec$ is not relevant to the proof as long as (1) $\prec$ is well-founded, and (2) the following lemma holds:

Lemma 2.2.

$$\forall \sigma \in \Sigma,\ i, j \in [0, N]\ \bullet [\![\forall c_0\colon C_i,\ x_0\colon T'_i,\ c_1\colon C_j,\ x_1\colon T'_j\ \bullet \\ \#m_i(c_0, x_0) \prec \#m_j(c_1, x_1)\ \Rightarrow\ \#m_i(c_0, x_0) \not\prec \#m_j(c_1, x_1)]\!]\sigma$$

In Sect. 4, we present a definition of $\prec$ that is suitable for our proof system. Proof obligation $PO2$, defined below, requires every method function call in the specification of m_i to lie below $\#m_i(\texttt{this}, \texttt{p}_i)$ in the order $\prec$ in every state in which the result of the call is relevant.

Definition 2.4 ($PO2$)**.** *Let $i, j \in [0, N]$. Let $NrOfCalls_{i,j}$ be the number of calls to $\#m_j$ in $Spec_i$. If $l + 1 = NrOfCalls_{i,j}$, and $k \in [0, l]$, then*
$Call_{i,j,k}$ is the expression that is the k'th call to $\#m_j$ in $Spec_i$
$Spec_{i,j,k}$ is $Spec_i$, but with a fresh variable substituted for the k'th call to $\#m_j$

$$\begin{array}{lll}
Smaller_{i,j,k} & \equiv & Call_{i,j,k} \prec \#m_i(\texttt{this}, \texttt{p}_i) \\
NotRel_{i,j,k} & \equiv & \forall\,\textbf{result}\colon T_i,\ x\colon T'_j \bullet Spec_{i,j,k} = Spec_i \\
PO2_{i,j,k} & \equiv & \forall \sigma \in \Sigma \bullet\ [\![\forall\,\texttt{this}\colon C_i,\ \texttt{p}_i\colon T'_i \bullet Smaller_{i,j,k} \vee NotRel_{i,j,k}]\!]\sigma \\
PO2_{i,j} & \equiv & PO2_{i,j,0} \wedge \ldots \wedge PO2_{i,j,l} \\
PO2_i & \equiv & PO2_{i,0} \wedge \ldots \wedge PO2_{i,N} \\
PO2 & \equiv & PO2_0 \wedge \ldots \wedge PO2_N
\end{array}$$

The intuition behind *NotRel* is that $Call_{i,j,k}$ in $Spec_i$ is not relevant in $\sigma \in \Sigma$ if the result value of $Call_{i,j,k}$ is not relevant to the value of $\#m_i(\texttt{this}, \texttt{p}_i)$ in σ. That is, for any value of **result**, the value of $Spec_i$ is the same for any result of $Call_{i,j,k}$. As an extreme example, suppose $Spec_i$ is `false` $\Rightarrow$ **result** `=` `this.`m_i`(p) + 1`. Then $Smaller_{i,i,0}$ never holds, but $NotRel_{i,i,0}$ always holds as $\forall \sigma \in \Sigma \bullet [\![\forall\,\texttt{this}\colon C_i,\ \texttt{p}_i\colon T'_i,\ \textbf{result}\colon T_j,\ x\colon T'_j \bullet$ `(false` $\Rightarrow$ **result** `= this.`m_i`(p) + 1) = (false` $\Rightarrow$ **result** `= x + 1)` $]\!]\sigma$. Then $PO2_{i,i,0}$ is met, and hence $PO2_i$ is met. We show a more realistic example in Sect. 4.1.

In this section, we formalized the problem sketched in the introduction. Furthermore, we introduced high-level proof obligations that ensure that the extension of the *Prelude* with the axiomatization of pure methods does not introduce inconsistencies: in Ap. A we prove that Thm. 2.1 holds if $POs \equiv PO1 \wedge PO2$. In the next two sections, we address two remaining practical concerns: we provide heuristics to prove $PO1$, and define the partial ordering $\prec$ used in $PO2$.

3 Heuristics for Establishing $PO1$

Proof obligation $PO1$ poses serious difficulties for automatic verification. Even in simple cases, automatic theorem provers are unable to come up with a witness for the existential quantification $\exists\,\mathbf{result} : T_i \bullet \mathit{Spec}_i$ in $PO1_i$. As a solution, [DL07] proposes only to allow a pure method m_i when (1) it has a postcondition of the form **result** *op* E or E *op* **result**, where *op* is a binary operator from the set $\{=, \geq, \leq, \Rightarrow, \Leftrightarrow\}$, and (2) E is an expression that does not contain **result**. If these conditions are met, then E is a witness for the quantification, i.e., $\forall \sigma \in \Sigma \bullet\ [\![\forall \mathtt{this} : C_i, \mathtt{p}_i : T_i' \bullet \mathit{Spec}_i[E/\mathbf{result}]]\!]\sigma$, and therefore $PO1_i$ holds.

This solution has the advantage that it only requires a simple syntactic check. However, it is quite restrictive. Unfortunately, not much more can be done with syntactic checks. For instance, consider method `findInsertPosition` from Fig. 1.5. Here, 0 is a witness (as $\mathtt{0} \leq \mathtt{N} \Rightarrow \mathtt{0} \leq \mathtt{0} \wedge \mathtt{0} \leq \mathtt{N}$). However, a syntactic check cannot establish that $0 \leq \mathtt{N}$. Our solution is to leverage the power of the theorem prover. Consider the scheme below.

1. Find a witness candidate E.
2. If $\forall \sigma \in \Sigma \bullet\ [\![\forall \mathtt{this} : C_i, \mathtt{p}_i : T_i' \bullet \mathit{Spec}_i[E/\mathbf{result}]]\!]\sigma$ can be established by the theorem prover, then $PO1_i$ holds. Otherwise, the program is rejected.

This scheme is more powerful than the syntactic check of [DL07]. For instance, unlike the syntactic check, it allows `findInsertPosition`, assuming that 0 is found as a witness candidate. Before we discuss how to find witness candidates, we improve on the scheme above in one important way. Consider method `max` from Fig. 1.5. $PO1$ cannot be established for `max` using the scheme above, no matter which witness candidate is found. In particular, neither $\mathit{Spec}_{\mathtt{max}}[\mathtt{x}/\mathbf{result}]$ nor $\mathit{Spec}_{\mathtt{max}}[\mathtt{y}/\mathbf{result}]$ holds. The problem is that the scheme requires that there is a witness that holds in all cases. $PO1$ only requires that in all cases, there is a witness. The latter is true for `max`, but the former is not. If $\mathtt{x} \leq \mathtt{y}$, then `y` is a witness. If $\mathtt{y} \leq \mathtt{x}$, then `x` is a witness. That is, $\mathit{Spec}_{\mathtt{max}}[\mathtt{x}/\mathbf{result}] \vee \mathit{Spec}_{\mathtt{max}}[\mathtt{y}/\mathbf{result}]$ holds. Therefore, $\exists \mathbf{result} \colon \mathbf{int} \bullet \mathit{Spec}_{\mathtt{max}}$ holds, and $PO1$ holds. Based on this reasoning, the scheme presented above is replaced by the more liberal scheme below.

1. Find witness candidates $E_0, \ldots, E_n$.
2. If $\forall \sigma \in \Sigma \bullet\ [\![\forall \mathtt{this} \colon C_i,\ \mathtt{p}_i \colon T_i' \bullet \mathit{Spec}_i[E_0/\mathbf{result}] \vee \ldots \vee \mathit{Spec}_i[E_n/\mathbf{result}]]\!]\sigma$ can be established by the theorem prover, then $PO1_i$ holds. Otherwise, the program is rejected.

Next, we present an algorithm to find witness candidates for a pure method. We assume that there is a function $\mathit{kind} : \mathit{Type} \rightarrow \{\mathit{Bool}, \mathit{Enum}, \mathit{Num}, \mathit{Ref}\}$ that distinguishes four kinds of types. The algorithm uses a Haskell-like switch that uses pattern matching and does not fall through. For example, case A of B → C D → E _ → F should be read as 'if A matches B, then C, else if A matches D, then E, else F'. The witness candidates for a pure method m_i with return type T_i and postcondition Post_i are given by $\mathit{wcs}(T_i, \mathit{Post}_i)$. Below, *wcs* and its helper functions are defined, discussed and illustrated by a number of examples. Note that $\mathit{ExprSet} \equiv \mathit{Set\ of\ Expression}$, and that $|\,S\,|$ returns the size of set S.

Definition 3.1 (*wcs*).

$$
\begin{array}{l}
\mathit{wcs} : \mathit{Type} \times \mathit{Predicate} \to \mathit{ExprSet} \\
\mathit{wcs}(T, P) \overset{\mathit{def}}{=} \ \mathit{case}\ \mathit{kind}(T)\ \mathit{of} \\
\quad \mathit{Bool} \ \ \to \{\mathit{true}, \mathit{false}\} \\
\quad \mathit{Enum} \to \textit{the enumerator list (i.e. the list of enumeration constants) of } T \\
\quad \mathit{Ref} \ \ \ \to \mathit{let}\ \mathit{euld}(P) = \langle S_0, S_1, S_2, S_3 \rangle\ \mathit{in}\ S_0 \cup \{\texttt{null}\} \\
\quad \mathit{Num} \ \to \mathit{let}\ \mathit{euld}(P) = \langle S_0, S_1, S_2, S_3 \rangle\ \mathit{in} \\
\qquad\qquad S_0 \cup \mathit{dupl}(S_1, |\,S_3\,|, \mathit{true}) \cup \mathit{dupl}(S_2, |\,S_3\,|, \mathit{false}) \cup \mathit{dupl}(\{1\}, |\,S_3\,|, \mathit{true})
\end{array}
$$

Definition 3.2 (euld).

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{euld} : \mathit{Predicate} \to \mathit{ExprSet} \times \mathit{ExprSet} \times \mathit{ExprSet} \times \mathit{ExprSet}} \\
\multicolumn{3}{l}{\mathit{euld}(P) \overset{\mathit{def}}{=} \ \mathit{case}\ P\ \mathit{of}} \\
\multicolumn{3}{l}{\quad \mathbf{result} = E\ \ \mathit{or}\ E = \mathbf{result} \to \langle \{E\}, \{\}, \{\}, \{\} \rangle} \\
\multicolumn{3}{l}{\quad \mathbf{result} \geq E\ \ \mathit{or}\ E \leq \mathbf{result} \to \langle \{\}, \{E\}, \{\}, \{\} \rangle} \\
\multicolumn{3}{l}{\quad \mathbf{result} \leq E\ \ \mathit{or}\ E \geq \mathbf{result} \to \langle \{\}, \{\}, \{E\}, \{\} \rangle} \\
\multicolumn{3}{l}{\quad \mathbf{result} \neq E\ \ \mathit{or}\ E \neq \mathbf{result} \to \langle \{\}, \{\}, \{\}, \{E\} \rangle} \\
\multicolumn{3}{l}{\quad \mathbf{result} > E\ \ \mathit{or}\ E < \mathbf{result} \to \mathit{euld}(\mathbf{result} \geq E + 1)} \\
\multicolumn{3}{l}{\quad \mathbf{result} < E\ \ \mathit{or}\ E > \mathbf{result} \to \mathit{euld}(\mathbf{result} \leq E - 1)} \\
\quad P_0 \vee P_1\ \ \mathit{or}\ P_0 \wedge P_1 & \to & \mathit{let}\ \mathit{euld}(P_0) = \langle S_0, S_1, S_2, S_3 \rangle,\ \mathit{and} \\
 & & \qquad \mathit{euld}(P_1) = \langle S'_0, S'_1, S'_2, S'_3 \rangle\ \ \mathit{in} \\
 & & \langle S_0 \cup S'_0, S_1 \cup S'_1, S_2 \cup S'_2, S_3 \cup S'_3 \rangle \\
\quad \neg P_0 & \to & \mathit{let}\ \ \mathit{euld}(P_0) = \langle S_0, S_1, S_2, S_3 \rangle\ \ \mathit{in} \\
 & & \langle S_3, \mathit{addOrSub1}(S2, \mathit{true}), \mathit{addOrSub1}(S1, \mathit{false}), S_0 \rangle \\
\quad P_0 \Rightarrow P_1\ \ \mathit{or}\ P_1 \Leftarrow P_0 & \to & \mathit{euld}(\neg P_0 \vee P_1) \\
\quad P_0 \Leftrightarrow P_1 & \to & \mathit{euld}((P_0 \wedge P_1) \vee (\neg P_0 \vee \neg P_1)) \\
\quad P_0\ ?\ P_1 : P_2 & \to & \mathit{euld}((P_0 \Rightarrow P_1) \wedge (\neg P_0 \Rightarrow P_2)) \\
\quad \_ & \to & \langle \{\}, \{\}, \{\}, \{\} \rangle
\end{array}
$$

Definition 3.3 ($\mathit{addOrSub1}$).

$$
\begin{array}{l}
\mathit{addOrSub1} : \mathit{ExprSet} \times \mathit{Bool} \to \mathit{ExprSet} \\
\mathit{addOrSub1}(\{E_0, \ldots, E_n\}, \mathit{isAdd}) \overset{\mathit{def}}{=} \\
\quad (\mathit{isAdd}\ ?\ \{E_0 + 1, \ldots, E_n + 1\} : \{E_0 - 1, \ldots, E_n - 1\})
\end{array}
$$

Definition 3.4 (dupl).

$$
\begin{array}{l}
\mathit{dupl} : \mathit{ExprSet} \times \mathbb{N} \times \mathit{Bool} \to \mathit{ExprSet} \\
\mathit{dupl}(\{E_1, \ldots, E_n\}, \mathit{duplCnt}, \mathit{isAdd}) \overset{\mathit{def}}{=} \\
\quad \mathit{duplExpr}(E_1, \mathit{duplCnt}, \mathit{isAdd}) \cup \ldots \cup \mathit{duplExpr}(E_n, \mathit{duplCnt}, \mathit{isAdd})
\end{array}
$$

Definition 3.5 ($\mathit{duplExpr}$).

$$
\begin{array}{l}
\mathit{duplExpr} : \mathit{Expression} \times \mathbb{N} \times \mathit{Bool} \to \mathit{ExprSet} \\
\mathit{duplExpr}(E, \mathit{duplCnt}, \mathit{isAdd}) \overset{\mathit{def}}{=} \\
\quad (\mathit{isAdd}\ ?\ \{E + 0, \ldots, E + \mathit{duplCnt}\} : \{E - 0, \ldots, E - \mathit{duplCnt}\})
\end{array}
$$

The intuition behind the $wcs(T, P)$ definition is as follows. If *kind(T)* $\in$ {*Bool, Enum*}, then there is no need to scan the postcondition for witness candidates. Instead, we make full use of the possibility to select multiple candidates and let every value of the type be a witness candidate. If *kind(T)* $\in$ {*Num, Ref*}, then function *euld* is used to scan P for *e*qualities, *u*pper bounds, *l*ower bounds, and *d*isequalities that contain **result**. More precisely, assume $euld(P) = (S_0, S_1, S_2, S_3)$. Let $cnf(P)$ yield the conjunctive normal form of P, and let $test(P, E) \equiv cnf(P)[E/\textbf{result}]$. Let E_0, E_1, E_2 and E_3 be elements of S_0, S_1, S_2 and S_3, respectively. Then for every $n \in \mathbb{N}$, each of $test(P, E_0)$, $test(P, E_1 + n)$ and $test(P, E_2 - n)$ has a satisfied conjunct. Also, at least one conjunct of $test(P, E_3)$ contains an unsatisfied disjunct. From *euld*'s result, witness candidates are extracted and where needed duplicated using function *dupl*.

We illustrate with several examples. Let $kind(T_i) = Num$. If $Post_i$ is **result** $= 4$, or **result** > 3, or **result** ≤ 4, then $euld(Post_i)$ is $\langle\{4\}, \{\}, \{\}, \{\}\rangle$, $\langle\{\}, \{4\}, \{\}, \{\}\rangle$, or $\langle\{\}, \{\}, \{4\}, \{\}\rangle$, respectively. In each case, $wcs(T_i, Post_i)$ $= \{4, 1\}$. As $Post_i[4/\textbf{result}]$ holds, $Post_i[4/\textbf{result}] \vee Post_i[1/\textbf{result}]$ holds as well and $PO1_i$ is satisfied. Default witness 1 is included to handle, e.g., the case where $Post_i$ is **result** $\neq 4$. Then $euld(Post_i) = \langle\{\}, \{\}, \{\}, \{4\}\rangle$. Then $wcs(T_i, Post_i) = \{1, 2\}$. As $Post_i[1/\textbf{result}]$ holds, $PO1_i$ is satisfied.

We track upper and lower bounds and the number of disequalities N to handle, e.g., the case where $Post_i$ is **result** $> 4 \wedge$ **result** $\neq 5$. Then $euld(Post_i) = \langle\{\}, \{5\}, \{\}, \{5\}\rangle$, and $wcs(T_i, Post_i) = \{5, 6, 1, 2\}$. As $Post_i[6/\textbf{result}]$ holds, $PO1_i$ is satisfied. More generally, by trying N different candidates that all satisfy the bound, we are sure to find at least one that satisfies the disequality.

We combine the candidates found in subpredicates of conjunctions and disjunctions to handle, e.g., the case where $Post_i$ is (**result** = 4 $\vee$ **result** > 8) $\wedge$ **result** > 7. In this case, we know that $euld(Post_i) = \langle\{4\}, \{9, 8\}, \{\}, \{\}\rangle$, and $wcs(T_i, Post_i) = \{4, 9, 8, 1\}$. As $Post_i[9/\textbf{result}]$ holds, $PO1_i$ is satisfied.

A predicate $\neg P$ is dealt with 'on the fly', which is more efficient than distributing the negation over the subexpressions of P. We interchange S_0 and S_3 as well as S_1 and S_2, and then add (subtract) 1 to each element of the new S_1 (S_2). The intuition is the following. As was stated above, if $E \in S_1$, then for every n a conjunct in $test(P, E + n)$ holds. Then for every n, a conjunct in $test(\neg P, E - 1 - n)$ holds. For example, $\neg$(**result** $\geq E$) equals **result** $\leq E - 1$.

As an aside, note that in the cases where P is either $P_0 \Leftrightarrow P_1$ or $P_0 \; ? \; P_1 \; : \; P_2$, $euld(P_0)$ and $euld(P_1)$ are evaluated twice. These cases can be optimized at the expense of a more complicated definition.

4 Defining the Ordering $\prec$

Our definition of $\prec$ builds on work in [DL07, DM06]. It uses a function *Order* (defined below) that associates a tuple of numbers with an expression $\#m_i(E_0, E_1)$ in a state σ. Our definition of $\prec$ ensures that $\prec$ is a well-founded strict partial order, and that Lem. 2.2 holds (as long as *Order* is well-defined):

Definition 4.1 ($\prec$).

$$[\![\#m_i(E_0, E_1) \prec \#m_j(E_2, E_3)]\!]\sigma \overset{def}{=}$$
$$Order(\#m_i, E_0, E_1, \sigma) \textit{ is lexicographically ordered below } Order(\#m_j, E_2, E_3, \sigma)$$

As Def. 4.2 shows, the definition of *Order* uses three functions. *RootDistance* associates a number with an object based on the well-founded strict partial order on objects provided by ownership, an existing specification technique (Sect. 4.1). *RTVal* associates a number with a method function based on a numbering scheme that can be largely inferred automatically (Sect. 4.2). *MeasuredBy* yields a tuple of numbers that is determined by a pure method's `measuredBy` clause, and that depends only on the values of the numerical parameters (Sect. 4.3). The definition uses $\triangleright$ to denote sequence concatenation.

Definition 4.2 (*Order*)**.**

$$Order : Method\ Function \times Expression \times Expression \times \Sigma \rightarrow Sequence\ of\ \mathbb{Z}$$
$$Order(\#m_i, E_0, E_1, \sigma) \overset{def}{=} \langle j, RTVal(\#m_i)\rangle \triangleright MeasuredBy(\#m_i, E_1, \sigma),$$

$$where\ j\ is\ (\ [\![E_0]\!]\sigma \in \texttt{Object}\ ?\ -RootDistance(\ [\![E_0]\!]\sigma, \sigma) : 0)$$

Note that if $\#m_j(E_0, E_1)$ occurs in the specification of m_i, and σ does not map E_0 to an object, then the first element of $Order(\#m_j, E_0, E_1, \sigma)$ is 0, thus requiring that the call is not relevant in σ if $PO2$ is to hold.

4.1 Root Distance

Ownership, originally developed to enforce state encapsulation [Cla01, Mül02], is a commonly used technique to make whole/part relations explicit in specifications (often applied to the modular verification of invariants [BDF$^+$04, LM04, MHKL08c, MPHL06]). The set of *owners* consists of the set of objects and the special purpose owner **root**. In any given state, every object x is *directly owned* by exactly one owner o, $o \neq x$. The *owned* relation is the transitive closure of the directly owned relation. The intention is that an object x owns the objects that are part of x, i.e., that belong to x's representation. Objects that are not part of any other object are directly owned by **root**. The owned relation is required to be irreflexive, as a whole is not a part of one of its parts. Therefore, ownership is a well-founded strict partial order, which makes it suitable for use in the definition of $\prec$.

In [DM06], it is suggested that 'the height of an object in the ownership hierarchy' can be used to allow direct recursion. We formalize this notion and apply it to general recursion. The owned relation ensures that every object is owned by **root**. Let function $RootDistance : Object \times \Sigma \rightarrow \mathbb{N}$ be such that $RootDistance(x, \sigma) = n$ iff x is owned by exactly n objects in σ (we say x has *RootDistance* n in σ). Then *RootDistance* induces a well-founded strict partial order that is an extension of ownership: if object x is owned by object y in state σ, then $RootDistance(x, \sigma) > RootDistance(y, \sigma)$. Additionally, *RootDistance* orders objects that are not ordered by ownership. For instance, if x and y have the same direct owner in state σ, and object z is owned by y, then x and z are not ordered by ownership, but $\mathit{RootDistance}(x,\sigma) < \mathit{RootDistance}(z,\sigma)$.

Note that given Defs. 4.1 and 4.2, $[\![\#m_i(E_0,E_1) \prec \#m_j(E_2,E_3)]\!]\sigma = \mathit{true}$ when $[\![E_0]\!]\sigma = x$, $[\![E_2]\!]\sigma = y$, and $\mathit{RootDistance}(x,\sigma) > \mathit{RootDistance}(y,\sigma)$.

It is not necessary, and usually not possible, to determine an object's absolute *RootDistance* during static program verification. Rather, if m_i's specification contains a call $\#m_j(E_0,E_1)$, one has to establish that the *RootDistance* of the `this`-object is smaller than (or at least equal to) the *RootDistance* of the E_0-object. I.e., one reasons about the relative *RootDistance*. This involves reasoning about ownership, which is often made explicit by extending types with *ownership modifiers* [CPN98] like **rep** and **peer**. Consider a state σ in which an object x has a field f that refers to an object y. If the ownership modifier of f is **rep**, then x directly owns y and $\mathit{RootDistance}(y,\sigma) = \mathit{RootDistance}(x,\sigma)+1$. If it is **peer**, then x and y have the same direct owner and $\mathit{RootDistance}(y,\sigma) = \mathit{RootDistance}(x,\sigma)$. Alternatively, ownership can be encoded into existing proof system concepts using a specification-only field `owner` [LM04]. If x`.owner` evaluates to y in σ, then $\mathit{RootDistance}(x,\sigma) = \mathit{RootDistance}(y,\sigma) + 1$.

The use of *RootDistance* is illustrated by method `Node.count` in Fig. 1.4. Its specification contains one call, to $\#$`count(this.next, obj)`. There are two cases, each of which satisfies $PO2$.

1. $[\![\texttt{this.next} = \texttt{null}]\!]\sigma = \mathit{false}$.
   Then modifier **rep** on `next` allows the verifier to deduce that
   $\mathit{RootDistance}(\,[\![\texttt{this.next}]\!]\sigma,\sigma) = \mathit{RootDistance}(\,[\![\texttt{this}]\!]\sigma,\sigma) + 1$;
2. $[\![\texttt{this.next} = \texttt{null}]\!]\sigma = \mathit{true}$.
   Then $[\![(\texttt{this.next} = \texttt{null}\ ?\ \texttt{0} : \texttt{this.next.count(obj)})]\!]\sigma = 0$. Then *NotRel* holds, which means that the value of $[\![\#\texttt{count(this.next,obj)}]\!]\sigma$ is not relevant.

The extension of ownership provided by *RootDistance* is useful for non-hierarchical scenarios. For instance, Fig. 4.1 shows two classes and a possible object configuration of an administration system. In this system, a `Holding` consists of multiple `Companies`, and a `Person` that is part of the `Holding` can work for multiple of these `Companies`. A `Personnel` object manages (access to) these `Persons`. Classes `Personnel` and `Person` are omitted. Each has only one relevant field. `Personnel` has a field **rep** `Node myPers` which refers to a linked list of the `Persons`. `Person` has a field **rep** `Node worksFor` which refers to a linked list of the `Companies` that `Person` works for. Class `Node` is found in Fig. 1.4. Pure method `Company.myPersonnel` returns a linked list of all `Persons` that work for that `Company` (e.g., if called on C1, it returns a single node with `val` P1). Assume that it can be deduced that `Company.thePnel` and `Personnel.thePers` are never `null` (for instance because of an invariant or non-null annotation [FL03]). Then the `this.thePnel.myPers.has(p)` call in `Company.myPersonnel` is allowed as `myPers` is a rep field of a peer of `this` and thus has a higher *RootDistance*. More formally, in any state $\sigma \in \Sigma$ in which `this` evaluates to a `Company` ob-

```
class Holding {
  rep Node myComps;
  rep Personnel myPnel;
} //rest of class omitted

class Company {
  peer Personnel thePnel;
  pure Node myPersonnel()
    ensures ∀ Person p • (
      result.has(p) ⇔
      this.thePnel.myPers.has(p) ∧
      p.worksFor ≠ null ∧
      p.worksFor.has(this) );
} //rest of class omitted
```

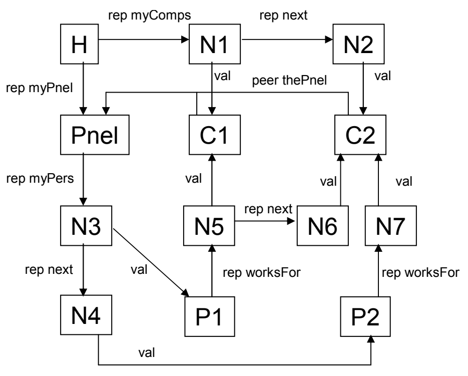


Figure 4.1: Administration System. H is a `Holding`, Pnel a `Personnel`, N's are `Nodes`, C's `Companies`, and P's `Persons`. Person P2 works only for C2, and P1 works for both C's.

ject, $\mathit{RootDistance}(\ [\![\texttt{this.thePnel.myPers}]\!]\sigma, \sigma) = \mathit{RootDistance}(\ [\![\texttt{this}]\!]\sigma, \sigma) + 1$. Then $[\![\#\texttt{has}(\texttt{this.thePnel.myPersons}, \texttt{p}) \prec \#\texttt{myPersonnel}(\texttt{this})]\!]\sigma$ holds. Likewise, the `p.worksFor.has(p)` call is allowed if one can deduce that the `Persons` in the list maintained by `p.thePnel` are owned by `p.thePnel` or by the `Holding` that owns `p.thePnel`. We discuss the **`result`**`.has(p)` call in Sect. 5.

4.2 Recursion Termination Value

For the second ordering, a *Recursion Termination Value* (RTV) is associated with each pure method [DL07]. A RTV is an element of the interval $[0, \mathit{maxRTV}]$, where *maxRTV* is a sufficiently large constant, e.g. *maxInt*. $\mathit{RTVal}(\#m_i)$ yields the RTV associated with pure method m_i.

Note that given Defs. 4.1 and 4.2,

$$[\![\#m_i(E_0, E_1) \prec \#m_j(E_2, E_3)]\!]\sigma = \mathit{true}$$

when $\mathit{RootDistance}(\ [\![E_0]\!]\sigma, \sigma) = \mathit{RootDistance}(\ [\![E_2]\!]\sigma, \sigma)$, and $\mathit{RTVal}(\#m_i) < \mathit{RTVal}(\#m_j)$.

The RTV can be specified explicitly. For instance, in Spec# explicit specification of the RTV is done using the `RecursionTermination` attribute that takes an integer parameter. The main advantage of the RTV ordering, however, is that it is largely inferred automatically. This inference is complicated by the desire for modular development (see Sect. 1).

Of course, the goal of the inference is to assign a RTV to every $\#m_i$ such that for every i, the inferred RTV is high enough to conclude $PO2_i$. When the the specification of m_i is changed, the previously inferred RTV for $\#m_i$ might no longer be high enough (for instance, because the specification of m_i now contains a method call). Therefore, the inference is rerun prior to re-verification. But as a consequence of modular development, it is not possible to re-infer every RTV. In particular, a RTV in a module that is hidden cannot be re-inferred. As a consequence, if an inferred RTV were publicly visible, a change to a specification that is hidden from a module M could indirectly invalidate the verification of M. That is, suppose that m_i and m_j are defined in different modules, and that the proof of $PO2_i$ depends on $RTVal(\#m_j) = n$. Suppose a part of the specification of m_j that is hidden from m_i is changed in such a way that re-inference of $RTVal(\#m_j)$ changes it to $n + 1$. Then the proof of $PO2_i$ no longer holds. While this does not go against modular development technically (re-inference of $RTVal(\#m_j)$ constitutes a change of public part of the specification of m_i), it is not intuitive (as the change is to an *implicit* part of the specification). Therefore, an inferred RTV is private, and an explicitly specified RTV is public. As the specifier has committed to the RTV, it is intuitive that changing it will require re-verification of modules to which it is visible. We discuss an algorithm to infer the RTVs for a module M in Ap. A. The outline is as follows. Construct a directed graph with a node N for every method visible in M, and with an edge from N to node N' iff N' occurs in the specification of N. For every N with an explicitly specified $RTV\ i$, label N with i. For every N with an RTV that is hidden from M, label N with $maxRTV$. For every remaining N, label N with the lowest value such that (1) N cannot reach a node with a higher RTV, and (2) if possible, such that N cannot reach a node with the same RTV. (1) is always possible, as $maxRTV$ can be assigned to all nodes. (2) can't be achieved for nodes that are part of a cycle, nor for nodes that can reach a $maxRTV$ node.

4.3 The measuredBy Clause

The third ordering allows for directly or mutually recursive method functions. We associate with pure method m_i, a *measuredBy clause* that specifies a tuple of numerical expressions $\langle E_1, \ldots, E_n \rangle$. $MeasuredBy(\#m_i, E, \sigma)$ is defined as $\langle\ [\![E_1[E/\mathtt{p}_i]]\!]\sigma, \ldots,\ [\![E_n[E/\mathtt{p}_i]]\!]\sigma\rangle$. For each such expression E_j, there is a proof obligation that $Pre_i \Rightarrow 0 \leq E_j$, which ensures that the ordering is well-founded. We restrict the free variables in these expressions to be the numerical formal parameters of m_i, but one can easily imagine allowing other variables, too, for example so that one can mention the *RootDistance* of a non-`this` object parameter. By default, the `measuredBy` clause is tuple $\langle 0 \rangle$.

The use of the `measuredBy` clause is illustrated by Fig. 1.6, where it allows the mutually recursive methods `isEven` and `isOdd`. For the call to `this.isOdd(n-1)` in the specification of `isEven`, the reasoning is as follows. Consider an arbitrary $\sigma \in \Sigma$. Assume $r_0, r_1, r_2, t_0, t_1, t_2 \in \mathbb{Z}$ such that $Order(\#isEven, \mathtt{this}, \mathtt{n}, \sigma) = \langle r_0, r_1, r_2 \rangle$, and $Order(\#isOdd, \mathtt{this}, \mathtt{n} - 1, \sigma) = \langle t_0, t_1, t_2 \rangle$. Then $r_0 = t_0$, as both are determined by the *RootDistance* of the `this`-object (see Sect. 4.1). Also, $r_1 = t_1$ as the same RTV is assigned to mutually recursive method functions (see Sect. 4.2). Finally, $r_2 > t_2$ as $r_2 = [\![2n]\!]\sigma$, and $t_2 = [\![(2m + 1)[n - 1/m]]\!]\sigma = [\![2n - 1]\!]\sigma$. Thus, $\langle t_0, t_1, t_2 \rangle$ is ordered lexicographically below $\langle r_0, r_1, r_2 \rangle$. So, if

`C` is the class that declares `isEven` and `isOdd`, then $\forall \sigma \in \Sigma \bullet [\![\forall\, \texttt{this} : \texttt{C}, \texttt{n} : \texttt{int} \bullet \#\mathit{isOdd}(\texttt{this}, \texttt{n} - 1) \prec \#\mathit{isEven}(\texttt{this}, \texttt{n})]\!]\sigma$. For the call to `isEven(m)` in the specification of `isOdd`, the reasoning is similar (the essential observation being that $2m + 1 > (2n)[m/n]$). Together, these properties establish that $PO2$ holds.

5 Related Work and Experience

Frame properties for a model field f declared in a class C (see Sect. 1) are discussed in [LM06]. Essentially, the idea is to add a specification-only field f to C, and to extend the deduction system with a second axiom $\forall \sigma \in \Sigma \bullet [\![\forall \texttt{this} \colon C \bullet P \Rightarrow \texttt{this}.f = \#f(\texttt{this})]\!]\sigma$, where P (defined by the methodology) describes the conditions under which the relation should hold. The methodology ensures that that $\#f(\texttt{this})$ is assigned to f whenever P becomes *true*. Breunesse and Poll suggest desugaring a model field using its satisfies clause [BP03]. This simplifies the treatment of model fields considerably, but does not account for recursion or for visibility constraints on satisfies clauses.

Modeling partial functions by underspecified total functions in the underlying logic can lead to unintuitive outcomes for the users of the specification language [Cha07]. Recent work by Rudich *et al.* [RDM08] discusses how to prevent such outcomes. The work also discusses how to allow conditional use of the axioms introduced by pure methods, as well as class invariants, when establishing $PO1$ (see Sect. 2). Essentially, the idea is that if $\mathit{Smaller}_{i,j,k}$ holds, then the axiom introduced by m_j, instantiated for $\mathit{Call}_{i,j,k}$, can be assumed when proving $PO1_i$ (see Defs. 2.2 and 2.4). More formally, let $P_{i,j,k} \equiv (\mathit{Smaller}_{i,j,k}?\ \mathit{Spec}_j[\mathit{Call}_{i,j,k}/\mathbf{result}] : \mathit{true})$. Then $PO1_i$ can be weakened to $\forall \sigma \in \Sigma \bullet [\![\forall \texttt{this} \colon C_i,\ \texttt{p}_i \colon T_i' \bullet P \Rightarrow \exists \mathbf{result} \colon T_i \bullet \mathit{Spec}_i]\!]\sigma$, where P consists of a conjunct $P_{i,j,k}$ for every $i, j \in [0, N]$, for every $k \in [0, \mathit{NrOfCalls}_{i,j} - 1]$.

In Sect. 4.1, we discussed how our approach allows a number of the calls in the specification of the `myPersonnel` method in Fig. 4.1. The call to **result**`.has()` in that specification, however, is problematic. The axiom introduced by the pure method describes a property that holds in every well-formed program state. Therefore, the resulting list of `Nodes` has to exist in each such state (and contain the right `Persons`). This is reflected in $PO1$, which cannot be proven to hold for this example. Possible solutions to this problem are suggested in [DM06, Nau07, BNSS04].

The heuristic guesses of candidate witnesses and the accompanying semantics checks in this chapter have been implemented in the Spec# programming system; there is a partial implementation of RootDistance and the RTV scheme [DL07]. Pure methods occur frequently in practice, partly because Spec# by default treats property getters as pure methods. The Spec#/Boogie test suite alone requires 148 consistency checks. From more than a year's use, we find that, with one exception, the heuristics adequately guess candidate witnesses that (for consistently specified pure methods) the semantic checks quickly verify to ensure consistency.

The one exception to this positive experience has been pure methods with a non-null return type. The only non-null candidate witnesses that our heuristics guess are fields or parameters of exactly those types—the heuristics cannot use calls to constructors, as this would require one to first prove the consistency of the specifications of such constructors. Luckily, this case has occurred only for property getters whose body returns a newly allocated object (see [LM08] for a technique that allows such methods to be considered observationally pure). In the cases we have found, these property getters were not used as pure methods, so we could circumvent the problem by explicitly marking them non-pure.

6 Conclusions

Pure methods and model fields are useful and common specification constructs that can be interpreted by the introduction of axioms in the underlying proof system. Care has to be taken that these axioms do not introduce an inconsistency into the proof system. In this chapter, we described and proved sound an approach that ensures no inconsistencies are introduced, and we described heuristics for the part of the approach that is problematic for trigger-based SMT solvers.

A Proof of Theorem

In this appendix, we prove Thm. 2.1. Ackermann's substitution lemma [Ack54, PS06] provides a way to prove the consistency of a system of axioms containing mutually recursive uninterpreted functions. To formulate this lemma, a number of shorthands are introduced first. Note that $\triangleright$ is used both to concatenate an element to a sequence, and to concatenate two sequences. Furthermore, note that shorthand $Call_{i,j,l}$ is introduced in Sect. 2.

Definition 1.1 ($Calls_i$)**.** *If* $0 \leq i \leq j \leq N$ *and* $0 \leq k < NrOfCalls_{i,j}$ *and* $l + 1 = NrOfCalls_{i,j}$*, then*

$Calls_{i,j}$ *is the sequence of expressions* $Call_{i,j,0}, \ldots, Call_{i,j,l}$

$Calls_i$ *is the sequence of expressions* $Call_{i,0} \triangleright \ldots \triangleright Call_{i,N}$

Roughly, Ackermann's substitution lemma says that a system of axioms that contains uninterpreted total functions is equivalent to the same system, but with (1) every $Call_{i,j,k}$ replaced by a fresh variable $x_{i,j,k}$, and (2) for each pair of calls to the same function $Call_{i,j,k}$ and $Call_{l,j,m}$, an additional axiom $\forall \sigma \in \Sigma \bullet\ [\![Call_{i,j,k} \sim Call_{l,j,m} \Rightarrow x_{i,j,k} = x_{l,j,m}]\!]\sigma$ that says that the two variables that replace the two calls are equal when the two calls are.

To account for item 1, in shorthand $NoMFSpec_i$ defined below, every expression $Call_{i,j,k}$ in $Spec_i$ is replaced by a variable $x_{i,j,k}$ (i.e., $NoMFSpec_i$ contains *no* *m*ethod *f*unctions). Note that *NoMFSpec* also replaces **result** by a variable x_i, as **result** is replaced by method function call $\#m_i(\texttt{this}, \texttt{p}_i)$ in the axiom added to the proof system (see the definition of $PureAx$ in Sect. 2). In the remainder of this document, $P[E_0/v_0]\dots[E_i/v_i]$ is abbreviated to $P[E_0,\dots,E_i/v_0,\dots,v_i]$.

Definition 1.2. *If $0 \le i \le j \le N$ and $k+1 = NrOfCalls_{i,j}$, then*

$Xs_{i,j}$	*is the sequence of variables* $x_{i,j,0},\dots,x_{i,j,k}$
Xs_i	*is the sequence of variables* $Xs_{i,0} \triangleright \dots \triangleright Xs_{i,N}$
Xs	*is the sequence of variables* $Xs_0 \triangleright \dots \triangleright Xs_N$
$TypedXs_{i,j}$	*is the sequence of variable declarations* $x_{i,j,0} : T_j,\dots,x_{i,j,k} : T_j$
$TypedXs_i$	*is the sequence of variable declarations* $x_i : T_i \triangleright (TypedXs_{i,0} \triangleright \dots \triangleright TypedXs_{i,N})$
$TypedXs$	*is the sequence of variable declarations* $TypedXs_0 \triangleright \dots \triangleright TypedXs_N$

Definition 1.3.

$$NoMFSpec_i : C_i \times T_i \times \underbrace{T_0 \times \dots \times T_0}_{NrOfCalls_{i,0}} \times \dots \times \underbrace{T_N \times \dots \times T_N}_{NrOfCalls_{i,N}} \rightarrow Pred$$

$$NoMFSpec_i(o, x, Xs_i) \overset{def}{=} (Pre_i[o/\texttt{this}] \Rightarrow Post_i[o, x/\texttt{this}, \textbf{result}])[Xs_i/Calls_i]$$

$$NoMFSpecs \equiv NoMFSpec_0(o_0, x_0, Xs_0) \wedge \dots \wedge NoMFSpec_N(o_N, x_N, Xs_N)$$

To account for item 2, that is, for functional consistency, shorthand FC is introduced.

Definition 1.4. *If $i, j, l \in [0, N]$ and $0 \le k < NrOfCalls_{i,j}$ and $0 \le m < NrOfCalls_{l,j}$, then*

$$\begin{array}{ll}
FC_{i,j,k,l,m} & \equiv Call_{i,j,k} \sim Call_{l,j,m} \Rightarrow x_{i,j,k} = x_{l,j,m} \\
FC_{i,j,k,l} & \equiv FC_{i,j,k,l,0} \wedge \dots \wedge FC_{i,j,k,l,n},\ \textit{where } n+1 = NrOfCalls_{l,j} \\
FC_{i,j,k} & \equiv FC_{i,j,k,i} \wedge \dots \wedge FC_{i,j,k,N} \wedge (Call_{i,j,k} \sim \#m_j(o_j, \mathtt{p}_j) \Rightarrow x_{i,j,k} = x_j) \\
FC_{i,j} & \equiv FC_{i,j,0} \wedge \dots \wedge FC_{i,j,n},\ \textit{where } n+1 = NrOfCalls_{i,j} \\
FC_i & \equiv FC_{i,0} \wedge \dots \wedge FC_{i,N} \\
FC & \equiv FC_0 \wedge \dots \wedge FC_N
\end{array}$$

As an aside, note that FC contains 'double elements' (e.g., $FC_{0,0,0,1,0} \Leftrightarrow FC_{1,0,0,0,0}$) because both $\sim$ and $=$ are symmetric. These double elements are not essential to the proof but simplify the FC-definition.

Together, $NoMFSpecs$ and FC allow for the formulation of $NoMFAx$, the analog of $PureAx$ defined in Sect. 2.

Definition 1.5 ($NoMFAx$).

$$\begin{array}{lll}
TypedFreeVars & \equiv & o_0 : C_0, \mathtt{p}_0 : T'_0, \dots o_N : C_N, \mathtt{p}_N : T'_N \\
NoMFAx & \equiv & \forall \sigma \in \Sigma \bullet \\
 & & [\![\forall TypedFreeVars \bullet \exists TypedXs \bullet NoMFSpecs \wedge FC]\!]\sigma
\end{array}$$

Lemma A.1. $NoMFAx \Rightarrow PureAx$

Proof. Lemma A.1 is a variant of Ackermann's substitution lemma. □

Lemma A.2. $Prelude \Rightarrow (POs \Rightarrow NoMFAx)$

Before we prove Lem. A.2, we prove the theorem formulated in Sect. 2.

Proof of Thm. 2.1. The theorem follows immediately from Lem. A.2, Lem. A.1, and the fact that the method functions do not occur in *Prelude*. □

To simplify the presentation of the proof of Lem. A.2, we assume that the pure methods are labeled in a certain order.

Definition 1.6.

$$\mathit{LabeledInOrder}(\sigma) \equiv \forall i, j \in \mathbb{N} \bullet i < j \le N \Rightarrow \; [\![\#m_j(o_j, \mathtt{p}_j) \not\prec \#m_i(o_i, \mathtt{p}_i)]\!]\sigma$$

Note that the assumption is not essential to the proof: given a state σ, the pure methods can always be relabeled in such a way that $\mathit{LabeledInOrder}(\sigma)$ holds, as $\prec$ is a strict partial order. Given this ordering, the essential property is that if $i < j$, then the value of $\#m_i(o_i, \mathtt{p}_i)$ is not constrained by the definition of $\#m_j$. Let $\mathbb{V}$ be the set of values that an expression can evaluate to, e.g., the union of the sets of objects, numerical values and booleans. The proof is based around the following, rather technical definition that effectively functions as an induction hypothesis.

Definition 1.7.

$$\begin{array}{l}
\mathit{IH}(n) \overset{\mathit{def}}{=} n \le N+1 \Rightarrow \\
\quad \forall \sigma_0 \in \Sigma \bullet \mathit{LabeledInOrder}(\sigma_0) \wedge \sigma_0 \textit{ maps TypedFreeVars to values} \Rightarrow \\
\quad\quad \exists \sigma_1 \in \Sigma, u_0, \ldots, u_{n-1}, v_0, \ldots v_N \in \mathbb{V} \bullet \sigma_1 \textit{ is the state like } \sigma_0 \textit{ except that} \\
\quad\quad\quad \forall i \in \mathbb{N} \bullet i < n \Rightarrow \\
\quad\quad\quad\quad \textit{(1)} \;\; [\![x_i]\!]\sigma_1 = u_i \\
\quad\quad\quad\quad \textit{(2)} \;\; \forall j, k \in \mathbb{N} \bullet \;\; n \le j \le N \wedge k < \mathit{NrOfCalls}_{i,j} \;\; \Rightarrow \;\; [\![x_{i,j,k}]\!]\sigma_1 = v_j \\
\quad\quad\quad\quad \textit{(3)} \;\; \forall j, k \in \mathbb{N} \bullet \;\; j < n \wedge k < \mathit{NrOfCalls}_{i,j} \Rightarrow \\
\quad\quad\quad\quad\quad\quad \textit{if } [\![\mathit{Call}_{i,j,k} \sim \#m_j(o_j, \mathtt{p}_j)]\!]\sigma_1, \;\; \textit{then} \;\; [\![x_{i,j,k}]\!]\sigma_1 = u_j, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textit{else} \;\; [\![x_{i,j,k}]\!]\sigma_1 = v_j \\
\quad\quad\quad\quad \textit{(4)} \;\; [\![\mathit{NoMFSpec}_i(o_i, x_i, \mathit{Xs}_i)]\!]\sigma_1
\end{array}$$

The definition of $\mathit{IH}(n)$ allows to separate the proof of Lem. A.2 into two separate concerns formulated by the two lemmas below.

Lemma A.3. $\mathit{POs} \Rightarrow (\forall n \in \mathbb{N} \bullet \mathit{IH}(n))$

Lemma A.4. $\mathit{IH}(N+1) \Rightarrow \mathit{NoMFAx}$

Proof of Lem. A.2. The lemma follows immediately from Lems. A.3 and A.4. □

What remains to be proven are Lems. A.3 and A.4. Recall from Sect. 2 that $\mathit{POs} \equiv \mathit{PO}1 \wedge \mathit{PO}2$. Proof of Lem. A.3 relies on an intermediate property that states that $\mathit{PO}1_i$ is strong enough to establish that $\mathit{NoMFSpec}_i$ holds.

Definition 1.8.
$PO1'_i \equiv \forall \sigma \in \Sigma \bullet \; [\![\forall o : C_i, \mathtt{p}_i : T'_i, \mathit{TypedXs}_i \bullet \exists x : T_i \bullet \mathit{NoMFSpec}_i(o, x, \mathit{Xs}_i)]\!]\sigma$
$PO1' \equiv PO1'_0 \wedge \ldots \wedge PO1'_N$

Lemma A.5. $PO1_i \Rightarrow PO1'_i$

Proof. Method functions in PO_i are uninterpreted functions. Method functions do not occur in the prelude (i.e., the value of a method function is not restricted by the prelude). Therefore, the proof of $PO1$ is valid no matter what values the method functions evaluate to. □

Proof of Lem. A.3. Proof is by induction on n. To simplify the presentation, we only prove a weaker version of the theorem in which $PO2$ (see Sect. 2) does not contain the *NotRel* disjuncts. That is, for the purpose of this proof, $PO2_{i,j,k} \overset{\text{def}}{=} \forall \sigma \in \Sigma \bullet \; [\![\forall \mathtt{this} : C_i, \mathtt{p}_i : T'_i \bullet \mathit{Smaller}_{i,j,k}]\!]\sigma$. Extending the proof to account for the *NotRel* disjuncts is fairly straightforward given the definition of *NotRel*.
Base $(n = 0)$**:** Trivial, as the domain $0 \le i < n$ is empty (i.e., take $\sigma_1 = \sigma_0$).
Step: We assume *POs* and $\mathit{IH}(n)$ and prove $\mathit{IH}(n+1)$. If $n \ge N+1$, then $\mathit{IH}(n+1)$ holds trivially. That leaves the case where $n < N+1$. Let $\sigma_0 \in \Sigma$ be such that σ_0 maps *TypedFreeVars* to values and let $\mathit{LabeledInOrder}(\sigma_0)$ hold. Then, due to $\mathit{IH}(n)$, there is a $\sigma_1 \in \Sigma$ such that $\exists u_0, \ldots, u_{n-1}, v_0, \ldots v_N \in \mathbb{V} \bullet \sigma_1$ is the state like σ_0 except that $\forall i \in \mathbb{N} \bullet i < n \Rightarrow$

(A1) $[\![x_i]\!]\sigma_1 = u_i$
(A2) $\forall j, k \in \mathbb{N} \bullet \; n \le j \le N \wedge k < \mathit{NrOfCalls}_{i,j} \;\; \Rightarrow \;\; [\![x_{i,j,k}]\!]\sigma_1 = v_j$
(A3) $\forall j, k \in \mathbb{N} \bullet \; j < n \wedge k < \mathit{NrOfCalls}_{i,j} \Rightarrow$
$\quad$ *if* $[\![\mathit{Call}_{i,j,k} \sim \#m_j(o_j, \mathtt{p}_j)]\!]\sigma_1$, *then* $[\![x_{i,j,k}]\!]\sigma_1 = u_j$,
$\quad$ *else* $[\![x_{i,j,k}]\!]\sigma_1 = v_j$
(A4) $[\![\mathit{NoMFSpec}_i(o_i, x_i, \mathit{Xs}_i)]\!]\sigma_1$

As *POs* holds, $PO1$ holds. Then $PO1'$ holds (Lem. A.5).
Then $[\![\forall o : C_n, \mathtt{p}_n : T'_n, \mathit{TypedXs}_n \bullet \exists x : T_n \bullet \mathit{NoMFSpec}_n(o, x, \mathit{Xs}_n)]\!]\sigma_1$.
Then $[\![\forall \mathit{TypedXs}_n \bullet \exists x : T_n \bullet \mathit{NoMFSpec}_n(o_n, x, \mathit{Xs}_n)]\!]\sigma_1$ (as $o : C_n, \mathtt{p}_n : T'_n \in$ *TypedFreeVars*, and *TypedFreeVars* are mapped to values by σ_1).

Let $\sigma_2 \in \Sigma$ be the state like σ_1, except that
(B1) $\forall j, k \in \mathbb{N} \bullet \; n \le j \le N \wedge k < \mathit{NrOfCalls}_{n,j} \;\; \Rightarrow \;\; [\![x_{n,j,k}]\!]\sigma_2 = v_j$
(B2) $\forall j, k \in \mathbb{N} \bullet \; j < n \wedge k < \mathit{NrOfCalls}_{n,j} \Rightarrow$
$\quad$ *if* $[\![\mathit{Call}_{n,j,k} \sim \#m_j(o_j, \mathtt{p}_j)]\!]\sigma_2$, *then* $[\![x_{n,j,k}]\!]\sigma_2 = u_j$,
$\quad$ *else* $[\![x_{n,j,k}]\!]\sigma_2 = v_j$

Then (B3) $[\![\exists x : T_n \bullet \mathit{NoMFSpec}_n(o_n, x, \mathit{Xs}_n)]\!]\sigma_2$ (as TypedX_n are mapped to values by σ_2).

First, we prove that
(P1) $\forall i, k \in \mathbb{N} \bullet i < n \wedge k < \mathit{NrOfCalls}_{i,n} \Rightarrow [\![\mathit{Call}_{i,n,k} \not\sim \#m_n(o_n, \mathtt{p}_n)]\!]\sigma_2$.
Consider arbitrary $i, k \in \mathbb{N}$ such that that $i < n \wedge k < \mathit{NrOfCalls}_{i,n+1}$ (if no such i, k exist, then P1 holds trivially).
Then $[\![\#m_n(o_n, \mathtt{p}_n) \not\prec \#m_i(o_i, \mathtt{p}_i)]\!]\sigma_2$ (due to $\mathit{LabeledInOrder}(\sigma_0)$).

Furthermore, $[\![\mathit{Call}_{i,n,k} \prec \#m_i(o_i, \mathtt{p}_i)]\!]\sigma_2$ (due to $\mathit{PO}2$).
Then $[\![\mathit{Call}_{i,n,k} \not\sim \#m_n(o_n, \mathtt{p}_n)]\!]\sigma_2$ (due to Lem. 2.2). Then P1 holds.

Next, we prove that
(P2) $\forall k \in \mathbb{N} \bullet k < \mathit{NrOfCalls}_{n,n} \Rightarrow [\![\mathit{Call}_{n,n,k} \not\sim \#m_n(o_n, \mathtt{p}_i)]\!]\sigma_2$.
Consider an arbitrary $k \in \mathbb{N}$ such that $k < \mathit{NrOfCalls}_{n,n}$ (if no such k exists, then P2 holds trivially).
Then $[\![\mathit{Call}_{n,n,k} \prec \#m_n(o_n, \mathtt{p}_i)]\!]\sigma_2$ (due to $\mathit{PO}2$).
As $\prec$ is irreflexive, $[\![\#m_n(o_n, \mathtt{p}_i) \not\prec \#m_n(o_n, \mathtt{p}_i)]\!]\sigma_2$.
Then $[\![\mathit{Call}_{n,n,k} \not\sim \#m_n(o_n, \mathtt{p}_i)]\!]\sigma_2$ (due to Lem. 2.2). Then P2 holds.

Now let $\sigma_3 \in \Sigma$ be the state like σ_2, except that it maps $x_n : T_n$ to a value u_n such that
$[\![\mathit{NoMFSpec}_n(o_n, x_n, \mathit{Xs}_n)]\!]\sigma_3$ (such a u_n exists due to B3).
Then it follows directly from B1 that
(C1) $\forall j, k \in N \bullet \;\; n < j \leq N \wedge k < \mathit{NrOfCalls}_{n,j} \;\; \Rightarrow \;\; [\![x_{n,j,k}]\!]\sigma_3 = v_j$
Furthermore, it follows from B2, P1 and P2 that
(C2) $\forall j, k \in \mathbb{N} \bullet \;\; j \leq n \wedge k < \mathit{NrOfCalls}_{n,j} \Rightarrow$
$\quad$ *if* $[\![\mathit{Call}_{n,j,k} \sim \#m_j(o_j, \mathtt{p}_j)]\!]\sigma_2$, *then* $[\![x_{n,j,k}]\!]\sigma_2 = u_j$,
$\quad$ *else* $[\![x_{n,j,k}]\!]\sigma_2 = v_j$

Then, $\forall i \in \mathbb{N} \bullet i < n + 1$,
(D1) $[\![x_i]\!]\sigma_3 = u_i$ (due to A1 and the definition of σ_3)
(D2) $\forall j, k \in \mathbb{N} \bullet \;\; n + 1 \leq j \leq N \wedge k < \mathit{NrOfCalls}_{i,j} \;\; \Rightarrow \;\; [\![x_{i,j,k}]\!]\sigma_3 = v_j$
(due to A2 and C1)
(D3) $\forall j, k \in \mathbb{N} \bullet \;\; j < n + 1 \wedge k < \mathit{NrOfCalls}_{i,j} \Rightarrow$
$\quad$ *if* $[\![\mathit{Call}_{i,j,k} \sim \#m_j(o_j, \mathtt{p}_j)]\!]\sigma_3$, *then* $[\![x_{i,j,k}]\!]\sigma_3 = u_j$,
$\quad$ *else* $[\![x_{i,j,k}]\!]\sigma_3 = v_j$
(due to A3 (case $i < n \wedge j < n$), and C2 (case $i = n \wedge j \leq n$), and P1 and A2 (case $i < n \wedge j = n$))
(D4) $[\![\mathit{NoMFSpec}_i(o_i, x_i, \mathit{Xs}_i)]\!]\sigma_3$ (due to A4 and the definition of σ_3)

Then $\mathit{IH}(n + 1)$ holds (σ_3 meets the properties required from σ_1 in the definition of $\mathit{IH}(n + 1)$). That concludes the proof of the step case. □

Proof of Lem. A.4. Let state $\sigma_0 \in \Sigma$ be such that (1) σ_0 maps *TypedFreeVars* to values, and (2) $\mathit{LabeledInOrder}(\sigma_0)$. Then, due to $\mathit{IH}(N + 1)$, there is a $\sigma_1 \in \Sigma$ such that $\exists u_0, \ldots, u_N, v_0, \ldots v_N \in \mathbb{V} \bullet \sigma_1 : \Sigma$ is the state like σ_0 except that $\forall i \in \mathbb{N} \bullet 0 \leq i < N + 1 \Rightarrow$
(A1) $[\![x_i]\!]\sigma_1 = u_i$
(A2) $\forall j, k \in \mathbb{N} \bullet \;\; j < N + 1 \wedge k < \mathit{NrOfCalls}_{i,j} \Rightarrow$
$\quad$ *if* $[\![\mathit{Call}_{i,j,k} \sim \#m_j(o_j, \mathtt{p}_j)]\!]\sigma_1$, *then* $[\![x_{i,j,k}]\!]\sigma_1 = u_j$,
$\quad$ *else* $[\![x_{i,j,k}]\!]\sigma_1 = v_j$
(A3) $[\![\mathit{NoMFSpec}_i(o_i, x_i, \mathit{Xs}_i)]\!]\sigma_1$

Then (B1) $[\![\mathit{NoMFSpecs}]\!]\sigma_1$ (due to A3).

Next, we prove that (B2) $[\![\mathit{FC}]\!]\sigma_1$
Consider arbitrary $i, j, k, l, m \in N$ such that $i, j, l \leq N \wedge k < \mathit{NrOfCalls}_{i,j} \wedge m < \mathit{NrOfCalls}_{l,j}$ (if no such i, j, k, l, m exist, then B2 holds trivially). Four cases can

be distinguished:
or (1) $[\![Call_{i,j,k} \sim \#m_j(o_j, \mathtt{p}_j) \wedge Call_{l,j,m} \sim \#m_j(o_j, \mathtt{p}_j)]\!]\sigma$,
or (2) $[\![Call_{i,j,k} \not\sim \#m_j(o_j, \mathtt{p}_j) \wedge Call_{l,j,m} \not\sim \#m_j(o_j, \mathtt{p}_j)]\!]\sigma$,
or (3) $[\![Call_{i,j,k} \sim \#m_j(o_j, \mathtt{p}_j) \wedge Call_{l,j,m} \not\sim \#m_j(o_j, \mathtt{p}_j)]\!]\sigma$,
or (4) $[\![Call_{i,j,k} \not\sim \#m_j(o_j, \mathtt{p}_j) \wedge Call_{l,j,m} \sim \#m_j(o_j, \mathtt{p}_j)]\!]\sigma$.
In case 1, $[\![x_{i,j,k}]\!]\sigma_1 = [\![x_{l,j,m}]\!]\sigma_1 = u_j$ (due to A2).
In case 2, $[\![x_{i,j,k}]\!]\sigma_1 = [\![x_{l,j,m}]\!]\sigma_1 = v_j$ (due to A2).
In either case, $[\![x_{i,j,k} = x_{l,j,m}]\!]\sigma_1$.
In case 3 and case 4, $[\![Call_{i,j,k} \not\sim Call_{l,j,m}]\!]\sigma_1$.
Then in all four cases, $[\![Call_{i,j,k} \sim Call_{l,j,m} \Rightarrow x_{i,j,k} = x_{l,j,m}]\!]\sigma_1$.
Then $[\![FC_{i,j,k,l,m}]\!]\sigma_1$.
Furthermore, if $[\![Call_{i,j,k} \sim \#m_j(o_j, \mathtt{p}_j)]\!]\sigma_1$, then $[\![x_{i,j,k}]\!]\sigma_1 = [\![x_j]\!]\sigma_1 = u_j$ (due to A2 and A1).
Then $[\![Call_{i,j,k} \sim \#m_j(o_j, \mathtt{p}_j) \Rightarrow x_{i,j,k} = x_j]\!]\sigma_1$.
Then B2 holds.

Then $[\![\exists\, TypedXs \bullet NoMFSpecs \wedge FC]\!]\sigma_0$ (due to B1, B2, and the definition of σ_1).
Then $\forall \sigma \in \Sigma \bullet [\![\forall\, TypedFreeVars \bullet \exists\, TypedXs \bullet NoMFSpecs \wedge FC]\!]\sigma$ (as σ_0 maps *TypedFreeVars* to arbitrary values, and as *LabeledInOrder*(σ_0) was introduced for convenience but is not essential to the proof).
Then *NoMFAx* holds. □ □

Then every lemma used in the proof of Thm. 2.1 has been proven to hold, which concludes this appendix.

CHAPTER 7

Conclusions

In this chapter we revisit the research questions posed in Chap. 1, discuss our contributions, and give directions for future work related to the topics of this thesis.

1 Question 1: Client-level Algebraic Specification

The first research question is the following.

Question 1 What are the syntax and semantics of a client specification based on algebraic specification, independent of the implementation used?

Contributions. We have contributed a specification technique in which a client specification consists of an algebraic specification and a canonicity function. (Chap. 2). The signature of the algebraic specification formalizes the vocabulary of the client's problem domain, and its axioms formalize equalities between terms in that vocabulary. The technique is suitable when the client problem can be formulated in terms of the desired input/output behavior of the implementation, where an implementation is viewed as a black box that rewrites terms into other terms. The semantics of a client specification is a set of black boxes that rewrites terms into an equivalent canonical (i.e. most basic) form. Which terms are of a most basic form is formalized by the canonicity function.

As the technique views an implementation as a black box, any implementation technique can be applied, be it functional or imperative. To apply an implementation technique, a notion of satisfaction needs to be defined that abstracts the implementation to a black box. We have contributed a notion of satisfaction for class-based implementations that uses an additional problem-independent presentation layer to bridge the gap between the input and output of a black box and the input and output of a class-based implementation.

Future Work. A useful extension would be to provide a notion of satisfaction for a full OO implementation rather than a class-based implementation. Other future work is to apply the technique to other implementation paradigms, e.g. the functional paradigm.

2 Question 2: Programmer-level Algebraic Specification

The second research question is the following.

> **Question 2** When reasoning about an OO implementation, how can we use programmer-level algebraic specification to capture and exploit an abstract view of a state?

Contributions. We have contributed an implementation approach that formalizes and extends ideas from [Hoa72], Hoare's seminal paper on data abstraction at the semantical level (Chap. 2). The approach uses a client specification based on algebraic specification as a starting point for development. We complement the Hoare-style approach, which essentially encodes the term provided as input by the client in an OO state, with a special purpose method that is responsible for 1) making the result canonical, and 2) encoding the canonical result in a way that is suitable for display to the client. The approach uses an algebraic specification to reason about executions using an abstract view of the state, where the behavior of a method is abstracted to a function that is interpreted using a model of the specification. It also uses algebraic specifications to formalize the relation between the abstract view of a user of a(n object of a) class, and the more concrete view of the implementer of that class.

Future Work. Future work is to extend the approach to account for layers of specifications, where part of the implementation consists of reused library code that comes with its own specification. Another possible direction for future research is to combine the approach with techniques based on programmer-level OO specifications. This would allow for implementations that rely in part on cooperation between objects that together implement a common purpose, thus mitigating the inherent limitations of the approach.

3 Question 3: Programmer-Level OO Specification

The third research question is the following.

Question 3 When reasoning about an OO implementation, how can we use programmer-level OO specification to capture and exploit an abstract view of a state?

Overall Contribution and Future Work. OO implementations typically contain many implementation patterns that cannot all be captured in the same way. We have contributed techniques to capture several common patterns (described in more detail below), and exploit them through more liberal semantics of invariants. Future work is to combine these techniques with existing techniques that capture and exploit abstract views for other implementation patterns, as well as to come up with specification techniques for other implementation patterns. Another interesting topic for future research is to make the abstract view a more central part of the semantics of specifications, as current OO specification techniques capture abstractions, but do not really reason at the level of the abstract view. That is, current techniques still deal with the entire OO state rather than just the abstract values of the currently relevant objects.

Contribution - Cooperation-based Invariants. We have contributed the programmer-level specification constructs *inc* and *coop* (Chap. 3). The inc construct allows a method specification to make explicit that a certain enumeration of invariants does not have to hold when that method is executed. The coop construct allows a field specification to make explicit that a certain enumeration of invariants might be invalidated when the field is updated. This allows for the specification and verification of OO designs in which in the process of updating one object, other objects with which it together implements a common purpose must be updated as well. We have generalized these constructs to consider sets rather than enumerations, and have removed a limitation on method calls in control flow statements. (Chap. 4).

Contribution - Layer-based Invariants. We have contributed a programmer-level specification technique to capture *layers* in OO architectures, and we have exploited these layers by providing a more liberal semantics of class invariants (Chap. 5). We have motivated that OO architectures often consists of several layers, where an object in a higher layer is not relevant to the purpose of an object in a lower layer. We have shown that several object structures in higher layers may share a sub-structure in a lower layer. Our specification technique makes the layers in a design explicit using a simple numbering scheme. We have observed that an object in a higher layer is not part of the abstract view from an object in a lower layer. We have exploited this by a more liberal semantics of invariants that allows that the invariant of a higher-layer object does not hold when a method of a lower-layer object is executing.

4 Question 4: Verification

The fourth research question is the following.

Question 4 Having used the techniques from research questions 2 and 3, how can we provide matching verification approaches?

Verification has been been discussed for every topic studied in this thesis. Additionally, we have contributed a verification approach for consistency of pure methods and model fields. Specific contribution as well as future work are discussed in more detail below.

Contribution - A verification approach for cooperation-based invariants. We have contributed a verification approach for cooperation-based invariants that uses flow analysis to determine which invariants must be established at which points in the method body (Chap. 4).

Contribution - A verification approach for layer-based invariants. We have contributed a verification approach for layer-based invariants that extends the ownership technique from [MPHL06] with constructs for capturing layers (Chap. 5). The technique allows static reasoning (i.e., reasoning using simple syntactic restrictions like those of a type system) about layer relations between objects. It uses these layer relations to make explicit, through proof obligations, which invariants must be established at which points in a method body. The proof obligations are formulated in a way that is suitable for automatic verification using the Boogie verification system [BLS05].

Contribution - A verification approach for consistency of pure methods and model fields. We have contributed a verification technique for pure methods and model fields, which are existing specification techniques for capturing an abstract view of the state in OO specifications (Chap. 6). Both can be interpreted by the introduction of an axioms in the underlying proof system. The verification technique allow to ensure that such an axiom does not introduce an inconsistency into the proof system. The verification technique comes with heuristics that that make it amenable to automatic verification with trigger-based SMT solvers.

Future Work. We have sketched a verification approach to establish that an implementation that has been developed using algebraic specifications to reason about the execution at an abstract level, satisfies a client specification based on algebraic specifications and a canonicity function. Future work is to further develop this approach.

It may be useful to investigate a verification approach for cooperation-based invariants that is based on assumptions and assertions at specific points in the method body. Such an approach may be more intuitive though better separation of concerns and would be more directly suitable to automatic verification using existing verification systems. A similar approach was taken in [SD10], which build on the work in this chapter. Other future work would be to place the layer-based invariants approach in the unified framework from [DFMS08].

The experience with the verification approach for consistency of pure methods and model fields in the test suites for the Spec# and Boogie specification and verification tools has been positive. However, the heuristics approach can be improved to correctly guess witnesses in more cases. In particular, the heuristics can be improved to deal with pure methods that return a newly created object. Furthermore, it seems possible to apply this verification technique to prove that algebraic specifications of abstraction operators as discussed in Chap. 2, do not introduce inconsistencies into the proof system (which is a verification requirement formulated in Sect. 5.3 of Chap. 2). Other future work is to apply the approach to more specifications from the 'real world'.

Bibliography

[Ack54] Wilhelm Ackermann. *Solvable Cases of the Decision Problem.* North-Holland Publishing Co., Amsterdam, 1954.

[BCD+06] 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 Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, September 2006.

[BDF+04] 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(6):27–56, June 2004. Special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs (FTfJP).

[BHKW01] Hubert Baumeister, Rolf Hennicker, Alexander Knapp, and Martin Wirsing. Ocl component invariants. In *Monterey Workshop 2001, Engineering Automation for Software Intensive System Integration, Monterey, USA*, pages 208–215, June 18–22 2001.

[BJM97] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 67–92. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0030589.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS '04)*, volume 3362 of *LNCS*. Springer, 2005.

[BN04] Michael Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction (MPC '04)*, volume 3125 of *LNCS*, pages 54–84. Springer, 2004.

[BNSS04] Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specifications. In *The ECOOP'04 workshop on Formal Techniques for Java-like Programs (FTfJP)*, pages 11–18. Technical Report NIII-R0426, University of Nijmegen, 2004.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin–Cummings, Redwood City, Calif., 2nd edition, 1994.

[BP03] Cees-Bart Breunesse and Erik Poll. Verifying JML specifications with model fields. In *the ECOOP'2003 workshop on Formal Techniques for Java-like Programs (FTfJP)*, pages 51–60. Technical Report 408, ETH Zurich, 2003.

[Bra07] Guillaume Brat. Experiences in the Static Analysis of Embedded Software. In *Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL 07*, 2007.

[CDE+02] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narcise Martí-Oliet, José Meseguer, and José F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187 – 243, 2002.

[CdPL09] Lawrence Chung and Julio do Prado Leite. On non-functional requirements in software engineering. In Alexander Borgida, Vinay Chaudhri, Paolo Giorgini, and Eric Yu, editors, *Conceptual Modeling: Foundations and Applications*, volume 5600 of *Lecture Notes in Computer Science*, pages 363–379. Springer Berlin / Heidelberg, 2009.

[Cha07] Patrice Chalin. Are the logical foundations of verifying compiler prototypes matching user expectations? *Form. Asp. Comput.*, 19(2):139–158, 2007.

[Cla01] Dave Clarke. *Object Ownership and Containment.* PhD thesis, University of New South Wales, July 2001.

[CLSE05] Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: cleanly supporting abstraction in design by contract: Research articles. *Softw. Pract. Exper.*, 35(6):583–599, 2005.

[CMR98] Maura Cerioli, Till Mossakowski, and Horst Reichel. From total equational to partial first order logic. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specifications*, 1998.

[Cok05] David R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, 2005.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33:10 of *ACM SIGPLAN Notices*, pages 48–64, New York, October 18-22 1998. ACM Press.

[CS02] Michael Colon and Henny Sipma. Practical methods for proving program termination. In Ed Brinksma and Kim Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 227–240. Springer Berlin / Heidelberg, 2002.

[DFMS08] Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. A unified framework for verification techniques for object invariants. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 412–437. Springer-Verlag, 2008.

[DL07] Ádám Darvas and K. Rustan M. Leino. Practical reasoning about invocations and implementations of pure methods. In Matthew B. Dwyer and Antónia Lopes, editors, *Fundamental Approaches to Software Engineering (FASE '07)*, volume 4422 of *LNCS*, pages 336–351. Springer, 2007.

[DM05] Werner Dietl and Peter Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[DM06] Ádám Darvas and Peter Müller. Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology*, 5(5):59–85, June 2006. Special issue: ECOOP 2005 workshop on Formal Techniques for Java-like Programs (FTfJP)).

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[EKP79] Hartmut Ehrig, Hans-Jörg Kreowski, and Peter Padawitz. Algebraic implementation of abstract data types: an announcement. *SIGACT News*, 11(2):25–29, 1979.

[FHT05] John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors. *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, volume 3582 of *LNCS*. Springer, 2005.

[FL03] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In Ron Crocker and Guy L. Steele Jr., editors, *OOPSLA*, pages 302–312. ACM, 2003.

[Fru04] Nicu G. Fruja. The Correctness of the Definite Assignment Analysis in C#. *Journal of Object Technology*, 3(9):29–52, October 2004. Special issue: .NET Technologies 2004 workshop.

[GH93] John V. Guttag and James J. Horning. *Larch: languages and tools for formal specification.* Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GS95] David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, number 1000, pages 366–373. Springer-Verlag, New York, N.Y., 1995.

[HK00] Kees Huizing and Ruurd Kuiper. Verification of object oriented programs using class invariants. In T. S. E. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE '00)*, volume 1783 of *LNCS*, pages 208–221. Springer, 2000.

[HK01] Cornelis Huizing and Ruurd Kuiper. Reinforcing fragile base classes. In *3rd ECOOP workshop on Formal Techniques for Java Programs, Budapest*, 2001.

[HK03] Cornelis Huizing and Ruurd Kuiper. Dynamic contracts for adaptive specification. In *Proceedings of the 7th World Multiconference on Systems, Cybernetics and Informatics (SCI 2003), Orlando, Florida*, 2003.

[Hoa72] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[JSPS07] Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. A simple sequential reasoning approach for sound modular verification of mainstream multithreaded programs. *Electron. Notes Theor. Comput. Sci.*, 174(9):23–47, 2007.

[Kas06] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Formal Methods (FM '06)*, volume 4085 of *LNCS*, pages 268–283. Springer, 2006.

[LBR06] 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.

[Lei95] K. Rustan M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995.

[Lei08] K. Rustan M. Leino. This is Boogie 2. Manuscript KRML 178, 2008. Available at `http://research.microsoft.com/en-us/um/people/leino/papers.html`.

[Lin93] Huimin Lin. Procedural implementation of algebraic specification. *ACM Trans. Program. Lang. Syst.*, 15(5):876–895, 1993.

[LM04] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP '04)*, volume 3086 of *LNCS*, pages 491–516. Springer, 2004.

[LM05] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In Fitzgerald et al. [FHT05], pages 26–42.

[LM06] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In Peter Sestoft, editor, *ESOP*, volume 3924 of *LNCS*, pages 115–130. Springer, 2006.

[LM08] K. Rustan M. Leino and Peter Müller. Verification of equivalent-results methods. In Sophia Drossopoulou, editor, *ESOP*, volume 4960 of *Lecture Notes in Computer Science*, pages 307–321. Springer, 2008.

[LM09] K. Rustan M. Leino and Ronald Middelkoop. Proving consistency of pure methods and model fields. In Marsha Chechik and Martin Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2009.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition.* Prentice-Hall, New Jersey, 1997.

[MHKL07a] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik Luit. Cooperation-based Invariants for OO Languages. In Zhiming Liu and Luís Barbosa, editors, *Proceedings of the International Workshop on*

Formal Aspects of Component Software (FACS 2005), volume 160 of *Electronic Notes in Theoretical Computer Science*, pages 225–237. Elsevier, 2007.

[MHKL07b] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik J. Luit. A new interpretation of invariants exploiting layers in OO designs. In *Proceedings of the Workshop on Concurrency, Specification and Programming (CS&P'07), September 27-29, 2007, Łagós, Poland*, 2007.

[MHKL08a] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik J. Luit. Invariants for non-hierarchical object structures. In L. Ribeiro and A. Martins Moreira, editors, *Proceedings of the 9th Brazilian Symposium on Formal Methods (SBMF'06)*, volume 195C of *Electronic Notes in Theoretical Computer Science*, pages 211–229. Elsevier, 2008.

[MHKL08b] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik J. Luit. A proof system for invariants in layered OO designs. Technical Report CSR 08-01, Department of of Mathematics and Computer Science, Technische Universiteit Eindhoven, 2008.

[MHKL08c] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik J. Luit. Specification and Verification of Invariants by Exploiting Layers in OO Designs. *Fundamenta Informaticae*, 85(1-4):377–398, 2008. Special issue: Concurrency, Specification and Programming (CS&P'07).

[Moi82] Abha Moitra. Direct implementation of algebraic specification of abstract data types. *IEEE Trans. Software Eng.*, 8(1):12–20, 1982.

[MP74] Zohar Manna and Amir Pnueli. Axiomatic approach to total correctness of programs. *Acta Informatica*, 3:243–263, 1974.

[MPHL06] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, 2006.

[Mül02] Peter Müller. *Modular Specification and Verification of Object Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[Nau07] David A. Naumann. Observational purity and encapsulation. *Theor. Comput. Sci.*, 376(3):205–224, 2007.

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Par05] Matthew Parkinson. *Local Reasoning for Java.* PhD thesis, University of Cambridge, 2005.

[PB05] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Proceedings of 32nd POPL*, volume 40 of *ACM SIGPLAN Notices*, pages 247–258, January 2005.

[PCdB04] Cees Pierik, Dave Clarke, and Frank S. de Boer. Creational invariants. In *Proceedings of the 6th workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2004.

[PCdB05] Cees Pierik, Dave Clarke, and Frank S. de Boer. Controlling object allocation using creation guards. In Fitzgerald et al. [FHT05], pages 59–74.

[PH97] Arnd Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs.* Habilitationsschrift, Technische Universität München, 1997.

[PHM99] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S.D. Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *LNCS*, pages 162–176. Springer-Verlag, 1999.

[Pie06] Cornelis Pierik. *Validation Techniques for Object-Oriented Proof Outlines.* PhD thesis, Universiteit Utrecht, 2006.

[PNCB05] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Featherweight generic ownership. In *Formal Techniques for Java-like Programs (FTfJP)*, 2005.

[PS06] Amir Pnueli and Ofer Strichman. Reduced functional consistency of uninterpreted functions. *ENTCS*, 144(2):53–65, 2006.

[RBL$^+$90] James R. Rumbaugh, Michael R. Blaha, William Lorensen, Frederick Eddy, and William Premerlani. *Object-Oriented Modeling and Design.* Prentice Hall, October 1990.

[RDM08] Arsenii Rudich, Ádám Darvas, and Peter Müller. Checking well-formedness of pure-method specifications. In J. Cuellar and T. Maibaum, editors, *Formal Methods (FM)*, volume 5014 of *LNCS*, pages 68–83. Springer-Verlag, 2008.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Third Annual Symposium on Logic in Computer Science*, Copenhagen, Denmark, 2002. IEEE Computer Society.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999. see www.awl.com/cseng/otseries/.

[SD10] Alexander J. Summers and Sophia Drossopoulou. Considerate reasoning and the composite design pattern. In *Verification, Model Checking and Abstract Interpretation (VMCAI 2010)*, pages 328–344, 2010.

[SSB01] Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine—Definition, Verification, Validation.* Springer-Verlag, 2001.

[ST99] Donald Sannella and Andrzej Tarlecki. Algebraic preliminaries. In Egidio Astesiano, Hans-Jörg Kreowski, and Bernd Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, chapter 2. Springer, 1999.

[Wan82] Mitchell Wand. Specifications, models, and implementations of data abstractions. *Theor. Comput. Sci.*, 20:3–32, 1982.

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise modeling with UML.* Addison-Wesley, 1999.

[YC78] Edward Yourdon and Larry L. Constantine. *Structured design: fundamentals of a discipline of computer program and systems design.* Yourdon Press, 1978.

Summary

Capturing and Exploiting Abstract Views of States in OO Verification

In this thesis, we study several implementation, specification and verification techniques for Object-Oriented (OO) programs. Our focus is on capturing conceptual structures in OO states in abstractions, and then exploiting such an abstract view of the state in specification and implementation approaches in a way that allows for formal verification.

Generally, an OO state consists of many objects that reference each other in possibly complicated ways. At the same time, at any one point in the execution of the program, we can often reason about what is happening using an abstract view of the state that is much less complicated. To further improve the quality of implementations, better techniques must be developed for 1) specification of the abstract views that are used by the client and the programmer, and 2) the verification that an implementation satisfies its specification. This thesis contributes to that effort.

We distinguish between client-level and programmer-level specification. A client-level specification acts as a contract between the client and the implementer. A programmer-level specification allows to reason formally about the implementation. We consider two specification formalisms that differ in the basic abstract view that is used: Algebraic Specification and OO Specification.

We consider both client-level and programmer-level specifications based on algebraic specification. We contribute a novel syntax and semantics for the former, and we contribute an implementation approach for OO implementations based on the latter. We show that the implementation approach is suitable for problem-independent verification.

We propose the programmer-level OO specification constructs *inc* and *coop*. The

inc construct allows a method specification to make explicit that a certain enumeration of invariants does not have to hold when that method is executed. The coop construct allows a field specification to make explicit that a certain enumeration of invariants might be invalidated when the field is updated. This allows for the specification and verification of OO designs in which in the process of updating one object, other objects with which it together implements a common purpose must be updated as well.

We then generalize the inc and coop constructs by removing a restriction to enumerations of invariants. For instance, this is needed in the well-known Observer Pattern, where a `Subject` can have an arbitrary and dynamically changing number of `Observers`. A more general interpretation of invariants and accompanying proof system are provided as well.

We contribute a programmer-level OO specification technique to capture layers in OO architectures, and we exploit these layers by providing a more liberal semantics of class invariants. We also provide a verification technique for the semantics. Layers are an abstraction at the architectural level in OO implementations that designate certain object structures in the design as sub-structures that are shared by other structures. An object in a higher layer is not relevant to the purpose of an object in the sub-structure. Given this intuition, an object in a higher layer is not part of the abstract view from an object in a lower layer. Therefore, the invariant of a higher layer object does not have to hold when a method of a lower-layer object is executing.

Finally, we contribute a verification technique for pure methods and model fields, which are existing specification techniques for capturing an abstract view of the state in OO specifications. A method that is pure can be used as a function in predicates in class specifications. The function is axiomatized using the pre- and postcondition that are specified for the method. A model field abstracts part of the concrete state of an object into an abstract value. This too introduces an additional axiom in the underlying reasoning. The technique contributed establishes that such additional axioms do no introduce inconsistencies into the formal reasoning. It comes with heuristics that that make it amenable to automatic verification.

Curriculum Vitae

Ronald Middelkoop was born on 29-06-1979 in Utrecht, the Netherlands. After finishing his VWO in 1997 at Dr. F. H. de Bruijne Lyceum in Utrecht, he studied Computer Science at the Eindhoven University of Technology. In 2003 he graduated within the Formal Methods group on a proof system for object oriented programming using separation logic. In 2004 he started a PhD project with the Software Engineering and Technology group, the results of which are presented in this dissertation. In 2007, he spent 3 months with Microsoft Research where he worked on the Spec# specification and verification tool. In 2009 and 2010 he worked as a software developer with FelineSoft in Bristol in the U.K. Since August 2010 he works as a software developer for ISAAC in Eindhoven.

P. Zoeteweij. *Composing Constraint Solvers.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell.* Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model.* Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems.* Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality.* Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting.* Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs.* Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations.* Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data.* Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space.* Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices.* Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization.*

trical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers.* Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented*