Program synthesis from domain specific object models

David Faitelson
St. Cross College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Hilary 2008

Acknowledgments

It is a rare opportunity to work on a project that is both interesting from a theoretical point of view as well as successful in practice. I would like to thank my supervisor Jim Davies for introducing me to the Booster project, and for the intense and productive discussions in which many of the ideas in this thesis were produced. During my research I had the pleasure to work together with Jim Davies and James Welch. I would like to thank both of them for this wonderful experience. I would like to thank Ib Sørenson for taking the time and the patience to explain to me the principles of Booster. I would like to thank Ib Sørenson, Dave Neilson and Ed Crichton for their work on Booster. I would like to thank Douglas Creager for his excellent tea, and for listening to my ideas with enthusiasm, while keeping a sharp watch on any ideas that were weak or incorrect. A special thanks goes to my examiners, Daniel Jackson and Jeremy Gibbons, whose careful reading and meticulous attention to every detail have greatly improved the quality of the thesis. Finally, I would like to thank my wife Anat, for constantly reminding me that there is more to life than weakest preconditions.

Abstract

Automatically generating a program from its specification eliminates a large source of errors that is often unavoidable in a manual approach. While a general purpose code generator is impossible to build, it is possible to build a practical code generator for a specific domain. This thesis investigates the theory behind Booster — a domain specific, object based specification language and automatic code generator. The domain of Booster is information systems — systems that consist of a rich object model in which the objects refer to each other to form a complicated network of associations. The operations of such systems are conceptually simple (changing the attributes of objects, adding or removing new objects and creating or destroying associations) but they are tricky to implement correctly. The thesis focuses on the theoretical foundation of the Booster approach, in particular on three contributions: semantics, model completion, and code generation. The semantics of a Booster model is a single abstract data type (ADT) where the invariants and the methods of all the classes in the model are promoted to the level of the ADT. This is different from the traditional view that considers each class as a separate ADT. The thesis argues that the Booster semantics is a better model of object oriented systems. The second important contribution is the idea of model completion — a process that augments the postconditions of methods with additional predicates that follow from the system's invariant and the method's original intention. The third contribution describes a simple but effective code generation technique that is based on interpreting postconditions as executable statements and uses weakest preconditions to ensure that the generated code refines its specification.

Contents

Chapter 1. Introduction 1

Chapter 2. Introducing Booster 3
- 2.1. The structure of a Booster model 3
- 2.2. Compilation 7
- 2.3. A case study 11
- 2.4. Related work 13
- 2.5. Summary 16

Chapter 3. A Mathematical Foundation 17
- 3.1. Programs 17
- 3.2. Specification statements 29
- 3.3. Discussion 35

Chapter 4. The structure of Booster models 37
- 4.1. Abstract syntax 37
- 4.2. Identifying entities in the model 39
- 4.3. The types of classes 40
- 4.4. The types of terms 40
- 4.5. The types of formulas 43
- 4.6. Well formed methods 44
- 4.7. Well formed models 46

Chapter 5. The Semantics of Booster 48
- 5.1. Component states 48
- 5.2. The outside world 50
- 5.3. Terms 50
- 5.4. Formulas 53
- 5.5. The meaning of methods 56
- 5.6. Models 57
- 5.7. Soundness 57
- 5.8. Modeling the case study 59
- 5.9. Related work 64
- 5.10. Discussion 73

Chapter 6. Model completion 74
- 6.1. Revealing information to the code generator 75
- 6.2. Maintaining association invariants 75
- 6.3. Related work 83
- 6.4. Discussion 84

Chapter 7. From predicates to programs 87
7.1. Booster's guarded command language 87
7.2. Dealing with undefined values 94
7.3. Generating code from primitive methods 95
7.4. Strengthening the precondition 98
7.5. Generating code from method combinators 101
7.6. Modular implementation of conjoined specifications 106
7.7. Discussion 112

Chapter 8. Discussion 115
8.1. The development of the theory 115
8.2. Contribution to Booster in practice 116
8.3. Booster in practice 118
8.4. Model completion 119
8.5. Programming with specifications 119
8.6. Domain specific formal methods 120
8.7. Solving problems at the right level 120
8.8. Object oriented specifications 121
8.9. Classes are not abstract data types 121
8.10. Limitations and future work 122

Appendix A. The concrete syntax of Booster 126
A.1. Specifications 126
A.2. Programs 127

Appendix B. Formal Booster operators 128

Appendix C. Term rewriting 130
Modeling rewrite rules 130
Confluence 132
Termination 132

Appendix D. An overview of Z 134
Describing data types 134
Useful properties of sets and relations 136
Schemas 138
Modeling behavior 140

Bibliography 141

CHAPTER 1

Introduction

> This process of constructing instruction tables should be very fascinating. There need be no real danger of it ever becoming a drudge, for any processes that are quite mechanical may be turned over to the machine itself.
>
> — Alan Turing

We often think of software development with formal methods as a top down approach. We begin with an abstract specification, written in a mathematical notation. We then gradually refine it, either by removing nondeterminism or by selecting more concrete data types, until we get a deterministic and concrete implementation. Each time we refine the specification we generate a set of theorems which we must prove, to ensure that the refinement is valid.

Unfortunately, there are two major problems that prevent formal methods from becoming widespread software development techniques [**8**]. The first problem is that it takes significant skill and knowledge in order to develop the mathematical models. This severely limits the number of people that can use the technique. The second problem is that in many cases insisting on proofs conflicts with the fact that the requirements frequently change. It takes a lot of time and effort to discharge the proof obligations, but when the requirements change they generate new obligations and the proofs must be done all over again.

Another approach that attempts to solve the problem is *program synthesis*. Program synthesis [**39, 50, 49, 23**] is the automatic translation of a specification into a program. The idea is that because we are so bad at writing correct code, we should be removed from the process and replaced by machines. In this brave new world, the only thing we will provide will be the specification, which according to some researchers will be in a natural language [**52**]. The machine will do the rest.

However, despite many years of research, programmers are still human, and there is no indication that this situation is going to change. The problem is not technical, but fundamental. In order to automate a task we must know how to perform it so well that it already appears to us mechanical. This is why robots have successfully replaced workers at the car factory, and this is why (mechanical) computers have successfully replaced the human computers at the beginning of the 20th century.

The greatest benefit of automation is that it relieves us from the manual labor of mechanical and repetitive tasks at which we are poor, and leaves us to focus on new problems which we do not yet understand. As time goes by our understanding grows, until what was once considered a difficult subject becomes so well understood that it is routinely solved, then mechanically solved, and finally embedded in an actual machine.

Automating programming is a particularly difficult problem because we use computers to solve an unbounded variety of problems: Rendering images to create animations, calculating salaries, predicting the weather, controlling washing machines, playing chess, and so on and so forth. Each class of problems has its own domain of knowledge: to render images we must know a lot about analytical geometry, to predict the weather we must know a lot about differential equations, and so on. Each domain has its own mathematics which effectively encodes our knowledge for that particular domain.

Therefore we can automate programming only when we can identify a domain with such a well known body of knowledge, that existing implementations are produced (or may be produced) in a routine and obvious fashion[1].

One particular domain in which implementations are routine, is the area of information management systems (IMS). An IMS is an application in which the database is the most important and complicated component. In an IMS the database models a *complicated business* domain (for example it keeps track of the organizational structure of a company, or of the insurance policies of the customers in a large insurance company). By business we mean that the entities in the database are at an abstraction level that is suitable for domain experts (employees and managers not maps and hash tables). By complicated we mean that the entities in the domain refer to each other to form a network of relationships. This is in contrast to more simple records that are kept in transaction processing databases (for example purchase records).

The operations of an information management system consist of querying and maintaining the database: adding new objects, removing old objects, updating the attributes of existing objects, and maintaining the associations between the objects. The implementation of such operations is routine and conceptually simple. They do not involve interesting algorithms, only the shuffling of information from one place to another. But it is often tricky to get them right because they are numerous, and because the objects that they manipulate are linked to each other in intricate ways, and often changing the state of one object must be accompanied by a change in the state of another object in order to keep the system consistent.

Thus, this domain has precisely the conditions in which it is worth while to attempt automation[2]. It therefore offers a good context in which to investigate the principles of formally based domain specific automatic program generators. One such example is the Booster[**14**] toolkit. In this thesis we investigate the theoretical principles behind Booster.

We begin, in chapter 2, with an informal introduction to the language and to the compilation process: A Booster specification consists of a set of classes, each having attributes, associations and method specifications. The compilation process begins with a *model completion* step, which reveals the consequence of the interaction between the methods and the global constraints of the model, followed by a consistency enforcement step which ensures that the specification is implementable, and finally code generation. The introduction to the language in chapter 2 motivates the theoretical discussion in the rest of the work. In chapter 3 we build a simple relational theory of specifications and programs. This theory provides the mathematical framework in which we explain the semantics of Booster and the code generation process. In chapter 4 we describe the abstract syntax and the type rules of the Booster language. In chapter 5 we define the meaning of a Booster model. In chapter 6 we explain in detail the process of model completion and prove that it retains the semantics of the original model. In chapter 7 we describe the code generation process, explain how the programs are generated from their method specifications and prove that the generated code refines the specification. We conclude the thesis with a chapter that reflects on the wider implications of the theoretical analysis and points out directions for further work.

[1] Of course, the implementation of the mechanical device that automates these tasks is far from being routine and obvious.

[2] We are not the first to notice this. See for example the papers by Gregory Ruth [**50**] and N. S. Prywes [**48**]

CHAPTER 2

Introducing Booster

In this chapter we present the key constructs of the Booster language: associations, methods, and method combinators. We explain how the model is expanded to actively maintain association invariants, and how it is strengthened to ensure that it is consistent. Finally, we explain how the model is implemented by translating it to a guarded command language.

The tone of this chapter is informal and the concepts are introduced by way of examples, deferring a more rigorous treatment to later chapters.

2.1. The structure of a Booster model

A Booster model is a collection of classes, each defining the behavior of a particular kind of object. A class has a list of attributes, associations, and methods. For example, the following model defines two simple classes:

```
CLASS A
  ATTRIBUTE
    an : NAT
    ab : [B]
END ;

CLASS B
  ATTRIBUTE
    bn : NAT
    ba : SET(A)
END
```

In this model, each object of class A has two attributes: an is a primitive attribute, of type NAT; ab is an optional attribute (indicated by the square brackets), which may contain a reference to an object of class B. Each object of class B has an attribute ba representing the set of all A's (as a set of references to A objects).

In general an attribute may denote a primitive value, such as an integer or string, or a reference to another object (an instance of another class). Attributes may be optional, mandatory, or set-valued. The type of an optional attribute is enclosed with square brackets; a set-valued attribute is indicated by the keyword SET.

If the type of an attribute is another class, then its declaration introduces an association. Although the Booster language supports unidirectional associations of the kind seen in the first example— ab : [B] —the use of bidirectional associations is strongly recommended: an explicit association in one direction creates an implicit association in the other; making that association explicit can aid navigation, understanding, and analysis.

A bidirectional association is introduced by identifying the corresponding attribute of the target class. In our example, we may replace the two unidirectional associations, represented by the attributes `ab` and `ba`, with a single bidirectional association.

```
CLASS A
  ATTRIBUTE
    an : NAT
    ab : [B.ba]
END ;
CLASS B
  ATTRIBUTE
    bn : NAT
    ba : SET(A.ab)
END
```

An association between two classes introduces a global *association invariant* on the model. In the example above the association between `ab` and `ba` means that given any object `a` of class `A`, the reference `a.ab` points to the object `b` if and only if `a` is a member of the set `b.ba`.

When a pair of attributes represents a single bidirectional association, any method that changes the value of one must also change the value of the other, in order to maintain the association invariant.

2.1.1. Specifying behavior. Methods specify the operations of the model. Methods can be either primitive or constructed from other methods using *combinators*. Each primitive method is declared as a pair: a *precondition* that describes the initial states from which the method can be called, and a *postcondition* that describes the intended effect of the method. The first of these is a predicate upon attribute values and inputs. The second is a relation between values before and after the operation. Consider the method `AM`, defined upon objects of class `A`:

```
CLASS A
  ATTRIBUTES
    an : NAT
    ...
  METHODS
    AM(n_in : NAT & n_in > 0 | an = an_0 + n_in)
```

The precondition insists that the method can be applied only if the input variable `n_in` is a natural number and its value is strictly greater than zero. The postcondition insists that the value of `an` after the operation completes should be equal to the original value, plus the input.

As in the Refinement Calculus [**44**], the annotation `_0` is used to denote the value of a particular variable before an operation.

The annotation `_in` denotes the input values for the current operation. The type of each input variable must be clearly specified in the precondition by asserting that it is a member of its type. In the example above we have declared `n_in` to be a natural number by asserting that it is a member of `NAT`.

Because Booster is a specification language, encapsulation is not necessary. Every attribute in the model already corresponds to a visible property of its class. As a result, a method in Booster may access in addition to the attributes of its own class, also the attributes of any other object that is reachable from any of its variables. In the example below the method `BM` is referencing the `ab` member of the input variable `a_in`:

```
CLASS B
 ATTRIBUTES
   ...
 METHODS
  BM(a_in : A & a_in.ab.the /= this | a_in : ba)
```

The variable `this` always refers to the object on which we apply the method. The Booster language uses the notation `x /= y` to mean *x is not equal to y*, and the notation `e : s` to mean *e is a member of the set s*. Similarly, `e /: s` means, *e is not a member of the set s*.

The postfix operator `.the` retrieves the value of an optional attribute. In the example above the use of `.the` means that the method is available only if `a_in.ab` refers to an object *and* the object is different from the current object. This prevents the method adding `a_in` to `ba` when `a_in.ab` does not point to any object. In order to always add `a_in` we have to change the precondition into

```
  a_in.ab = {} or a_in.ab.the /= this
```

2.1.2. New objects. Each class defines an implicit set of identifiers, called the *extent* of the class, which holds all the identifiers of the instances of the class. As we create new objects, their class extent grows to accommodate them.

To specify that a variable refers to a new object of class `A` we assert that it is a member of the set `new(A)`. The set `new(A)` represents all the instances that appear in the extent of `A` but not in the extent of `A_0`. For example,

```
  B_Create(true | this.a : new(A))
```

sets `this.a` to a new instance of class `A`.

It is not necessary to specify the values of optional attributes of new objects, but mandatory attributes must be specified. For example, consider the following class `C` which has a mandatory association to the class `A`:

```
CLASS C
  ATTRIBUTES
    c_a : A
  METHODS
    C_Create(a_in : A | c : new(C) & c.c_a = a_in)
```

If we remove the predicate that specifies the value of `c.c_a`, the result is an invalid specification and will be rejected by the Booster compiler.

2.1.3. Method combinators. As the number of details in a primitive method increases, the pre- and postconditions become more complicated. Without some means of making them more modular, such specifications may become difficult to understand. Method combinators solve this problem by providing a way to construct methods from other methods. Given that `M1` and `M2` are methods,

- `M1 AND M2` is a method that will be available (its precondition will be true) whenever both `M1` and `M2` are available; it has the combined effect of both methods;
- `M1 OR M2`[1] is available whenever either `M1` or `M2` is available; it has the effect of `M1` if that method is available; otherwise it has the effect of `M2`.
- `M1 THEN M2` has the effect of `M1` followed by that of `M2`; it is available whenever `M1` is available *and* the effect of `M1` would make `M2` available.

2.1.3.1. *The `AND` combinator.* We require that the two arguments to `AND` are independent, in the sense that the changes made by one method do not affect any variables that are used by the other method and vice versa. For example, the specification

```
(x_in : NAT | x = x_in) AND (true | y = x)
```

is illegal because the variable `x` which is being changed by the first method, is used by the second method. In contrast, the specification

[1] Unlike what the name suggests, the `OR` combinator is not symmetric. However we keep the name `OR` because this is the name of the combinator in the Booster compiler and because with a different name we will lose the intuitive understanding of its effect. Programmers are very familiar with asymmetric logical operators such as the C and Java `||` and `&&` operators, so perhaps the choice of the name is not so bad after all.

```
(x_in : NAT & x_in<10 | x = x_in) AND (y > 0 | y = y_0-1)
```

is legal, and has the meaning

```
(x_in : NAT & x_in < 10 & y > 0 | x = x_in & y = y_0-1)
```

2.1.3.2. *The Guard combinator.* A method may be guarded with an additional constraint; this has the effect of adding that constraint to the overall precondition. For example, the method

```
( n_in : NAT & n_in <  100 | AM)
```

is equivalent to the primitive method

```
(n_in : NAT & n_in <  100 & n_in > 0 | an = an_0 + n_in)
```

2.1.3.3. *The `OR` combinator.* We use the `OR` combinator to perform different things depending on the precondition. The following example demonstrates this idea by defining a method that sets `x` to the absolute value of the input variable `x_in`:

```
Positive(x_in : NAT & x_in >= 0 | x = x_in)
Negative(x_in : NAT & x_in < 0 | x = - x_in)
Abs(Positive OR Negative)
```

This definition of the method `Abs` is equivalent to the primitive method

```
Abs(x_in : NAT |
    (x_in >= 0 => x = x_in) &
    (x_in < 0 => x = -x_in))
```

2.1.3.4. *The `SKIP` combinator.* The combinator `SKIP` is always available, and its execution has no effect upon the state of the system. It may be used, in combination with `OR`, to make a composite method that is always available. For example:

```
(AM OR SKIP)
```

The precondition of this method is `true`, which means that the method can be applied in any state, even if the value of `n_in` is not positive. However, when the value of `n_in` is not positive the method does nothing. It is equivalent to the primitive method

```
  (n_in : NAT | (n_in > 0 => an = an_0 + n_0) &
                (n_in <= 0 & true => true))
```

2.1.3.5. *The `THEN` combinator.* We use the `THEN` combinator to specify activities that require an explicit sequencing of actions. For example, the specification

```
(x_in : NAT | x = x_in) THEN (x > 0 | y = x - 1)
```

first sets the value of `x` to the input `x_in` and then sets the value of `y` to `x-1`. However, because the precondition of the second method requires that `x` must be positive, the method won't execute unless the first method ends in a state where `x` is positive. In other words, the precondition of the `THEN` combination will be `x_in > 0`.

In general, the precondition of a `THEN` combinator ensures that the first method will end up in a state that satisfies the precondition of the second method. This behavior sets the `THEN` combinator apart from simple sequential composition since it turns the combination of methods into a transaction. Either the whole sequence of methods is executed, or none of the methods are executed.

2.1.3.6. *The Reference combinator.* Finally, we can compose a method by referring to other methods in the model. To illustrate this idea let us extend our model with a new class `D` which uses a method reference to specify one of its own methods:

```
CLASS D
  ATTRIBUTES
   da : A
  METHODS
   DM(this.da.AM)
```

```
END
```

In the expression `this.da.AM` we replace the reference to `this` within `AM` by the expression `this.da`. The meaning of the method `DM` is therefore equivalent to the following direct definition:

```
DM(n_in : NAT & n_in > 0 | this.da.an = this.da.an_0 + n_in)
```

References are not recursive. That is, it is illegal to use the name of a method while it is being defined.

2.1.4. Input variables. Even though input variables may appear anywhere in a method their value is taken at the beginning of the method's execution. This applies both to primitive methods and to combinators. In particular this is true for methods that are constructed using the `THEN` combinator, as the following example demonstrates:

```
(x_in > 0 | x = x_in) THEN (y_in > 0 | y = y_in + x)
```

Even though it appears as if `y_in` is read only when the second method executes, in fact both `y_in` and `x_in` are read (and tested) before the method starts.

The reason is that otherwise it is impossible to define preconditions. To see why, consider the precondition of the example above. When can we execute the method? we must have `x_in > 0` since this is the precondition of the first method, but what about the value of `y_in`? if by the time the first method finishes executing the value of `y_in` becomes negative, the second method cannot be executed.

2.2. Compilation

Like any other compiler, the Booster compiler translates source code into a working program. Unlike traditional compilers, however, the Booster compiler analyses and modifies the source code to ensure that it is consistent and that it reflects the intentions of the specifier. The compiler generates for each primitive method additional pre- and post-conditions, based upon the types of the attributes involved (and hence the association information in the model). The resulting *expanded model* is then ready for subsequent implementation.

2.2.1. Completion. When we write a postcondition like `b : a.ab` we expect the system to add `b` to `a.ab`. If `ab` participates in an association we also expect that the system will add the corresponding object to the other end of the association: `a : b.ba`.

In fact, this complementary intention exists in the model. Since `ab` and `ba` are the two inverses of the same association it follows that if `b` is a member of `a.ab` then `a` must be a member of `b.ba`.

The problem is that the information is implicit, so it is not visible to the Booster compiler. By making the complete intention explicit in the model, we provide the compiler with the necessary predicates to generate code that actively achieves the intended behavior of the methods. For example, the compiler completes the method postcondition of `BM` with the predicate

```
a_in.ab.the = this &
(a_in.ab_0 /= {} =>
  (a_in.ab_0.the /= this =>
    (a_in /: a_in.ab_0.the.ba)))
```

which means that if the old value of the optional attribute `ab_0` in `a_in` refers to an object that is different from the current object, then `a_in` must be removed from the collection `ba` of that object.

This is however, just one possible way to complete the postcondition. We can complete the postcondition by adding any predicate that is implied by the model invariant and the postcondition. In addition, if we allow to strengthen the assumptions of the method (by strengthening the precondition), then there are even more ways in which we can complete the method.

But there is no single completion strategy that is better than the others. This is because the purpose of completion is to free the developer from manually specifying the usual consequences of his intentions. What these consequences are, depends on the domain of application. Therefore the particular completion

strategy depends on the domain of application. Indeed, we may regard the completion rules that we define in this work as a detailed example that illustrates the mathematical framework in which we can develop and reason about completion.

In all the examples we have seen so far it was always possible to perform an operation on an association from both its ends. But this is not always the case. Consider for example the following model:

```
CLASS E
  ATTRIBUTES
    ef : F.fe
END
CLASS F
  ATTRIBUTES
    fe : SET(E.ef)
  METHODS
    FM(e_in : fe | e_in /: this.fe)
END
```

The problem with the method `FM` is that the attribute `ef` of the object `e_in` must always point to some object, but because we remove `e_in` from `this.fe` we cannot keep `e_in.ef` pointing to `this`. We must use a different object. But which one? the specification is silent, and using an arbitrary object is probably not such a good idea. Can we remove `e_in` from the system? this would save us the trouble of worrying about its mandatory attributes, but it introduces another problem. Other objects may have mandatory attributes that refer to `e_in`. Shall we then remove these objects as well? there is a danger that the cascading effect of this seemingly innocent operation will end up removing large parts of the system. We said earlier that the approach we have taken is to do as much as we possibly can. But following this approach to the very end means that we may find ourselves in a situation where the system deleted the whole database in order to maintain the association invariants. Therefore we do not allow to remove an object from the 'many' side of a mandatory-to-many association.

The fact that we cannot remove `e_in` from the many side of a mandatory-to-many association, does not prevent us from removing it from the association. We can always set `e_in.ef` to a different `F` object, in which case `e_in` will be removed from `this.fe`.

2.2.2. Definedness and consistency. The next step in the compilation process is to ensure that the specifications of each method are consistent. By that we mean two things: First, every expression and predicate in the model must have a meaning. Second, the postconditions of every method must be free from contradictions.

A predicate might be meaningless if it contains a partial function that is applied to a value that is not in its domain. For example, the operator `.the` is a partial function because it has no meaning when applied to an empty optional attribute. Consider for example the method `BM`:

```
BM(a_in.ab.the /= this | a_in : ba)
```

The term `a_in.ab.the` is meaningless when the optional association `a_in.ab` is empty, so the Booster compiler strengthens the precondition to avoid this case:

```
BM(a_in.ab /= {} & a_in.ab.the /= this | a_in : ba)
```

In general the Booster compiler strengthens any predicate, whether it appears in the pre- or the postcondition, with an additional constraint that ensures that the predicate is always meaningful.

The second kind of inconsistency is visible in the following example:

```
(true | x_in : s & y_in /: s)
```

If `x_in` and `y_in` refer to the same object it is impossible to achieve the postcondition. Therefore, the Booster compiler strengthens the precondition of this method with the predicate

```
x_in /= y_in
```

which ensures that the method can be implemented.

2.2.3. Code generation. The last thing that the Booster compiler does is to transform the model into one in which each method specification is replaced by a program. This step consists of two parts. The first generates a program in a platform independent programming language, and the second transforms that to a platform specific language. The platform independent programming language is based on Dijkstra's guarded command language, extended with constructs to support object creation, navigation and update, and with primitive set operations. For example, the implementation of the method `BM` looks like this:

```
BM(a_in : A) {
  local before1 = a_in.ab
  in
    add(this.ba, a_in);
    a_in.ab := this;
    before1 /= emptyset ->
        before1.the /= this ->
            remove(before1.the.ba, a_in)
        []
        before1.the = this -> skip
    []
    before1 = emptyset -> skip
  end
}
```

Instead of a specification, the method is now followed by a list of input variables and a program. The program begins by capturing the old value of `a_in.ab` in the local variable `before1`. The next line adds `a_in` to the set `this.ba`, and the third line sets `a_in.ab` to `this`. The rest of the code ensures that if `a_in.ab` was previously referencing an object (that is different from the current object) then `a_in` is removed from that object.

In general, the programming language has the same structural elements as the Booster modeling language. That is, it uses the same syntax to represent classes and attributes, and it has the same basic types: integers, strings, sets, etc.

2.2.3.1. *Deriving programs from postconditions.* Of the two parts that make a method, the postcondition is the more important because it determines the intended effect of the method. The precondition is then used to ensure that the postcondition can be implemented.

Accordingly, the Booster compiler focuses on the postconditions when it implements the methods. Each predicate in the postcondition is translated to a program: equality is translated to an assignment statement; a test for set membership is translated into an operation that adds the element to the set; conjunction between two predicates is translated into a sequential composition of their corresponding programs; implication is translated into a conditional statement. For example, the predicate

```
x = e & (a /= x_0 => a : s)
```

becomes the program

```
local x_0 = x
in
  x := e ;
  a /= x_0 -> add(s, a)
  []
  a = x_0 -> skip
end
```

Not every predicate has an obvious matching program. How should we implement negation? or disjunction? There doesn't seem to be a good rule that we can follow in such cases. But we must generate some program. The solution is to generate the program `skip` whenever we cannot find a good program for a particular predicate. Here we follow the rule: if you don't know how to do something then do nothing.

Of course, the generated program is not, in most cases, a correct implementation of the method. The next step in the code generation process takes care of this problem by strengthening the precondition of the method to ensure that the generated program will achieve its postcondition. The following example demonstrates this technique on a simple specification.

Original Precondition	`true`
Strengthened Precondition	`y < x + 1`
Postcondition	`x = x_0 + 1 & y < x`
Generated Program	`local x0 = x`

```
local x0 = x
in
   x := x0 + 1 ;
   skip
end
```

As we will explain in details later, the effect of using `skip` to implement a predicate is to propagate the predicate from the postcondition back to the precondition. This is demonstrated in the example above, where the predicate `y < x` in the postcondition is pushed back to the precondition where it becomes the predicate `y < x + 1`.

But, you may ask, why are such changes valid? isn't it a dirty trick to change the specification to match the generated code? and what arguments justify the completion of the postcondition?

We will answer these questions at length in chapters 5, 6 and 7. At this point we only give an informal overview of the main ideas:

- The meaning of a Booster model depends on the set of programs that can be generated based on the information in the model. Thus, the meaning of a Booster method is only partially given by the user's specification. There is already in place an implicit set of constraints dictated by the model, which the Booster compiler makes explicit during the compilation process.
- The completion rules are justified by observing that each completion can be deduced from the postcondition and the constraints that are imposed on the model by the associations. Therefore the completed predicates when taken together with the constraints of the model retain the original meaning of the model.

2.2.4. Implementing method combinators. We implement method combinators by combining the implementations of their arguments.

2.2.4.1. *The `AND` combinator.* The `AND` combinator is implemented by sequentially composing the implementations of its arguments. For example, the specification

```
(true | x = x_in) AND (true | y = y_0 - 1)
```

is implemented by the program

```
x := x_in ;  local y0 = y in y := y0 - 1 end
```

The order in which we compose the implementations of an `AND` combinator is not important.

2.2.4.2. *The `THEN` combinator.* In contrast, the order of the composition in the implementation of the `THEN` combinator is important. For example, the specification

```
(true | x = x_in) THEN (x > 0 | y = x - 1)
```

is implemented by the program

```
x := x_in ; y := x - 1
```

2.2.4.3. *The* `OR` *combinator.* The `OR` combinator is implemented by using the programming language choice operator to select the first method whose precondition holds. For example the specification

```
Abs(Positive OR Negative)
```

becomes the program

```
 x_in >= 0 -> x := x_in
 []
 x_in < 0 & x_in < 0 -> x := - x_in
```

The predicate `x_in < 0` appears twice in the second branch of the choice operator because it appears once as the negation of the precondition of the first method and once as the precondition of the second method.

2.2.4.4. *The guard combinator.* Finally, the guard combinator is implemented using the program of the guarded method without making use of the guard. For example the specification

```
( n_in <  100 | AM)
```

is implemented by the program

```
an := an_0 + n_in
```

The reason is that the guard contributes only to the precondition of the method. But for the method's implementation, the precondition is an assumption and need not appear as part of the code. The precondition only appears when the method is used either in another combination or when it is called by the user. For example if we place the example above in an `OR` combinator like this:

```
(n_in < 100 | AM) OR (n_in = 100 | AM)
```

then the Booster compiler will use the preconditions of both methods to decide which one to execute. The implementation of this combination will be:

```
n_in < 100 & n_in > 0 -> an := an_0 + n_in
[]
(n_in >= 100 or n_in <=0) &
  (n_in = 100 & n_in > 0) -> an := an_0 + n_in
```

The second choice is the result of conjoining the negation of the first method's precondition and the second method's precondition. Note that the precondition of a guarded method is the conjunction of the guard with the precondition of the method. This program can be simplified to the program

```
n_in < 100 & n_in > 0 -> an := an_0 + n_in
[]
n_in = 100 -> an := an_0 + n_in
```

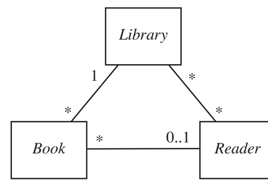
The precondition of the complete `OR` combinator, which is

```
(n_in < 100 & n_in > 0) or (n_in = 100 & n_in > 0)
```

does not appear in the implementation, but will be used by the compiler if the method will appear as a part of another combinator.

2.3. A case study

To illustrate how the compiler works, we consider a Booster model of a simple library system, containing multiple libraries, readers and books. The following diagram summarizes the associations between the elements of the library system:



The association between readers and libraries is a *many-to-many* association, that is, a reader can be a member of many libraries, and a library can be used by many readers. The association between books and readers is *optional-to-many*, that is, a book might be used by at most one reader, but a reader can borrow many books. Finally, the association between books and libraries is *one-to-many*, that is, every book must belong to exactly one library, but each library may hold many books.

Here is the library system described in Booster, with a few illustrative methods:

```
CLASS Reader
  ATTRIBUTES
    Reader_books : SET(Book.Book_reader)
    Reader_libraries : SET(Library.Library_readers)
  METHODS
    Reader_Borrow(book_in.Book_reader = {} |
                book_in : this.Reader_books)
END

CLASS Book
  ATTRIBUTES
    Book_reader : [Reader.Reader_books]
    Book_library : Library.Library_books
  METHODS
    Book_Transfer(true | this.Book_library = library_in)
END

CLASS Library
  ATTRIBUTES
    Library_books : SET(Book.Book_library)
    Library_readers : SET(Reader.Reader_libraries)
  METHODS
    Library_Add(true | reader_in : this.Library_readers)
END
```

2.3.1. Completion. Consider the method `Reader_Borrow` which adds a book to the set of books borrowed by a reader, say Alice. Alice's set of books and the book's `Book_reader` attribute are associated to each other, so the compiler must make sure that when the `Reader_Borrow` method terminates the book's reader is set to Alice. The completed `Reader_Borrow` method looks like this:

```
Reader_Borrow(book_in.Book_reader = {} |
 book_in : this.Reader_books &
 book_in.Book_reader.the = this &
 (book_in.Book_reader_0 /= {} =>
   (book_in.Book_reader_0.the /= this =>
    (book_in /: book_in.Book_reader_0.the.Reader_books))))
```

In this case we do not want the system to steal the book from an existing reader, so the method is protected with a precondition that ensures the new book has no reader. This makes the antecedent of the outer

implication in the postcondition false, preventing the compiler from removing the book from the old reader.

2.3.2. Code generation. The implementation of the `Reader_Borrow` method looks like this:

```
Reader_Borrow(book_in : Book) {

  local before3 = book_in.Book_reader
  in
    add(this.Reader_books, book_in);
    book_in.Book_reader := this;
    before3 /= {} ->
      before3.the /= this ->
        remove(before3.the.Reader_books, book_in)
      []
      before3.the = this -> skip
    []
    before3 = {} -> skip
  end
}
```

The first line in the program captures the old value of the book's reader in the local variable `before3`. Then the primitive operation `add` adds the new book to the reader. The next line sets the new book's reader to the current reader. This is followed by a conditional statement that, provided that the new book's previous reader is not equal to the current reader, removes the new book from its previous reader.

2.4. Related work

Booster grew out of the experience of using the B-method [**4**] to develop information management systems. Therefore it is worth while to give a short introduction to B, and to describe the similarities and differences between B and Booster.

2.4.1. The B-Method. The B-Method [**4**] is a formal language that supports the derivation of programs from their specifications. It consists of two notations: the Abstract Machine Notation (AMN), which is used to specify the structure of the system, and the Generalized Substituted Language (GSL), which is used to specify the operations of the system.

A B model is constructed from one or more *abstract machines*. Each abstract machine has a name and a set of sections that define the structure and the operations of the machine:

```
MACHINE
  machine name
VARIABLES
  a,b,c,...
INVARIANT
  a : NAT & b : NAT & c < a & ...
OPERATIONS
  op1 = ... ;
  op2 = ...
END
```

The operations are specified as generalized substitutions. For example,

```
  op1 =
    PRE
      a > 0
```

```
THEN
  a := a - 1
END ;
```

describes an operation `op1` that decrements the value of `a`, provided that `a` is not negative.

Generalized substitutions extend the weakest precondition calculus with partial operations. As in the weakest precondition calculus, the meaning of an operation is given as a function that maps postconditions to their weakest preconditions. For example,

$$wp(PRE\, q\, THEN\, s\, END, P) = q \wedge wp(s, P)$$

We use the weakest preconditions to ensure that each operation respects the machine's invariant. If I is the machine's invariant then for every operation op we must prove that

$$I \Rightarrow wp(op, I)$$

We can compose machines to form larger machines. When machine A *includes* machine S, the content of S becomes available to A. This means that A can refer to the variables and sets of S in its invariant and in its operations. The only restriction is that in order to change a variable of A we must use an operation of A.

The reason for the programmatic flavor of the B language is that it was designed to provide the basis for a toolkit. Indeed there are several such toolkits today. We use the B-toolkit [**62**] as an example.

The B-toolkit supports a complete development cycle beginning with the abstract specification of the system and ending with a running program. At each step of the development process the B-toolkit generates the proof obligations that we must discharge in order to show that the more detailed model refines the current model. The B-toolkit also provides theorem provers that attempt to automatically discharge as many proof obligations as possible. However, in most cases there are some proof obligations that we must discharge manually. The development stops when we construct an *implementation*, which is a restricted machine that can be translated into an executable program. The B-toolkit then generates a C program that implements the original model.

2.4.2. B and Booster. The development of Booster grew out of the experience of using the B-toolkit to develop the student management system of the software engineering school at the University of Oxford. The system keeps track of students, lectures and modules: which students have registered to which modules, their payment details, their grades, which lecturers are teaching which modules and so on. Two major problems with B led to the development of Booster.

First, every time the requirements changed, the process of refinement and implementation had to be redone. This was both time consuming and tedious. However the developers of the student management system noted that the relationship between the specification and the implementation is quite straightforward and that most of it could be mechanically deduced from a description of the system's structure (which entities refer to which entities and what is the nature of the reference) and operations.

In addition, the nature of the system called for an object oriented model since it was easy to think of each entity as a class: student, supervisor, course, and so on. We can then model the properties of each entity as attributes of the corresponding class and model the relationships between the entities as references between the objects.

However the B language does not provide a convenient notation for modeling in this style. It does not have a concept of a class and it does not have a built in notion of object references.

In particular, a machine is not equivalent to a class. This is for a number of reasons. First, we cannot create instances of machines at run time. This seems like a technical reason (maybe it is possible to modify B slightly so that we will be able to create machines at run time) . But there is a deeper reason why machines and classes are different.

Machines partition a complex specification into modules. The dependency between machines is strictly top down, forming a tree in which each leaf machine can be understood in isolation, and the

higher level machines can be understood from the specification of their children and from the way these children are combined to implement the higher level machines.

In contrast, the relations between classes often form a graph of references. In many cases the relations are bidirectional — a parent refers to its children and a child references its parent. Whenever we operate on such structures, we cannot consider them in isolation.

If we are to model such structures in a language like B we have no choice but to represent the attributes of each entity as functions that associate each object with the attribute's value. For example, here is a machine that closely mimics the simple Booster model we used in chapter 2:

```
MACHINE example
SETS
  ExtentA, ExtentB
VARIABLES
  an,ab,bn,ba, A, B
INVARIANT
  A <: ExtentA &
  B <: ExtentB &
  an : A -> NAT &
  ab : A +-> B &
  bn : B -> NAT &
  ba : B <-> A &
  ab = ba~
INITIALIZATION
  A = {} & B = {} & an = {} & ab = {} & bn = {} & ba = {}
OPERATIONS
  AM(n_in) =
    PRE n_in : NAT & n_in > 0 THEN
      an := an + n_in
    END
  BM(a_in,this) =
    PRE a_in : A & this : B & a_in |-> this /: ab THEN
      ba := ba \/ this |-> a_in
    END
END
```

Instead of classes, we have two sets that hold the potential instances of each class (`A` and `B`), and two sets that hold the actual instances of each class (`ExtentA` and `ExtentB`). The members of these sets are not the objects, but rather the unique identifiers of the objects. Each attribute is represented using a function. To find out the value of `a.an` we apply the function `an` to the variable `a`.

There are two important things we may learn from this example. First, compared to the same Booster model this version is more difficult to understand. It is not practical to model a realistic system using this technique.

However, this style of specification reduces an object model that consists of classes, objects, and object references, into a mathematical model that consists of functions and relations. The translation is very simple: every class introduces a set of functions that map the (unique identifiers of the) objects to their attributes. It is therefore an ideal foundation for the underlying semantics of an object oriented specification language.

Indeed such a translation is the basis for the UML-B profile [**54**] which translates a UML model into an equivalent B model. UML-B translates all the classes in the class diagram into a single B machine. Each class becomes a set, and the attributes and associations (only unidirectional associations are supported) of the class become functions from the class to their type.

The same translation appears in the paper [**36**] which describes another technique for translating UML to B. In addition, this paper describes an approach that is similar to the one we develop in this thesis: that the structure of a constraint can be interpreted as a program for maintaining the constraint, and that the precondition of an event must be strengthened if such a program cannot be generated. However the approach is only described briefly on a very particular case (the synthesis of event code) and lacks many details.

A direct translation to B is appealing because we can reuse the theory and the tools that were developed for B. However the freedom we get by creating a new language makes it possible to support features that are difficult to implement in an existing system which was not designed with these features in mind.

2.5. Summary

In this chapter we have introduced the Booster language and illustrated the process that translates a Booster model to a platform independent programming language. We have seen that a Booster model consists of a set of classes interconnected by associations, and that the behavior of the model is specified using methods and method combinators. We have seen that the meaning of a Booster model depends not only on the explicit definitions in each method, but also on the interaction between these definitions and the constraints that are imposed on the model by its structure. We have introduced two key phases of the compilation process, namely *Completion* and *Consistency enforcement* and have illustrated their application. This informal introduction leaves many questions unanswered:

- How can we be sure that the completion rules respect the semantics of the model?
- What are the details of the process that strengthen the preconditions? how can we be sure that it respects the semantics of the model?
- What is the precise meaning of expressions and statements that involve old variables?
- What is the precise semantics of the implementation language?
- In particular, what is the meaning of assignment to attributes?
- How can we be sure that the generated program is a valid refinement?

In order to answer these questions we develop in the rest of this work a formal model of the Booster language that precisely defines the syntax, structure, and semantics of the language. We also formally define the algorithms that make up the compilation process, and prove that they respect the semantics of the model. At the end of this work we will have a precise and clear answer to each of these questions.

CHAPTER 3

A Mathematical Foundation

Booster takes ideas from a number of formal notations. The primitive methods in Booster are based on the specifications statements of the refinement calculus, while method combinators are based on the schema operations of Z. The Booster notation is based on the syntax of B, but the code that we generate from Booster models is based on Dijkstra's guarded command language. The theory in this chapter is the result of trying to create a single mathematical framework for all the relevant aspects of these notations.

3.1. Programs

In this section we define programs as relations between states. We also define what it means for one program to refine another program, and how programs can be composed. We use a total correctness model, meaning that for any state in the domain of the program, the program is guaranteed to terminate. We explain how we change the sequential composition operator in order for the model to be faithful to the behavior of programs.

3.1.1. States. A computer program reads values from memory, performs calculations on these values and stores the results back to the memory.[1] This process repeats until the program stops and we can examine the result. We often use named variables to represent the memory locations that are relevant to the computation. Each assignment of values to the variables is a state of the computation. The *initial state* of the computation is the state just before the programs begins to run, and the *final state* of the computation, is the state immediately after the program has finished running.

Formally, we model states as partial functions that map the name of each variable (that is visible to the program) to its value.

$[Name]$

$State == Name \rightarrow\!\!\!\!\!+ \ Value$

We call the set of variables that a program observes (and modifies), the *alphabet* of the program. The alphabet of a program is by definition fixed. This means the states of a program, taken as functions from alphabets to values, are total functions.

If we ignore the intermediate steps of a computation and focus only on the outcome, then we can model a program as a relation between initial and final states:

$$Program : \mathbb{P}(State \leftrightarrow State)$$

$$\forall p : State \leftrightarrow State \bullet p \in Program \Leftrightarrow (\exists Alpha : \mathbb{P}\, Name \bullet p \in (Alpha \rightarrow Value) \leftrightarrow (Alpha \rightarrow Value))$$

By allowing states to be partial functions but insisting on a common alphabet for programs, we make it possible to model local variables. We say more about this matter in chapter 7. In the rest of this chapter we always use programs with the same alphabets.

[1]It is true that there are programming methodologies such as functional and logic programming where the concept of a program does not involve assignments that modify a store. However in this thesis we focus on an imperative semantics because the Booster compiler generates imperative programs.

3.1.2. Values. The theory that we describe in the rest of this chapter does not depend on any particular kind of value. However, to describe the semantics of Booster it is useful if we have values for the following data types:

- the result of an undefined calculation
- object identifiers
- sets of object identifiers
- binary relations between object identifiers
- functions from object identifiers to values

$$\begin{aligned} \mathit{Value} \quad ::= &\ \bot \mid \mathit{objectid}\langle\langle \mathit{ObjectID}\rangle\rangle \\ \mid &\ \mathit{set}\langle\langle \mathbb{P}\,\mathit{ObjectID}\rangle\rangle \\ \mid &\ \mathit{function}\langle\langle \mathit{ObjectID} \rightarrow\!\!\!\!\!+ \mathit{Value}\rangle\rangle \\ \mid &\ \mathit{relation}\langle\langle \mathit{ObjectID} \leftrightarrow \mathit{ObjectID}\rangle\rangle \end{aligned}$$

$[\mathit{ObjectID}]$

3.1.3. Reasoning about programs. We specify the behavior of programs using predicates. One popular notation is the Hoare triple [**27**][2]

$$\{pre\}program\{post\}$$

which asserts that if *program* starts in a state that satisfies *pre* then it terminates in a state that satisfies *post*. We call the predicate that specifies the initial states the *precondition* of the program, and the predicate the specifies the final states the *postcondition* of the program. For example, the assertion

$$\{x > 0\}x := x + 1\{x = x_0 + 1\}$$

says that in any initial state in which x is greater than zero, the program $x := x + 1$ terminates in a state in which x is equal to the old (initial) value of x plus one.

The precondition $x > 0$ describes the set of all states in which x is greater than zero. The postcondition $x = x_0 + 1$ describes the set of all *pairs* of before/after states in which x in the after state is equal to the value of x in the before state plus one. Therefore, this predicate describes not a set of states but a relation between states.

In general, predicates that appear as preconditions describe a set of states that constrain the domain of application of the program, while predicates that appear as postconditions describe relations between states that capture the effect of the program.

We use the term *Constraint* to denote the meaning of preconditions, but because we model programs as relations, we make no distinction between the meaning of a program and the meaning of a postcondition.

$\mathit{Constraint} == \mathbb{P}\,\mathit{State}$

We can now give a meaning to assertions. The assertion $\{pre\}prog\{post\}$ means that given any state s in *pre*, if the program *prog* terminates in a state s' then the pair $s \mapsto s'$ must be in *post*. This means that the set *pre* is a subset of *prog*'s domain, and that the domain restriction of *pre* on *prog* is a subset of *post*[3]

DEFINITION 3.1.1.

$$\{pre\}prog\{post\} \Leftrightarrow pre \subseteq \mathrm{dom}\, prog \wedge pre \lhd prog \subseteq post$$

[2]Our version of Hoare triples differs from the original version in two respects. First, we follow the later trend of surrounding the predicates in curly braces rather than the program. Second, our semantics is total whereas the original semantics is partial.

[3]The meaning of the domain restriction operator, $A \lhd R$, is the relation that consists of all the pairs in R whose first member is in A. See also the discussion on relations in appendix D.

Note that if we consider the program in the definition above as a parameter then the Hoare triple describes a specification. It specifies all the programs that satisfy the right hand side of the equivalence in the definition above.

A weaker precondition means that the program can achieve the postcondition in a larger set of initial states. Therefore, the most general case in which a program achieves a postcondition is given by its weakest precondition [**20**].

3.1.4. Weakest preconditions. The standard way to define weakest preconditions is syntactic: we define substitution rules for each kind of statement in the programming language. However, weakest preconditions also have an important role in our relational model, in particular as we shall see in the next section, they are essential for the description of sequential composition. We define the relational weakest precondition function as follows:

$$
\begin{array}{|l}
wp : Program \rightarrow Program \rightarrow Constraint \\
\hline
\forall\, program, post : Program \bullet \\
\qquad wp\; program\; post = \operatorname{dom} program \setminus \operatorname{dom}(program \cap \overline{post})
\end{array}
$$

where $\overline{p} = (State \times State) \setminus p$

By removing from the domain of *program* all the initial states that appear in pairs that are not in *post*, we ensure that all the remaining states appear only in pairs that are in *post*.

We can illustrate this definition with a state space of just two states (which we call 0 and 1):

$$
\begin{aligned}
wp\; \begin{array}{ccc}0&\text{—}&0\\1&\text{—}&1\end{array}\; \begin{array}{ccc}0&\overline{\diagdown}&0\\1&&1\end{array} &= \operatorname{dom} \begin{array}{ccc}0&\text{—}&0\\1&\text{—}&1\end{array} \setminus \operatorname{dom}\left(\begin{array}{ccc}0&\text{—}&0\\1&\text{—}&1\end{array} \cap \overline{\begin{array}{ccc}0&\overline{\diagdown}&0\\1&&1\end{array}} \right) \\
&= \{0,1\} \setminus \operatorname{dom}\left(\begin{array}{ccc}0&\text{—}&0\\1&\text{—}&1\end{array} \cap \begin{array}{ccc}0&&0\\1&\underline{\diagup}&1\end{array} \right) \\
&= \{0,1\} \setminus \operatorname{dom} \begin{array}{ccc}0&&0\\1&\text{—}&1\end{array} \\
&= \{0,1\} \setminus \{1\} \\
&= \{0\}
\end{aligned}
$$

The following theorem is useful for checking if a particular state is a member of the weakest precondition:

THEOREM 3.1.1. *Let x be a state and let prog and post be programs. Then x is a member of $wp\; prog\; post$ iff*

$$x \in dom\, prog \wedge \forall\, y : State \bullet x \mapsto y \in prog \Rightarrow x \mapsto y \in post$$

PROOF. Let x be a state and let *prog* and *post* be programs. We argue as follows:

$$
\begin{array}{lr}
x \in wp\; prog\; post & \\
\Leftrightarrow x \in \operatorname{dom} prog \setminus \operatorname{dom}(prog \cap \overline{post}) & \text{definition of } wp \\
\Leftrightarrow x \in \operatorname{dom} prog \wedge \neg\, (x \in \operatorname{dom}(prog \cap \overline{post})) & \text{definition of } \setminus \\
\Leftrightarrow x \in \operatorname{dom} prog \wedge \neg\, (\exists\, y : State \bullet x \mapsto y \in prog \cap \overline{post}) & \text{set theory} \\
\Leftrightarrow x \in \operatorname{dom} prog \wedge \neg\, (\exists\, y : State \bullet x \mapsto y \in prog \wedge x \mapsto y \notin post) & \text{set theory} \\
\Leftrightarrow x \in \operatorname{dom} prog \wedge \forall\, y : State \bullet x \mapsto y \notin prog \vee x \mapsto y \in post & \text{de-Morgan} \\
\Leftrightarrow x \in \operatorname{dom} prog \wedge \forall\, y : State \bullet x \mapsto y \in prog \Rightarrow x \mapsto y \in post & \text{predicate logic}
\end{array}
$$

□

The relational weakest precondition function is consistent with the meaning we gave to Hoare assertions:

THEOREM 3.1.2.

$$\{pre\}prog\{post\} \Leftrightarrow pre \subseteq wp\,prog\,post$$

PROOF. Let *pre* be a set of states and let *prog* and *post* be relations between states. We argue as follows:

$pre \subseteq wp\,prog\,post$

$\Leftrightarrow pre \subseteq \mathrm{dom}\,prog \setminus (\mathrm{dom}\,prog \cap \overline{post})$ — definition of *wp*

$\Leftrightarrow pre \subseteq \mathrm{dom}\,prog \wedge pre \cap (\mathrm{dom}\,prog \cap \overline{post}) = \{\}$ — set theory

$\Leftrightarrow pre \subseteq \mathrm{dom}\,prog \wedge (\mathrm{dom}\,pre \lhd (prog \cap \overline{post})) = \{\}$ — set theory

$\Leftrightarrow pre \subseteq \mathrm{dom}\,prog \wedge (\mathrm{dom}(pre \lhd prog) \cap \overline{post}) = \{\}$ — set theory

$\Leftrightarrow pre \subseteq \mathrm{dom}\,prog \wedge (\mathrm{dom}\,pre \lhd prog) \subseteq post$ — lemma D.0.1

$\Leftrightarrow \{pre\}prog\{post\}$ — definition 3.1.1

□

The relational weakest precondition has all the properties of the familiar syntactic version of the weakest precondition function, as the following theorems show:

LEMMA 3.1.3. *Let p be a program, then*

$$wp\,p\,State = \mathrm{dom}\,p$$

PROOF. Let p be a program, we argue as follows:

$wp\,p\,State$

$= (\mathrm{dom}\,p) \setminus (p \rhd \overline{State})$ — definition of *wp*

$= (\mathrm{dom}\,p) \setminus (p \rhd \{\})$ — set theory

$= (\mathrm{dom}\,p) \setminus \{\}$ — set theory

$= (\mathrm{dom}\,p)$ — set theory

□

THEOREM 3.1.4. *The weakest precondition function distributes through conjunction*

$$wp\,r\,(p \cap q) = (wp\,r\,p) \cap (wp\,r\,q)$$

PROOF. Let r, p and q be programs. We argue as follows:

$wp\,r\,(p \cap q)$

$= (\mathrm{dom}\,r) \setminus \mathrm{dom}(r \cap \overline{p \cap q})$ — definition of *wp*

$= (\mathrm{dom}\,r) \setminus \mathrm{dom}(r \cap (\overline{p} \cup \overline{q}))$ — set theory

$= (\mathrm{dom}\,r) \setminus \mathrm{dom}((r \cap \overline{p}) \cup (r \cap \overline{q}))$ — set theory

$= (\mathrm{dom}\,r) \setminus (\mathrm{dom}(r \cap \overline{p}) \cup \mathrm{dom}(r \cap \overline{q}))$ — dom distributes through union

$= ((\mathrm{dom}\,r) \setminus \mathrm{dom}(r \cap \overline{p})) \cap ((\mathrm{dom}\,r) \setminus \mathrm{dom}(r \cap \overline{q}))$ — $A \setminus (B \cup C) = (A \setminus B) \cap (A \setminus C)$

$= (wp\,r\,p) \cap (wp\,r\,q)$ — definition of *wp*

□

THEOREM 3.1.5. *The weakest precondition function is monotonic in its second argument*

$$p \subseteq q \Rightarrow wp\, r\, p \subseteq wp\, r\, q$$

PROOF. Let p, q and r be programs. We argue as follows:

$wp\, r\, p \subseteq wp\, r\, q$	
$\Leftrightarrow \mathrm{dom}\, r \setminus \mathrm{dom}(r \cap \overline{p}) \subseteq \mathrm{dom}\, r \setminus \mathrm{dom}(r \cap \overline{q})$	definition of *wp*
$\Leftarrow \mathrm{dom}(r \cap \overline{q}) \subseteq \mathrm{dom}(r \cap \overline{p})$	set theory
$\Leftarrow r \cap \overline{q} \subseteq r \cap \overline{p}$	set theory
$\Leftrightarrow r \setminus q \subseteq r \setminus p$	$A \cap B = A \setminus \overline{B}$
$\Leftarrow p \subseteq q$	set theory

□

We can use a more simple law to calculate the weakest precondition of a functional program:

THEOREM 3.1.6. *For every functional program prog and postcondition post*

$$wp\, prog\, post = dom(prog \cap post)$$

PROOF. Let *prog* and *post* be programs, and assume that *prog* is a function. Let x be an arbitrary member of $wp\, prog\, post$. We argue as follows:

$x \in wp\, prog\, post$	
$\Leftrightarrow x \in \mathrm{dom}\, prog \wedge \forall y : State \bullet x \mapsto y \in prog \Rightarrow x \mapsto y \in post$	theorem 3.1.1
$\Leftrightarrow x \in \mathrm{dom}\, prog \wedge x \mapsto prog\, x \in post$	*prog* is a function
$\Leftrightarrow x \mapsto prog\, x \in prog \wedge x \mapsto prog\, x \in post$	set theory
$\Leftrightarrow x \mapsto prog\, x \in prog \cap post$	
$\Rightarrow x \in \mathrm{dom}(prog \cap post)$	

This argument shows that $wp\, prog\, post$ is a subset of $\mathrm{dom}(prog \cap post)$.

Assume now that x is an arbitrary member of $\mathrm{dom}(prog \cap post)$. We argue as follows:

$x \in \mathrm{dom}(prog \cap post)$	
$\Leftrightarrow \exists y : State \bullet x \mapsto y \in prog \cap post$	definition of dom
$\Leftrightarrow \exists y : State \bullet x \mapsto y \in prog \wedge x \mapsto y \in post$	set theory
$\Leftrightarrow \exists y : State \bullet x \mapsto y \in prog \wedge x \mapsto y \in post \wedge$	
$\quad \forall z : State \mid x \mapsto z \in prog \bullet (z = y)$	*prog* is a function
$\Rightarrow \exists y : State \bullet x \mapsto y \in prog \wedge x \mapsto y \in post \wedge$	
$\quad \forall z : State \mid x \mapsto z \in prog \bullet (x \mapsto z \in post)$	since $z = y$ and $x \mapsto y \in post$
$\Leftrightarrow x \in \mathrm{dom}\, prog \wedge \forall z : State \mid x \mapsto z \in prog \bullet (x \mapsto z \in post)$	definition of dom
$\Leftrightarrow x \in wp\, prog\, post$	theorem 3.1.1

This argument shows that $\mathrm{dom}(prog \cap post)$ is a subset of $wp\, prog\, post$, and concludes the proof. □

3.1.5. Sequential composition. Informally, the program $p_1 \mathbin{;} p_2$ operates by first executing p_1 and then executing p_2. This means that every final state of p_1 must be an initial state of p_2.

It is tempting to define the sequential composition of programs using the relational composition operator. Unfortunately this does not work. Consider the following example:

0—0 ⨾ 0—0 = 0—0
1 ╲ 1 ⨾ 1 1 = 1 1

The first program may end in a state that lies outside the domain of the second program. This may cause the second program to start in an invalid state, resulting in undefined behavior. But as we can see in the diagram above, this possibility is not described by the relational composition.

This model fails to describe how programs behave in practice. The standard way to fix it is to totalize the relations by adding an explicit undefined state, $\perp$, and connecting every initial state that is not in the domain of the program to the undefined state [**16, 61**]. As a result the relational composition maps the top state to the undefined state $\perp$, correctly reflecting the behavior of the program:

0—0 ⨾ 0—0 = 0—0
1 1 ⨾ 1 1 = 1 1
⊥—⊥ ⨾ ⊥—⊥ = ⊥—⊥

It is, however, possible to avoid the need for an explicit undefined state by changing the way we model the composition operator. Our version of sequential composition uses the weakest precondition function to exclude the initial states of the first program that may escape from the domain of the second program:

$$_ \mathbin{;} _ : \mathit{Program} \times \mathit{Program} \rightarrow \mathit{Program}$$

$$\forall p_1, p_2 : \mathit{Program} \bullet$$
$$p_1 \mathbin{;} p_2 = (\mathit{wp}\, p_1\, (\mathit{lift}(\operatorname{dom} p_2)) \lhd p_1) \mathbin{⨾} p_2$$

where the function *lift* transforms the domain of the second program into the final states of a relation by associating every initial state to a state in the domain:

$$\mathit{lift}\; c = \mathit{State} \times c$$

If we revisit the example using this version of sequential composition, we get a description that is consistent with the way we expect the programs to behave:

$$
\begin{aligned}
\begin{smallmatrix}0&—&0\\&\diagdown&\\1&&1\end{smallmatrix} \; ; \; \begin{smallmatrix}0&—&0\\&&\\1&&1\end{smallmatrix} &= \left(wp \left(\begin{smallmatrix}0&—&0\\&\diagdown&\\1&&1\end{smallmatrix} \right) \mathit{lift} \left(\operatorname{dom} \begin{smallmatrix}0&—&0\\&&\\1&&1\end{smallmatrix} \right) \right) \lhd \left(\begin{smallmatrix}0&—&0\\&\diagdown&\\1&&1\end{smallmatrix} \; ⨾ \; \begin{smallmatrix}0&—&0\\&&\\1&&1\end{smallmatrix} \right) \\
&= \left(wp \left(\begin{smallmatrix}0&—&0\\&\diagdown&\\1&&1\end{smallmatrix} \right) \mathit{lift} \left(\operatorname{dom} \begin{smallmatrix}0&—&0\\&&\\1&&1\end{smallmatrix} \right) \right) \lhd \begin{smallmatrix}0&—&0\\&&\\1&&1\end{smallmatrix} \\
&= \left(wp \; \begin{smallmatrix}0&—&0\\&\diagdown&\\1&&1\end{smallmatrix} \; \begin{smallmatrix}0&—&0\\&\diagup&\\1&&1\end{smallmatrix} \right) \lhd \begin{smallmatrix}0&—&0\\&&\\1&&1\end{smallmatrix} \\
&= \left(\left(\operatorname{dom} \begin{smallmatrix}0&—&0\\&\diagdown&\\1&&1\end{smallmatrix} \right) \setminus \operatorname{dom} \left(\begin{smallmatrix}0&—&0\\&\diagdown&\\1&&1\end{smallmatrix} \cap \overline{\begin{smallmatrix}0&—&0\\&\diagup&\\1&&1\end{smallmatrix}} \right) \right) \lhd \begin{smallmatrix}0&—&0\\&&\\1&&1\end{smallmatrix} \\
&= \left(\{0\} \setminus \operatorname{dom} \left(\begin{smallmatrix}0&—&0\\&\diagdown&\\1&&1\end{smallmatrix} \cap \begin{smallmatrix}0&&0\\&\diagdown&\\1&—&1\end{smallmatrix} \right) \right) \lhd \begin{smallmatrix}0&—&0\\&&\\1&&1\end{smallmatrix} \\
&= \left(\{0\} \setminus \operatorname{dom} \begin{smallmatrix}0&&0\\&\diagdown&\\1&&1\end{smallmatrix} \right) \lhd \begin{smallmatrix}0&—&0\\&&\\1&&1\end{smallmatrix} \\
&= (\{0\} \setminus \{0\}) \lhd \begin{smallmatrix}0&—&0\\&&\\1&&1\end{smallmatrix} \\
&= \{\}
\end{aligned}
$$

The weakest precondition of the first program on the domain of the second program is the empty set, so the result of the composition is the empty relation. In other words, there is no initial state from which the composed program is certain to terminate.

3.1.6. The weakest precondition of sequential composition. In the standard definition of weakest preconditions, the postcondition is a predicate on the after state of the system. That is, it may not refer to old variables. A relation that corresponds to such a predicate is a relation that does not restrict its domain. That is, it connects every possible initial state to the final states of the postcondition. Therefore, the relational equivalent of the constraint p is the relation $\mathit{State} \times p$.

In order to simplify the analysis of such postconditions, we overload the weakest precondition function with a version that takes a constraint as a postcondition:

$$
\begin{array}{|l}
wp : \mathit{Program} \rightarrow \mathit{Constraint} \rightarrow \mathit{Constraint} \\
\hline
\forall \mathit{program} : \mathit{Program};\ \mathit{post} : \mathit{Constraint} \bullet \\
\qquad wp\ \mathit{program}\ \mathit{post} = \operatorname{dom} \mathit{program} \setminus \operatorname{dom}(\mathit{program} \rhd \overline{\mathit{post}})
\end{array}
$$

where $\overline{p} = \mathit{State} \setminus p$

This special case of the weakest precondition function enjoys the following useful law:

THEOREM 3.1.7. *Let p_1 and p_2 be two programs, and let q be a constraint, then*

$$wp\,(p_1\,;\,p_2)\,q = wp\,p_1\,(wp\,p_2\,q)$$

To see why this law fails when the postcondition refers to initial variables, assume that we have a state space of just one binary variable x, and let prog be the following program:

$$x = 0 \rightarrow x := 1 \;\square\; x = 1 \rightarrow x := 0$$

now consider the weakest precondition of the program $\mathit{prog};\ \mathit{prog}$ on the postcondition $x = x_0 \wedge x = 0$:

$$
\begin{aligned}
& wp\,(\mathit{prog}\,;\mathit{prog})\,(x = x_0 \wedge x = 0) \\
& = wp\,(x = 0 \rightarrow x := 0 \;\square\; x = 1 \rightarrow x := 1)\,(x = x_0 \wedge x = 0)
\end{aligned}
$$

$= x = 0 \wedge wp(x := 0, (x = x_0 \wedge x = 0)) \vee x = 1 \wedge wp(x := 1, (x = x_0 \wedge x = 0))$
$= x = 0 \wedge 0 = x_0 \wedge x = 0 \vee x = 1 \wedge 1 = x_0 \wedge x = 0$
$= x = 0 \vee \mathit{false}$
$= x = 0$

However, if we use the composition law we get:

$wp\ (prog\ ; prog)\ (x = x_0 \wedge x = 0)$
$= wp\, prog\ (wp\, prog, x = x_0 \wedge x = 0)$
$= wp\, prog\ (wp\ (x = 0 \rightarrow x := 1 \,\square\, x = 1 \rightarrow x := 0, x = x_0 \wedge x = 0))$
$= wp\, prog\ (x = 0 \wedge wp(x := 1, x = x_0 \wedge x = 0) \vee$
$\qquad x = 1 \wedge wp(x := 0, x = x_0 \wedge x = 0))$
$= wp\, prog\ (x = 0 \wedge 1 = x_0 \wedge 1 = 0 \vee x = 1 \wedge 0 = x_0 \wedge 0 = 0)$
$= wp\, prog\ (\mathit{false} \vee \mathit{false})$
$= \mathit{false}$

The correct law, taking account of postconditions that refer to old variables, is the following:

THEOREM 3.1.8. *For every two programs p_1 and p_2 and for every postcondition q*

$$wp\ (p_1\ ; p_2)\ q = wp\, p_1\ ((wp\, p_2\, q[x_0 := t])[t := x_0])$$

where x_0 refers to the sequence of old variables that appear in q, t is a sequence of fresh variables and $q[n := e]$ is the substitution of the variable n by the expression e.

Using this theorem we may derive the correct result for the example above:

$wp\ (prog\ ; prog)\ (x = x_0 \wedge x = 0)$
$= wp\, prog\ (wp\, prog, (x = x_0 \wedge x = 0)[x_0 := t])[t := x_0]$
$= wp\, prog\ (wp\, prog, (x = t \wedge x = 0))[t := x_0]$
$= wp\, prog\ (wp\ (x = 0 \rightarrow x := 1 \,\square\, x = 1 \rightarrow x := 0, x = t \wedge x = 0))[t := x_0]$
$= wp\, prog\ (x = 0 \wedge wp(x := 1, x = t \wedge x = 0) \vee$
$\qquad x = 1 \wedge wp(x := 0, x = t \wedge x = 0))[t := x_0]$
$= wp\, prog\ (x = 0 \wedge 1 = t \wedge 1 = 0 \vee x = 1 \wedge 0 = t \wedge 0 = 0)[t := x_0]$
$= wp\, prog\ (\mathit{false} \vee t = 0)[t := x_0]$
$= wp\, prog\ (x_0 = 0)$
$= wp\ (x = 0 \rightarrow x := 1 \,\square\, x = 1 \rightarrow x := 0, x_0 = 0)$
$= x = 0 \wedge wp(x := 1, x_0 = 0) \vee x = 1 \wedge wp(x := 0, x_0 = 0)$
$= x = 0 \wedge x_0 = 0 \vee x = 1 \wedge x_0 = 0$
$= x = 0$

Note that when the postcondition q does not refer to any old variables, theorem 3.1.8 reduces to the familiar law we have described earlier in theorem 3.1.7.

We now prove theorem 3.1.7:

PROOF. Let p_1 and p_2 be two programs, and let q be a constraint. We argue as follows:

$wp\, p_1\ (wp\, p_2\, q)$
$= (wp\, p_1\ (\mathrm{dom}\, p_2)) \cap wp\, p_1\, \overline{\mathrm{dom}\, p_2 \rhd \overline{q}}$ — lemma 3.1.9
$= (wp\, p_1\ (\mathrm{dom}\, p_2)) \cap (\mathrm{dom}\, p_1 \setminus \mathrm{dom}(p_1 \rhd \mathrm{dom}\, p_2 \rhd \overline{q}))$ — definition of *wp*

$= (wp\, p_1\, (\mathrm{dom}\, p_2)) \setminus \mathrm{dom}(p_1 \rhd \mathrm{dom}\, p_2 \rhd \overline{q})$ — $wp\, p_1\, (\mathrm{dom}\, p_2) \subseteq \mathrm{dom}\, p_1$

$= (wp\, p_1\, (\mathrm{dom}\, p_2)) \setminus \mathrm{dom}(p_1 \fatsemi (p_2 \rhd \overline{q}))$ — lemma D.0.2

$= (wp\, p_1\, (\mathrm{dom}\, p_2)) \setminus \mathrm{dom}((p_1 \fatsemi p_2) \rhd \overline{q}))$ — set theory

$= wp\, (p_1\, ;\, p_2)\, q$ — lemma 3.1.10

□

LEMMA 3.1.9. *Let p_1 and p_2 be two programs and let q be a constraint. Then,*

$$wp\, p_1\, (wp\, p_2\, q) = (wp\, p_1\, (\mathrm{dom}\, p_2)) \cap (wp\, p_1\, \overline{\mathrm{dom}(p_2 \rhd \overline{q})}$$

PROOF. Let p_1 and p_2 be two programs and let q be a constraint. We argue as follows:

$wp\, p_1\, (wp\, p_2\, q)$

$= (\mathrm{dom}\, p_1) \setminus \mathrm{dom}(p_1 \rhd \overline{wp\, p_2\, q})$ — definition of wp

$= (\mathrm{dom}\, p_1) \setminus \mathrm{dom}(p_1 \rhd \overline{\mathrm{dom}\, p_2 \setminus \mathrm{dom}(p_2 \rhd \overline{q})}))$ — definition of wp

$= (\mathrm{dom}\, p_1) \setminus \mathrm{dom}(p_1 \rhd \overline{(\mathrm{dom}\, p_2 \cap \overline{\overline{\mathrm{dom}(p_2 \rhd \overline{q})}}}))$ — $A \setminus \overline{B} = A \cap B$

$= (\mathrm{dom}\, p_1) \setminus \mathrm{dom}(p_1 \rhd (\overline{\mathrm{dom}\, p_2} \cup \mathrm{dom}(p_2 \rhd \overline{q})))$ — de-Morgan

$= (\mathrm{dom}\, p_1) \setminus \mathrm{dom}(p_1 \rhd \overline{\mathrm{dom}\, p_2} \cup p_1 \rhd \mathrm{dom}(p_2 \rhd \overline{q}))$ — set theory

$= (\mathrm{dom}\, p_1) \setminus (\mathrm{dom}(p_1 \rhd \overline{\mathrm{dom}\, p_2}) \cup \mathrm{dom}(p_1 \rhd \mathrm{dom}(p_2 \rhd \overline{q})))$ — $\mathrm{dom}(A \cup B) = \mathrm{dom}\, A \cup \mathrm{dom}\, B$

$= ((\mathrm{dom}\, p_1) \setminus \mathrm{dom}(p_1 \rhd \overline{\mathrm{dom}\, p_2})) \cap ((\mathrm{dom}\, p_1) \setminus \mathrm{dom}(p_1 \rhd \mathrm{dom}(p_2 \rhd \overline{q})))$
— $A \setminus (B \cup C) = (A \setminus B) \cap (A \setminus C)$

$= (wp\, p_1\, (\mathrm{dom}\, p_2)) \cap (wp\, p_1\, \overline{\mathrm{dom}\, p_2 \rhd \overline{q}})$ — definition of wp

□

LEMMA 3.1.10. *Let p_1 and p_2 be programs and let q be a constraint. Then,*

$$wp\, (p_1\, ;\, p_2)\, q = wp\, p_1\, (\mathrm{dom}\, p_2) \setminus \mathrm{dom}(p_1 \fatsemi (p_2 \rhd \overline{q}))$$

PROOF. Let p_1 and p_2 be programs and let q be a constraint, we argue as follows:

$wp\, (p_1\, ;\, p_2)\, q$

$= \mathrm{dom}(p_1\, ;\, p_2) \setminus \mathrm{dom}((p_1\, ;\, p_2) \rhd \overline{q})$ — definition of wp

$= \mathrm{dom}((wp\, p_1\ \mathrm{dom}\, p_2) \lhd p_1 \fatsemi p_2) \setminus \mathrm{dom}((wp\, p_1\ \mathrm{dom}\, p_2) \lhd (p_1 \fatsemi p_2 \rhd \overline{q}))$ — definition of ;

$= (wp\, p_1\ \mathrm{dom}\, p_2) \cap \mathrm{dom}(p_1 \fatsemi p_2) \setminus ((wp\, p_1\ \mathrm{dom}\, p_2) \cap \mathrm{dom}(p_1 \fatsemi p_2 \rhd \overline{q}))$ — set theory

$= (wp\, p_1\ \mathrm{dom}\, p_2) \setminus ((wp\, p_1\ \mathrm{dom}\, p_2) \cap \mathrm{dom}(p_1 \fatsemi p_2 \rhd \overline{q}))$ — lemma 3.1.11

$= (wp\, p_1\ \mathrm{dom}\, p_2) \setminus \mathrm{dom}(p_1 \fatsemi p_2 \rhd \overline{q})$ — $A \setminus (A \cap B) = A \setminus B$

□

LEMMA 3.1.11. *Let p_1 and p_2 be programs. Then,*

$$wp\, p_1\ \mathrm{dom}\, p_2 \subseteq \mathrm{dom}(p_1 \fatsemi p_2)$$

PROOF. Let p_1 and p_2 be programs, and let x be a state that is a member of $wp\, p_1 \;\; \mathrm{dom}\, p_2$. We argue as follows:

$x \in wp\, p_1 \;\; \mathrm{dom}\, p_2$

$\Leftrightarrow x \in \mathrm{dom}\, p_1 \wedge \forall y : State \bullet x \mapsto y \in p_1 \Rightarrow y \in \mathrm{dom}\, p_2$ — theorem 3.1.1

$\Rightarrow \exists y : State \bullet x \mapsto y \in p_1 \wedge y \in \mathrm{dom}\, p_2$ — predicate logic

$\Leftrightarrow \exists y : State \bullet x \mapsto y \in p_1 \wedge \exists z : State \bullet y \mapsto z \in p_2$ — predicate logic

$\Leftrightarrow \exists z : State \bullet \exists y : State \bullet x \mapsto y \in p_1 \wedge y \mapsto z \in p_2$ — predicate logic

$\Leftrightarrow \exists z : State \bullet x \mapsto z \in p_1 \fatsemi p_2$ — definition of $\fatsemi$

$\Leftrightarrow x \in \mathrm{dom}(p_1 \fatsemi p_2)$ — definition of dom

□

Finally, we show that the overloaded definition of *wp* for constraint postconditions is indeed a special case of the general *wp* function:

THEOREM 3.1.12. *Let p be a program and q be a constraint. Then,*

$$wp\, p\, q = wp\, p\, (lift\, q)$$

PROOF. Let p be a program and let q be a constraint. We argue as follows:

$wp\, p\, (lift\, q)$

$= \mathrm{dom}\ p \setminus \mathrm{dom}(p \cap \overline{lift\, q})$ — definition of *wp*

$= \mathrm{dom}\ p \setminus \mathrm{dom}(p \cap (lift\, \overline{q}))$ — lemma 3.1.13

$= \mathrm{dom}\ p \setminus \mathrm{dom}(p \rhd \overline{p})$ — lemma 3.1.14

$= wp\, p\, q$ — definition of *wp*

□

LEMMA 3.1.13. *Let q be a constraint, then*

$$\overline{lift\, q} = lift\, \overline{q}$$

PROOF. Let q be a constraint, and let the pair of states $x \mapsto y$ be an arbitrary member of the relation $\overline{lift\, q}$. we argue as follows:

$x \mapsto y \in \overline{lift\, q}$

$\Leftrightarrow x \mapsto y \notin lift\, q$ — definition of $\overline{\square}$

$\Leftrightarrow x \mapsto y \notin State \times q$ — definition of *lift*

$\Leftrightarrow x \notin State \vee y \notin q$ — set theory

$\Leftrightarrow y \notin q$ — set theory

$\Leftrightarrow y \in \overline{q}$ — definition of $\overline{\square}$

$\Leftrightarrow x \in State \wedge y \in \overline{q}$ — every state is a member of *State*

$\Leftrightarrow x \mapsto y \in lift\, \overline{q}$ — definition of *lift*

Since $x \mapsto y$ is an arbitrary pair, we have shown that the two relations are equal. □

LEMMA 3.1.14. *Let p be a relation and q be a constraint. Then,*

$$p \cap lift\, q = p \rhd q$$

PROOF. Let p be a relation and q be a constraint, and let $x \mapsto y$ be an arbitrary member of $p \cap lift\, q$. We argue as follows:

$x \mapsto y \in p \cap lift\, q$

$\Leftrightarrow x \mapsto y \in p \wedge x \mapsto y \in lift\, q$ — set theory

$\Leftrightarrow x \mapsto y \in p \wedge x \mapsto y \in State \times q$ — definition of *lift*

$\Leftrightarrow x \mapsto y \in p \wedge x \in State \wedge y \in q$ — set theory

$\Leftrightarrow x \mapsto y \in p \wedge y \in q$ — set theory

$\Leftrightarrow x \mapsto y \in p \rhd q$ — definition of $\rhd$

Since $x \mapsto y$ is an arbitrary pair, we have shown that the two relations are equal. □

We will find the following property useful later in the thesis:

LEMMA 3.1.15. *Let p and q be two programs, then*

$$\mathrm{dom}(p\,;q) = wp\, p\, (\mathrm{dom}\, q)$$

PROOF. Let p and q be two programs, we argue as follows:

$\mathrm{dom}(p\,;q)$

$= wp\,(p\,;q)\, State$ — lemma 3.1.3

$= wp\, p\,(wp\, q\, State)$ — theorem 3.1.7

$= wp\, p\,(\mathrm{dom}\ q)$ — lemma 3.1.3

□

3.1.7. Refinement. We say that a concrete program *prog* refines a more abstract program *spec*, written $prog \sqsupseteq spec$, when *prog* behaves according to *spec*. That is, for every initial state of *spec* the program *prog* terminates, and it terminates in a state that is a final state of *spec*:

$$_ \sqsupseteq _ : Program \leftrightarrow Program$$

$$\forall\, spec, prog : Program \bullet$$
$$prog \sqsupseteq spec \Leftrightarrow (\mathrm{dom}\, spec) \lhd prog \subseteq spec \wedge \mathrm{dom}\, spec \subseteq \mathrm{dom}\, prog$$

A different way to define refinement is by using the weakest precondition function. According to this definition, the program *prog* refines the specification *spec* if given any postcondition *post*, every state in *wp spec post* is also a state in *wp prog post*. The following theorem shows that these definitions are equivalent:

THEOREM 3.1.16.

$\forall\, r, s : Program \bullet r \sqsupseteq s \Leftrightarrow \forall\, p : Program \bullet (wp\, s\, p) \subseteq (wp\, r\, p)$

PROOF. We prove the theorem by proving separately each implication in the equivalence. First we assume that $r \sqsupseteq s$. This means that $\mathrm{dom}\, s \lhd r \subseteq s$ and that $\mathrm{dom}\, s \subseteq \mathrm{dom}\, r$. Let p be an arbitrary postcondition, and let x be a member of $wp\, s\, p$. We argue as follows:

$x \in wp\, s\, p$

$\Leftrightarrow x \in \mathrm{dom}\ s \wedge x \notin \mathrm{dom}(s \cap \overline{p})$ — definition of *wp*

$\Leftrightarrow x \in \mathrm{dom}\ s \wedge \forall\, y : State \bullet x \mapsto y \in s \Rightarrow x \mapsto y \in p$ — set theory

From the first conjunct and the assumption $\mathrm{dom}\, s \subseteq \mathrm{dom}\, r$ we get that $x \in \mathrm{dom}\, r$. Therefore in order to show that x is a member of $wp\, r\, p$ we only have to show that $x \notin \mathrm{dom}(r \cap \overline{p})$. Assume that this is not the

case. This means that there exists a state y such that $x \mapsto y$ is a member of r but $x \mapsto y$ is not a member of p. We can now argue as follows:

$x \mapsto y \in r$

$\Rightarrow x \mapsto y \in \operatorname{dom} s \lhd r$ $\qquad x \in \operatorname{dom} s$

$\Rightarrow x \mapsto y \in s$ $\qquad$ using the assumption $\operatorname{dom} s \lhd r \subseteq s$

$\Rightarrow x \mapsto y \in p$ $\qquad$ since x is a member of $wp\, s\, p$.

This contradicts the assumption that $x \mapsto y \notin p$. Therefore it must be that for every state y, if $x \mapsto y$ is a member of r then $x \mapsto y$ is a member of p. Since we have already shown that $x \in \operatorname{dom} r$ this proves that x is a member of $wp\, r\, p$, which concludes the first part of the proof.

Now let us assume that $\forall p : Program \bullet wp\, s\, p \subseteq wp\, r\, p$. In particular, for $p = State \times State$ we may argue as follows:

$wp\, s\, (State \times State) \subseteq wp\, r\, (State \times State)$

$\Leftrightarrow \operatorname{dom} s \setminus \operatorname{dom}(s \cap \overline{State \times State}) \subseteq$

$\quad \operatorname{dom} r \setminus \operatorname{dom}(r \cap \overline{State \times State})$ $\qquad$ definition of wp

$\Leftrightarrow \operatorname{dom} s \setminus \operatorname{dom}(s \cap \{\}) \subseteq \operatorname{dom} r \setminus \operatorname{dom}(r \cap \{\})$ $\qquad$ definition of $\overline{State \times State}$

$\Leftrightarrow \operatorname{dom} s \subseteq \operatorname{dom} r$ $\qquad$ set theory

Now all we have to show is that $\operatorname{dom} s \lhd r \subseteq s$. Assume that this is not the case. That is, there is a pair $x \mapsto y$ that is a member of $\operatorname{dom} s \lhd r$ but is not a member of s. Let p be the postcondition $State \times State \setminus \{x \mapsto y\}$. We will simplify both ends of the predicate

$$wp\, s\, p \subseteq wp\, r\, p$$

and will see that we get a contradiction. First let us simplify the left hand side:

$wp\, s\, p$

$= \operatorname{dom} s \setminus \operatorname{dom}(s \cap \overline{State \times State \setminus \{x \mapsto y\}}$ $\qquad$ definition of p

$= \operatorname{dom} s \setminus \operatorname{dom}(s \cap \{x \mapsto y\}$ $\qquad$ set theory

$= \operatorname{dom} s \setminus \{\}$ $\qquad$ assumption

$= \operatorname{dom} s$

Now we simplify the right hand side:

$wp\, r\, p$

$= \operatorname{dom} r \setminus \operatorname{dom}(r \cap \overline{State \times State \setminus \{x \mapsto y\}}$ $\qquad$ definition of p

$= \operatorname{dom} r \setminus \operatorname{dom}(r \cap \{x \mapsto y\}$ $\qquad$ set theory

$= \operatorname{dom} r \setminus \{x\}$ $\qquad$ assumption

We see that x is a member of the left hand side but not of the right hand side, which contradicts our initial assumption. Therefore every pair of states $x \mapsto y$ in the relation $\operatorname{dom} s \lhd r$ is also in the relation s. In other words, $\operatorname{dom} s \lhd r \subseteq s$. This proves the other direction in the equivalence and therefore completes the proof. □

Refinement is monotonic with respect to sequential composition:

THEOREM 3.1.17. *Let p, q, a and b be programs. Then,*

$$p \sqsupseteq a \wedge q \sqsupseteq b \Rightarrow p\, ;\, q \sqsupseteq a\, ;\, b$$

PROOF. Let p, q, a and b be programs such that $p \sqsupseteq a$ and $q \sqsupseteq b$. We argue as follows:

$p \,;\, q \sqsupseteq a \,;\, b$

$\Leftrightarrow \mathrm{dom}(a \,;\, b) \subseteq \mathrm{dom}(p \,;\, q) \wedge \mathrm{dom}(a \,;\, b) \lhd (p \,;\, q) \subseteq a \,;\, b$ — definition of refinement

$\Leftrightarrow wp\, a\, (\mathrm{dom}\, b) \subseteq wp\, p\, (\mathrm{dom}\, q) \wedge wp\, a\, (\mathrm{dom}\, b) \lhd p \,;\, q \subseteq a \,;\, b$ — lemma 3.1.15

To prove the first conjunct, let x be a state. We argue as follows:

$x \in wp\, a\, (\mathrm{dom}\, b)$

$\Leftrightarrow x \in \mathrm{dom}\, a \wedge \forall y : State \bullet x \mapsto y \in a \Rightarrow y \in \mathrm{dom}\, b$ — theorem 3.1.1

$\Rightarrow x \in \mathrm{dom}\, p \wedge \forall y : State \bullet x \mapsto y \in a \Rightarrow y \in \mathrm{dom}\, q$ — because $\mathrm{dom}\, a \subseteq \mathrm{dom}\, p$ and $\mathrm{dom}\, b \subseteq \mathrm{dom}\, q$

$\Rightarrow x \in \mathrm{dom}\, p \wedge \forall y : State \bullet x \mapsto y \in p \Rightarrow y \in \mathrm{dom}\, q$ — because $\mathrm{dom}\, a \lhd p \subseteq a$

To prove the second conjunct, let x and z be two states. We argue as follows:

$x \mapsto z \in wp\, a\ \mathrm{dom}\, b \lhd p \fatsemi q$

$\Leftrightarrow x \in wp\, a\ \mathrm{dom}\, b \wedge x \mapsto z \in p \fatsemi q$ — definition of $\lhd$

$\Leftrightarrow \exists y : State \bullet x \mapsto wp\, a\ \mathrm{dom}\, b \wedge x \mapsto y \in p \wedge y \mapsto z \in q$ — definition of $\fatsemi$

$\Rightarrow \exists y : State \bullet x \in wp\, a\ \mathrm{dom}\, b \wedge x \mapsto y \in a \wedge y \mapsto z \in q$ — since $\mathrm{dom}\, a \lhd p \subseteq a$

$\Rightarrow \exists y : State \bullet x \in wp\, a\ \mathrm{dom}\, b \wedge x \mapsto y \in a \wedge y \mapsto z \in b$ — from lemma 3.1.18 and since $\mathrm{dom}\, b \lhd q \subseteq b$

$\Leftrightarrow x \in wp\, a\ \mathrm{dom}\, b \wedge x \mapsto z \in a \fatsemi b$ — definition of $\fatsemi$

$\Leftrightarrow x \mapsto z \in wp\, a\ \mathrm{dom}\, b \lhd a \fatsemi b$

□

LEMMA 3.1.18. *Let p be a program, s be a constraint and x and y states. Then,*

$$x \in wp\, p\, s \wedge x \mapsto y \in p \Rightarrow y \in s$$

PROOF. Let p be a program, s be a constraint and x and y states such that $x \in wp\, p\, s$ and $x \mapsto y \in p$. We argue as follows:

$y \in s$

$\Leftarrow x \mapsto y \in p \wedge \forall z : State \bullet x \mapsto z \in p \Rightarrow z \in s$ — take $z = y$

$\Leftrightarrow x \in wp\, p\, s$ — theorem 3.1.1

$\Leftarrow true$ — assumption

□

3.2. Specification statements

A specification statement consists of a precondition, a change list and a postcondition. It describes all the programs that when started from the precondition can establish the postcondition by changing only variables in the change list. For example the specification statement

$$(x \geq 0 \mid y \mid y * y = x_0)$$

describes all the programs that set y (and only y) to the square root of x provided that x is not negative.

As is the case with Hoare assertions, the precondition of a specification statement describes an initial set of states and the postcondition describes a relation between before and after states.

Because the discussion in this chapter focuses on a relational model we will use from now on the notation $(pre \mid post)$ to mean the constraint and relation that are described by *pre* and *post* and not their syntactical representation.

Let us first consider a specification statement that does not restrict the variables at all, that is, its change list includes all the variables in the system. In this case the specification statement describes all the programs that refine the postcondition when they are restricted to the initial states defined by the precondition. In other words it describes all the programs that refine the relation $pre \lhd post$:

DEFINITION 3.2.1.

$$(pre \mid post) = pre \lhd post$$

A more selective change list restricts the postcondition by insisting that the only variables that can change are the ones that appear in the change list, and the others remain the same. In other words, if we remove from every $(before, after)$ pair in the postcondition all the variables that appear in the change list, then *before* must be equal to *after*[4]:

$$\begin{array}{|l}
restrict : ChangeList \rightarrow Program \rightarrow Program \\
\hline
\forall\, cl : ChangeList;\ prog : Program \bullet \\
\quad restrict\ cl\ prog = \{s, s' : State \mid s \mapsto s' \in prog \land cl \mathbin{⩤} s = cl \mathbin{⩤} s'\}
\end{array}$$

Using *restrict* we can define the meaning of a specification statement with a change list, in terms of a specification statement without a change list:

DEFINITION 3.2.2.

$$(pre \mid cl \mid post) = (pre \mid restrict\ cl\ post)$$

3.2.1. Sequential composition of specification statements. Since specification statements are relations, we can compose them in sequence. The following theorem shows that the result is also a specification statement:

THEOREM 3.2.1.

$$(pre_1 \mid post_1)\ ;\ (pre_2 \mid post_2) = (pre_1 \cap wp\ post_1\ pre_2 \mid post_1\ ;\ post_2)$$

PROOF. We show that the weakest precondition function of both sides of the equation is identical and therefore they are equivalent. Let p be a predicate. We argue as follows:

$$\begin{array}{lr}
wp((pre_1 \mid post_1)\ ;\ (pre_2 \mid post_2))\ p & \\
= wp(pre_1 \mid post_1)(wp(pre_2 \mid post_2)\ p) & \text{theorem 3.1.7} \\
= wp(pre_1 \lhd post_1)(wp(pre_2 \lhd post_2) p) & \text{definition 3.2.1} \\
= pre_1 \cap wp\ post_1\ (pre_2 \cap wp\ post_2\ p) & \text{lemma 3.2.2} \\
= pre_1 \cap wp\ post_1\ pre_2 \cap wp\ post_1\ (wp\ post_2\ p) & \text{theorem 3.1.4} \\
= pre_1 \cap wp\ post_1\ pre_2 \cap wp(post_1\ ;\ post_2)\ p & \text{theorem 3.1.7} \\
= wp(pre_1 \cap wp\ post_1\ pre_2 \mid post_1\ ;\ post_2)\ p & \text{theorem 3.2.2}
\end{array}$$

□

[4]The meaning of the domain subtraction operator $A \mathbin{⩤} R$ is the relation that consists of all the pairs in R whose first member is not in A.

LEMMA 3.2.2.

$$wp(q \lhd r)\,p = q \cap (wp\,r\,p)$$

PROOF. Let q be a precondition, p be a postcondition and r be a program. We argue as follows:

$wp(q \lhd r)\,p$

$= \mathrm{dom}(q \lhd r) \setminus \mathrm{dom}((q \lhd r) \rhd \overline{p})$ — definition of wp

$= (q \cap \mathrm{dom}\,r) \setminus (q \cap \mathrm{dom}(r \rhd \overline{p}))$ — $\mathrm{dom}(A \lhd R) = A \cap (\mathrm{dom}\,R)$

$= ((q \cap \mathrm{dom}\,r) \setminus q) \cup ((q \cap \mathrm{dom}\,r) \setminus \mathrm{dom}(r \rhd \overline{p}))$ — set theory

$= \{\} \cup ((q \cap \mathrm{dom}\,r) \setminus \mathrm{dom}(r \rhd \overline{p}))$ — set thoery

$= q \cap ((\mathrm{dom}\,r) \setminus \mathrm{dom}(r \rhd \overline{p}))$ — $(A \cap B) \setminus C = A \cap (B \setminus C)$

$= q \cap (wp\,r\,p)$ — definition of wp

□

3.2.2. Sequential composition of statements with change lists. We have just seen that specification statements are closed under sequential composition. However the definition of the composition operator was given in terms of Z style specification statements, that is, without any reference to change lists. To find a similar equation for the composition of specifications with change lists, we can plug definition 3.2.2 into theorem 3.2.1 and get:

THEOREM 3.2.3.

$$(pre_1 \mid cl_1 \mid post_1)\,;\,(pre_2 \mid cl_2 \mid post_2) = \\ (pre_1 \cap wp\,(restrict\,cl_1\,post_1)\,pre_2 \\ \mid cl_1 \cup cl_2 \mid (restrict\,cl_1\,post_1)\,;\,(restrict\,cl_2\,post_2))$$

The precondition ensures that the first program (while changing only the variables in cl_1) will end in a state that satisfies the precondition of the second program. The change list reflects the fact that the composition can change the variables of both programs. The postcondition insists that the effect of the composed program is achieved by the sequential composition of the individual postconditions, properly restricted to their change lists.

PROOF.

$(pre_1 \mid cl_1 \mid post_1)\,;\,(pre_2 \mid cl_2 \mid post_2)$

$= (pre_1 \mid restrict\,cl_1\,post_1)\,;\,(pre_2 \mid restrict\,cl_2\,post_2)$ — definition 3.2.2

$= (pre_1 \cap wp\,(restrict\,cl_1\,post_1)\,pre_2 \mid (restrict\,cl_1\,post_1)\,;\,(restrict\,cl_2\,post_2))$ — theorem 3.2.1

$= (pre_1 \cap wp\,(restrict\,cl_1\,post_1)\,pre_2 \mid restrict\,(cl_1 \cup cl_2)\,((restrict\,cl_1\,post_1)\,;\,(restrict\,cl_2\,post_2)))$ — lemma 3.2.4

$= (pre_1 \cap wp\,(restrict\,cl_1\,post_1)\,pre_2 \mid cl_1 \cup cl_2 \mid (restrict\,cl_1\,post_1)\,;\,(restrict\,cl_2\,post_2))$ — definition 3.2.2

□

LEMMA 3.2.4.

$$restrict\,k\,p\,;\,restrict\,l\,q = restrict\,(k \cup l)\,(restrict\,k\,p\,;\,restrict\,l\,q)$$

PROOF. Let $r = wp\ (restrict\ k\ p)\ (\mathrm{dom}(restrict\ l\ q))$. We argue as follows:

$restrict\ k\ p\ ;\ restrict\ l\ q$	
$= (r \lhd (restrict\ k)\ p) \mathbin{⨾} restrict\ l\ q$	definition of ;
$= restrict\ k\ (r \lhd\ p) \mathbin{⨾} restrict\ l\ q$	lemma 3.2.7
$= restrict\ (k \cup l)\ (restrict\ k\ (r \lhd\ p) \mathbin{⨾} restrict\ l\ q)$	lemma 3.2.5
$= restrict\ (k \cup l)\ (r \lhd (restrict\ k)\ p \mathbin{⨾} restrict\ l\ q)$	lemma 3.2.7
$= restrict\ (k \cup l)\ (restrict\ k\ p\ ;\ restrict\ l\ q)$	

□

LEMMA 3.2.5.

$$restrict\ k\ p \mathbin{⨾} restrict\ l\ q = restrict\ (k \cup l)\ (restrict\ k\ p \mathbin{⨾} restrict\ l\ q)$$

PROOF. Lemma 3.2.6 tells us that the right hand side is a subset of the left hand side. To show that the right hand side is a subset of the left hand side, we let $s \mapsto s'$ be an arbitrary pair in $restrict\ k\ p \mathbin{⨾} restrict\ l\ q$ and argue as follows:

$s \mapsto s' \in restrict\ k\ p \mathbin{⨾} restrict\ l\ q$	
$\Leftrightarrow \exists\, s'' \bullet s \mapsto s'' \in restrict\ k\ p \wedge s'' \mapsto s' \in restrict\ l\ q$	definition of *comp*
$\Leftrightarrow \exists\, s'' \bullet s \mapsto s'' \in restrict\ k\ p \wedge s'' \mapsto s' \in restrict\ l\ q \wedge$	
$\quad k \lhd s = k \lhd s'' \wedge l \lhd s'' = l \lhd s'$	definition of *restrict*
$\Rightarrow \exists\, s'' \bullet s \mapsto s'' \in restrict\ k\ p \wedge s'' \mapsto s' \in restrict\ l\ q \wedge$	
$\quad k \cup l \lhd s = k \cup l \lhd s'$	lemma D.0.3
$\Leftrightarrow s \mapsto s' \in restrict\ k\ p \mathbin{⨾} restrict\ l\ q \wedge k \cup l \lhd s = k \cup l \lhd s'$	set theory
$\Leftrightarrow s \mapsto s' \in restrict(k \cup l)\ (restrict\ k\ p \mathbin{⨾} restrict\ l\ q)$	definition of *restrict*

□

LEMMA 3.2.6.

$$restrict\ k\ p \subseteq p$$

PROOF. Let s and s' be two states. We argue as follows:

$s \mapsto s' \in restrict\ k\ p$	
$\Leftrightarrow s \mapsto s' \in p \wedge k \lhd s = k \lhd s'$	definition of *restrict*
$\Rightarrow s \mapsto s' \in p$	

Since s and s' are arbitrary states we have shown that every pair in $restrict\ k\ p$ must also be a pair in p and therefore $restrict\ k\ p$ is a subset of p. □

LEMMA 3.2.7.

$$restrict\ k\ (c \lhd p) = c \lhd (restrict\ k\ p)$$

PROOF. Let s and s' be two states. We argue as follows:

$s \mapsto s' \in restrict\, k\, (c \lhd p)$	
$\Leftrightarrow s \mapsto s' \in c \lhd p \wedge k \unlhd s = k \unlhd s'$	definition of *restrict*
$\Leftrightarrow s \in c \wedge s \mapsto s' \in p \wedge k \unlhd s = k \unlhd s'$	definition of $\lhd$
$\Leftrightarrow s \in c \wedge s \mapsto s' \in restrict\, k\, p$	definition of *restrict*
$\Leftrightarrow s \mapsto s' \in c \lhd (restrict\, k\, p)$	definition of $\lhd$

Since s and s' are arbitrary states we have shown that every pair of states is a member of the left hand side if and only if it is a member of the right hand side and therefore the two sides are equal. □

3.2.3. Linking specification statements to assertions and weakest preconditions.

THEOREM 3.2.8. *If we weaken the precondition of a specification statement we get a refinement:*

$$pre_2 \subseteq pre_1 \Rightarrow (pre_1 \mid post) \sqsupseteq (pre_2 \mid post)$$

PROOF. We argue as follows:

$(pre_1 \mid post) \sqsupseteq (pre_2 \mid post)$	
$\Leftrightarrow \mathrm{dom}(pre_2 \mid post) \subseteq \mathrm{dom}(pre_1 \mid post) \wedge$	
$\quad \mathrm{dom}(pre_2 \mid post) \lhd (pre_1 \mid post) \subseteq (pre_2 \mid post)$	definition of refinement
$\Leftrightarrow (pre_2 \cap (\mathrm{dom}\, post)) \subseteq (pre_1 \cap (\mathrm{dom}\, post)) \wedge$	
$\quad (pre_2 \cap (\mathrm{dom}\, post)) \lhd (pre_1 \lhd post) \subseteq pre_2 \lhd post$	definition 3.2.1
$\Leftarrow pre_2 \subseteq pre_1 \wedge (pre_2 \cap pre_1 \cap (\mathrm{dom}\, post)) \lhd post \subseteq pre_2 \lhd post$	set theory and assumption
$\Leftarrow pre_2 \subseteq pre_1$	set theory

□

THEOREM 3.2.9. *Let prog and post be programs, and let pre be a constraint. If the domain of post is a subset of pre then*

$$prog \sqsupseteq (pre \mid post) \Leftrightarrow \{pre\}prog\{post\}$$

PROOF. Let *prog* and *post* be programs, and let the constraint *pre* be such that $\mathrm{dom}\, post$ is a subset of *pre*. We argue as follows:

$prog \sqsupseteq (pre \mid post)$	
$\Leftrightarrow prog \sqsupseteq pre \lhd post$	definition of specification statements
$\Leftrightarrow prog \sqsupseteq post$	assumption
$\Leftrightarrow wp\, prog\, post = \mathrm{dom}\, post$	lemma 3.2.10
$\Leftrightarrow wp\, prog\, post \subseteq \mathrm{dom}\, post \wedge \mathrm{dom}\, post \subseteq wp\, prog\, post$	set theory
$\Leftrightarrow true \wedge \mathrm{dom}\, post \subseteq wp\, prog\, post$	lemma 3.2.11
$\Leftrightarrow \{\mathrm{dom}\, post\}prog\{post\}$	theorem 3.1.2
$\Leftrightarrow \{pre\}prog\{post\}$	theorem 3.2.8 weaken precondition

□

LEMMA 3.2.10. *Let prog and spec be programs, then*

$$prog \sqsupseteq spec \Leftrightarrow wp\, prog\, spec = \operatorname{dom} spec$$

PROOF. Let *prog* and *spec* be programs. We prove the lemma by proving each implication separately. We already know that *wp prog spec* is a subset of dom *spec*, so we only need to prove that dom *spec* is a subset of *wp prog spec*. First, assume that *prog* refines *spec*. This means that

$$\operatorname{dom} spec \subseteq \operatorname{dom} prog \wedge \operatorname{dom} spec \lhd prog \subseteq spec$$

Now, let x be an element of *wp prog spec*. We argue as follows:

$x \in wp\, prog\, spec$

$\Leftrightarrow x \in \operatorname{dom} prog \wedge \forall y : State \bullet x \mapsto y \in prog \Rightarrow x \mapsto y \in spec$ — definition of *wp*

$\Leftarrow x \in \operatorname{dom} spec \wedge \forall y : State \bullet x \mapsto y \in prog \Rightarrow x \mapsto y \in spec$ — first conjunct of assumption

$\Leftrightarrow x \in \operatorname{dom} spec \wedge \{x\} \lhd prog \subseteq \{x\} \lhd spec$ — set theory

$\Leftarrow x \in \operatorname{dom} spec \wedge true$ — second conjunct of assumption

Because x is an arbitrary element of dom *spec* we have shown that every element of dom *spec* is also an element of *wp prog spec*, in other words $\operatorname{dom} spec \subseteq wp\, prog\, spec$.

Now let us assume that $wp\, prog\, spec = \operatorname{dom} spec$. Let x be an element of dom *spec*. We argue as follows:

$x \in \operatorname{dom} spec$

$\Leftrightarrow x \in wp\, prog\, spec$ — assumption

$\Leftrightarrow x \in \operatorname{dom} prog \wedge \forall y : State \bullet x \mapsto y \in prog \Rightarrow x \mapsto y \in spec$ — definition of *wp*

$\Rightarrow x \in \operatorname{dom} prog$

Therefore we get that $\operatorname{dom} spec \subseteq \operatorname{dom} prog$. Now let $x \mapsto z$ be a member of $(\operatorname{dom} spec) \lhd prog$. We argue as follows:

$x \mapsto z \in (\operatorname{dom} spec) \lhd prog$

$\Leftrightarrow x \in \operatorname{dom} spec \wedge x \mapsto z \in prog$ — set theory

$\Rightarrow x \in \operatorname{dom} prog \wedge x \mapsto z \in prog$ — since $\operatorname{dom} spec \subseteq \operatorname{dom} prog$

$\Rightarrow x \mapsto z \in spec$ — assumption

The last implication follows from the assumption by substituting the y in the universal quantifier with the particular z. □

LEMMA 3.2.11. *The weakest precondition of prog on spec is always a subset of spec's domain:*

$$wp\, prog\, spec \subseteq \operatorname{dom} spec$$

PROOF. Let x be a member of *wp prog spec*. We argue as follows:

$x \in wp\, prog\, spec$

$\Leftrightarrow x \in \operatorname{dom} prog \wedge \forall y : State \bullet x \mapsto y \in prog \Rightarrow x \mapsto y \in spec$ — definition of *wp*

$\Rightarrow x \in \operatorname{dom} spec$ — since there must be at least one such y

□

3.3. Discussion

In this chapter we have developed a relational model of programs and specifications. We have shown that with the aid of the weakest precondition function it is possible to model total correctness with partial relations. In addition, we have shown that weakest preconditions have a simple relational meaning and that they play an important role in the definition of sequential composition.

We do not need an explicit relational model to reason about specifications and programs. The early works on assertions [**27**] and weakest preconditions [**20**] are formalized as a logic (in the case of Hoare assertions) or as higher order logical operators (in the case of Dijkstra's weakest preconditions, Back's refinement calculus[**7**] and Morgan's refinement calculus[**44**]). Another work we should mention is Nelson's paper "A generalization of Dijkstra's calculus" [**45**], which describes several ways to define the semantics of programs (a relational semantics being one of them). However the focus of Nelson's work is on extending the guarded command language with partial programs and on the analysis of termination. It does not link the different semantics into a single mathematical theory.

All these works share an informal underlying relational model, which is adequate for their purpose because they are mostly used to analyze individual programs rather than systems. Most of the structure of an individual program is found in the algorithm itself and not in the data, which is often simple. In contrast, when we design a system, most of the structure is located in the data. In many management information systems, for example, the most complicated part is the database schema. Therefore, it is reasonable to use an explicit model for data, and it appears that the relational model is a particularly good fit for this purpose.

Even though using a relational model to describe the semantics of programs is not new, see for example [**16, 4, 19**], most of these works use total relations to describe the behavior of programs. To model the effect of applying an operation in an undefined state, every state that is not in the (original) domain of the program is associated to the special undefined state.

In contrast, the relational model we define here does not use the undefined state. Instead undefined states are simply taken out of the domain of the program. This means that we lose the ability to describe states from which it is possible both to end in a valid state and in an invalid state. However, this is not such a great loss because in practice we must always be cautious and consider a state that may lead to undefined behavior as a state that always leads to undefined behavior.

Both the B method and the refinement calculus are based on predicate transformers, not on relations. The refinement calculus does not use an explicit relational model. Instead it defines the semantics of a specification statement by mapping it to a predicate transformer:

$$wp(w : [pre, post], q) = pre \wedge \forall x' \bullet post \Rightarrow q$$

This axiom is used to justify the laws of the refinement calculus.

The B method uses a set theoretical model to give an implicit definition of predicate transformers (see theorem 6.4.1 in the B-book). But the purpose of the model is to make it easier to develop the mathematics, not to point out the relational character of the semantics.

In contrast, in Booster we specify behavior using a construct that, even though it looks similar to Morgan's specification statement, actually has the semantics of a Z operation schema, that is a relation. A Booster method is a relation, and the theory is used to make this notion precise.

Because we use relations and not predicate transformers, we lose the ability to model some specifications. For example the specification statement:

$$w : [false, true]$$

describes a program that is undefined in any initial state, while

$$w : [true, false]$$

is a program that is defined for any initial state but establishes the impossible *false* state. These are two entirely different programs as we can see by examining their weakest precondition function:

$$wp\,(w : [false, true])\, p = false \wedge \forall\, w \bullet true \Rightarrow p = false$$
$$wp\,(w : [true, false])\, p = true \wedge \forall\, w \bullet false \Rightarrow p = true$$

But the only relation that corresponds to both statements is the empty relation, because no pair of states can ever be a member of either statement.

Fortunately, this is not a problem in practice. The only specifications we cannot describe are miraculous specifications, which we cannot implement anyway.

The closest work to the model we present here is that of Woodcock and Cavalcanti on the weakest precondition semantics of Z [**10**]. They describe a relational model of programs, taking into account that postconditions are relations between initial and final states, and they define a relational weakest precondition function that is very similar to the function we have developed in this chapter. However they use a special undefined state to model undefined behavior. This makes their relations equivalent to predicate transformers but also means that not every relation is a program. They must define healthiness conditions that select the subset of relations that correspond to programs. In contrast, we do not use the undefined state, so every relation corresponds to a program.

The more powerful predicate transformers are essential only when we analyze the termination of recursive programs. But in this work we are not concerned with the termination of programs because we do not use loops or recursion. Instead we assume that any operations that may require a loop (such as sorting a sequence or copying the content of a set) are provided as primitive operations.

CHAPTER 4

The structure of Booster models

In this chapter we use the Z language to describe the abstract syntax and the type rules of the Booster language.

4.1. Abstract syntax

Since we are interested in the meaning of models and not in the concrete syntax of the language we focus the description on the abstract syntax. The concrete syntax is summarized in appendix A.

As we have already seen in chapter 2, a booster model is composed of a collection of named classes, each presenting a collection of named methods, attributes, and associations. Associations must refer to existing classes in the model: the target of each association must refer to a class in the model, and the mirror of each association must refer to an association in the model.

Model

$$class : Name \rightarrow\!\!\!\!\!\!+\; Class$$

$$\begin{array}{l} \forall\, c : \mathrm{ran}\, class \bullet \\ \quad \forall\, a : \mathrm{ran}(c.association) \bullet \\ \qquad a.target \in \mathrm{dom}(class) \land \\ \qquad a.mirror \in \mathrm{dom}((class\; a.target).association) \end{array}$$

The *Class* schema defines the structure of a Booster class. We require that the names of attributes and associations are mutually exclusive.

Class

$$\begin{array}{l} attribute : Name \rightarrow\!\!\!\!\!\!+\; Attribute \\ association : Name \rightarrow\!\!\!\!\!\!+\; Association \\ method : Name \rightarrow\!\!\!\!\!\!+\; Method \end{array}$$

$$\mathrm{dom}\, attribute \cap \mathrm{dom}\, association = \{\}$$

Each attribute has a type, corresponding to a range of primitive values.

Attribute

$$type : Type$$

As we have already mentioned, each association has a target type, a multiplicity, and a mirror.

Association

$$\begin{array}{l} target, mirror : Name \\ multiplicity : Multiplicity \end{array}$$

$$Multiplicity ::= mandatory \mid optional \mid many$$

A method can be either primitive, a reference to another method in the model, or constructed using method combinators.

$$
\begin{aligned}
\mathit{Method} ::= {} & \mathit{primitive}\langle\langle \mathit{PrimitiveMethod} \rangle\rangle \\
\mid {} & \mathit{methodref}\langle\langle \mathit{Term} \times \mathit{Name} \rangle\rangle \\
\mid {} & \mathit{guardm}\langle\langle \mathit{Formula} \times \mathit{Method} \rangle\rangle \\
\mid {} & \mathit{conj}\langle\langle \mathit{Method} \times \mathit{Method} \rangle\rangle \\
\mid {} & \mathit{disj}\langle\langle \mathit{Method} \times \mathit{Method} \rangle\rangle \\
\mid {} & \mathit{then}\langle\langle \mathit{Method} \times \mathit{Method} \rangle\rangle
\end{aligned}
$$

Each primitive method has two named components denoting its pre- and post-conditions.

PrimitiveMethod

$pre, post : Formula$

A formula is a description of a predicate. A formula can be either the literal true or false, a basic formula, or any of the standard logical operators of first order logic[1].

$$
\begin{aligned}
\mathit{Formula} ::= {} & \mathit{true} \mid \mathit{false} \\
\mid {} & \mathit{basic}\langle\langle \mathit{BasicFormula} \times \mathit{Term} \times \mathit{Term} \rangle\rangle \\
\mid {} & \mathit{negation}\langle\langle \mathit{Formula} \rangle\rangle \\
\mid {} & \mathit{conjunction}\langle\langle \mathit{Formula} \times \mathit{Formula} \rangle\rangle \\
\mid {} & \mathit{disjunction}\langle\langle \mathit{Formula} \times \mathit{Formula} \rangle\rangle \\
\mid {} & \mathit{implication}\langle\langle \mathit{Formula} \times \mathit{Formula} \rangle\rangle
\end{aligned}
$$

Each of the basic formulas that we have described informally in the introduction is associated with a basic formula name.

$$\mathit{BasicFormula} ::= \mathit{isMember} \mid \mathit{isNonMember} \mid \mathit{isEqual} \mid \mathit{isNotEqual}$$

We use formulas to describe the desirable properties of *terms*. Terms represent the entities on which the model operate: integers, object identifiers, sets, and so on. We create terms either from variables or by applying a function which calculates a new term based on its arguments[2].

Variables in Booster represent several distinct entities. We may refer to ‘before’ values which describe the state of an entity just before the method is executed, or to ‘after’ values which represent the state of the entity just after the method has finished executing. In addition, we may refer to a variety of sources: attributes, inputs, or the special variable *this* which represents the current object.

[1] An essential part of the compilation process is the elimination of negations. However, we want to retain the implications in the formulas even after we have eliminated the negations. Therefore we define implications directly instead of describing them in terms of negations and disjunctions.

[2] We do not offer literals because their treatment is standard and does not offer any interesting insight to the semantics of Booster.

$$
\begin{aligned}
Term ::= {} & this \\
\mid {} & inputvar\langle\langle Name\rangle\rangle \\
\mid {} & new\langle\langle Name\rangle\rangle \\
\mid {} & newobject\langle\langle Name\rangle\rangle \\
\mid {} & before\langle\langle Name\rangle\rangle \\
\mid {} & after\langle\langle Name\rangle\rangle \\
\mid {} & mkrel\langle\langle Term \times Term\rangle\rangle \\
\mid {} & path\langle\langle Term \times Term\rangle\rangle \\
\mid {} & the\langle\langle Term\rangle\rangle \\
\mid {} & union\langle\langle Term \times Term\rangle\rangle \\
\mid {} & setminus\langle\langle Term \times Term\rangle\rangle \\
\mid {} & update\langle\langle Term \times Term\rangle\rangle \\
\mid {} & cond\langle\langle Formula \times Term \times Term\rangle\rangle
\end{aligned}
$$

The definition above contains just a few primitive functions such as *union*, *setminus*, and *update*. However as we shall see later, they are enough to demonstrate how we treat functions in Booster in general. Any additional functions (like integer operators and other useful functions) may be treated in the same way.

Using the constructors of *Formula* and *Term* is inconvenient. Even in simple cases, the resulting expressions are very difficult to read. To overcome this problem we define for each constructor an operator that takes the appearance of the familiar logical or set theoretic operator. To avoid confusing these operators with their *Z* equivalents we paint them with a different color.

For example, the variables b and a in the formula $b \in a.ab$ are arbitrary terms, and the expression ab denotes a variable whose name is ab. The formula $b \in a.ab$ is equivalent to the formula

$$basic(isMember, b, path(a, after(ab)))$$

but the former is easier to read. A complete list of these operators is available in appendix B.

4.2. Identifying entities in the model

Following [**15**] we will use schemas to represent particular constructs of the model. The schemas declare the identifiers of the corresponding schema types and relate their values to those of identifiers in the enclosing scope:

IdentifyClass

$Model;\ Class;\ thisClass : Name$

$thisClass \mapsto \theta Class \in class$

IdentifyAttribute

$IdentifyClass;\ Attribute;\ thisAttribute : Name$

$thisAttribute \mapsto \theta Attribute \in attribute$

IdentifyAssociation

$IdentifyClass;\ Association;\ thisAssociation : Name$

$thisAssociation \mapsto \theta Association \in association$

IdentifyMethod

$IdentifyClass;\ theMethod : Method;\ thisMethod : Name$

$thisMethod \mapsto theMethod \in method$

4.3. The types of classes

Each class in the model has a type that contains the set of all the potential object identifiers of this class. The types of all the classes in the model partition the set of all object identifiers.

ClassTypes

$Model$
$classtype : Name \rightarrowtail\!\!\!\!\!\!+ \ \mathbb{P}\, ObjectID$

$\mathrm{dom}\, classtype = \mathrm{dom}\, class$
$\forall\, n_1, n_2 : \mathrm{dom}\, classtype \mid n_1 \neq n_2 \bullet classtype\, n_1 \cap classtype\, n_2 = \{\}$
$\bigcup(\mathrm{ran}\, classtype) = ObjectID$

4.4. The types of terms

The following definition describes the possible types in our model of the Booster language:

$$
\begin{aligned}
Type ::= {} & reference\langle\!\langle Name\rangle\!\rangle \\
\mid {} & settype\langle\!\langle Type\rangle\!\rangle \\
\mid {} & opttype\langle\!\langle Type\rangle\!\rangle \\
\mid {} & relationtype\langle\!\langle Type \times Type\rangle\!\rangle \\
\mid {} & functiontype\langle\!\langle Type \times Type\rangle\!\rangle
\end{aligned}
$$

Every type defines a set of values, which we call the values of the type. The function *values* associates each type with its values. We add the undefined value to each type because an undefined calculation may occur in any type.

TypeValues

$ClassTypes$
$values : Type \rightarrow \mathbb{P}\, Value$

$$
\begin{aligned}
& \forall\, n, m : Name;\ t : Type \mid \{n, m\} \subseteq \mathrm{dom}\, classtype \bullet \\
& \quad values\,(reference\, n) = objectid(\!|\ classtype\, n\ |\!) \cup \{\bot\} \wedge \\
& \quad values\,(settype\,(reference\, n)) = set(\!|\ \mathbb{P}(classtype\, n)\ |\!) \cup \{\bot\} \wedge \\
& \quad values\,(opttype\,(reference\, n)) = \\
& \qquad \{obj : classtype\, n \bullet set\,\{obj\}\} \cup set\,\{\} \cup \{\bot\} \wedge \\
& \quad values\,(relationtype(reference\, n, reference\, m)) = \\
& \qquad relation(\!|\ (classtype\, n) \leftrightarrow (classtype\, m)\ |\!) \cup \{\bot\} \wedge \\
& \quad values\,(functiontype(reference\, n, t)) = \\
& \qquad function(\!|\ (classtype\, n) \rightarrowtail\!\!\!\!\!\!+ \ values\, t \cup \{\bot\}\ |\!)
\end{aligned}
$$

The type of a term depends on the model, on the method in which it appears, and on the class in which the method appears. Therefore we use a schema to hold the method, the class and the model as the context of the function *wtt* that maps each term to its type.

The type of an input variable must be specified by a top level conjunct in the precondition of the primitive method in which it appears.

WellTypedInput

$$
\begin{array}{l}
\mathit{IdentifyMethod} \\
\mathit{wtt} : \mathit{Term} \pfun \mathit{Type} \\
\hline
\exists \mathit{PrimitiveMethod} \mid \mathit{theMethod} = \mathit{primitive}\ \theta \mathit{PrimitiveMethod} \bullet \\
\quad \forall n : \mathit{Name};\ c : \mathrm{dom}\, \mathit{class} \bullet \mathit{wtt}\ (\mathit{inputvar}\ n) = \mathit{reference}\ c \Leftrightarrow \\
\quad\quad (\exists f : \mathit{Formula} \bullet \mathit{pre} = n \in c \land f)
\end{array}
$$

The type of the variable *this* is always a reference to the current class.

WellTypedThis

$$
\begin{array}{l}
\mathit{IdentifyClass} \\
\mathit{wtt} : \mathit{Term} \pfun \mathit{Type} \\
\hline
\mathit{wtt}\ \mathit{this} = \mathit{reference}\ \mathit{thisClass}
\end{array}
$$

The term $\mathit{new}(A)$ identifies all the instances of class A that appear in the after extent but not in the before extent. Its type is therefore a set of references of class A.

WellTypedNewObject

$$
\begin{array}{l}
\mathit{Model} \\
\mathit{wtt} : \mathit{Term} \pfun \mathit{Type} \\
\hline
\forall c : \mathit{Name} \bullet \mathit{wtt}\ (\mathit{new}(c)) = \mathit{settype}(\mathit{reference}\ c) \Leftrightarrow c \in \mathrm{dom}\, \mathit{class}
\end{array}
$$

The term $\mathit{newobject}(A)$ is used by the code generator to refer to a particular new object. Its type is therefore a reference to an instance of class A.

WellTypedNewObject

$$
\begin{array}{l}
\mathit{Model} \\
\mathit{wtt} : \mathit{Term} \pfun \mathit{Type} \\
\hline
\forall c : \mathit{Name} \bullet \mathit{wtt}\ (\mathit{newobject}(c)) = \mathit{reference}\ c \Leftrightarrow c \in \mathrm{dom}\, \mathit{class}
\end{array}
$$

The term $t.n$ has the type tt if and only if t refers to an object of a class that contains an attribute or association with the name n whose type is tt.

WellTypedPath

$$
\begin{array}{l}
\mathit{wtt} : \mathit{Term} \pfun \mathit{Type} \\
\hline
\forall t : \mathit{Term};\ n : \mathit{Name};\ tt : \mathit{Type} \bullet \mathit{path}(t, \mathit{before}\ n) = tt \\
\quad \Leftrightarrow \exists \mathit{thisAttribute}, \mathit{thisAssociation}, \mathit{thisClass} : \mathit{Name}; \\
\quad\quad \mathit{Attribute};\ \mathit{Association};\ \mathit{Class} \bullet \mathit{TypeOfAttr} \lor \mathit{TypeOfManyAssoc} \lor \\
\quad\quad\quad \mathit{TypeOfOptionalAssoc} \lor \mathit{TypeOfMandatoryAssoc}
\end{array}
$$

TypeOfAttr

$$
\begin{array}{l}
\mathit{IdentifyAttribute} \\
t : \mathit{Term};\ n : \mathit{Name};\ tt : \mathit{Type} \\
\hline
n = \mathit{thisAttribute} \\
\mathit{wtt}\ t = \mathit{reference}\ \mathit{thisClass} \\
\mathit{type} = tt
\end{array}
$$

TypeOfAssoc

$IdentifyAssociation$
$t : Term;\ n : Name;\ tt : Type$

$n = thisAssociation$
$wtt\ t = reference\ thisClass$

TypeOfMandatoryAssoc

$TypeOfAssoc$

$multiplicity = mandatory$
$tt = reference\ target$

TypeOfManyAssoc

$TypeOfAssoc$

$multiplicity = optional$
$tt = set\ (reference\ target)$

TypeOfOptionalAssoc

$TypeOfAssoc$

$multiplicity = optional$
$tt = opttype\ (reference\ target)$

WellTypedThe

$wtt : Term \leftrightarrow Type$

$\forall t : Term;\ tt : Type \bullet wtt(the(t)) = tt \Leftrightarrow$
$\quad wtt\ t = opttype\ tt \wedge tt \in \operatorname{ran} reference$

WellTypedMkRel

$wtt : Term \leftrightarrow Type$

$\forall t_1, t_2 : Term;\ tt : Type \bullet wtt(mkrel(t_1, t_2)) = tt \Leftrightarrow$
$\quad wtt\ t_1 \in \operatorname{ran} reference \wedge wtt\ t_2 \in \operatorname{ran} reference \wedge tt = relationtype(wtt\ t_1, wtt\ t_2)$

WellTypedConditional

$wtt : Term \leftrightarrow Type$
$wff : \mathbb{P}\ Formula$

$\forall test : Formula;\ alt, consq : Term;\ tt : Type \bullet wtt(cond(test, alt, consq)) = tt \Leftrightarrow$
$\quad wtt\ alt = tt \wedge wtt\ consq = tt \wedge test \in wff$

WellTypedOperator

$$wtt : Term \pfun Type$$

$$\forall t_1, t_2 : Term \bullet \\ \quad wtt\, t_1 = wtt\, t_2 \land \\ \quad wtt\, t_1 \in compatible \land \\ \quad wtt(union(t_1, t_2)) = wtt\, t_1 \land \\ \quad wtt(setminus(t_1, t_2)) = wtt\, t_1 \land \\ \quad wtt(update(t_1, t_2)) = wtt\, t_1$$

The predicate *compatible* selects all the types that are compatible with the operations *union*, *setminus*, and *update*.

$$compatible : \mathbb{P}\, Type$$

$$compatible = \{w : Type \bullet settype\, w\} \cup \\ \quad \{v, w : Type \bullet relationtype(v, w)\} \cup \\ \quad \{v, w : Type \bullet functiontype(v, w)\}$$

Finally, we combine all the schemas into a single definition of well typed terms.

$$WellTypedTerms \mathrel{\widehat{=}} WellTypedInput \land WellTypedThis \land \\ \quad WellTypedAttribute \land WellTypedAssociation \land WellTypedThe \land \\ \quad WellTypedFunctionalPath \land WellTypedRelationalPath \land \\ \quad WellTypedOperator \land WellTypedNewObject$$

4.5. The types of formulas

To define the set of all the well formed formulas, we take the set of formulas *wff*, and create a schema that constrains *wff* for each kind of formula. We then conjoin all these constraints into a single schema. Because we have covered all the possible ways to construct formulas, we are certain that the set *wff* contains all the well defined formulas and only the well defined formulas.

wfBasicFormula

$$WellTypedTerms \\ wff : \mathbb{P}\, Formula$$

$$\forall f : BasicFormula;\ t_1, t_2 : Term \bullet \\ \quad basic(f, t_1, t_2) \in wff \Leftrightarrow \{t_1, t_2\} \subseteq (\mathrm{dom}\, wtt)$$

wfConjunction

$$wff : \mathbb{P}\, Formula$$

$$\forall p, q : Formula \bullet p \land q \in wff \Leftrightarrow (p \in wff \land q \in wff)$$

wfDisjunction

$$wff : \mathbb{P}\, Formula$$

$$\forall p, q : Formula \bullet p \lor q \in wff \Leftrightarrow (p \in wff \land q \in wff)$$

wfImplication

$$wff : \mathbb{P}\, Formula$$

$$\forall p, q : Formula \bullet p \Rightarrow q \in wff \Leftrightarrow (p \in wff \land q \in wff)$$

wfNegation

$wff : \mathbb{P}\, Formula$

$\forall\, p : Formula \bullet \neg\, p \in wff \Leftrightarrow p \in wff$

The schema *WellFormedFormulas* defines the set *wff* to hold all the well defined formulas in the context of a model *Model* and a class *Class*.

$WellFormedFormulas \mathrel{\widehat=} wfBasicFormula \land wfConjunction \land$
$\quad wfDisjunction \land wfImplication \land wfNegation$

4.6. Well formed methods

Every method in the model must be *well formed*. A primitive method is well formed if its pre and postconditions are well typed.

wfPrimitiveMethod

$WellFormedFormulas$
$wfm : \mathbb{P}\, Method$

$\forall\, PrimitiveMethod \bullet$
$\quad primitive\, \theta PrimitiveMethod \in wfm \Leftrightarrow pre \in wff \land post \in wff$

A guarded method is well formed if its guard is well formed and its guarded method is well formed.

wfGuard

$WellFormedFormulas$
$wfm : \mathbb{P}\, Method$

$\forall\, g : Formula;\ m : Method \bullet$
$\quad guardm(g, m) \in wfm \Leftrightarrow g \in wff \land m \in wfm$

Similarly the disjunctive, and sequence combinators are well formed if their arguments are well formed.

wfOrCombinator

$wfm : \mathbb{P}\, Method$

$\forall\, m_1, m_2 : Method \bullet$
$\quad conj(m_1, m_2) \in wfm \Leftrightarrow m_1 \in wfm \land m_2 \in wfm$

wfThenCombinator

$wfm : \mathbb{P}\, Method$

$\forall\, m_1, m_2 : Method \bullet$
$\quad then(m_1, m_2) \in wfm \Leftrightarrow m_1 \in wfm \land m_2 \in wfm$

A method reference is well formed if its path denotes some class, say *thisClass*, and the reference' name refers to the name of a method in *thisClass*.

$$
\begin{array}{l}
\textit{wfMethodReference} \\
\hline
\textit{WellTypedTerms} \\
\textit{wfm} : \mathbb{P}\,\textit{Method} \\
\hline
\forall \textit{path} : \textit{Term};\ \textit{thisMethod} : \textit{Name} \bullet \\
\quad \textit{methodref}(\textit{path}, \textit{thisMethod}) \in \textit{wfm} \Leftrightarrow \\
\quad\quad (\exists \textit{thisClass} : \textit{Name};\ \textit{Class};\ \textit{theMethod} : \textit{Method} \bullet \\
\quad\quad\quad \textit{IdentifyMethod} \land \textit{wtt}\,\textit{path} = \textit{reference}\,\textit{thisClass})
\end{array}
$$

An AND combinator is well formed if both its arguments are well formed and syntactically independent.

DEFINITION 4.6.1. *Two methods are syntactically independent if their alphabets are disjoint.*

$$
\begin{array}{l}
\textit{wfAndCombinator} \\
\hline
\textit{wfm} : \mathbb{P}\,\textit{Method} \\
\hline
\forall m_1, m_2 : \textit{Method} \bullet \\
\quad \textit{conj}(m_1, m_2) \in \textit{wfm} \Leftrightarrow \\
\quad\quad m_1 \in \textit{wfm} \land m_2 \in \textit{wfm} \land \textit{alpha}\,m_1 \cap \textit{alpha}\,m_2 = \{\}
\end{array}
$$

The function *alpha* calculates the alphabet of a method. The alphabet of a primitive method is the union of the free attributes of its pre- and postconditions. The alphabet of a combinator is the union of the alphabets of its arguments.

$$
\begin{array}{l}
\textit{alpha} : \textit{Method} \rightarrow \mathbb{P}\,\textit{Name} \\
\hline
\forall \textit{PrimitiveMethod} \bullet \\
\quad \textit{alpha}(\textit{primitive}(\theta \textit{PrimitiveMethod})) = \textit{free}(\textit{pre}) \cup \textit{free}(\textit{post}) \\
\forall g : \textit{Formula};\ M : \textit{Method} \bullet \\
\quad \textit{alpha}(\textit{guardm}(g, M)) = \textit{free}(g) \cup \textit{alpha}(M) \\
\forall M_1, M_2 : \textit{Method} \bullet \\
\quad \textit{alpha}(\textit{conj}(M_1, M_2)) = \textit{alpha}(M_1) \cup \textit{alpha}(M_2) \land \\
\quad \textit{alpha}(\textit{disj}(M_1, M_2)) = \textit{alpha}(M_1) \cup \textit{alpha}(M_2) \land \\
\quad \textit{alpha}(\textit{then}(M_1, M_2)) = \textit{alpha}(M_1) \cup \textit{alpha}(M_2)
\end{array}
$$

$$
\begin{array}{l}
\textit{free} : \textit{Formula} \rightarrow \mathbb{P}\,\textit{Name} \\
\hline
\textit{free}\,\textit{true} = \{\} \\
\textit{free}\,\textit{false} = \{\} \\
\forall n : \textit{BasicFormula};\ t_1, t_2 : \textit{Term} \bullet \\
\quad \textit{free}\,(\textit{basic}(n, t_1, t_2)) = \textit{freev}\,t_1 \cup \textit{freev}\,t_2 \\
\forall p, q : \textit{Formula} \bullet \\
\quad \textit{free}\,(p \land q) = \textit{free}\,p \cup \textit{free}\,q \land \\
\quad \textit{free}\,(p \lor q) = \textit{free}\,p \cup \textit{free}\,q \land \\
\quad \textit{free}\,(p \Rightarrow q) = \textit{free}\,p \cup \textit{free}\,q
\end{array}
$$

$$
\begin{array}{|l}
freev : Term \rightarrow \mathbb{P}\, Name \\
\hline
freev\ this = \{\} \\
\forall\, n : Name \bullet \\
\quad freev\ (before\ n) = \{n\} \land \\
\quad freev\ (after\ n) = \{n\} \\
\forall\, t_1, t_2 : Term \bullet \\
\quad freev\ (mkrel(t_1, t_2)) = freev\ t_1 \cup freev\ t_2 \land \\
\quad freev\ (path(t_1, t_2)) = freev\ t_1 \cup freev\ t_2 \land \\
\quad freev\ (the(t_1)) = freev\ t_1 \land \\
\quad freev\ (union(t_1, t_2)) = freev\ t_1 \cup freev\ t_2 \land \\
\quad freev\ (setminus(t_1, t_2)) = freev\ t_1 \cup freev\ t_2 \land \\
\quad freev\ (update(t_1, t_2)) = freev\ t_1 \cup freev\ t_2
\end{array}
$$

The following schema defines the set *wfm* which contains all the potential well formed methods of a particular model.

$$
\begin{array}{l}
wfMethod \mathrel{\widehat=} \\
\quad \exists\, wtt : Term \mathrel{+\!\!\!\!\rightarrow} Type;\ wff : \mathbb{P}\, Formula \bullet \\
\qquad wfPrimitiveMethod \land wfMethodReference \land \\
\qquad wfOrCombinator \land wfThenCombinator \land wfAndCombinator \land wfGuard
\end{array}
$$

To know if a particular method is well formed or not we use the following schema, which uses *IdentifyMethod* to identify the method in question:

$$
\begin{array}{l}
MethodWellFormed \mathrel{\widehat=} \\
\quad \exists\, wfm : \mathbb{P}\, Method \bullet [wfMethod;\ IdentifyMethod \mid theMethod \in wfm]
\end{array}
$$

We have hidden the set of all well formed methods because we are only interested to know if a particular method is well formed.

4.7. Well formed models

A model is well formed if all of its methods are well formed, and no method definition refers to itself. We have already defined the necessary schemas to specify the first requirement. Let us therefore see how we can prevent recursive method definitions.

We prevent recursive methods by first defining when a method is recursive and then insisting that such a situation is not allowed.

A method is recursive if its definition contains a reference to its own name. This may happen directly, that is, there may be such reference in the methods that make up the definition of our method. But it may also happen indirectly, that is, there may be a reference to another method which in turn refers back to our method.

First, let us define a function that finds all the methods names that are being used by a method:

$$
\begin{array}{|l}
usedby : Method \rightarrow \mathbb{P}\, Name \\
\hline
\forall\, PrimitiveMethod;\ n : Name \bullet \\
\quad usedby \theta PrimitiveMethod\ n = \{\} \\
\forall\, m_1, m_2 : Method \bullet \\
\quad usedby\ conj(m_1, m_2)\ n = usedby\ m_1 \cup usedby\ m_2 \land \\
\quad usedby\ disj(m_1, m_2)\ n = usedby\ m_1 \cup usedby\ m_2 \land \\
\quad usedby\ then(m_1, m_2)\ n = usedby\ m_1 \cup usedby\ m_2 \land \\
\forall\, test : Formula;\ m : Method \bullet \\
\quad usedby\ guard(test, m) = usedby\ m \\
\forall\, path : Term;\ n : Name \bullet usedby\ methodref(path, n) = \{n\}
\end{array}
$$

Now we can define when a method name, say *thisMethod*, directly references another name, say *m*. This happens when the name *m* is used by the method associated to *thisMethod*:

$$
\begin{array}{|l}
\textit{DirectReference} \\
\hline
\textit{Model} \\
\textit{dirref} : \textit{Name} \leftrightarrow \textit{Name} \\
\hline
\forall \textit{thisMethod}, m : \textit{Name} \bullet \textit{thisMethod} \mapsto m \in \textit{dirref} \Leftrightarrow \\
\quad \exists \textit{thisClass} : \textit{Name};\ \textit{Class};\ \textit{theMethod} : \textit{Method} \mid \textit{IdentifyMethod} \bullet \\
\quad\quad m \in \textit{usedby}\ \textit{theMethod} \\
\hline
\end{array}
$$

Finally, we can use the transitive closure of *dirref* to form the indirect reference relation.

Here is therefore the definition of a well formed model. It is well formed when every method is well formed, and when no method name is directly or indirectly referencing itself:

$$
\begin{array}{|l}
\textit{wfModel} \\
\hline
\textit{Model} \\
\hline
\forall \textit{thisClass} : \textit{Name};\ \textit{Class};\ \textit{thisMethod} : \textit{Name};\ \textit{theMethod} : \textit{Method} \mid \\
\quad \textit{IdentifyMethod} \bullet \textit{MethodWellFormed} \land \textit{thisMethod} \mapsto \textit{thisMethod} \notin \textit{dirref}^{+} \\
\hline
\end{array}
$$

CHAPTER 5

The Semantics of Booster

In this chapter we describe the semantics of Booster by fixing a particular structure on the state space of the relational model we have defined in chapter 3. We define the meaning of terms, formulas and models by mapping them to values, predicates and abstract data types. We explain how the structure of the model, the association invariants and the method specifications are combined to form the semantics of the Booster language.

5.1. Component states

We define the meaning of Booster models by mapping each instance to an abstract data type, comprising of a set of states, and a collection of named programs upon these states.

ADT

$$state : \mathbb{P}\, ComponentState$$
$$operation : Name \rightarrow\!\!\!\!\!+\; Program$$

$$\operatorname{ran} operation \subseteq state \leftrightarrow state$$

The abstract data type represents the semantics of a *component*, specified by the collection of associated classes in the model. The following schema describes the set of all possible component states:

ComponentState

$$extent : Name \rightarrow\!\!\!\!\!+\; \mathbb{P}\, ObjectID$$
$$link : Name \rightarrow\!\!\!\!\!+\; (ObjectID \leftrightarrow ObjectID)$$
$$value : Name \rightarrow\!\!\!\!\!+\; (ObjectID \rightarrow\!\!\!\!\!+\; Value)$$
$$local : Name \rightarrow\!\!\!\!\!+\; Value$$

$$\forall c_1, c_2 : \operatorname{dom} extent \mid c_1 \neq c_2 \bullet extent\, c_1 \cap extent\, c_2 = \{\}$$
$$\forall n : \operatorname{dom} link \bullet$$
$$\quad \operatorname{dom}(link\, n) \subseteq \bigcup(\operatorname{ran} extent) \wedge$$
$$\quad \operatorname{ran}(link\, n) \subseteq \bigcup(\operatorname{ran} extent)$$
$$\forall n : \operatorname{dom} value \bullet \operatorname{dom}(value\, n) \subseteq \bigcup(\operatorname{ran} extent)$$
$$UniqueNamesAssumption$$

The extent contains all the valid object identifiers for each class. For each association, the function *link* holds all the object identifier pairs that make up the association; the function *value* associates every object with the values of its attributes; finally, the function *local* holds the values of local variables.

The constraint in the *ComponentState* schema insists that:

(1) object identifiers appear only in the extent of a single class.
(2) The associations and the attributes refer to existing objects.

In addition we assume that the names of the different entity functions (*extent*, *link*, and so on) are mutually exclusive. This assumption is not essential because we can always disambiguate the entity by considering the name of the class as well as the name of the attribute, but it simplifies the analysis.

UniqueNamesAssumption

$extent : Name \pfun \mathbb{P}\, ObjectID$
$link : Name \pfun (ObjectID \leftrightarrow ObjectID)$
$value : Name \pfun (ObjectID \pfun Value)$
$local : Name \pfun Value$

$\mathbf{let}\ ns == \langle \mathrm{dom}\, extent, \mathrm{dom}\, link, \mathrm{dom}\, value, \mathrm{dom}\, local \rangle \bullet$
$\quad \forall\, i, j : \mathrm{dom}\, ns \mid i \neq j \bullet (ns\, i) \cap (ns\, j) = \{\}$

For a state to belong to the state space of a given model its members must respect the structure defined by the model: the state's extent must contain a binding for every class in the model; every relation in the state must correspond to precisely one association in the model; every function in *value* must correspond to precisely one attribute in the model. In the rest of this section we describe a set of schemas that define precisely the state space of Booster models.

Every attribute must have a corresponding attribute function; the domain of the attribute function must consist of all the objects in the extent of the attribute's class; the values that are associated with the attribute must belong to its type.

AttributeConstraint

$IdentifyAttribute;\ ComponentState$

$thisAttribute \in \mathrm{dom}(value)$
$\mathrm{dom}(value\ thisAttribute) = extent\ thisClass$
$\mathrm{ran}(value\ thisAttribute) \subseteq values\ type$

Every association a must have a relation r in *link*; the relation r must be between a's class extent and a's target extent; finally, the relation that corresponds to a's mirror must be the inverse of r.

AssociationConstraint

$IdentifyAssociation;\ ComponentState$

$thisAssociation \in \mathrm{dom}(link)$
$link\ thisAssociation \in extent\ thisClass \leftrightarrow extent\ target$
$link\ thisAssociation = (link\ mirror)^{\sim}$

In the full blown Booster compiler, attributes or associations that have the same name in different classes are disambiguated by using the name of the class as well as the name of the entity in question. However, because such details are not essential to the description and just complicate the analysis, we assume instead that the names of attributes and associations from different classes are unique.

We use the following schema to define the state space of any well formed model:

ModelConstraint

$wfModel$
$ComponentState$

$\mathrm{dom}\, extent = \mathrm{dom}\, class$
$\forall\, thisClass : \mathrm{dom}\, class \bullet \exists\, Class \bullet$
$\quad (\forall\, thisAttribute : \mathrm{dom}\, attribute \bullet$
$\qquad \exists\, Attribute \bullet AttributeConstraint) \land$
$\quad (\forall\, thisAssociation : \mathrm{dom}\, association \bullet$
$\qquad \exists\, Association \bullet AssociationConstraint)$

5.1.1. Relating component states to program states. The function *stos* maps the structured schema *ComponentState* to the unstructured function *State*.

$$stos : ComponentState \rightarrowtail\!\!\!\!\!\!\mapsto State$$

$$\forall s : ComponentState \bullet$$
$$\quad stos\, s = s.extent \mathbin{⨾} set \cup s.link \mathbin{⨾} relation \cup s.value \mathbin{⨾} function \cup s.local$$

Because the names of the different components do not overlap, their union is a function. The range of *stos* is the set of all unstructured states that correspond to valid Booster states.

5.2. The outside world

A complete Booster system contains a run time component that is responsible for communicating with the environment. It provides a way for the environment to select objects and the operations the environment would like to apply to these objects. The Booster run time component checks that the precondition of the method holds before executing the method. Otherwise it does not execute the method and returns an error code to the environment.

However, because we focus in this thesis on the semantics of Booster, and not on its interaction with the environment, we will not mention the environment again, but assume that the necessary inputs (which include the input variables and the current object) are available and have the appropriate types (the types of the input variables must be asserted in the precondition).

5.3. Terms

In most imperative programming languages the meaning of terms is defined by mapping them to values in the context of a state. The state is used to look up the values of the variables.

In the case of Booster, in addition to variables that refer to the current state, we also have variables that refer to the 'before' state. Accordingly we use two states: one to look up variables that refer to the current (after) state, and the other to look up variables that refer to the old (before) state.

The meaning of a variable in the context of a pair of states is given by looking up the variable's name either in the inputs, or in the before state, or in the after state, depending on the nature of the variable.

VariableMeaning

$$input : Name \nrightarrow Value$$
$$currentObject : ObjectID$$
$$s, s' : State$$
$$value : Term \nrightarrow Value$$

$$\forall n : Name \bullet$$
$$\quad value\,(inputvar\, n) = input\, n \land$$
$$\quad value\,(before\, n) = s\, n \land$$
$$\quad value\,(after\, n) = s'\, n \land$$
$$\quad value\; this = objectid\; currentObject$$

The term $new(c)$ denotes the set of all identifiers that appear in c's extent but not in c_0's extent.

NewMeaning

$$s, s' : State$$
$$value : Term \nrightarrow Value$$

$$\forall c : Name \bullet value\,(new(c)) = value\,(c \setminus c_0)$$

The term $newobject(A)$ denotes an object identifier that does not appear in the extent (of class A) in the current state[1].

$$
\begin{array}{l}
\textit{NewObjectMeaning} \\
\hline
s, s' : State \\
value : Term \rightarrowtail Value \\
\hline
\forall c : Name \bullet value\,(newobject(c)) = objectid\,(uniqueId\,s)
\end{array}
$$

An identifier is unique if it does not appear in the extent of any class.

$$
\begin{array}{l}
uniqueId : State \rightarrow ObjectID \\
\hline
\forall s : State \bullet \mathbf{let}\ extent == (stos^{\sim}\,s).extent \bullet \\
\quad uniqueId\,s \notin (\bigcup(\mathrm{ran}\,extent))
\end{array}
$$

The term $path(id, r)$ has a meaning only when r is a relation (or function) and id is an object identifier in the domain of r. When r is a relation the meaning is the relational image of r on id. When r is a function the meaning is $r(id)$.

$$
\begin{array}{l}
\textit{PathMeaning} \\
\hline
value : Term \rightarrowtail Value \\
\hline
\forall t_1, t_2 : Term;\ v : Value \bullet value(path(t_1, t_2)) = v \Leftrightarrow \\
\quad ((\exists\, id : ObjectID;\ r : ObjectID \leftrightarrow ObjectID \bullet \\
\quad\quad value\,t_1 = objectid\,id \wedge value\,t_2 = relation\,r \wedge v = set\,(r(\!|\ \{id\}\ |\!))) \vee \\
\quad (\exists\, id : ObjectID;\ f : ObjectID \rightarrowtail Value \bullet \\
\quad\quad value\,t_1 = objectid\,id \wedge value\,t_2 = function\,f \wedge id \in \mathrm{dom} f \wedge v = f\ id))
\end{array}
$$

The meaning of the *path* term gives us the following useful *dot composition* rule:

THEOREM 5.3.1. *Let obj refer to an object and let assoc be an association of obj, then*

$$x \in obj.assoc \Leftrightarrow obj \mapsto x \in assoc$$

.

The term $the\,s$ has a meaning only when s is a set of identifiers. When the set contains a single element id, the meaning of $the\,s$ is id. In any other case the meaning of $the\,s$ is the undefined value $\perp$.

$$
\begin{array}{l}
\textit{TheMeaning} \\
\hline
value : Term \rightarrowtail Value \\
\hline
\forall t : Term;\ v : Value \bullet value(the(t)) = v \Leftrightarrow \\
\quad ((\exists\, id : ObjectID \bullet value\,t = set\,\{id\} \wedge v = objectid\,id) \vee \\
\quad (\exists\, ids : \mathbb{P}\,ObjectID \bullet \#ids \neq 1 \wedge value\,t = set\,ids \wedge v = \perp))
\end{array}
$$

$$
\begin{array}{l}
\textit{MkRelMeaning} \\
\hline
value : Term \rightarrowtail Value \\
\hline
\forall t_1, t_2 : Term;\ v : Value \bullet value(mkrel(t_1, t_2)) = v \Leftrightarrow \\
\quad (\exists\, id_1, id_2 : ObjectID \bullet value\,t_1 = objectid\,id_1 \wedge value\,t_2 = objectid\,id_2 \wedge \\
\quad\quad v = relation\,\{id_1 \mapsto id_2\})
\end{array}
$$

[1] The term $newobject(c)$ and the function *uniqueId* are not a part of the specification language, only of the implementation language, where they are used to give the meaning of the object creation statement. Their roles will become clear in chapter 7.

UnionMeaning

$$value : Term \rightarrowtail\!\!\!\!\!+ Value$$

$$\begin{aligned}
&\forall t_1, t_2 : Term;\ v : Value \bullet value(union(t_1, t_2)) = v \Leftrightarrow \\
&\quad ((\exists ids_1, ids_2 : \mathbb{P}\, ObjectID \bullet value\, t_1 = set\, ids_1 \land value\, t_2 = set\, ids_2 \land \\
&\qquad v = set\,(ids_1 \cup ids_2)) \lor \\
&\quad (\exists r_1, r_2 : ObjectID \leftrightarrow ObjectID \bullet \\
&\qquad value\, t_1 = relation\, r_1 \land value\, t_2 = relation\, r_2 \land v = relation\,(r_1 \cup r_2)))
\end{aligned}$$

SetMinusMeaning

$$value : Term \rightarrowtail\!\!\!\!\!+ Value$$

$$\begin{aligned}
&\forall t_1, t_2 : Term;\ v : Value \bullet value(setminus(t_1, t_2)) = v \Leftrightarrow \\
&\quad ((\exists ids_1, ids_2 : \mathbb{P}\, ObjectID \bullet value\, t_1 = set\, ids_1 \land value\, t_2 = set\, ids_2 \land \\
&\qquad v = set\,(ids_1 \setminus ids_2)) \lor \\
&\quad (\exists r_1, r_2 : ObjectID \leftrightarrow ObjectID \bullet \\
&\qquad value\, t_1 = relation\, r_1 \land value\, t_2 = relation\, r_2 \land v = relation\,(r_1 \setminus r_2)))
\end{aligned}$$

UpdateMeaning

$$value : Term \rightarrowtail\!\!\!\!\!+ Value$$

$$\begin{aligned}
&\forall t_1, t_2 : Term;\ v : Value \bullet value(update(t_1, t_2)) = v \Leftrightarrow \\
&\quad (\exists r_1, r_2 : ObjectID \leftrightarrow ObjectID \bullet \\
&\qquad value\, t_1 = relation\, r_1 \land value\, t_2 = relation\, r_2 \land v = relation\,(r_1 \oplus r_2))
\end{aligned}$$

CondMeaning

$$\begin{aligned}
&s : State \\
&value : Term \rightarrowtail\!\!\!\!\!+ Value \\
&\mathcal{C} : Formula \rightarrowtail\!\!\!\!\!+ Constraint
\end{aligned}$$

$$\begin{aligned}
&\forall g : Formula;\ t_1, t_2 : Term \bullet \\
&\quad s \in \mathcal{C} g \Rightarrow value(cond(g, t_1, t_2)) = value\, t_1 \land \\
&\quad s \in \mathcal{C}(\neg\, g) \Rightarrow value(cond(g, t_1, t_2)) = value\, t_2
\end{aligned}$$

We now combine the meaning of the individual terms into a single schema.

$$\begin{aligned}
&TermMeaningInState \mathrel{\widehat=} VariableMeaning \land PathMeaning \land TheMeaning \land \\
&\quad MkRelMeaning \land UnionMeaning \land SetMinusMeaning \land \\
&\quad UpdateMeaning \land NewObjectMeaning
\end{aligned}$$

This definition gives the meaning of a term in the context of a specific pair of states. But in many cases it is more useful to have a definition that gives the meaning of a term as a function of pairs of states.

TermMeaning

$$\begin{aligned}
&input : Name \rightarrowtail\!\!\!\!\!+ Value \\
¤tObject : ObjectID \\
&eval : State \times State \rightarrow Term \rightarrow Value
\end{aligned}$$

$$\begin{aligned}
&\forall s, s' : State \bullet \\
&\quad \exists value : Term \rightarrow Value \bullet \\
&\qquad TermMeaningInState \land eval\,(s, s') = value
\end{aligned}$$

Note that because have used the function $\mathcal{C}$ to describe the meaning of the formula that appears in the head of a conditional term, the meaning of terms is well defined only in conjunction with the meaning of formulas which we define below.

5.4. Formulas

Formulas appear in two places in a Booster model: in the precondition and in the postcondition. When a formula appears in the precondition it restricts the set of initial states of the method, and therefore acts as a *constraint* (a set of states). When a formula appears in the postcondition it defines a *relation* between the 'before' (old) values and the 'after' (current) values of the method, and therefore represents a *program*. Accordingly we provide two interpretations for formulas, one that defines the meaning of a formula when it appears as a precondition and one that defines the meaning of a formula when it appears as a postcondition.

An additional detail we need to explain is how we treat negation. The problem is that because we allow for undefined terms, we cannot model negation using set subtraction. For example, the meaning of the formula $x/y = 1$ is a set that contains all the states in which x is equal to y, except for states in which y is zero. Similarly, the meaning of the formula $\neg\ (x/y = 1)$ contains all the states in which x is different from y, *except* for states in which y is zero. Therefore, it is not true that $\mathcal{C}(\neg\ (x/y = 1)) = State \setminus \mathcal{C}(x/y = 1)$.

To solve this problem we eliminate negation using de-Morgan's laws. We push the negation inwards, until we reach the basic formulas. Then we replace each negated basic formula, for example $\neg\ (x = y)$, by an equivalent basic formula: $x \neq y$. This means that we restrict the set of basic formulas only to formulas that have a corresponding negated form, but this is not a problem in practice because all the basic formulas that we need have this property.

LogicalOpsMeaning

$$\mathcal{C} : Formula \nrightarrow Constraint$$
$$\mathcal{P} : Formula \nrightarrow Program$$

$$\begin{aligned}
&\forall p, q : Formula \bullet \\
&\quad \mathcal{C}(false) = \{\} \land \\
&\quad \mathcal{C}(true) = State \land \\
&\quad \mathcal{C}(\neg\ p) = \mathcal{C}(pos(\neg\ p)) \land \\
&\quad \mathcal{C}(p \land q) = \mathcal{C}(p) \cap \mathcal{C}(q) \land \\
&\quad \mathcal{C}(p \lor q) = \mathcal{C}(p) \cup \mathcal{C}(q) \land \\
&\quad \mathcal{C}(p \Rightarrow q) = \mathcal{C}(\neg\ p \lor q) \land \\
&\quad \mathcal{P}(false) = \{\} \land \\
&\quad \mathcal{P}(true) = State \times State \land \\
&\quad \mathcal{P}(\neg\ p) = \mathcal{P}(pos(\neg\ p)) \land \\
&\quad \mathcal{P}(p \land q) = \mathcal{P}(p) \cap \mathcal{P}(q) \land \\
&\quad \mathcal{P}(p \lor q) = \mathcal{P}(p) \cup \mathcal{P}(q) \land \\
&\quad \mathcal{P}(p \Rightarrow q) = \mathcal{P}(\neg\ p \lor q)
\end{aligned}$$

The meaning of a basic formula depends on the values of its terms, which in turn depend on the input and on the current object (all available in the *TermMeaning* schema).

BasicFormulaMeaning

$$
\begin{array}{l}
TermMeaning \\
\mathcal{C} : Formula \pfun Constraint \\
\mathcal{P} : Formula \pfun Program \\
\hline
\forall n : BasicFormula;\ t_1, t_2 : Term \bullet \\
\quad \mathcal{C}(basic(n, t_1, t_2)) = \\
\quad\quad \{s : State \mid (test\, n)(eval(s, s)\, t_1, eval(s, s)\, t_2) = true\} \land \\
\quad \mathcal{P}(basic(n, t_1, t_2)) = \\
\quad\quad \{s, s' : State \mid (test\, n)(eval(s, s')\, t_1, eval(s, s')\, t_2) = true\}
\end{array}
$$

The schema *FormulaMeaning* combines both schemas to form complete the definition of the meaning of formulas:

$FormulaMeaning \mathrel{\widehat{=}} BasicFormulaMeaning \land LogicalOpsMeaning$

The function *test* applies a basic formula to its arguments. Each pair of values is mapped to a boolean to indicate if it satisfies the formula (*true*), does not satisfy the formula (*false*) or if it is undefined ($\perp$).

$Bool ::= true \mid \perp \mid false$

Because the semantics of the basic formulas is obvious we won't specify it formally here. The only thing we would like to make explicit is that no basic formula will accept tuples that contain the undefined value.

$$
\begin{array}{l}
test : BasicFormula \rightarrow (Value \times Value \rightarrow Bool) \\
\hline
\forall f : BasicFormula;\ v_1, v_2 : Value \bullet \\
\quad v_1 = \perp \lor v_2 = \perp \Rightarrow test\, f\, (v_1, v_2) = false
\end{array}
$$

The function *pos* takes a formula and returns a new formula with the same meaning but without negations.

$$
\begin{array}{l}
pos : Formula \rightarrow Formula \\
\hline
\forall p, q : Formula;\ t_1, t_2 : Term;\ n : BasicFormula \bullet \\
\quad pos(true) = true \land \\
\quad pos(false) = false \land \\
\quad pos(\lnot\ true) = false \land \\
\quad pos(\lnot\ false) = true \land \\
\quad pos(\lnot\ (\lnot\ p)) = p \land \\
\quad pos(\lnot\ (basic(n, t_1, t_2))) = basic(negate\, n, t_1, t_2) \land \\
\quad pos(p \land q) = pos(p) \land pos(q) \land \\
\quad pos(p \lor q) = pos(p) \lor pos(q) \land \\
\quad pos(p \Rightarrow q) = pos(p) \Rightarrow pos(q) \land \\
\quad pos(\lnot\ (p \land q)) = pos(\lnot\ p) \lor pos(\lnot\ q) \land \\
\quad pos(\lnot\ (p \lor q)) = pos(\lnot\ p) \land pos(\lnot\ q) \land \\
\quad pos(\lnot\ (p \Rightarrow q)) = pos(p) \land pos(\lnot\ q)
\end{array}
$$

The function *negate* associates each basic formula with another basic formula, that selects all the tuples that are rejected by the original formula.

$$\begin{array}{|l} negate : BasicFormula \to BasicFormula \\ \hline negate\ isMember = isNonMember \\ negate\ isNonMember = isMember \\ negate\ isEqual = isNotEqual \\ negate\ isNotEqual = isEqual \end{array}$$

5.4.1. Properties of $\mathcal{C}$ and $\mathcal{P}$. There is a simple connection between the meaning of a formula as a program and as a constraint. We can always generate the meaning of the formula as a constraint from its meaning as a program by restricting the program to the identity relation:

THEOREM 5.4.1. *For every formula p*

$$\mathcal{C}\,p = dom(\mathcal{P}\,p \cap id)$$

where $id = \{s : State \bullet s \mapsto s\}$.

PROOF. We prove the theorem by induction on the structure of formulas.

$$\begin{array}{ll} \mathcal{C}\,true & \\ = State & \text{definition of } \mathcal{C} \\ = \text{dom}\,id & id \text{ is total} \\ = \text{dom}(id \cap State \times State) & \text{set theory} \\ = \text{dom}(id \cap \mathcal{P}\,true) & \end{array}$$

$$\begin{array}{ll} \mathcal{C}\,false & \\ = \{\} & \text{definition of } \mathcal{C} \\ = \text{dom}\{\} & \text{set theory} \\ = \text{dom}(id \cap \{\}) & \text{set theory} \\ = \text{dom}(id \cap \mathcal{P}\,false) & \end{array}$$

$$\begin{array}{ll} \mathcal{C}(p \wedge q) & \\ = \mathcal{C}\,p \cap \mathcal{C}\,q & \text{definition of } \mathcal{C} \\ = \text{dom}(\mathcal{P}\,p \cap id) \cap \text{dom}(\mathcal{P}\,q \cap id) & \text{induction hypothesis} \\ = \text{dom}(\mathcal{P}\,p \cap \mathcal{P}\,q \cap id) & \text{lemma D.0.6} \\ = \text{dom}(\mathcal{P}\,(p \wedge q) \cap id) & \text{definition of } \mathcal{P} \end{array}$$

$$\begin{array}{ll} \mathcal{C}(p \vee q) & \\ = \mathcal{C}\,p \cup \mathcal{C}\,q & \text{definition of } \mathcal{C} \\ = \text{dom}(\mathcal{P}\,p \cap id) \cup \text{dom}(\mathcal{P}\,q \cap id) & \text{induction hypothesis} \\ = \text{dom}(\mathcal{P}\,p \cap id \cup \mathcal{P}\,q \cap id) & \text{set theory} \\ = \text{dom}((\mathcal{P}\,p \cup \mathcal{P}\,q) \cap id) & \text{set theory} \\ = \text{dom}(\mathcal{P}\,(p \vee q) \cap id) & \text{definition of } \mathcal{P} \end{array}$$

$$\begin{array}{ll} \mathcal{C}(basic(n, t_1, t_2)) & \\ = \{s : State \mid test\,n\,(eval(s,s)\,t_1, eval(s,s)\,t_2) = true\} & \text{definition of } \mathcal{C} \\ = \text{dom}(\{s, s' : State \mid s = s' \wedge test\,n\,(eval(s,s')\,t_1, eval(s,s')\,t_2) = true\}) & \\ & \text{set theory} \\ = \text{dom}(\{s, s' : State \mid test\,n\,(eval(s,s')\,t_1, eval(s,s')\,t_2) = true\} \cap id) & \text{set theory} \end{array}$$

$= \mathrm{dom}(\mathcal{P}(basic(n, t_1, t_2)) \cap id)$

□

5.5. The meaning of methods

In this section we define the meaning of methods by mapping each kind of method to a corresponding relation.

The meaning of a primitive method with precondition *pre* and postcondition *post* is equal to the meaning of the specification statement $(pre \mid post)$[2], restricted to the state space of the model. Therefore if the method starts in a valid model state it must end in a valid model state, thus maintaining the invariants of the model.

PrimitiveMethodMeaning

$Model$
$FormulaMeaning$
$meaning : Method \rightarrow\!\!\!\!\!+ Program$

$$\textbf{let } state == stos(\!| \{ComponentState \mid ModelConstraint\} |\!) \bullet$$
$$\forall PrimitiveMethod \bullet$$
$$meaning(primitive\ \theta PrimitiveMethod) =$$
$$(state \cap \mathcal{C}(pre)) \lhd (\mathcal{P}(post) \rhd state)$$

5.5.1. The guard combinator.

The meaning of a guard combinator is equal to the meaning of its body, constrained to the initial states defined by its guard.

GuardMethodMeaning

$FormulaMeaning$
$meaning : Method \rightarrow\!\!\!\!\!+ Program$

$$\forall guard : Formula;\ body : Method \bullet$$
$$meaning(guardm(guard, body)) = \mathcal{C}(guard) \lhd meaning\ body$$

5.5.2. The **AND** combinator.

The meaning of an `AND` combinator is equal to the intersection of the meanings of its arguments.

AndMethodMeaning

$meaning : Method \rightarrow\!\!\!\!\!+ Program$

$$\forall m_1, m_2 : Method \bullet$$
$$meaning(conj(m_1, m_2)) = meaning\ m_1 \cap meaning\ m_2$$

5.5.3. The **OR** combinator.

The meaning of an `OR` combinator is equal to the union of the first method and a restricted form of the second method. The second method is restricted to the states that are not in the precondition of the first method. This means that if the two methods are deterministic then their `OR` is deterministic as well, even if their preconditions intersect.

OrMethodMeaning

$meaning : Method \rightarrow\!\!\!\!\!+ Program$

$$\forall m_1, m_2 : Method \bullet$$
$$meaning(disj(m_1, m_2)) = meaning\ m_1 \cup \mathrm{dom}\,(meaning\ m_1) \mathbin{⩤} (meaning\ m_2)$$

[2] See also the discussion on the semantics of specification statements in chapter 3.

5.5.4. The `THEN` combinator. The meaning of a `THEN` combinator is equal to the sequential composition of the two methods:

$$\begin{array}{l} \textit{ThenMethodMeaning} \\ \hline \textit{meaning} : \textit{Method} \rightarrow\!\!\!\!\!\!+ \textit{Program} \\ \hline \forall m_1, m_2 : \textit{Method} \bullet \\ \quad \textit{meaning}(\textit{then}(m_1, m_2)) = \textit{meaning}\, m_1 \,;\, \textit{meaning}\, m_2 \end{array}$$

5.5.5. The method-reference combinator. The meaning of a method reference is the result of substituting the reference's path instead of the variable *this* in the referenced method.

$$\begin{array}{l} \textit{MethodRefMeaning} \\ \hline \textit{Model} \\ \textit{meaning} : \textit{Method} \rightarrow\!\!\!\!\!\!+ \textit{Program} \\ \hline \forall \textit{path} : \textit{Term};\ \textit{thisMethod} : \textit{Name};\ \textit{program} : \textit{Program} \bullet \\ \quad \textit{meaning}(\textit{methodref}(\textit{path}, \textit{thisMethod})) = \textit{program} \Leftrightarrow \\ \quad\quad (\exists \textit{thisClass} : \textit{Name};\ \textit{Class};\ \textit{theMethod} : \textit{Method} \bullet \\ \quad\quad\quad \textit{IdentifyMethod} \wedge \textit{meaning}\, \textit{theMethod}[\textit{this} := \textit{path}] = \textit{program}) \end{array}$$

The meaning of methods is the result of conjoining the meaning schemas of each method.

$$\begin{array}{l} \textit{MethodMeaning} \mathrel{\widehat=} \textit{PrimitiveMethodMeaning} \wedge \textit{GuardMethodMeaning} \wedge \\ \quad \textit{AndMethodMeaning} \wedge \textit{OrMethodMeaning} \wedge \\ \quad \textit{ThenMethodMeaning} \wedge \textit{MethodRefMeaning} \end{array}$$

5.6. Models

Finally, we can define the semantics of a model by mapping it to an instance of an abstract data type:

$$\begin{array}{l} \textit{ComponentSemantics} \\ \hline \textit{MethodMeaning} \\ \textit{Model} \\ \textit{ADT} \\ \hline \textit{state} = \textit{stos}(\!|\ \{\textit{ComponentState} \mid \textit{ModelConstraint}\}\ |\!) \\ \forall \textit{thisMethod} : \textit{Name};\ \textit{program} : \textit{Program} \bullet \\ \quad \textit{thisMethod} \mapsto \textit{program} \in \textit{operation} \Leftrightarrow \\ \quad\quad (\exists \textit{thisClass} : \textit{Name};\ \textit{Class};\ m : \textit{Method} \bullet \textit{meaning}\, m = \textit{program}) \end{array}$$

Thus, an operation *op* is an operation of the abstract data type if and only if there exists a class that has a method whose meaning is equal to that of *op*, and they both have the same name.

5.7. Soundness

In this section we prove that the type rules, the model constraints, and the semantics of terms and formulas, are sound. That is, that every well typed term has a value that is a member of the term's type.

THEOREM 5.7.1. *Let t be a term that is well typed in the context of a model and a class, and let s and s' be two valid states of the model. Let currentObject refer to an instance of thisClass. Then the value of t is a member of the values of t's type.*

$$\textit{value}\ t \in \textit{values}(\textit{wtt}\ t)$$

PROOF. Let $\theta Model$ be a model and *thisClass* be the name of a class in the the model. Let s and s' be valid states of the model. Finally, let t be a well typed term.

We prove the theorem by induction on the structure of terms. Assume that t is the term *this*. We may argue as follows:

$value\ this \in value(wtt\ this)$

$\Leftrightarrow value\ this \in value(reference\ thisClass)$ — definition of *wtt*

$\Leftrightarrow values\ this \in objectid(\!|\ classtype\ thisClass\ |\!)$ — definition of *values*

$\Leftrightarrow objectid\ currentObject \in objectid(\!|\ classtype\ thisClass\ |\!)$ — definition of *value*

$\Leftarrow currentObject \in classtype\ thisClass$ — set theory

$\Leftrightarrow true$ — assumption

Now assume that t is a *before* variable of some attribute *thisAttribute*. We argue as follows:

$value(before\ thisAttribute) \in values(wtt(before\ thisAttribute))$

$\Leftrightarrow value(before\ thisAttribute) \in values(functiontype(reference\ thisClass, type))$

definition of *wtt*

$\Leftrightarrow value(before\ thisAttribute) \in function(\!|\ classtype\ n \mathrel{+\!\!\!\rightarrow} (values\ type)\ |\!)$

definition of *values*

$\Leftrightarrow s\ thisAttribute \in function(\!|\ classtype\ n \mathrel{+\!\!\!\rightarrow} (values\ type)\ |\!)$ — definition of *value*

$\Leftrightarrow ((stos^{\sim})\ s).value\ thisAttribute \in classtype\ n \mathrel{+\!\!\!\rightarrow} values\ type$ — property of *stos*

$\Leftrightarrow true$ — *AttributeConstraint*

The case when t is an *after* variable is the same except that we use s' instead of s.

Now assume that t is a *before* variable of some association *thisAssociation*. We argue as follows:

$value(before\ thisAssociation) \in values(wtt(before\ thisAssociation))$

$\Leftrightarrow value(before\ thisAssociation) \in$
$values(relationtype(reference\ thisClass, reference\ target)$ — definition of *wtt*

$\Leftrightarrow value(before\ thisAssociation) \in$
$relation(\!|\ classtype\ thisClass \leftrightarrow classtype\ target\ |\!)$ — definition of *values*

$\Leftrightarrow s\ thisAssociation \in$
$relation(\!|\ classtype\ thisClass \leftrightarrow classtype\ target\ |\!)$ — definition of *value*

$\Leftrightarrow ((stos^{\sim})\ s).link\ thisAssociation \in$
$classtype\ thisClass \leftrightarrow classtype\ target\ |\!)$ — property of *stos*

$\Leftrightarrow true$ — *AssociationConstraint*

The case when t is an *after* association is the same except that we use s' instead of s.

Now assume that t is of the form $mkrel(t_1, t_2)$. By inspecting the definition of *WellTypedMkRel* we see that for $mkrel(t_1, t_2)$ to be well typed, the types of the terms t_1 and t_2 must be class references. Let us therefore assume that the type of t_1 is *reference n* and that the type of t_2 is *reference m*. We can use the induction hypothesis to deduce that t_1 either evaluates to $objectid\ id_1$ or to the undefined value, and t_2 evaluates to $objectid\ id_2$ or to the undefined value, where id_1 is a member of *classtype n* and id_2 is a member of *classtype m*. If either sub term evaluates to the undefined value, the value of the whole term is also undefined which belongs to the values of every type. Otherwise, we can now argue as follows:

$value(mkrel(objectid\ id_1, objectid\ id_2)) \in$
$values(wtt(mkrel(reference\ n, reference\ m)))$

$\Leftrightarrow relation(id_1 \mapsto id_2) \in$
$values(wtt(mkrel(reference\, n, reference\, m)))$ — definition of *value*
$\Leftrightarrow relation(id_1 \mapsto id_2) \in$
$values(relationtype(classtype\, n, classtype\, m))$ — definition of *wtt*
$\Leftrightarrow relation(id_1 \mapsto id_2) \in$
$relation(\!| classtype\, n \leftrightarrow classtype\, m |\!)$ — definition of *values*
$\Leftrightarrow id_1 \mapsto id_2 \in classtype\, n \leftrightarrow classtype\, m$ — set theory
$\Leftrightarrow id_1 \in classtype\, n \wedge id_2 \in classtype\, m$ — set theory
$\Leftrightarrow true$

The case when t is a *path* is very similar to the one above, so we won't describe it here.

Assume that t is the term $the(t_1)$. By inspecting *WellTypedThe* we see that the type of t_1 must be a reference to a set of identifiers of some class. Let us therefore assume that $wtt(t_1) = settype(reference\, n)$. From the induction hypothesis we can deduce[3] that t_1 evaluates to a set, *ids*, of object identifiers that are members of *classtype n*. We may argue as follows:

$value(the(set\, ids)) \in values(wtt(the(set\, ids)))$
$\Leftrightarrow value(the(set\, ids)) \in values(reference\, n)$ — definition of *wtt*
$\Leftrightarrow value(the(set\, ids)) \in objectid(\!| classtype\, n |\!) \cup \{\perp\}$ — definition of *values*

Now there are two options. Either the set *ids* contains just one member *id*, or it does not. In the first case we may argue as follows:

$value(the(set\, \{id\})) \in objectid(\!| classtype\, n |\!)$
$\Leftrightarrow objectid\, id \in objectid(\!| classtype\, n |\!) \cup \{\perp\}$ — definition of *value*
$\Leftrightarrow id \in classtype\, n$ — set theory
$\Leftrightarrow true$

In the second case we may argue as follows:

$value(the(set\, ids)) \in objectid(\!| classtype\, n |\!)$
$\Leftrightarrow \perp \in objectid(\!| classtype\, n |\!) \cup \{\perp\}$ — definition of *value*
$\Leftrightarrow true$

□

5.8. Modeling the case study

To illustrate the formal semantics we demonstrate it on a model of the library system. We begin by defining the names of all the entities in the model. These include the names of classes, attributes, associations, and methods:

$$Reader, Book, Library, ReaderBooks, ReaderLibraries,$$
$$BookReader, BookLibrary, LibraryBooks, LibraryReaders,$$
$$ReaderBorrow, BookTransfer, LibraryAdd : Name$$

[3] if t_1 evaluates to an undefined value then $the(t_1)$ also evaluates to an undefined value.

5.8.1. The structure of the library system. The library system model consists of the classes: *ReaderClass*, *BookClass*, and *LibraryClass*. The particular structure of each class is defined in its corresponding class model schema (for example, the structure of *LibraryClass* is defined in the *LibraryClassModel* schema).

LibrarySystemModel

$Model$
$ReaderClassModel$
$BookClassModel$
$LibraryClassModel$

$class = \{Reader \mapsto ReaderClass,$
$Book \mapsto BookClass,$
$Library \mapsto LibraryClass\}$

Each schema that models a class has a similar structure. We begin with the schema *ReaderClassModel* which defines the structure of the *ReaderClass* class.

The *Reader* class has two associations: the association *ReaderBooks* which references a reader's books, and the association *ReaderLibraries* which references the libraries to which a reader is subscribed.

ReaderClassModel

$Class$
$ReaderClass : Class$
$ReaderBooksAssociationModel$
$ReaderLibrariesAssociationModel$
$ReaderBorrowMethodModel$

$ReaderClass = \theta Class$
$attribute = \{\}$
$association = \{ReaderBooks \mapsto ReaderBooksAssociation,$
$ReaderLibraries \mapsto ReaderLibrariesAssociation\}$
$method = \{ReaderBorrow \mapsto primitive\ ReaderBorrowMethod\}$

The association *ReaderBooks* is a many-to-optional association, whose mirror is the association *BookReader*. Here we only see the *many* part of the association. To see that the mirror is *optional* we must look at the definition of *BookReader*.

ReaderBooksAssociationModel

$Association$
$ReaderBooksAssociation : Association$

$ReaderBooksAssociation = \theta Association$
$target = Book$
$multiplicity = many$
$mirror = BookReader$

The association *ReaderLibraries* is a many-to-many association, whose mirror is the association *LibraryReaders*.

ReaderLibrariesAssociationModel

Association
$ReaderLibrariesAssociation : Association$

$ReaderLibrariesAssociation = \theta Association$
$target = Library$
$multiplicity = many$
$mirror = LibraryReaders$

BookClassModel

Class
$BookClass : Class$
BookReaderAssociationModel
BookLibraryAssociationModel
BookTransferMethodModel

$BookClass = \theta Class$
$attribute = \{\}$
$association = \{BookReader \mapsto BookReaderAssociation,$
$BookLibrary \mapsto BookLibraryAssociation\}$
$method = \{BookTransfer \mapsto primitive\ BookTransferMethod\}$

BookReaderAssociationModel

Association
$BookReaderAssociation : Association$

$BookReaderAssociation = \theta Association$
$target = Reader$
$multiplicity = optional$
$mirror = ReaderBooks$

BookLibraryAssociationModel

Association
$BookLibraryAssociation : Association$

$BookLibraryAssociation = \theta Association$
$target = Library$
$multiplicity = mandatory$
$mirror = LibraryBooks$

LibraryClassModel

$$
\begin{array}{l}
Class \\
LibraryClass : Class \\
LibraryReadersAssociationModel \\
LibraryBooksAssociationModel \\
LibraryAddMethodModel \\
\hline
LibraryClass = \theta Class \\
attribute = \{\} \\
association = \{LibraryReaders \mapsto LibraryReadersAssociation, \\
LibraryBooks \mapsto LibraryBooksAssociation\} \\
method = \{LibraryAdd \mapsto primitive\ LibraryAddMethod\}
\end{array}
$$

LibraryReadersAssociationModel

$$
\begin{array}{l}
Association \\
LibraryReadersAssociation : Association \\
\hline
LibraryReadersAssociation = \theta Association \\
target = Reader \\
multiplicity = many \\
mirror = ReaderLibraries
\end{array}
$$

LibraryBooksAssociationModel

$$
\begin{array}{l}
Association \\
LibraryBooksAssociation : Association \\
\hline
LibraryBooksAssociation = \theta Association \\
target = Book \\
multiplicity = many \\
mirror = BookLibrary
\end{array}
$$

We can check that *LibrarySystemModel* satisfies the constraints of *Model*. There are three classes in the range of *class*, and for each we have to show that every association in the class refers to a target that is an association in another class and that their mirror fields are compatible. For the first association of the class *ReaderClass* we may argue as follows:

$ReaderBooksAssociation.target \in \mathrm{dom}\, class$

$\Leftrightarrow Book \in \mathrm{dom}\, class$ — definition of *ReaderBooksAssociation*

$\Leftrightarrow Book \in \{Reader, Book, Library\}$ — definition of *class*

$\Leftrightarrow true$

And similarly,

$ReaderBooksAssociation.mirror \in$
$\quad \mathrm{dom}(class\ ReaderBooksAssociation.target).association)$

$\Leftrightarrow BookReader \in \mathrm{dom}(class\ ReaderBooksAssociation.target).association)$
— definition of *ReaderBooksAssociation*

$\Leftrightarrow BookReader \in \mathrm{dom}(class\ Book).association$
— definition of *ReaderBooksAssociation*

$\Leftrightarrow BookReader \in \mathrm{dom}\, BookClass.association$ — definition of *class*

$\Leftrightarrow BookReader \in \{BookReader, BookLibrary\}$ definition of *BookClass*

$\Leftrightarrow true$

We may argue in a similar way to show that the other associations satisfy the constraints of *Model*.

5.8.2. The behavior of the library system. We use the following schema to describe the method `ReaderBorrow` of the reader class. The green operators make the syntactic description of the pre and postconditions easier to read.

ReaderBorrowMethodModel

$PrimitiveMethod$
$ReaderBorrowMethod : PrimitiveMethod$

$ReaderBorrowMethod = \theta PrimitiveMethod$
$pre = Book_in.BookReader = \{\}$
$post = Book_in \in this.ReaderBooks$

BookTransferMethodModel

$PrimitiveMethod$
$BookTransferMethod : PrimitiveMethod$

$BookTransferMethod = \theta PrimitiveMethod$
$pre = true$
$post = this.BookLibrary = Library_in$

LibraryAddMethodModel

$PrimitiveMethod$
$LibraryAddMethod : PrimitiveMethod$

$LibraryAddMethod = \theta PrimitiveMethod$
$pre = true$
$post = Reader_in \in this.LibraryReaders$

Earlier we said that the meaning of an operation in Booster depends not only on the definitions of its pre and postconditions but also on the fact that it must respect the constraints in the model. In particular, *AssociationConstraint* requires the relation of every association to be the inverse of its mirror.

We can demonstrate this property by observing the effect of *AssociationConstraint* on the meaning of the method *LibraryAdd*. The meaning of this method is, according to *MethodMeaning*,

$$states \lhd \mathcal{P}(Reader_in \in this.LibraryReaders) \rhd states$$

where $states = \{ComponentState \mid ModelConstraint\}$.

From the definition of $\mathcal{P}$ and $\in$ we can see that the unconstrained postcondition includes all the final states in which the relation that is bound to *LibraryReaders* contains the pair $this \mapsto Reader_in$:

$\mathcal{P}(Reader_in \in this.LibraryReaders)$
$\quad = \{s, s' : State \mid this \mapsto input\, Reader_in \in s'.link\, LibraryReaders\}$

However, this program is constrained to states that satisfy, among other constraints:

$$s.link\, LibraryReaders = (s.link\, ReaderLibraries)^{\sim}$$

Therefore *LibraryAdd* does not only specify that *Reader_in* should be a member of the readers of this library, but also that *ReaderLibraries* must be equal to the inverse of *LibraryReaders*.

5.9. Related work

We have presented a semantics for object models that addresses association constraints (which we interpret as global invariants on the state of the model) and operation specifications.

Although others have identified the shortcomings of semantic approaches that do not address this constraint information—[**57**] gives a particularly good account—we are not aware of another approach in which it has been successfully incorporated.

The potential impact of global constraints, and the issue of aliasing, is addressed in an early attempt at object-oriented modeling using the Z notation [**26**], in which the author explores the use of relations to capture reference information (although rejects this for modeling purposes) and observes that the specification of an operation may address attributes and associations outside the scope of the current class.

However, despite the importance of associations, and perhaps because most of the research on the formal foundations of object oriented systems is focused on programming, many authors have suggested a semantics for models in which associations are treated as attributes of the source class, and classes are then treated as self-contained components. The model semantics is then presented simply as a combination of class semantics, with the semantics of each class determined separately.

While this is consistent with the view of models as collections of classes, and that of classes as implementations of abstract data types:

> *A class—you may have heard this quite a few times by now—is an implementation of an abstract data type, whether formally specified or (as in many cases) just implicitly understood.* [**43**]

the resulting semantics—although appropriate for programming implementation, where other logics and tools may be applied— proves inadequate for the analysis of object models; we cannot even express the requirement that each reference should point to an existing object of the appropriate class.

Another example is the work of Abadi and Cardelli [**3**]. In this work the authors develop the σ-calculus— a mathematical theory that explains the mechanisms that underlie object oriented programming, in particular the semantics of dynamic method invocations. However the σ-calculus model has no concept of specifications or refinement, and even more important to our case, it does not deal with references. Like many other important texts on object oriented programming, the authors claim that "Objects form natural data abstraction boundaries" and that "objects [are] behaviorally autonomous". This point of view may be true when we consider the operation of methods in a programming language — to understand the effect of a single method (indeed, all the examples in the book are of single objects) on the state of the system it is enough to read the source code of the method. However, when we consider the effects of methods from the perspective of the system's specification we find that this is not the case.

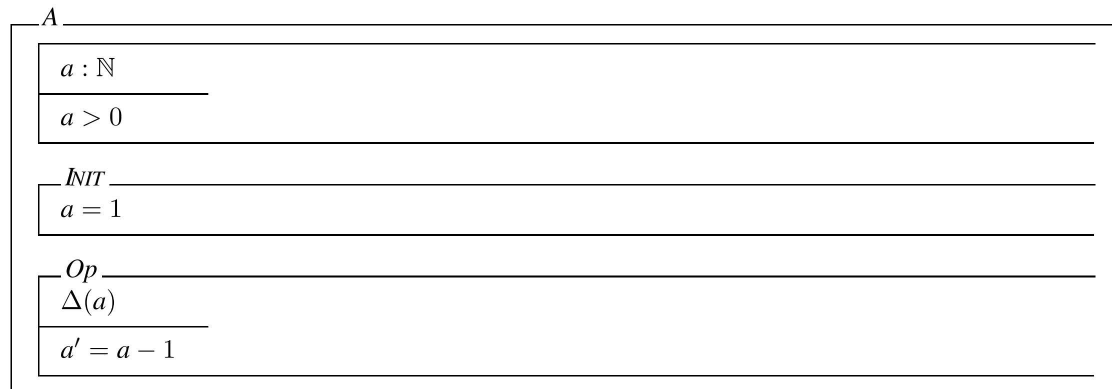
This point is acknowledged in the design of the Unified Modeling Language (UML): in class diagrams, modelers are encouraged to distinguish between attributes and associations; associations are considered at the same level as classes; and the constraint language OCL, now a fundamental component of the UML, is explicitly intended for the description of constraints involving attributes and associations from different classes.

5.9.1. Object-Z. The problems that occur when we try to import ideas from a programming language into a specification language may be demonstrated by the Object-Z [**53**] specification language.

The Object-Z specification language extends Z with classes and objects and offers full support for reference based semantics. Object-Z is similar to Booster in a number of areas: both have an underlying reference semantics, both have an explicit notion of classes, and both use combinators to build methods out of other methods.

However the semantics of Object-Z is different from the semantics of Booster, in particular with regards to the treatment of global constraints. As we shall see, the programmatic flavor of Object-Z makes it difficult to use Object-Z to reason about object models that involve global constraints.

Object-Z extends Z with an explicit description of classes. A class is a named box that contains three nested components: a schema that describes the state of the class, an initialization schema, and zero or more operation schemas.



The initialization schema has no declaration part. It defines the possible states in which an object of the class may be before any operation was applied to the object. In the example above every object of class A begins in a state in which a is equal to 1.

Each class has a set of operations that are defined using special operation schemas. Each operation schema has a delta list which specifies which variables of the class the operation may change, and a predicate which specifies the nature of the change.

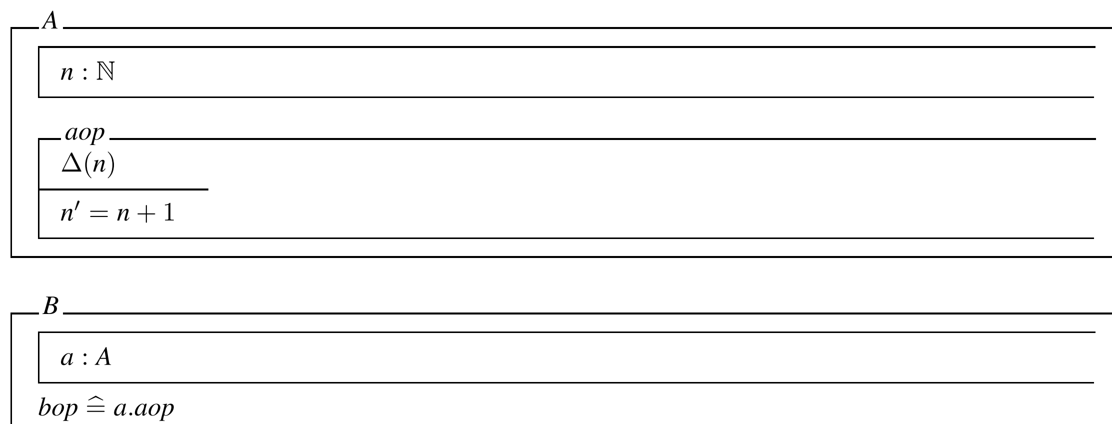
Together with the initialization schema, the operations of the class define all the states in which an object of the class may be. Even though it is not mentioned explicitly we assume that the permissible states of the class are also restricted to satisfy the predicate in the class state schema.

In Z, an operation is specified as a schema that defines the required relation between the initial and final states of the operation. For states that fall outside the operation's precondition, the operation is unspecified and therefore in such states the implementation is free to behave as it pleases.

In contrast, the operations of classes in Object-Z have a blocking semantics. An operation is available for execution only in the states that satisfy its precondition. In any other state the operation is *blocked*.

Object-Z provides operators, which like Booster combinators, make it possible to compose operations in various ways: operation promotion, conjunction, parallel composition, choice, and sequential composition.

Operation promotion makes it possible to use an operation of an object that is part of the state of the current class. For example,



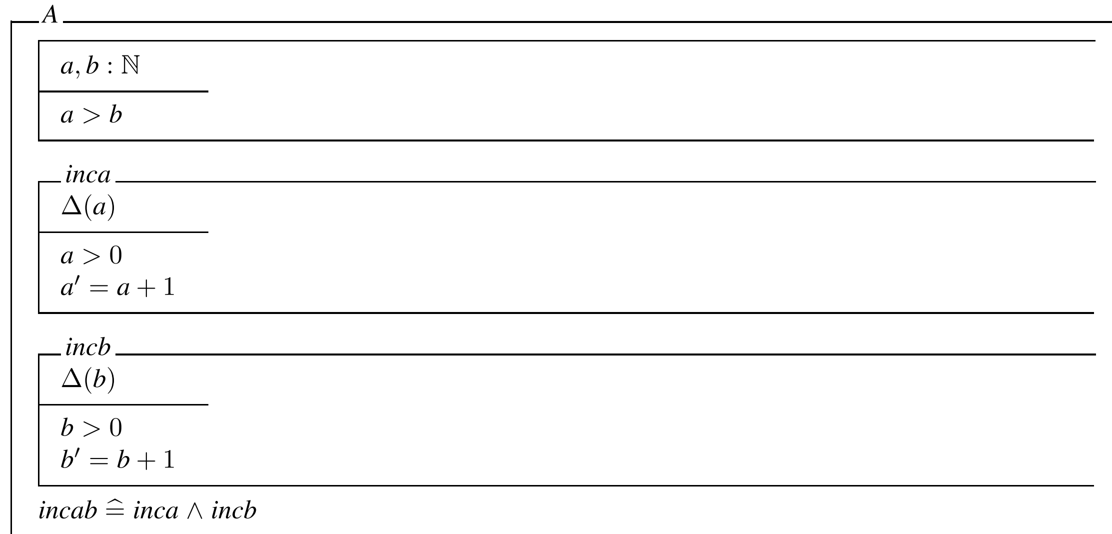
B

$a : A$

$bop \mathrel{\widehat{=}} a.aop$

The operation *bop* denotes an operation which models the application of *aop* to *a*.

Operation conjunction makes it possible to apply two operations at once. For example, in the following Object-Z model, the operation *incab* increments both *a* and *b* provided that both of them are positive:



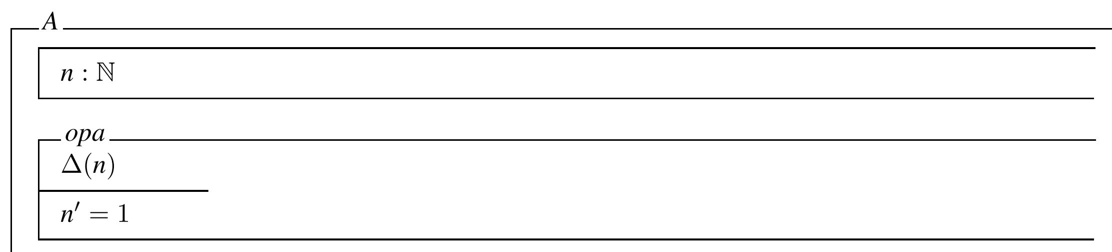
In general, conjoining two operations, written $op_1 \land op_2$, yields a new operation whose predicate is the conjunction of the predicates of its arguments and whose delta list is the union of the delta lists of its arguments.

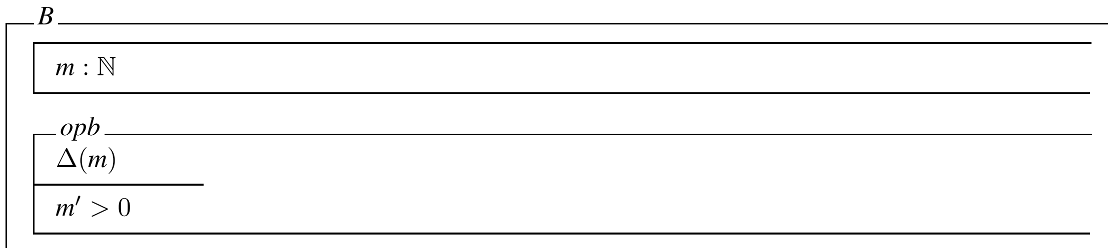
Parallel composition, written $op_1 \parallel op_2$, is a special form of conjunction in which the output variables of the first operation and the input variables of the second operation are equated and hidden. The effect is as if the first operation has communicated its output to the input of the second operation.

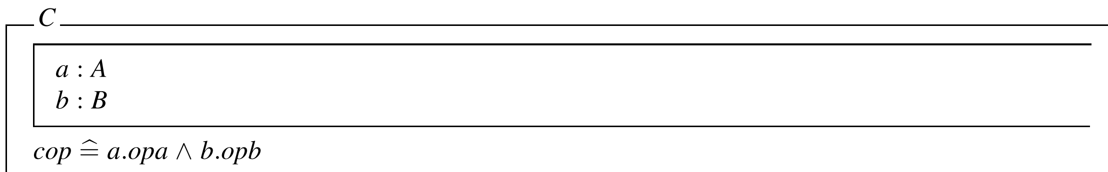
A *choice* between two operations, written, $op_1 [] op_2$ creates an operation by disjoining the predicates of its arguments. The delta list is the union of the delta list of its arguments.

The *sequential composition* of two operations, written $op_1 \mathbin{⨟} op_2$ is an atomic transaction that is enabled only in the states in which the first operation establishes the precondition of the second operation. Similarly to parallel composition, the output variables of the first operation are equated to the input variables of the second operations and then hidden.

So far the description of the language seems innocent, and indeed quite similar to Booster. However, a closer inspection reveals many problems. Most of the problems occur because the semantics insists that methods can change only the state of their class. Consider for example the following model:

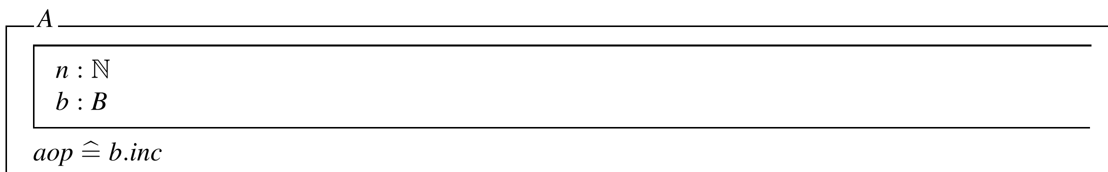


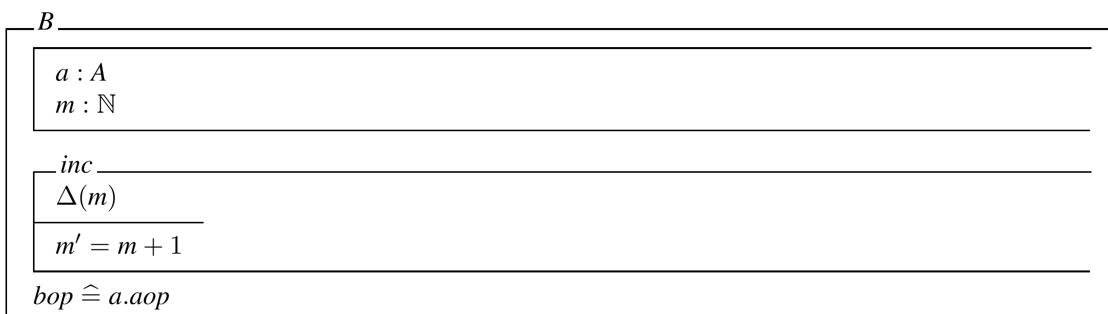




How can we reason about the operation *cop*? The meaning of operation conjunction is the conjunction of the predicate part of the arguments and the union of their delta list. However, if we conjoin the bodies of *opa* and *opb* we get an operation that changes the state of other classes, which is illegal. Therefore we cannot reduce $a.opa \wedge b.opb$ to a more simple form. So what is the meaning of this expression? how can we analyze it? For example, how can we find out the precondition of this operation?

Consider another example:





Assume that we have arranged for a and b to refer to each other. Now when we apply the operation *bop* it calls the operation $a.aop$ which in turns calls the operation $b.inc$ which modifies the state of b. But according to the definition of the language this is illegal because, as we read in page 34 of [**53**]:

> The operation $a.Op$ does not change any variables of the class in which it is defined.

Another example is provided in section 2.3, page 36 of [**53**]:

B

$$\begin{array}{l} a : A \\ y : \mathbb{N} \\ \hline a.x = y \end{array}$$

$IncrementY$

$$\begin{array}{l} \Delta(y) \\ \hline y' = y + 1 \end{array}$$

According to the definition of the language, the operation *IncrementY* can never be applied by itself, because it increments y but y must be equal to $a.x$ and the operation cannot modify the state of another object.

It is true that the semantics of the language protects itself from committing the sin of changing the state of other classes, but in fact all it does is to throw the problem at the door of the specifier.

In every case where an object uses the attributes of another object we must examine the operations of both classes and find out when they block. If we need to change the state of both objects we must create a new operation that performs the change indirectly through a composition of operations from each class (and in many cases we would have to write these operations specifically for the purpose of this particular operation).

This makes writing specifications in which attributes of several classes must be modified in a single transaction, unnecessarily difficult.

The source of these difficulties is the belief that objects are abstract data types — that they are independent entities that can be understood in isolation. At the beginning of section 2.3 in page 36 we read that:

> A modular specification is one which comprises a number of separate parts, each of which can be understood in isolation. For example, an object-oriented specification is modular due to the structure provided by its classes.

But the section that follows this quote presents a specification in which the invariant of one class uses an attribute of another class. Is it really possible to understand such classes in isolation?

The answer is No. Indeed, we are not the first to point out this problem. It was initially recognized in

> ... *encapsulation is not sufficient to render a system fully modular. That is, the operations defined in a class may prescribe the invocation on one of the referenced objects. As such, the meaning of such operations is not confined to the referencing object—it is not modular.*
>
> [**25**]

and again in

> ... *Object-Z allows coupling constraints between classes which, on the one hand, facilitate specification at a high level of abstraction, but, on the other hand, make class refinement non-compositional.*
>
> [**41**]

However, the suggested solution is to transform and complete the specification, losing the abstraction, and much of the value of the model. Yet without this, Object-Z descriptions that include class invariants such as

$$\forall s : supermarkets \bullet \\ \quad \mathrm{dom}(s.database.itemRec) \subseteq \mathrm{dom}(warehouse.stock)$$

[**21**]

do not have a formal semantics adequate for the analysis of method specifications in the context of model constraints; neither do they fully support the semantic comparison of two versions of the same model.

5.9.2. Z and Booster. The semantics of Booster's primitive methods is similar to the semantics of Z operation schemas: a Z operation schema is a relation between the current and the after states of the operation. This is equivalent to the postcondition section of a Booster primitive method. In addition, the Z schema may contain constraints that apply only to the initial states of the operation. This corresponds to the precondition of the primitive method, except that in Z the precondition is embedded in the schema's predicate. However the precondition may be extracted (by existentially quantifying the after state) in which case the meaning of the two notations becomes even closer.

Alternatively we can transform any primitive Booster method $(pre \mid post)$ into the equivalent method $(true \mid pre_0 \wedge post)$ — where pre_0 means that we have replaced all the variables in *pre* by their old equivalents — in which all the information appears in the (relational) postcondition. Except for the different conventions for naming the before and after states this form is identical to a Z operation schema.

However the similarities between Z and Booster are deeper than the similarities between the semantics of the operations.

Z encourages a specification style in which the system is modeled as a single sequential entity with operations that may potentially affect any part of the model. Even though we often use schemas to break down the description of the state and the operations into smaller parts, they are eventually put together. This makes it possible to check that the separate concerns work together and do not conflict with each other.

Even though such a style is bad for designing and implementing systems, it is essential for their specifications. The reason is that a specification must often describe global constraints that must be respected by all the operations in the system. It is essential to describe such constraints as concisely and simply as possible or otherwise it will be very difficult to analyze the specification.

By making the global constraints an invariant of the system we make it clear that the constraint must be respected by every operation of the system.

It is important to understand that when we combine the schema that describes the effect of an operation with a schema that describes the valid states of the system, the constraints on the state and the description of the operation are conjoined so that the operation must by definition respect all the global invariants of the model. Any states in which the description of the operation conflicts with the invariants are therefore excluded from the relation described by the operation schema.

An important step in the development of a Z model is to make sure that all the interactions between the constraints in the model and the description of the operations yield the expected constraints.

This style of specification is also a fundamental principle of the Booster language. This is the reason why the semantics of methods has a global effect on the system. Methods in Booster are specifications, not operations. Therefore they have no choice but to respect the invariants of the model.

5.9.3. Alloy and Booster. Alloy [**35**] is an abstract version of Z with additional support for modeling objects. Alloy models are analyzed using a model checker. Like Z, Alloy does not insist on a particular logic (most model checkers use temporal logic) and it does not assume an underlying state machine model (like for example the labeled transition systems of FDR [**9**] or the transition automata of Spin [**30**] and NuSMV [**12**]). Like Z, we can use Alloy to reason not only about software systems but also about discrete mathematics in general.

Because we build Alloy models in order to model check them, Alloy is not as expressive as Z. For example, the data types in Alloy are relations over atoms (atoms are Alloy's equivalent of Z's basic types) but the relations are always first order. We cannot create relations of relations or sets of sets.

In addition, unlike Z but very much like Booster, Alloy supports the modeling of object systems with reference semantics. Alloy's *signature* construct is very similar to a Booster class. It introduces a new

basic type to the model that we can associate with attributes. For example, here is an Alloy model of the library system:

```
sig Library {
  readers : set Reader,
  books : set Book
}

sig Book {
  reader : lone Reader,
  library : Library
}

sig Reader {
  books : set Book,
  libraries : set Library
 }
```

The semantics of these declarations is very close to the Booster semantics. Each attribute is a relation between the signature in which it is embedded and the signature that denotes its type. The keyword `set` is equivalent to Booster's set attribute, the keyword `lone` is equivalent to Booster's optional attribute, and a plain type denotes (like in Booster) a mandatory attribute.

Alloy does not provide bidirectional associations but these can be specified as facts in a straightforward way:

```
fact
{
  readers = ~libraries
  books = ~library
  books = ~reader
}
```

Alloy's relational model is more simple than Booster's because Booster distinguishes between attributes and associations (which are function and relations) and between other data types (like object identifiers, integers, and so on). In contrast, in Alloy every value is a relation. A set is a relation of arity 1 and a scalar is a relation of arity 0.

However, even though using the Alloy semantics in Booster simplifies the description of navigation, it makes the description of simple operations more difficult to understand. For example, division between integers must now be described in terms of division between sets (relations) of integers. It also means that many combinations that do not make sense in Booster are now valid according to the semantics of the language. For example attributes in Alloy are first class entities. we can reason about them without applying them to a particular object, but this makes no sense for Booster models.

The Alloy model we have described above is sufficient to analyze the static structure of systems. However, if we want to model operations we must introduce an explicit representation of state. There are many ways of introducing such a notion of state. One way for example is to create a state signature that holds the attributes of all the classes, much as the B model of the library system that we have described in the discussion section of chapter 2. However such a model destroys the conceptual partitioning of the problem domain and makes the description of invariants and of the operations cumbersome and difficult to understand. Instead, we can keep the original classes, but turn every attribute into a function that maps the value of the attribute to a particular state:

```
sig State {}
```

```
sig Library {
  readers : set Reader -> one State,
  books : set Book -> one State
}

sig Book {
  reader : Reader lone -> one State,
  library : Library -> one State
}

sig Reader {
  books : set Book -> one State,
  libraries : set Library -> one State
 }

pred inv[s : State] {
  readers . s = ~(libraries . s)
  books . s = ~(library . s)
  reader . s = ~(books . s)
}
```

To refer to the value of an attribute in a particular state we have to join the attribute with the state. For example if *s*0 and *s* are the before and after states of an operation we may write *obj*.*attr*.*s*0 to refer to the old value and *obj*.*attr*.*s* to refer to the new value.

We can now describe operations using Alloy's predicate functions. The predicate takes as arguments the before and after states, and defines the meaning of the operation by describing a relation between them.

It is relatively easy to map Booster's method specifications into corresponding Alloy predicates. To model a primitive method we conjoin the precondition and the invariant on the before state (which we achieve by joining each attribute with *s*0) to the postcondition and to the invariant (this time on the after state). This is the semantics of primitive methods exactly as we have defined it in this chapter.

To model a method combinator we may first convert each argument to the corresponding Alloy predicate function, and then combine the arguments according to the nature of the combinator. For example, if *A* is the meta function that converts Booster methods into Alloy predicates, then

```
 A(guard -> m1) = pred m[s,s' : State]
                    { A(guard)[s] & A(m1)[s,s'] }

    A(m1 OR m2) = pred m[s,s' : State]
                    { A(m1)[s,s'] or not A(pre(m1)) & A(m2)[s,s'] }

   A(m1 AND m2) = pred m[s,s' : State]
                    { A(m1)[s,s'] & A(m2)[s,s'] }
```

and so on.

Because Alloy and Booster have such a similar semantics it is easy to use Alloy to analyze Booster models. Indeed it may even be possible to perform the translation from Booster to Alloy automatically (by using for example abstract interpretation [**13**] to replace the detailed data types of Booster— integers, strings and so on — by domains with a much smaller size).

What would we gain by translating Booster models to Alloy? First, we will have another complementary technique to analyze the specification. Using Alloy we can ask many questions about the

specification (and get answers!) that we cannot ask with Booster. For example, in the case of the library system we may ask the following questions:

- Must every book be have a reader?

```
pred SomeBooksAreNotBorrowed[] {
  some b : Book | b.reader = none
}
```

- Can the same book be located in more than one library?

```
assert NoBookIsHeldByMoreThanOneLibrary {
  all  x, y : Library | x != y => x.books & y.books = none
}
```

- Does every book that is borrowed by a reader belong to one of the reader's libraries?

```
assert BooksMustBeBorrowedOnlyFromMyLibraries {
  Reader <: books in libraries . books
}
```

- Can the same book be borrowed by more than one reader?

```
assert BookIsBorrowedByAtMostOneReader {
  all x, y : Reader | x != y => x.books & y.books = none
}
```

Second, we may be able to validate that the Booster compiler is correct by translating Booster's guarded command implementation of each method to an equivalent Alloy predicate and then using Alloy to check that the relation that was defined by the implementation refines the relation that was defined by the specification.

To summarize, Alloy and Booster share the same object model and many other ideas (since they are both heavily influenced by Z), but they also have fundamental differences that arise from the different purposes for which they were created. Alloy models are models in the sense of traditional engineering models — abstract constructions we create in order to reason about. Whereas Booster models are specifications — constructions that we must follow to the letter in order to create the right system.

Both Alloy and Booster have developed the same underlying model for objects, which in both cases, has proven to be very effective in describing and reasoning about objects. This may indicate that even though it is less intuitive than the traditional object model, representing attributes as relations between the unique identifiers of the objects and their values captures the essence of objects better than the traditional model.

5.10. Discussion

The ideas in this chapter were first described in [**15**]. However, even though that paper described the essential principles of Booster's semantics, it left out many important details. It did not present the semantics of method combinators and it did not describe how the syntactic definition of the model (which was introduced in [**22**]) is mapped to the semantics. In this chapter we have fixed these deficiencies by giving a formal syntax to Booster, and by mapping the formal syntax to the semantic definition of the language.

The specification statement that we use in Booster does not include a change list. A change list makes sense when we follow a top down approach (as for example is the case in the refinement calculus) because it defines the final specification right at the start and this includes which entities may change and which entities must remain fixed. However, when we work in a bottom up style we would like to create building blocks that can be combined as easily as possible with other building blocks. It is therefore better to only specify which variables are changed and leave the other variables unconstrained.

We have chosen to construct the state space of Booster models by breaking the encapsulation of objects as entities that own their attributes. Instead of mapping each object identifier to a function that associates the names of the object's attributes to their values, we map each attribute name to a function that associates each object identifiers to the value of the attribute for this particular object.

The problem with the standard representation is that it hides the fact that, because objects have reference semantics, attributes are global entities, and therefore changes to an attribute of one object may affect the integrity of other objects (as can be seen for example in the case of association invariants).

The standard representation is a model of the organization of objects in most object oriented programming languages, but it is not necessarily a suitable model for object oriented specification languages.

In contrast, the approach that we describe here has two important properties that make it more suitable for the specification of object based systems. First, it provides a simple way to model associations. An association between two attributes is described by asserting that their corresponding functions are inverses of each other. Second, as we shall see in chapter 7, the model offers a simple weakest precondition rule for assignment to attributes, which deals correctly with aliasing that may be caused by object references. Finally, the model provides a simple way to understand formulas and statements that use old variables. For example, the formula $a_0.b = e$ means that the new (after) value of the function b applied to the old (before) value of a must be equal to e, while the formula $a.b_0 = e$ means that the old (before) value of the function b applied to the new (after) value of a must be equal to e.

This model is similar to the theory used by the Simplify theorem prover [**18**] to describe updates and accesses to object fields. However, there the description is given as an axiomatic definition, not as a weakest precondition law:

> Third is the theory of maps, which contains the two functions select and store and the two axioms:
>
> $$(\forall\, a, i, x : select(store(a, i, x), i) = x)$$
> $$(\forall\, a, i, j, x : i \neq j \Rightarrow select(store(a, i, x), j) = select(a, j))$$
>
> These are called the unit and non-unit select-of-store axioms respectively. In our applications to program checking, maps are used to represent arrays, sets, and object fields, for example.[**18**, page 5]

CHAPTER 6

Model completion

Completion is the process by which we augment the postcondition of a method with formulas that ensure that the program that we generate from the postcondition will have a specific effect, without modifying the semantics of the original (uncompleted) postcondition. Completion exploits the fact that we can add to a predicate p any predicate that is implied by p without changing p's meaning. In this chapter we formalize this idea and show how it can be used to justify Booster's automatic maintenance of association invariants.

The semantics of Booster takes into account not only the pre- and postconditions of each method but also the invariants that ensure the integrity of the associations in the model. The problem is that this information is implicit, and as a result it cannot be used by the code generator. For example, assume that `ab` and `ba` are two ends of a many-to-many association and we want to add `b` to `a.ab`:

```
CLASS A
  ATTRIBUTES
    ab : SET(B.ba)
  METHODS
    M(true | b : a.ab)
END
CLASS B
  ATTRIBUTES
    ba : SET(A.ab)
  ...
END
```

The model's invariant implies that `a` must be a member of `b.ba`. But because the postcondition of `M` contains only the formula `b: a.ab`, the Booster compiler generates the program[1]

```
add(a.ab, b)
```

which adds `b` to `a.ab`, but does not add `a` to `b.ba`. However, because the compiler insists on maintaining the invariant, the requirement `a : b.ba` is pushed to the precondition. Since the invariant holds immediately before the method is executed, the formula `a : b.ba` is equivalent (in the precondition) to the formula `b : a.ab`. The result is a specification that was strengthened to the point of being useless:

```
CLASS A
  ATTRIBUTES
    ab : SET(B.ba)
  METHODS
    M(b : a.ab | b : a.ab)
...
```

The only time we may apply `M` is when its goal has already been achieved.

In contrast, if we add the formula `a : b.ba` to the postcondition,

[1] We will define the meaning of programs in the next chapter, for the moment it is enough to know that $add(s, e)$ adds e to the set s, and that $remove(s, e)$ removes e from s.

```
CLASS A
  ATTRIBUTES
    ab : SET(B.ba)
  METHODS
    M(true | b : a.ab & a : b.ba)
...
```

then the meaning of the formula remains the same, but the generated code updates both ends of the association:

```
add(a.ab, b) ; add(b.ba, a)
```

The result is a program that achieves the postcondition and maintains the invariant from any initial state.

By transforming the postcondition into a version that includes additional information, we made it possible for the compiler to use this extra information to generate a better program.

6.1. Revealing information to the code generator

The idea that makes it possible to perform this kind of transformation in general is the fact that if $p \Rightarrow q$ then $p = p \wedge q$. That is, we can add to a formula information that we already know, without changing its meaning.

This simple result means that given a model and a method $m(p \mid q)$, we can transform the model into one in which m is now of the form $m(p \mid q \wedge r)$ where r is implied by the conjunction of q and the model's semantics. This transformation makes r available to the code generator without changing the meaning of the method.

This rule justifies the transformations of the postconditions that are used in the treatment of the association invariants. Consider for example the most simple case of a many-to-many transformation. We have two relations: ab and ba, that are inverses of each other: $ab = ba^{\sim}$, and we have a postcondition that intends to insert a new pair into ab:

$$x \mapsto y \in ab$$

We may now conjoin the association invariant to the postcondition to get the formula:

$$ab = ba^{\sim} \wedge x \mapsto y \in ab$$

From which we can deduce that $y \mapsto x \in ba$, so we can add this predicate to the postcondition to get the postcondition:

$$x \mapsto y \in ab \wedge y \mapsto x \in ba$$

The new postcondition has the same semantics as the old postcondition but the form of the new postcondition makes it possible for the code generator to implement the required program.

In the rest of the section we explain in details how this principle is applied to the completion of Booster's association invariants.

6.2. Maintaining association invariants

Our purpose is to show that a completed postcondition is implied by the original postcondition in the context of a model. To show that this is the case we will first show how each completed intention is implied by the original intention, and then show how we can apply these completions to an arbitrary formula.

The completion rules depend on the multiplicity of both ends of the association, so we define a schema that makes it easy to find them in a model. The schema constrains the names ab and ba to refer to two associations that are mirrors of each other. That is, if $ab : A.ba$ then $ba : B.ab$.

AssociationPair

$abClassName, baClassName : Name$
$abAssoc, baAssoc : Association$
$ab, ba : Name$

$abAssoc.target = baClassName$
$abAssoc.mirror = ba$
$baAssoc.target = abClassName$
$baAssoc.mirror = ab$

We can now define the different kinds of associations using the following schemas:

ManyToMany

$AssociationPair$

$abAssoc.multiplicity = many$
$baAssoc.multiplicity = many$

OptionalToMany

$AssociationPair$

$abAssoc.multiplicity = optional$
$baAssoc.multiplicity = many$

ManyToOptional

$AssociationPair$

$abAssoc.multiplicity = many$
$baAssoc.multiplicity = optional$

OptionalToOptional

$AssociationPair$

$abAssoc.multiplicity = optional$
$baAssoc.multiplicity = optional$

MandatoryToMany

$AssociationPair$

$abAssoc.multiplicity = mandatory$
$baAssoc.multiplicity = many$

ManyToMandatory

$AssociationPair$

$abAssoc.multiplicity = many$
$baAssoc.multiplicity = mandatory$

MandatoryToOptional

$$
\begin{array}{l}
AssociationPair \\
\hline
abAssoc.multiplicity = mandatory \\
baAssoc.multiplicity = optional
\end{array}
$$

OptionalToMandatory

$$
\begin{array}{l}
AssociationPair \\
\hline
abAssoc.multiplicity = optional \\
baAssoc.multiplicity = mandatory
\end{array}
$$

MandatoryToMandatory

$$
\begin{array}{l}
AssociationPair \\
\hline
abAssoc.multiplicity = mandatory \\
baAssoc.multiplicity = mandatory
\end{array}
$$

An intention always involves two terms a and b and an association ab. The intention may be to add b to $a.ab$, or to remove b from $a.ab$, or if ab is mandatory or optional, to set $a.ab$ to b. In addition, if $a.ab$ is optional we may want to clear it.

Intention

$$
\begin{array}{l}
a, b : Term \\
ab : Name \\
intention? : Formula \\
completion! : Formula
\end{array}
$$

AddIntention

$$
\begin{array}{l}
Intention \\
\hline
intention? = b \in a.ab
\end{array}
$$

RemoveIntention

$$
\begin{array}{l}
Intention \\
\hline
intention? = b \notin a.ab
\end{array}
$$

SetIntention

$$
\begin{array}{l}
Intention \\
\hline
intention? = (a.ab).\mathrm{the} = b
\end{array}
$$

ClearIntention

$$
\begin{array}{l}
Intention \\
\hline
intention? = a.ab = \{\}
\end{array}
$$

Now we can examine the individual completion rules. Let us begin by completing the intention to add an element to a many-to-many association.

To prove that the completion rules do not change the meaning of the model we prove for each kind of completion rule the following theorem, where the constraints, intentions, and completions are taken from the specific rule (depending on the nature of the association and the nature of the intention).

THEOREM 6.2.1. *Let m be a model and let ab and ba be the two relations that correspond to the two ends of an association in m. Then for any intention intention?, the meaning of intention? in the context of the model is identical to the meaning of intention? $\wedge$ completion!.*

6.2.1. Many-to-Many. When the intention is of the form $b \in a.ab$ and the names ab and ba refer to the two ends of a many-to-many association in the model, the completion adds a to $b.ba$[2].

ManyToManyAdd

ManyToMany; *AddIntention*

$completion! = a \in b.ba$

PROOF. To show that this completion does not change the meaning of the intention we argue as follows:

$b \in a.ab$

$\Leftrightarrow a \mapsto b \in ab$ — dot composition

$\Leftrightarrow b \mapsto a \in ab^{\sim}$ — set theory

$\Leftrightarrow b \mapsto a \in ba$ — model invariant

$\Leftrightarrow a \in b.ba$ — dot composition

Since the constraints in the model and the intention imply the completion, we can conjoin the completion to the postcondition without changing the meaning of the method. □

Let us now describe the completion rule for removing an element from one end of a many-to-many association.

ManyToManyRemove

ManyToMany; *RemoveIntention*

$completion! = a \notin b.ba$

PROOF. To show that this completion does not change the meaning of the intention we argue as follows:

$b \notin a.ab$

$\Leftrightarrow a \mapsto b \notin ab$ — dot composition

$\Leftrightarrow b \mapsto a \notin ab^{\sim}$ — set theory

$\Leftrightarrow b \mapsto a \notin ba$ — model invariant

$\Leftrightarrow b \notin a.ab$ — dot composition

Since the constraints in the model and the intention imply the completion, we can conjoin the completion to the postcondition without changing the meaning of the method. □

[2]When the intention does not match, the completion becomes the trivial formula *true*

6.2.2. Optional-to-Many.

OptionalToManySet

$OptionalToMany;\ SetIntention$

$$completion! = a \in b.ba \land (a.ab_0 \neq \{\} \Rightarrow ((a.ab_0).\mathrm{the} \neq b \Rightarrow a \notin (a.ab_0)\ .\mathrm{the}\ .ba))$$

PROOF. To show that this completion does not change the meaning of the intention we argue in two steps. First, we show that it is safe to add the formula $a \in b.ba$ to the postcondition, and then we show that it is safe to add the rest of the completion.

$a.ab.the = b$

$\Leftrightarrow a \mapsto b \in ab$ dot composition

$\Leftrightarrow b \mapsto a \in ab^{\sim}$ set theory

$\Leftrightarrow b \mapsto a \in ba$ model invariant

$\Leftrightarrow a \in b.ba$ dot composition

Because the formula $a \in b.ba$ is implied by the intention and the model constraints, we can add it to the postcondition without changing the meaning of the method.

To see why we can add the second formula to the postcondition, we note that *ab* is a function and therefore maps *a* to at most a single value. We can therefore argue as follows:

$\forall y : \mathrm{ran}\, ab \bullet a \mapsto y \in ab \Rightarrow y = b$

$\Leftrightarrow \forall y : \mathrm{dom}\, ba \bullet y \mapsto a \in ba \Rightarrow y = b$ assumption

In particular, if $a.ab_0$ is not empty and $a.ab_0.the$ is different from *b*, we can substitute it for *y* in the universal quantifier above, and argue:

$a.ab_0.the \mapsto a \in ba \Rightarrow a.ab_0.the = b$

$\Leftrightarrow a.ab_0.the \mapsto a \notin ba \lor a.ab_0.the = b$ predicate logic

$\Leftrightarrow a.ab_0.the \mapsto a \notin ba$ assumption

$\Leftrightarrow a \notin (a.ab_0.the).ba$ dot composition

By using our assumptions as the antecedents of an implication we get the second conjunct in the completion:

$$a.ab_0 \neq \{\} \Rightarrow (a.ab_0.the \neq b \Rightarrow a \notin a.ab_0.the.ba)$$

Since we have shown that this formula can be deduced from the model invariant and the postcondition, we can add it to the postcondition without changing the meaning of the method. □

OptionalToManyClear

$OptionalToMany;\ ClearIntention$

$$completion! = (a.ab_0) \neq \{\} \Rightarrow a \notin (a.ab_0)\ .\mathrm{the}\ .ba$$

PROOF. To show that this completion does not change the meaning of the method, we assume that $a.ab_0$ is not empty and argue as follow:

$a.ab = \{\}$

$\Leftrightarrow a \notin \text{dom}\, ab$ — dot composition

$\Leftrightarrow a \notin \text{ran}\, ba$ — model invariant

$\Leftrightarrow \forall y : \text{dom}\, ba \bullet y \mapsto a \notin ba$ — set theory

$\Rightarrow a.ab_0.the \mapsto a \notin ba$ — universal elimination and assumption

$\Leftrightarrow a \notin a.ab_0.the.ba$ — dot composition

By adding the assumption as an antecedent, we get the completion formula. Because this completion follows from the intention and the model invariant, we can add it to the postcondition without changing the meaning of the method. □

6.2.3. Many-to-Optional.

ManyToOptionalAdd

ManyToOptional; *AddIntention*

$$\begin{aligned} completion! = {} & ((b.ba).\text{the} = a) \wedge \\ & (b.ba_0 \neq \{\} \Rightarrow \\ & \quad ((b.ba_0).\text{the} \neq a \Rightarrow \\ & \quad\quad b \notin (b.ba_0)\ .\text{the}\ .ab)) \end{aligned}$$

PROOF. We show that the completion follows from the model invariant and the intention by proving it separately for each conjunct in the completion. We begin with the first conjunct:

$b \in a.ab$

$\Leftrightarrow a \mapsto b \in ab$ — dot composition

$\Leftrightarrow b \mapsto a \in ab^{\sim}$ — set theory

$\Leftrightarrow b \mapsto a \in ba \wedge ba \in ObjectId \nrightarrow ObjectId$ — model invariant

$\Leftrightarrow (b.ba).the = a$ — dot composition and definition of *.the*

To prove that the second conjunct may be deduced from the invariant and the intention, we assume that $b.ba_0$ is not empty and $b.ba_0.the$ is different from a and argue as follows:

$a \mapsto b \in ab$

$\Leftrightarrow b \mapsto a \in ba$ — model invariant

$\Rightarrow \forall y : \text{ran}\, ba \bullet b \mapsto y \in ba \Rightarrow y = a$ — *ba* is a function

$\Rightarrow b \mapsto b.ba_0.the \in ba \Rightarrow b.ba_0.the = a$ — universal elimination

$\Leftrightarrow b.ba_0.the \mapsto b \in ab \Rightarrow b.ba_0.the = a$ — model invariant

$\Leftrightarrow b.ba_0.the \mapsto b \notin ab \vee b.ba_0.the = a$ — predicate calculus

$\Leftrightarrow b.ba_0.the \mapsto b \notin ab$ — assumption

$\Leftrightarrow b \notin (b.ba_0.the).ab$ — dot composition

By adding the assumption as an antecedent to the result, we get the second conjunct in the completion formula. We have therefore shown that the completion formula can be deduced from the model invariant and the intention and therefore adding it to the postcondition does not change the meaning of the method. □

ManyToOptionalRemove

$ManyToOptional; RemoveIntention$

$completion! = a \notin b.ba$

PROOF.

$b \notin a.ab$

$\Leftrightarrow a \mapsto b \notin ab$ — dot composition

$\Leftrightarrow b \mapsto a \notin ab^{\sim}$ — set theory

$\Leftrightarrow b \mapsto a \notin ba$ — model invariant

$\Leftrightarrow a \notin b.ba$ — dot composition

□

At first it may seem this completion is too weak, and that in particular it could be possible to demand that the optional attribute $b.ba$ will be empty. However, we cannot deduce that $b.ba$ is empty because it can be pointing to an object that is different from a.

6.2.4. Optional-to-Optional.

OptionalToOptionalSet

$OptionalToOptional; SetIntention$

$$completion! = (b.ba).\mathrm{the} = a \wedge (a.ab_0 \neq \{\} \Rightarrow ((a.ab_0).\mathrm{the} \neq b \Rightarrow (a \notin (a.ab_0)\ .\mathrm{the}\ .ba))) \wedge (b.ba_0 \neq \{\} \Rightarrow ((b.ba_0).\mathrm{the} \neq a \Rightarrow (b \notin (b.ba_0)\ .\mathrm{the}\ .ab)))$$

The argument that justifies this completion rule is very similar to the one that justifies the many-to-optional completion rule, in particular when cannot demand that the attributes $a.ab_0$ and $b.ba_0$ will be cleared because they can point to objects that are different from a and b and still maintain the invariant.

OptionalToOptionalClear

$OptionalToOptional; ClearIntention$

$completion! = (a.ab_0) \neq \{\} \Rightarrow a \notin (a.ab_0)\ .\mathrm{the}\ .ba$

This completion is the same as that for optional-to-many clear. In particular it does not follow that we must clear the other end of the optional attribute because it may point to a different object than a and still maintain the invariant.

6.2.5. Mandatory-to-Many.

MandatoryToManySet

$MandatoryToMany; SetIntention$

$completion! = a \in b.ba \wedge (b \neq a.ab_0 \Rightarrow a \notin a.ab_0.ba)$

The justification for the first conjunct in the completion rule is exactly the same as in the case of the corresponding optional-to-many completion rule and we do not repeat it here. The justification for the

second conjunct is similar but more simple than the corresponding conjunct in the optional-to-many rule, because the function *ab* in this case is total so there is no need to test if $a.ab$ is empty.

6.2.6. Many-to-Mandatory.

ManyToMandatoryAdd

ManyToMandatory; *AddIntention*

$completion! = (b.ba = a) \wedge (b.ba_0 \neq a \Rightarrow b \notin b.ba_0.ab)$

The justification for this completion rule is essentially the same as the corresponding many-to-optional rule except that because the function *ba* is total there is no need to check that $b.ba$ is non-empty.

6.2.7. Problematic combinations. Some combinations of associations, even though they may appear to be useful, create many difficulties for the completion process. These include: mandatory-to-optional, optional-to-mandatory, and mandatory-to-mandatory.

For example, assume that *ab* and *ba* are the two ends of a mandatory to mandatory association, and we want to set $a.ab$ to b. The implications of this goal can be unraveled by the following argument:

$a.ab = b$

$\Rightarrow b.ba = a$ $\qquad ab = ba^{\sim}$

$\Rightarrow b.ba_0 \neq a \Rightarrow b \mapsto b.ba_0 \notin ba$ $\qquad$ *ba* is a function

$\Rightarrow b.ba_0 \mapsto b \notin ab$ $\qquad ab = ba^{\sim}$

$\Leftrightarrow (b.ba_0).ab \neq b$

this means that $b.ba_0.ab$ can no longer point to b, but since *ab* is mandatory it must be set to some object. But to which object? the compiler has no information that can be used to find such an object. Currently we forbid such combinations by setting the completion to the predicate *false*:

MandatoryToMandatorySet

MandatoryToMandatory; *SetIntention*

$completion! = false$

An interesting way to admit such combinations was recently suggested by James Welch. James suggested that in cases such as the above we can introduce a new input variable and demand that the problematic attribute ($b.ba_0.ab$ in the case above) will be equal to this input variable. However this strategy requires more rigorous study before it can be used safely. For example we have to rule out an avalanche effect that may occur if by applying the completion rules to the new input variable we introduce new input variables, which introduce new input variables and so on.

6.2.8. Applying the rules to an arbitrary postcondition. So far we have defined the completion rules for each kind of intention and shown that they can be deduced from the context and the intention. If intentions appear alone in postconditions then this will be enough. However in most cases the intentions are usually embedded in a bigger postcondition and it is not always clear which part of the postcondition we have to change in order for the completion to be valid. For example, we cannot in general deduce completion from disjunctions.

To solve this problem we use a function, *complete*, which takes a semantics preserving function (like the completion functions we have defined above) and applies it to a formula in a way that ensures that the final formula has the same meaning as the original formula. The idea is that *complete* avoids troublesome constructs like negations and disjunctions and applies the function f only when it is certain that the semantics preserving properties of the construct can be deduced from those of its sub formulas.

$$
\begin{array}{l}
complete : (Formula \nrightarrow Formula) \rightarrow (Formula \nrightarrow Formula) \\
\hline
\forall f : Formula \nrightarrow Formula;\ p, q : Formula;\ b : BasicFormula;\ s, t : Term \bullet \\
\quad complete\, f\ true = true \land \\
\quad complete\, f\ false = false \land \\
\quad complete\, f\ (\neg\ p) = \neg\ p \land \\
\quad complete\, f\ (p \land q) = complete\, f\ p \land complete\, f\ q \land \\
\quad complete\, f\ (p \lor q) = p \lor q \land \\
\quad complete\, f\ (p \Rightarrow q) = p \Rightarrow complete\, f\ q \\
\quad complete\, f\ basic(b, s, t) = f\ (basic(b, s, t))
\end{array}
$$

DEFINITION 6.2.1. *We say that a function f is semantics preserving in the context of a model if the meaning of any formula of the form $f(q)$ is identical to the meaning of the formula q for every state that satisfies the model constraints.*

THEOREM 6.2.2. *If the function f is semantics preserving in the context of a formula r then the function $complete\, f$ is also semantics preserving in the same context.*

PROOF. We will prove the theorem by induction on the structure of *Formula*. For the base cases, negation, and disjunction it is easy to see by inspecting the definition of *complete* that it leaves the formulas unmodified and therefore the theorem holds. For implication, let p and q be formulas. We argue as follows:

$$
\begin{array}{lr}
\mathcal{P}(r \land complete\, f\ (p \Rightarrow q)) & \\
= \mathcal{P}(r \land (p \Rightarrow complete\, f\ q)) & \text{definition of } complete \\
= \mathcal{P}(r) \cap \mathcal{P}(p \Rightarrow complete\, f\ q)) & \text{definition of } \mathcal{P} \\
= \mathcal{P}(r) \cap (\mathcal{P}(pos(\neg\ p)) \cup \mathcal{P}(complete\, f\ q)) & \text{definition of } \mathcal{P} \\
= (\mathcal{P}(r) \cap \mathcal{P}(pos(\neg\ p))) \cup (\mathcal{P}(r) \cap \mathcal{P}(complete\, f\ q)) & \text{set theory} \\
= (\mathcal{P}(r) \cap \mathcal{P}(pos(\neg\ p))) \cup \mathcal{P}(r \land (complete\, f\ q)) & \text{definition of } \mathcal{P} \\
= (\mathcal{P}(r)) \cap \mathcal{P}(pos(\neg\ p))) \cup (\mathcal{P}(r) \land q) & \text{induction hypothesis} \\
= (\mathcal{P}(r) \cap \mathcal{P}(pos(\neg\ p))) \cup (\mathcal{P}(r) \cap \mathcal{P}(q)) & \text{definition of } \mathcal{P} \\
= \mathcal{P}(r) \cap (\mathcal{P}(\neg\ p) \cup \mathcal{P}(q)) & \text{set theory} \\
= \mathcal{P}(r) \cap \mathcal{P}(p \Rightarrow q) & \text{definition of } \mathcal{P} \\
= \mathcal{P}(r \land (p \Rightarrow q)) & \text{definition of } \mathcal{P}
\end{array}
$$

Proving that the theorem holds for conjunction is similar. Finally, let t and u be terms. We argue as follows:

$$
\begin{array}{lr}
\mathcal{P}(r \land complete\, f\ (t \in u)) & \\
= \mathcal{P}(r \land f\ (t \in u)) & \text{definition of } complete \\
= \mathcal{P}(r \land t \in u) & f \text{ is semantics preserving}
\end{array}
$$

The same argument applies to the other basic formulas. □

6.3. Related work

Although we are unaware of similar approaches involving the application of generic formal methods, a considerable amount of relevant work has been reported in the area of database design [**24, 31, 33, 51, 11**]. Maintaining association invariants is particularly important when valuable data may be accrued and updated over a long period of time.

The simplest approach is to ask the programmer to maintain the associations explicitly [**24, 31**]; however, the programmer would receive no assistance in designing local operations to maintain global

properties. A degree of protection may be provided at run-time by translating each association invariant into a corresponding assertion, and rolling back any operation that ends in a state in which the assertion does not hold.

Unfortunately it is not always possible to roll back the effects of an operation. Accelerating a car cannot always be undone by braking.

Another approach is to design *triggers* that detect problematic situations and invoke operations (which can, in some cases, be generated automatically from the model) that attempt to re-establish the invariant [**33, 17**]. This approach is effective only for simple association invariants; in other cases, an attempt to re-establish one invariant may result in another being violated [**11**].

The trigger approach has also been used to enforce business rules in knowledge representation systems [**6**], and has been the subject of a considerable amount of research: see, for example [**40**], in which invariants are 'repaired' by finding a suitable order in which to invoke a selection of compensating actions (however, again, no such order may exist, and the repair operation can fail to terminate).

It has also been suggested [**17**] that invariants could be checked at intervals, rather than after every operation. However, if it is acceptable for invariants to be broken, then these invariants are a poor characterization of correctness. That such a compromise should be suggested is indicative of a realization that it may be impossible to maintain invariants on-the-fly: prior analysis at the level of a model or specification is required.

The approach closest to the one taken here is that of Schewe et al [**51**], who suggest that, given a set of invariant properties, we may transform the design of operations so that they are properly preserved. However, applying this approach at the level of a programming language offered little benefit: it is far more effective to perform such transformations at the specification level, where the intended behavior of each operation is explicit, and the appropriate constraints can be expressed and incorporated.

As an application of formal methods, this approach may be characterized as 'automatic refinement'. It is quite distinct from the more familiar type of formally-driven development in which a concrete implementation may be proved correct with regard to abstract specification. Here, an abstract, declarative specification may be used to construct, automatically, a complete working implementation, on the basis of specific assumptions about the application domain.

We should remark that others have used functions from object identities to attributes to provide a semantics for class models—see, for example, [**26**]. Even though such a semantics may seem unwieldy when applied to a specific model, it is ideally suited for the analysis required here.

6.4. Discussion

In this chapter we have shown that it is possible to develop the completion rules based only on the relational semantics of the associations and simple predicate logic. The completion rules do not need to rely on a particular implementation strategy. Their meaning is implicit within the model's semantics and the completion rules only make this meaning explicit.

6.4.1. Active vs. passive maintenance of invariants. We use the term 'association maintenance' to refer to the task of making sure that operations never end in a state that breaks the invariant. In a traditional programming language this task rests on the shoulders of the developers. In contrast, in Booster this task is the responsibility of the compiler.

Regardless of the identity of the maintainer we can approach this problem in two different ways: either by doing more, or by restricting the domain of application. In Booster we do more by adding more predicates to the postcondition, and we restrict the domain of application by adding more predicates to the precondition. Adding more predicates to the postcondition results in active maintenance, while adding more predicates to the precondition results in passive maintenance.

It is best to illustrate the different options with an example. Let us look at a simple model with a single optional-to-many association:

```
CLASS A
  ATTRIBUTES
    ab : [B.ba]
  METHODS
    M(true | ab = b_in)
CLASS B
  ATTRIBUTES
    ba : SET(A.ab)
```

And consider the different ways in which we can ensure that an operation that sets the optional attribute `ab` can be completed to ensure that it does not break the association invariant:

- We can decide to do nothing and only apply the operation if the final condition already holds in the beginning:

  ```
  M(ab = b_in | ab = b_in)
  ```

  We can always implement such specifications using the very simple program ‘skip’ which does nothing. Such a completely passive implementation ‘works’ simply because it is applied only when the goal it needs to achieve was already met.
- We can be more active by adding a predicate to the postcondition which explicitly states that we want to modify the other end of the association:

  ```
  M(ab = {} | ab = b_in & this : b_in.ba)
  ```

  In this case the generated code will be more active than the previous implementation. It will set the optional field *ab* to *b_in* and it will also add *this* to the other end of the association. However, this will happen only if the optional field is empty.
- Finally, we can be even more active by adding even more predicates to the postcondition to handle the case when *ab* is not empty:

  ```
  M(true | ab = b_in & this : b_in.ba &
          (ab /= {} =>
            (ab /= b_in =>
              this /: a.ab_0.ba)))
  ```

  In this case, if the optional attribute is not empty and is not equal to the input *b_in*, we remove the current object from the other end of the old association.

As we can see the more predicates we add to the postcondition the more active the resulting implementation becomes. This is because more predicates in the postcondition give the code generator more material from which it can generate code.

The extent to which a completion strategy is active or passive depends on the problem domain. For the kinds of problems we have used Booster so far it seems that a middle ground, that is the second option above, is a good fit, but other systems may find that a more active or a more passive strategy is a better fit.

The idea of completion may be applied to more than association invariants. Many domains can be characterized by a set of constraints. For example, in financial domains the flow of incoming and outgoing money must always balance out, making their sum zero. Such constraints are sometimes also called *business rules*.

Since not all business rules are invariants we will use the term *business invariants* to mean business rules that must be true in every state of the system.

Today in order to specify and implement operations within a domain, engineers must describe not only the intent of the operation, but also any additional effects that are needed in order to maintain the business invariants of the domain. Often these additional effects are more complicated to describe than the original intention.

Using the techniques illustrated in this chapter, it may be possible to let the engineer specify only the intention of the operation, and let the system infer the additional information that is necessary to maintain the business invariants. This may greatly simplify the task of describing the operations and remove a large source of errors.

CHAPTER 7

From predicates to programs

In this chapter we describe how the Booster compiler transforms specifications into programs. We begin by describing our implementation language, then we describe how we treat the problem of undefined expressions (for example division by zero) and then we describe the code generation process.

7.1. Booster's guarded command language

As an implementation language, we use a version of Dijkstra's guarded command language. The language is small, has a clear and simple semantics and can easily be translated to a specific programming language. In this section we describe the statements that make up the language, and define their meaning by mapping each statement to a program.

The free type *Statement* defines the abstract syntax of the guarded command language:

$$
\begin{array}{lll}
\mathit{Statement} & ::= & \mathit{skip} \mid \mathit{extend}\langle\!\langle \mathit{Name} \times \mathit{Term} \rangle\!\rangle \\
& \mid & \mathit{assignment}\langle\!\langle \mathit{Name} \times \mathit{Term} \rangle\!\rangle \\
& \mid & \mathit{store}\langle\!\langle \mathit{Term} \times \mathit{Name} \times \mathit{Term} \rangle\!\rangle \\
& \mid & \mathit{add}\langle\!\langle \mathit{Term} \times \mathit{Name} \times \mathit{Term} \rangle\!\rangle \\
& \mid & \mathit{remove}\langle\!\langle \mathit{Term} \times \mathit{Name} \times \mathit{Term} \rangle\!\rangle \\
& \mid & \mathit{sequence}\langle\!\langle \mathit{Statement} \times \mathit{Statement} \rangle\!\rangle \\
& \mid & \mathit{guard}\langle\!\langle \mathit{Formula} \times \mathit{Statement} \rangle\!\rangle \\
& \mid & \mathit{choice}\langle\!\langle \mathit{Statement} \times \mathit{Statement} \rangle\!\rangle \\
& \mid & \mathit{block}\langle\!\langle \mathit{Block} \rangle\!\rangle
\end{array}
$$

where a block consists of a list of local variables, a list of terms that initialize the local variables, and a body:

Block

$\mathit{locals} : \mathrm{iseq}\, \mathit{Name}$
$\mathit{inits} : \mathrm{seq}\, \mathit{Term}$
$\mathit{body} : \mathit{Statement}$

$\#\mathit{locals} = \#\mathit{inits}$

Here is an informal summary of the behavior of each statement:

$skip$	does nothing
$extend(c, id)$	extends the extent of class c with id
$assignment(v, e)$	assigns the value of the term e to the variable v
$add(a, b, e)$	adds the pair (a, e) to the relation b
$store(a, b, e)$	replaces any old bindings of a in b with the pair (a, e)
$remove(a, b, e)$	removes the pair (a, e) from the relation b
$sequence(first, second)$	executes $first$ followed by $second$
$guard(g, s)$	executes s if g evaluates to $true$
$choice(s_1, s_2)$	executes either s_1 or s_2
$block(locals, inits, body)$	executes $body$ in a state whose local variables are extended with the variables in the list $locals$. The ith variable in $locals$ is initialized to the value of the ith term in $inits$

In the rest of this section we give to each statement both its relational semantics and its weakest precondition semantics, and prove that they are equivalent.

7.1.1. Skip.

SkipMeaning

$meaning : Statement \nrightarrow Program$

$meaning\ skip = \{s : State \bullet s \mapsto s\}$

SkipWp

$wp : Statement \rightarrow Formula \rightarrow Formula$

$\forall p : Formula \bullet wp\ skip\ p = p$

To prove that the weakest precondition function of *skip* is consistent with the semantics of *skip* we show that for any formula p

$$wp\ meaning(skip)\ \mathcal{P}(p) = \mathcal{C}(wp\ skip\ p)$$

PROOF. We argue as follows:

$wp\ (meaning\ skip)\ \mathcal{P}(p)$

$= \text{dom}(meaning\ skip \cap \mathcal{P}(p))$ — theorem 3.1.6

$= \text{dom}(id \cap \mathcal{P}(p))$ — definition of *meaning skip*

$= \mathcal{C}(p)$ — theorem 5.4.1

$= \mathcal{C}(wp\ skip\ p)$ — definition of *wp skip*

□

7.1.2. Assignment. The meaning of the assignment statement $v := e$ is to update the state by replacing the current binding of v with the value of e:

AssignmentMeaning

TermMeaning

$meaning : Statement \nrightarrow Program$

$\forall v : Name;\ e : Term \bullet$
$\quad meaning(v := e) = \{s : State \bullet s \mapsto s \oplus \{v \mapsto eval(s, s)\ e\}\}$

The weakest precondition of assignment is a substitution:

AssignmentWp

$wp : Statement \rightarrow Formula \rightarrow Formula$

$\forall\, v : Name;\ e : Term;\ p : Formula \bullet wp(v := e)\, p = p[v := e]$

We now show that the semantic definition and the syntactic definition of assignment are the same. We write $p[x := e]$ to denote the substitution of e for x in p.

THEOREM 7.1.1. *Let e be a term, x be a variable, and p be a formula. Then,*

$$wp\,(meaning(x := e))\, \mathcal{P}(p) = \mathcal{C}(p[x := e])$$

PROOF. Let e be a term, x be a variable, and p be a formula. We argue as follows:

$wp\,(meaning(x := e))\, \mathcal{P}(p)$
$= \mathrm{dom}(meaning(x := e) \cap \mathcal{P}(p))$ — theorem 3.1.6
$= \mathrm{dom}(\{s : State \bullet s \mapsto s \oplus \{x \mapsto eval(s,s)\, e\}\} \cap \mathcal{P}(p))$ — definition of $meaning(x := e)$
$= \mathrm{dom}(\{s : State \bullet s \mapsto s'\} \cap \mathcal{P}(p))$ — we take $s' = s \oplus \{x \mapsto eval(s,s)\, e\}$
$= \mathrm{dom}(\{s : State \mid s \mapsto s' \in \mathcal{P}(p) \bullet s \mapsto s'\})$ — set theory
$= \{s : State \mid s \mapsto s' \in \mathcal{P}(p)\}$ — set theory
$= \{s : State \mid s \mapsto s \in \mathcal{P}(p[x := e])\}$ — lemma 7.1.2
$= \mathrm{dom}(\mathcal{P}(p[x := e]) \cap id)$ — set theory
$= \mathcal{C}(p[x := e])$ — theorem 5.4.1

□

LEMMA 7.1.2. *For every state s and every formula p,*

$$s \mapsto s \in \mathcal{P}(p[x := e]) \Leftrightarrow s \mapsto s \oplus \{x \mapsto eval(s,s)\, e\} \in \mathcal{P}(p)$$

PROOF. We prove the lemma by induction on the structure of formulas. It is easy to see that the lemma holds for *true* and *false*. In addition, we may use a straight forward induction to prove that the lemma holds for each of the logical operators. The only interesting case is for basic formulas.

Assume therefore that p is a basic formula. Since all the basic formulas follow the same pattern it is easier to pick just one representative for the proof. Let us prove the lemma for the formula $t_1\ \alpha\ t_2$ where t_1 and t_2 are terms and α is a relational operator. Let $s' = s \oplus \{x \mapsto eval(s,s)\, e\}$. We may argue as follows:

$s \mapsto s' \in \mathcal{P}(t_1\ \alpha\ t_2)$
$\Leftrightarrow s \mapsto s' \in \{w, w' : State \mid eval(w,w')\, t_1 \mapsto eval(w,w')\, t_2 \in \alpha\}$ — definition of $\mathcal{P}$
$\Leftrightarrow eval(s,s')\, t_1 \mapsto eval(s,s')\, t_2 \in \alpha$ — set comprehension
$\Leftrightarrow eval(s,s)\, t_1[x := e] \mapsto eval(s,s')\, t_2[x := e] \in \alpha$ — lemma 7.1.3
$\Leftrightarrow s \mapsto s \in \{w, w' : State \mid eval(w,w')\, t_1[x := e] \mapsto eval(w,w')\, t_2[x := e] \in \alpha\}$ — set comprehension
$\Leftrightarrow s \mapsto s \in \mathcal{P}(t_1[x := e]\ \alpha\ t_2[x := e])$ — definition of $\mathcal{P}$
$\Leftrightarrow s \mapsto s \in \mathcal{P}((t_1\ \alpha\ t_2)[x := e])$

□

LEMMA 7.1.3. *For every state s, terms t and e and variable x,*

$$eval(s, s)\, t[x := e] = eval(s, s \oplus \{x \mapsto eval(s, s)\, e\})\, t$$

PROOF. We prove the lemma by induction on the term t. For literal terms both sides of the lemma evaluate to the same value because they do not use the state. Similarly, when t is a variable that is different from x, both s and $s \oplus \{x \mapsto eval(s, s)\, e\}$ return the same value for t. For any operator of the form $opname(t_1, \ldots, t_n)$ we can use the following straightforward argument:

$eval(s, s)\, (opname(t_1, \ldots, t_n))[x := e]$
$= eval(s, s)\, (opname(t_1[x := e], \ldots, t_n[x := e]))$ — definition of substitution
$= op(eval(s, s)\, t_1[x := e], \ldots, eval(s, s)\, t_n[x := e])$ — definition of *eval*
$= op(eval(s, s \oplus \{x \mapsto eval(s, s)\, e\})\, t_1, \ldots, eval(s, s \oplus \{x \mapsto eval(s, s)\, e\})\, t_n)$ — induction hypothesis
$= eval(s, s \oplus \{x \mapsto eval(s, s)\, e\})\, (opname(t_1, \ldots, t_n))$ — definition of *eval*

The only remaining case is when t is the variable x. Assume that $t = x$. We argue as follows:

$eval(s, s)\, x[x := e]$
$= eval(s, s)\, e$ — substitution
$= (s \oplus \{x \mapsto eval(s, s)\, e\})\, x$ — property of $\oplus$
$= eval(s, s \oplus \{x \mapsto eval(s, s)\, e\})\, x$ — definition of *eval*

□

7.1.3. Creating new objects. We do not provide a single statement for creating new objects. Instead, we create a new object using a combination of the function *newobject* which returns a new unique identifier, and the statement *extend* which extends the extent of a class. For example, to set the attribute $a.b$ to a new object of class B we may write

$$a.b := newobject(B)\ ;\ extend(c, newobject(B))$$

The meaning of the statement $extend(c, t)$ is to update the extent of class c with the value of t. We rely on the type rules of the language to ensure that the name c refers to a class (and therefore the global variable c refers to the extent of the class c) and that t is an object reference.

ExtendMeaning

$meaning : Statement \rightarrow\!\!\!\!\!+\ Program$

$\forall\, c : Name;\ t : Term \bullet meaning\,(extend(c, t)) = meaning(c := c \cup \{t\})$

Note that a newly created object has no attributes. It must be accompanied by appropriate assignment statements that ensure that all the attributes of the object and all its mandatory associations are properly initialized.

7.1.4. Other primitive statements. We give the meaning of *add*, *remove*, and *store* in terms of the assignment statement:

AddRemoveStoreMeaning

$meaning : Statement \rightarrowtail Program$

$\forall a, e : Term;\ b : Name \bullet$
$\quad meaning(add(a,b,e)) = meaning(b := b \cup \{a \mapsto e\}) \land$
$\quad meaning(remove(a,b,e)) = meaning(b := b \setminus \{a \mapsto e\}) \land$
$\quad meaning(store(a,b,e)) = meaning(b := b \oplus \{a \mapsto e\})$

7.1.5. Sequential composition. The definition of the sequential composition statement is based on the discussion in chapter 3, in particular theorem 3.1.8

SequenceMeaning

$meaning : Statement \rightarrowtail Program$

$\forall first, second : Statement \bullet$
$\quad meaning\ (first\ ;\ second) = meaning\ first\ ;\ meaning\ second$

SequenceWp

$wp : Statement \rightarrow Formula \rightarrow Formula$

$\forall first, second : Statement;\ p : Formula \bullet$
$\quad wp\ (first\ ;\ second)\ p = wp\ first\ (wp\ second\ p[v_0 := fresh])[fresh := v_0]$

7.1.6. Guards.

GuardStatementMeaning

FormulaMeaning
$meaning : Statement \rightarrowtail Program$

$\forall test : Formula;\ body : Statement \bullet$
$\quad meaning\ (guard(test, body)) = \mathcal{C}\ test \lhd meaning\ body$

Let p be an arbitrary predicate. We can calculate the weakest precondition of a guard on p in the following way:

$wp\ (meaning\ guard(test, body))\ p$

$= wp\ (\mathcal{C}\ test \lhd meaning\ body)\ p$	definition of *meaning*
$= \mathcal{C}\ test \cap wp\ (meaning\ body)\ p$	lemma 7.1.4

LEMMA 7.1.4. *Let p and q be programs and let r be a constraint. Then,*

$$wp\ (r \lhd p)\ q = r \cap (wp\ p\ q)$$

PROOF. Let p and q be programs and let r be a constraint. We argue as follows:

$wp\ (r \lhd p)\ q$

$= \mathrm{dom}(r \lhd p) \setminus \mathrm{dom}((r \lhd p) \cap \overline{q})$	definition of *wp*
$= (r \cap \mathrm{dom}\, p) \setminus \mathrm{dom}(r \lhd (p \cap \overline{q}))$	set theory
$= (r \cap \mathrm{dom}\, p) \setminus (r \cap \mathrm{dom}(p \cap \overline{q}))$	set theory
$= (r \cap \mathrm{dom}\, p) \setminus \mathrm{dom}(p \cap \overline{q}))$	$(A \cap B) \setminus (A \cap C) = (A \cap B) \setminus C$
$= r \cap (\mathrm{dom}\, p \setminus \mathrm{dom}(p \cap \overline{q}))$	$(A \cap B) \setminus C = A \cap (B \setminus C)$
$= r \cap (wp\ p\ q)$	definition of *wp*

□

Replacing the set theoretical operators by their equivalent Booster syntax yields the weakest precondition of guards:

GuardStatementWp

$wp : Statement \rightarrow Formula \rightarrow Formula$

$\forall\, test : Formula;\ body : Statement;\ p : Formula \bullet$
$\quad wp\ (guard(test, body))\ p = test \wedge wp\ body\ p$

7.1.7. Choice.

ChoiceMeaning

$meaning : Statement \nrightarrow Program$

$\forall\, left, right : Statement \bullet$
$\quad meaning\ (choice(left, right)) = meaning\ left \cup meaning\ right$

In order to find out the weakest precondition of choice we will calculate the semantic weakest precondition of the union of two relations. Let l and r be two relations, and let p be an arbitrary postcondition. First we prove a lemma that will help us in the final stages of the calculation[1]:

LEMMA 7.1.5.

$$\overline{\text{dom}\, r} \cup (wp\ r\ p) = \overline{\text{dom}(r \cap \overline{p})}$$

PROOF.

$\overline{\text{dom}\, r} \cup (wp\ r\ p)$

$= \overline{\text{dom}\, r} \cup ((\text{dom}\, r) \cap \overline{\text{dom}(r \cap \overline{p})})$ — definition of wp

$= (\overline{\text{dom}\, r} \cup (\text{dom}\, r)) \cap (\overline{\text{dom}\, r} \cup \overline{\text{dom}(r \cap \overline{p})})$ — set theory

$= \overline{\text{dom}\, r} \cup \overline{\text{dom}(r \cap \overline{p})}$ — $\overline{A} \cup A = State$ and $State \cap S = S$

$= \overline{(\text{dom}\, r) \cap (\text{dom}(r \cap \overline{p}))}$ — de-Morgan

$= \overline{\text{dom}(r \cap \overline{p})}$ — dom is monotonic

□

Now we can calculate the weakest precondition as follows:

$wp\ (l \cup r)\ p$

$= \text{dom}(l \cup r) \setminus \text{dom}((l \cup r) \cap \overline{p})$ — definition of wp

$= \text{dom}(l \cup r) \setminus \text{dom}((l \cap \overline{p}) \cup (r \cap \overline{p}))$ — set theory

$= \text{dom}(l \cup r) \setminus (\text{dom}(l \cap \overline{p}) \cup \text{dom}(r \cap \overline{p}))$ — dom distributes through union

$= ((\text{dom}\, l) \cup (\text{dom}\, r)) \setminus (\text{dom}(l \cap \overline{p}) \cup \text{dom}(r \cap \overline{p}))$ — dom distributes through union

$= (\text{dom}\, l) \setminus (\text{dom}(l \cap \overline{p}) \cup \text{dom}(r \cap \overline{p})) \cup$
$\quad (\text{dom}\, r) \setminus (\text{dom}(l \cap \overline{p}) \cup \text{dom}(r \cap \overline{p}))$ — $(A \cup B) \setminus C = (A \setminus C) \cup (B \setminus C)$

$= ((\text{dom}\, l) \setminus \text{dom}(l \cap \overline{p})) \setminus \text{dom}(r \cap \overline{p})) \cup$
$\quad ((\text{dom}\, r) \setminus \text{dom}(r \cap \overline{p})) \setminus \text{dom}(l \cap \overline{p}))$ — $A \setminus (B \cup C) = (A \setminus B) \setminus C = (A \setminus C) \setminus B$

[1] The expression $\overline{\text{dom}\, r} \cup (wp\ r\ p)$ can be seen as a relational version of the weakest liberal precondition because it consists of all the states that are either undefined or terminate in p.

$$= ((wp\,l\,p) \setminus \mathrm{dom}(r \cap \overline{p})) \cup ((wp\,r\,p) \setminus \mathrm{dom}(l \cap \overline{p})) \qquad \text{definition of } wp$$
$$= ((wp\,l\,p) \cap \overline{\mathrm{dom}(r \cap \overline{p})} \cup ((wp\,r\,p) \cap \overline{\mathrm{dom}(l \cap \overline{p})} \qquad A \setminus B = A \cap \overline{B}$$
$$= ((wp\,l\,p) \cap (\overline{\mathrm{dom}\,r} \cup (wp\,r\,p)) \cup ((wp\,r\,p) \cap (\overline{\mathrm{dom}\,l} \cup (wp\,l\,p)) \qquad \text{lemma 7.1.5}$$
$$= ((wp\,l\,p) \cap \overline{\mathrm{dom}\,r}) \cup ((wp\,l\,p) \cap (wp\,r\,p)) \cup ((wp\,r\,p) \cap \overline{\mathrm{dom}\,l}) \qquad \text{set theory}$$

We can now form the equivalent syntactic form by taking intersections into conjunctions, complement into negation, and unions into disjunctions. This results in the following weakest precondition formula:

ChoiceWp

$wp : Statement \to Formula \to Formula$

$$\begin{aligned}
&\forall\, left, right : Statement;\ p : Formula \bullet \\
&\quad wp\,(choice(left, right))\,p = \\
&\qquad wp\,left\,p \wedge \neg\,(wp\,right\,true) \vee \\
&\qquad wp\,left\,p \wedge wp\,right\,p \vee \\
&\qquad wp\,right\,p \wedge \neg\,(wp\,left\,true)
\end{aligned}$$

This weakest precondition ensures that if a state s is mapped by *left* to p, then either it is also mapped by *right* to p, or it is not in the domain of *right*. Similarly, if s is mapped by *right* to p then it is either mapped by *left* to p, or it is not in the domain of *left*.

If we attempt to simplify the rule, for example by taking only the disjunction of the weakest preconditions, then we may run the risk that a state s that is mapped by *left* to p is mapped by *right* to $\overline{p}$, which means that s cannot be in the union of both relations.

Fortunately we can simplify this result in the common case where the domains of the two relations are disjoint. In this case every state in the weakest precondition of *left* is not in the domain of *right* and vice versa:

$$((wp\,l\,p) \cap \overline{\mathrm{dom}\,r}) \cup ((wp\,l\,p) \cap (wp\,r\,p)) \cup ((wp\,r\,p) \cap \overline{\mathrm{dom}\,l})$$
$$= ((wp\,l\,p) \cap \overline{\mathrm{dom}\,r}) \cup \{\} \cup ((wp\,r\,p) \cap \overline{\mathrm{dom}\,l}) \qquad \mathrm{dom}\,r \cap \mathrm{dom}\,l = \{\}$$
$$= (wp\,l\,p) \cup (wp\,r\,p) \qquad A \cap B = \{\} \Rightarrow A \cap \overline{B} = A$$

which means that

$$wp\,(choice(left, right))\,p = wp\,left\,p \vee wp\,right\,p$$

provided $\mathrm{dom}\,left \cap \mathrm{dom}\,right = \{\}$.

7.1.8. Blocks. A block has a list of local variables that are made available to its body, but remain hidden from the neighboring statements. In other words, the local variables are part of the body's alphabet but not of the block's alphabet. Therefore to get at the states of the block we must remove the local names from the states of its body.

BlockMeaning

TermMeaning
$meaning : Statement \rightarrowtail\!\!\!\!\!\!\!+\; Program$

$$\begin{aligned}
&\forall\, Block \bullet meaning\,(block(\theta Block)) = \\
&\quad \{s, s' : State \mid s \mapsto s' \in meaning(body) \wedge \\
&\qquad \forall\, i : \mathrm{dom}\,locals \bullet s\,(locals\,i) = value\,(inits\,i) \bullet \\
&\qquad\quad \mathrm{ran}\,locals \lhd s \mapsto \mathrm{ran}\,locals \lhd s'\}
\end{aligned}$$

The universal quantifier makes sure that any initial state of the body binds every local variable to its initial value. We assume that the names in *locals* do not already appear in the alphabet of the block, otherwise

we should first replace them with fresh names. We also assume that no local variable appears free in any of the initializers.

Without initializing the local variables the semantics of a block becomes non deterministic, which makes it difficult to reason about. It is much better to insist that the local variables are always initialized, since this makes the semantics deterministic and simple.

The weakest precondition of a block with respect to a formula p is the simultaneous substitution in $wp(body, p)$ of each local variable by its initializer.

BlockWp

$wp : Statement \rightarrow Formula \rightarrow Formula$

$\forall Block;\ p : Formula \bullet wp\ (block(\theta Block))\ p = wp(body, p)[locals := inits]$

7.1.9. Putting it all together. To get the complete meaning of the statements we put the individual descriptions into a single schema.

$StatementMeaning \mathrel{\widehat{=}} SkipMeaning \wedge ExtendMeaning \wedge AssignmentMeaning \wedge ChoiceMeaning \wedge SequenceMeaning \wedge AddRemoveStoreMeaning \wedge GuardStatementMeaning \wedge BlockMeaning$

And similarly for the weakest preconditions:

$StatementWp \mathrel{\widehat{=}} SkipWp \wedge AssignmentWp \wedge SequenceWp \wedge GuardStatementWp \wedge ChoiceWp \wedge BlockWp$

7.2. Dealing with undefined values[2]

There are two problems with the way we have defined the meaning of formulas, both arising from the use of the undefined value. First, the formulas that appear in the specification of methods are eventually translated to an executable form to provide guards for their methods. However few programming languages support the semantics of Booster with regards to undefined values. Second, it is easy to create undefined formulas and never know that they occur in the specification. It is much better to highlight such formulas by providing some explicit indication for them in the specification.

It follows from the semantics of Booster that the specification $(pre \mid post)$ describes a program that is obliged to achieve *post* only in the states where *pre* holds, and in particular not in the states in which *pre* has no meaning. It is also meaningless to ask for the program to achieve a postcondition that has no meaning. Thus, the meaning of formulas in the context of a specification is as if they have been strengthened with formulas that ensure that they are always well defined.

Therefore, we can solve both problems by making Booster's semantics explicit. We conjoin to the formula f another formula, $\mathscr{D}f$, which is true if and only if the formula f is well defined.

$\mathscr{D} : Formula \rightarrow Formula$

$$\begin{aligned}
&\forall p, q : Formula;\ t_1, t_2 : Term;\ n : BasicFormula \bullet \\
&\quad \mathscr{D}true = true \wedge \\
&\quad \mathscr{D}false = true \wedge \\
&\quad \mathscr{D}(p \wedge q) = \mathscr{D}p \wedge \mathscr{D}q \wedge \\
&\quad \mathscr{D}(p \vee q) = \mathscr{D}p \vee \mathscr{D}q \wedge \\
&\quad \mathscr{D}(p \Rightarrow q) = pos(\neg\ (\mathscr{D}(pos(\neg\ p)))) \Rightarrow (\mathscr{D}q) \wedge \\
&\quad \mathscr{D}(basic(n, t_1, t_2)) = \mathscr{D}t_1 \wedge \mathscr{D}t_2 \wedge basic(n, t_1, t_2)
\end{aligned}$$

[2]There has been much discussion on the semantics of undefined terms and formulas [**56, 55, 29, 42, 5**]. In particular, the paper [**5**] provides a good overview of the different approaches that were used to overcome this problem. The approach we describe here is identical to approach 6 in this paper even though it was developed independently.

The formula $\mathscr{D}$ is defined so the result of $\mathscr{D}f \wedge f$ is always either true or false. It is true whenever f is true and it is *false* whenever f is either false or undefined.

Similarly, the function $\mathscr{D}t$ (we overload the symbol $\mathscr{D}$ but its use should be clear from the context) takes a term t and returns a formula $\mathscr{D}t$ that is true if t is well defined, and false otherwise.

$$
\begin{array}{l}
\mathscr{D} : \mathit{Term} \rightarrow \mathit{Formula} \\
\hline
\forall\, v : \mathit{Value};\ n : \mathit{Name};\ t_1, t_2 : \mathit{Term} \bullet \\
\quad \mathscr{D}(\mathit{literal}\, v) = \mathit{true} \wedge \\
\quad \mathscr{D}\mathit{this} = \mathit{true} \wedge \\
\quad \mathscr{D}(\mathit{inputvar}\, n) = \mathit{true} \wedge \\
\quad \mathscr{D}(\mathit{newobject}\, n) = \mathit{true} \wedge \\
\quad \mathscr{D}(\mathit{localvar}\, n) = \mathit{true} \wedge \\
\quad \mathscr{D}(\mathit{before}(n)) = \mathit{true} \wedge \\
\quad \mathscr{D}(\mathit{after}(n)) = \mathit{true} \wedge \\
\quad \mathscr{D}(\mathit{mkrel}(t_1, t_2)) = \mathscr{D}t_1 \wedge \mathscr{D}t_2 \wedge \\
\quad \mathscr{D}(\mathit{union}(t_1, t_2)) = \mathscr{D}t_1 \wedge \mathscr{D}t_2 \wedge \\
\quad \mathscr{D}(\mathit{setminus}(t_1, t_2)) = \mathscr{D}t_1 \wedge \mathscr{D}t_2 \wedge \\
\quad \mathscr{D}(\mathit{update}(t_1, t_2)) = \mathscr{D}t_1 \wedge \mathscr{D}t_2 \wedge \\
\quad \mathscr{D}(\mathit{path}(t_1, t_2)) = \mathscr{D}t_1 \wedge \mathscr{D}t_2 \wedge \\
\quad \mathscr{D}(\mathit{the}\, t_1) = \mathscr{D}t_1 \wedge (t_1 \neq \{\})
\end{array}
$$

Note how the definedness formula for *the* captures within it the domain of *the*. In a well formed model, the only case in which the application $t.\mathit{the}$ is undefined is when t evaluates to an empty set. But this is exactly the case against which the function $\mathscr{D}$ protects.

In the rest of the chapter we describe how the Booster compiler generates a program from each method in the model. We divide the description into two sections. First we describe the process for primitive methods, and then we describe the process for method combinators.

7.3. Generating code from primitive methods

To generate a program from a primitive method we follow the structure of its postcondition and generate a program fragment from each kind of formula we meet as we traverse the postcondition. Most of the hard work goes into the generation of code from the basic formulas. Accordingly, we divide the description into two parts: generating code from basic formulas and generating code from compound formulas.

7.3.1. Generating code from basic formulas. We divide the basic formulas into two classes: value determining and non value determining. Value determining formulas have corresponding program fragments. For example, an equality of the form $a = e$ corresponds to the program $a := e$. The non value determining formulas (for example, the formula $a \neq e$) do not have an effective translation, so we 'implement' them using *skip*.

$$
\begin{array}{l}
\mathit{ValueDetermining} : \mathbb{P}\, \mathit{Formula} \\
\hline
\mathit{ValueDetermining} = \\
\quad \{t : \mathit{Term};\ n, c : \mathit{Name} \bullet t.n \in \mathit{new}(c)\} \cup \\
\quad \{t, u : \mathit{Term};\ n : \mathit{Name} \bullet u.n = t\} \cup \\
\quad \{t, u : \mathit{Term};\ n : \mathit{Name} \bullet t \in u.n\} \cup \\
\quad \{t, u : \mathit{Term};\ n : \mathit{Name} \bullet t \notin u.n\} \cup
\end{array}
$$

Actually, the description above omits an important problem. The terms in a formula often contain references to old variables. Old variables in postconditions have clear semantics. But they lose their meaning when they become embedded in statements. If we look at the semantics of statements we see

that terms are always evaluated in the current state. Indeed, since we often use a sequence of assignments to implement a single method it would be very difficult to ensure that the old variable of each statement refers to the initial state of the entire sequence.

The solution is to observe that at the beginning of the program the values of the *before* and *after* variables are the same. We can therefore use local variables to capture these initial values and then replace any term that uses an old variable by the corresponding local variable.

But there is a problem that makes this process more difficult than what the description above suggests. In Booster old variables refer to attributes, but since attributes are modeled as functions, this means that if we follow the description above we have to store at the beginning of the program the whole attribute (as a function) in a local variable. This is terribly inefficient and very difficult to do because no imperative programming language represents attributes as functions.

Fortunately, we can avoid this problem by noting that attributes never appear alone, but are always embedded inside a path term. This means that we do not need to store the whole attribute function, only the specific applications (since we interpret $a.b$ as the application of the function b to a) that appears in the postcondition.

To generate the names of the local variables we use the injective function $\mathcal{L}$ that generates a unique name for each term of the form $t.v_0$.

$$\mathcal{L} : Term \rightarrowtail\!\!\!\!\rightarrow Name$$
$$\mathrm{dom}\,\mathcal{L} = \{t : Term;\ n : Name \bullet t.n_0\}$$

The function *repold* replaces every occurrence of old variables by local variables. The local variables are produced using the function $\mathcal{L}$:

$$repold : Term \rightarrow Term$$
$$repold = t.n_0 \longrightarrow localvar(\mathcal{L}(t.n_0))$$

The function *assign* associates a statement to every value formula:

$$assign : ValueDetermining \rightarrow Statement$$
$$\begin{array}{l}
\forall\, t : Term;\ n, c : Name \bullet \\
\quad assign\, t.n \in new(c) = \\
\qquad t.n := newobject(c)\ ;\ extend(c, newobject(c)) \\
\forall\, n : Name;\ t : Term \bullet \\
\quad assign\, u.n = t = store(repold\, u, n, repold\, t) \\
\forall\, n : Name;\ t, u : Term \bullet \\
\quad assign\, t \in u.n = add(repold\, u, n, repold\, t) \\
\forall\, n : Name;\ t, u : Term \bullet \\
\quad assign\, t \notin u.n = remove(repold\, u, n, repold\, t)
\end{array}$$

7.3.2. Generating code from arbitrary formulas. To generate a program from an arbitrary formula we recursively descend through the structure of the formula. When we reach a basic formula we use *assign* to generate the program. When we reach a compound formula we either use a program combinator, or the program *skip* (if we do not know how to generate a program from the formula):

ProgramFromBasic

$$program : Formula \rightarrow Statement$$
$$\begin{array}{l}
\forall\, post : ValueDetermining \bullet program\, post = assign\, t \\
\forall\, post : \mathrm{ran}\, basic \setminus ValueDetermining \bullet program\, post = skip
\end{array}$$

ProgramFromTrue

$program : Formula \rightarrow Statement$

$program\ true = skip$

We can generate the program *skip* to 'implement' the *false* specification even though it is impossible to implement, because the weakest precondition for this implementation will be *false*, preventing this method from ever being applied.

ProgramFromFalse

$program : Formula \rightarrow Statement$

$program\ false = skip$

ProgramFromDisjunction

$program : Formula \rightarrow Statement$

$\forall post : \operatorname{ran} disj \bullet program\ post = skip$

ProgramFromConjunction

$program : Formula \rightarrow Statement$

$\forall p, q : Formula \bullet program(p \wedge q) = program\ p\ ;\ program\ q$

When we generate a program from an implication we must remember to replace old variables in the antecedent by local variables. This is because the antecedent becomes the guard of the guarded command.

ProgramFromImplication

$program : Formula \rightarrow Statement$

$\forall p, q : Formula \bullet program(p \Rightarrow q) =$
$\quad choice(guard(repold\ p, program\ q), guard(pos(\neg\ (repold\ p)), skip))$

$ProgramFromFormula \mathrel{\widehat=} ProgramFromBasic \wedge ProgramFromConjunction \wedge$
$\quad ProgramFromFalse \wedge ProgramFromDisjunction \wedge ProgramFromTrue \wedge$
$\quad ProgramFromImplication$

The last step is to collect all the old variables and create a block that initializes each local variable to its corresponding initial value. To collect all the old variables we filter the postcondition through an old attribute pattern. A pattern is a function that matches sub-terms in compound terms. The expression $t \upharpoonright p$ returns a set of all the sub-terms in t that match the pattern p. Appendix C describes patterns in more details.

$oldies : Formula \rightarrow \operatorname{iseq} Term$

$\forall post : Formula \bullet oldies\ post = post \upharpoonright t.n_0$

We create the list of local variable names by applying the function $\mathcal{L}$ to each member of *odlies*:

$locals : \operatorname{iseq} Term \twoheadrightarrow \operatorname{iseq} Name$

$\forall ts : \operatorname{seq} Term \bullet locals\ ts = \{i : \mathbb{N};\ t : Term \mid i \mapsto t \in ts \bullet i \mapsto \mathcal{L}\ t\}$

And we create the initializers for the local variables by applying a rewrite rule to each term in *oldies*. A rewrite rule is a function that takes a pattern, a replacement term and a target term, and replaces all the

sub-terms in the target that match the pattern by the replacement. Rewrite rules are explained in more detail in appendix C.

$$inits : \mathrm{iseq}\, Term \rightarrow \mathrm{iseq}\, Term$$

$$\textbf{let}\ btoa == before(n) \longrightarrow after(n) \bullet$$
$$\quad \forall ts : \mathrm{iseq}\, Term \bullet inits\, ts = \{i : \mathbb{N};\ t : Term \mid i \mapsto t \in ts \bullet i \mapsto btoa\, t\}$$

Finally, we generate a program from a primitive method by generating a local variable name for each old variable, generating a program from the postcondition, replacing the old variables in the generated program by the local variables and placing the generated program into a block that declares the local variables:

ProgramFromPrimitiveMethod

$$ProgramFromFormula$$
$$execute : Method \rightarrow Statement$$

$$\forall PrimitiveMethod \bullet \textbf{let}\ old == oldies\, post \bullet$$
$$\quad execute\, primitive\, \theta PrimitiveMethod =$$
$$\quad\quad block(locals\, old, inits\, old, program\, post)$$

7.4. Strengthening the precondition

The process we have just described does not always ensure that the generated program refines its specification. This is for two reasons: first, the precondition might not be strong enough to achieve the result. For example, if we require both to include an element x in a set and to exclude an element y from the same set then x must be different from y. Second, there might be a program that can achieve the postcondition given the precondition, but the code generator is not smart enough to generate this program.

But since we have already generated a program, we can use its weakest precondition function to find the exact initial conditions under which it achieves its postcondition. We can then offer a modified version of the specification for which we know the generated program is a refinement.

For example, consider the specification:

$$(true \mid x \in a.s \wedge y \notin a.s)$$

By following the structure of the postcondition we generate the program

$$add(a.s, x);\ remove(a.s, y)$$

which obviously does not refine the specification because when x is equal to y it can never end up with $a.s$ containing x.

However if we calculate the weakest precondition of the program with respect to the postcondition, we get

$wp(add(a,s,x);\ remove(a,s,y), x \in a.s \wedge y \notin a.s)$

$= wp(add(a,s,x), wp(remove(a,s,y), x \in a.s \wedge y \notin a.s))$ — property of *wp*

$= wp(s := s \cup \{a \mapsto x\}, wp(s := s \setminus \{a \mapsto y\}, x \in a.s \wedge y \notin a.s))$ — semantics of *add* and *remove*

$= wp(s := s \cup \{a \mapsto x\}, x \in a.(s \setminus \{a \mapsto y\}) \wedge y \notin a.(s \setminus \{a \mapsto y\})))$ — semantics of assignment

$= wp(s := s \cup \{a \mapsto x\}, x \in a.s \wedge x \neq y \wedge (y \notin a.s \vee y = y))$ — set theory

$= wp(s := s \cup \{a \mapsto x\}, x \in a.s \wedge x \neq y)$ — predicate calculus

$= x \in a.(s \cup \{a \mapsto x\}) \wedge x \neq y$ — assignment

$= (x \in a.s \vee x = x) \wedge x \neq y$ set theory

$= x \neq y$

Now we can offer the specification

$$(x \neq y \mid x \in a.s \wedge y \notin a.s)$$

instead of the original specification, knowing that (by construction) the generated program refines this specification.

In this case there was no choice but to change the original specification because it is impossible to achieve the postcondition in any initial state. This is a typical example for the kind of mistakes that we make when we write specifications.

Because we model attributes as functions and relations, the weakest precondition rules for assignment to attributes contain terms that do not have a counterpart in any existing programming language. For example:

$wp(a.b := e, a.b = h)$

$\Leftrightarrow wp(b := b \oplus \{a \mapsto e\}, a.b = h)$

$\Leftrightarrow a.(b \oplus \{a \mapsto e\}) = h$

In order to overcome this problem we translate every term of the form

$$u.(v \oplus \{x \mapsto y\})$$

to the equivalent term

$$(u = x)\ ?\ y : u.v$$

where $p?e : e'$ is the concrete syntax of the conditional evaluation expression $cond(p, e, e')$ that evaluates to e if p is true and to e' if p is false.

This means that the example above becomes:

$\Leftrightarrow a.(b \oplus \{a \mapsto e\}) = h$

$\Leftrightarrow ((a = a)\ ?\ e : a.b) = h$

$\Leftrightarrow e = h$

We apply a similar transformation to terms that add or remove a pair from a relational attribute. The function *rewriteUpdates* encodes this translation formally:

$rewriteUpdates : Term \rightarrow Term$

$rewriteUpdates = t.(v \oplus \{x \mapsto y\}) \longrightarrow (t = x)\ ?\ y : t.v\ ⨾$
$\quad t.(s \setminus \{x \mapsto y\}) \longrightarrow (t = x)\ ?\ t.s \setminus \{y\} : t.s\ ⨾$
$\quad t.(s \cup \{x \mapsto y\}) \longrightarrow (t = x)\ ?\ t.s \cup \{y\} : t.s$

The following schema describes how we strengthen the precondition of primitive methods.

StrengthenPrecondition

ProgramFromFormula
StatementWp
$original, strengthened : PrimitiveMethod$

$post = original.post$
$strengthened.pre = original.pre \wedge rewriteUpdates(wp\ program\ post)$
$strengthened.post = original.post$

Finally, we can show that every strengthened primitive method is refined by the code that we use to strengthen it.

THEOREM 7.4.1. *Let M be a well formed model and let m be a well formed primitive method in M, and let sm be the result of strengthening m. Then,*

$$meaning(execute\,m) \sqsupseteq meaning\,sm$$

PROOF. Let *prog* be equal to *execute m*, *pre* be equal to *m*'s precondition and *post* be equal to *m*'s postcondition. We argue as follows:

$meaning(execute\,m) \sqsupseteq meaning\,sm$

$\Leftrightarrow$ definition of *prog*, *pre*, and *post*

$prog \sqsupseteq (pre \wedge wp(prog, post) \mid post)$

$\Leftarrow prog \sqsupseteq (wp(prog, post) \mid post) \wedge$

$(wp(prog, post) \mid post) \sqsupseteq (pre \wedge wp(prog, post) \mid post)$

refinement is transitive

The first conjunct above is true by theorem 3.2.9 and the second conjunct is true by the "weaken precondition" theorem (theorem 3.2.8). □

Of course it is easy to come up with specifications that have a simple implementation but for which the current implementation strategy produces a very restrictive solution. For example the specification

$$(true \mid x = 1 \vee x = 2)$$

can be implemented either by the program $x := 1$ or by the program $x := 2$. However, the current implementation strategy translates disjunctions in the postconditions into *skip*s, and as a result the only version of the specification it can implement is

$$(x = 1 \vee x = 2 \mid x = 1 \vee x = 2)$$

The question is, however, are such cases common in the context of the application domain? if they are common, then the implementation strategy should be changed to support them. If they are rare, then there is no reason to support them explicitly. We can often rewrite the same specification in a different way, maybe by using combinators, and find a description that can be implemented efficiently.

The implementation strategy represents our knowledge of the application domain. It captures the specification patterns that are common to the domain, and describes the general strategies by which such specifications can be implemented.

This is why the Booster approach is effective. It has a catalog of patterns — specification fragments – that match the problems that occur in a particular domain, and it expects the developers to describe their system as a combination of these specification fragments. If the designers of the implementation strategy had done a good job, then the description of the system in terms of these particular patterns will be natural to the developers and they will never notice the restricted nature of the platform.

7.5. Generating code from method combinators

We now describe how we generate code from combinators, and prove that the generated code refines its specification.

We generate the code for each method combinator by generating the code for each argument and combining the result into a bigger program. The function *execute* generates, for each method, a corresponding program:

MethodToStatement

$Model$
$ProgramFromPrimitiveMethod$
$execute : Method \rightarrow Statement$

$$
\begin{aligned}
&\forall m_1, m_2 : Method;\ name : Name;\\
&\qquad g : Formula;\ path : Term \bullet\\
&\quad execute\ (guardm(g, m_1)) = execute\ m_1\ \wedge\\
&\quad execute\ (conj(m_1, m_2)) = (execute\ m_1)\ ;\ (execute\ m_2)\ \wedge\\
&\quad execute\ (disj(m_1, m_2)) =\\
&\qquad pre\ m_1 \rightarrow (execute\ m_1)\\
&\qquad \Box\\
&\qquad \neg\ (pre\ m_1) \wedge pre\ m_2 \rightarrow (execute\ m_2)\\
&\quad execute\ (then(m_1, m_2)) = (execute\ m_1)\ ;\ (execute\ m_2)\ \wedge\\
&\quad execute\ (methodref(path, name)) =\\
&\qquad (\mathbf{let}\ m == (lookupmethod\ \theta Model\ name) \bullet\\
&\qquad\quad execute(m[this := path]))
\end{aligned}
$$

The function *pre* calculates for each well formed method a formula that represents the precondition of the method.

$pre : Method \nrightarrow Formula$

$$
\begin{aligned}
&\forall g : Formula;\ m_1, m_2 : Method;\ pm : PrimitiveMethod \bullet\\
&\quad pre\ pm = pm.pre\ \wedge\\
&\quad pre\ guardm(g, m_1) = g \wedge pre\ m_1\ \wedge\\
&\quad pre\ conj(m_1, m_2) = pre\ m_1 \wedge pre\ m_2\ \wedge\\
&\quad pre\ disj(m_1, m_2) = pre\ m_1 \vee pre\ m_2\ \wedge\\
&\quad pre\ then(m_1, m_2) = wp\ (execute\ m_1)\ (pre\ m_2)
\end{aligned}
$$

THEOREM 7.5.1. *For every well formed model M and for every method m in M*

$$\mathcal{C}(pre\ m) \subseteq dom(meaning(execute\ m))$$

PROOF. We prove the theorem by induction on the structure of methods. We begin with primitive methods. Let m be a strengthened primitive method. We may argue as follows:

$$
\begin{aligned}
&\mathcal{C}(m.pre) \subseteq \mathrm{dom}(meaning(execute\ m))\\
&\Leftarrow \mathcal{C}(m.pre) \cap \mathrm{dom}(\mathcal{P}(m.post)) \subseteq \mathrm{dom}(meaning(execute\ m))\\
&\qquad\qquad \text{because } \mathrm{dom}(\mathcal{P}(m.post)) \subseteq \mathcal{C}(m.pre)\\
&\Leftarrow \mathrm{dom}(\mathcal{C}(m.pre) \lhd \mathcal{P}(m.post)) \subseteq \mathrm{dom}(meaning(execute\ m)) \qquad \text{set theory}\\
&\Leftarrow \mathrm{dom}(meaning\ m) \subseteq \mathrm{dom}(meaning(execute\ m))\\
&\qquad\qquad \text{meaning of primitive methods}
\end{aligned}
$$

$\Leftarrow meaning(execute\ m) \sqsupseteq meaning\ m$ definition of refinement

$\Leftarrow true$ theorem 7.4.1

For the guard combinator we may argue as follows:

$\mathcal{C}(pre\ guardm(g,m)) \subseteq \text{dom}(meaning\ (execute\ guardm(g,m)))$

$\Leftrightarrow \mathcal{C}\ g \cap \mathcal{C}(pre\ m) \subseteq \text{dom}(meaning\ (execute\ guardm(g,m)))$ definition of $\mathcal{C}$

$\Leftrightarrow \mathcal{C}\ g \cap \mathcal{C}(pre\ m) \subseteq \text{dom}(meaning\ (execute\ m))$ definition of *execute*

$\Leftarrow \mathcal{C}(pre\ m) \subseteq \text{dom}(meaning(execute\ m))$ set theory

$\Leftarrow true$ induction hypothesis.

For the AND combinator, we assume that the methods m_1 and m_2 are syntactically independent and argue as follows:

$\mathcal{C}(pre\ conj(m_1,m_2)) \subseteq \text{dom}(meaning\ execute(conj(m_1,m_2)))$

$\Leftrightarrow \mathcal{C}(pre\ m_1) \cap \mathcal{C}(pre\ m_2) \subseteq \text{dom}(meaning\ execute(conj(m_1,m_2)))$
definition of $\mathcal{C}$

$\Leftrightarrow \mathcal{C}(pre\ m_1) \cap \mathcal{C}(pre\ m_2) \subseteq \text{dom}(meaning(execute\ m_1\ ;\ execute\ m_2))$
definition of *execute*

$\Leftrightarrow \mathcal{C}(pre\ m_1) \cap \mathcal{C}(pre\ m_2) \subseteq \text{dom}(meaning(execute\ m_1)\ ;\ meaning(execute\ m_2))$
definition of *meaning*

$\Leftarrow \mathcal{C}(pre\ m_1) \cap \mathcal{C}(pre\ m_2) \subseteq \text{dom}(meaning(execute\ m_1) \cap meaning(execute\ m_2))$
theorem 7.6.6

$\Leftrightarrow \mathcal{C}(pre\ m_1) \cap \mathcal{C}(pre\ m_2) \subseteq \text{dom}(meaning(execute\ m_1)) \cap \text{dom}(meaning(execute\ m_2))$
lemma 7.6.5

$\Leftarrow \mathcal{C}(pre\ m_1) \subseteq \text{dom}(meaning\ m_1) \wedge \mathcal{C}(pre\ m_2) \subseteq \text{dom}(meaning\ m_2)$ set theory

$\Leftarrow true$ induction hypothesis

To prove the theorem for the OR combinator we calculate the domain of the program that we generate from the combinator:

$\text{dom}(meaning\ execute(disj(m_1,m_2)))$

$= \text{dom}(meaning\ (pre\ m_1 \rightarrow (execute\ m_1) \,\Box\, \neg\ (pre\ m_1) \wedge pre\ m_2 \rightarrow (execute\ m_2)))$
definition of *execute*

$= \text{dom}(\mathcal{C}(pre\ m_1) \lhd meaning\ (execute\ m_1) \cup$
$\quad \mathcal{C}(\overline{pre\ m_1}) \cap \mathcal{C}(pre\ m_2) \lhd meaning\ (execute\ m_2))$ definition of *meaning*

$= \mathcal{C}(pre\ m_1) \cap \text{dom}\ meaning\ (execute\ m_1) \cup$
$\quad \mathcal{C}(\overline{pre\ m_1}) \cap \mathcal{C}(pre\ m_2) \cap \text{dom}\ meaning\ (execute\ m_2)$ set theory

$= \mathcal{C}(pre\ m_1) \cup \mathcal{C}(\overline{pre\ m_1}) \cap \mathcal{C}(pre\ m_2)$ induction hypothesis

$= \mathcal{C}(pre\ m_1) \cup \mathcal{C}(pre\ m_2)$ $A \cup (\overline{A} \cap B) = A \cup B$

$= \mathcal{C}(pre\ m_1 \vee pre\ m_2)$ definition of $\mathcal{C}$

$= \mathcal{C}(pre(disj(m_1,m_2)))$ definition of *pre*

Finally, to prove the theorem for the THEN combinator we argue as follows:

$\mathcal{C}(pre\ then(m_1,m_2)) \subseteq \text{dom}(meaning\ execute(then(m_1,m_2)))$

$\Leftrightarrow \mathcal{C}(pre\ then(m_1,m_2)) \subseteq \text{dom}(meaning\ (execute\ m_1\ ;\ execute\ m_2))$

definition of *execute*

$\Leftrightarrow \mathcal{C}(pre\ then(m_1, m_2)) \subseteq \mathrm{dom}(meaning\ (execute\ m_1)\ ;\ meaning\ (execute\ m_2))$

definition of *meaning*

$\Leftrightarrow \mathcal{C}(pre\ then(m_1, m_2)) \subseteq wp\ (meaning\ (execute\ m_1))\ \mathrm{dom}\ meaning\ (execute\ m_2)$

lemma 3.1.15

$\Leftrightarrow \mathcal{C}(wp\ (execute\ m_1)\ (pre\ m_2)) \subseteq$
$\quad wp\ (meaning\ (execute\ m_1))\ \mathrm{dom}\ meaning\ (execute\ m_2)$ definition of *pre*

$\Leftrightarrow wp\ (meaning\ execute\ m_1))\ \mathcal{C}(pre\ m_2) \subseteq$
$\quad wp\ (meaning\ (execute\ m_1))\ \mathrm{dom}\ meaning\ (execute\ m_2)$

$\Leftarrow \mathcal{C}(pre\ m_2) \subseteq \mathrm{dom}\ meaning\ (execute\ m_2)$ *wp* is monotonic

$\Leftarrow true$ induction hypothesis.

□

Now we can prove that *execute* produces programs that refine their method specifications.

THEOREM 7.5.2. *For every well formed model M and for every method m in M*

$$meaning(execute\ m) \sqsupseteq (meaning\ m)$$

The name *meaning* is overloaded according to the type of the entity. For methods it refers to the function that appears in the schema *MethodMeaning* and for statements it refers to the function that appears in the schema *StatementMeaning*.

PROOF. We have already shown that (strengthened) primitive methods are refined by their implementation. Let us consider therefore the guard combinator. Let g be a formula and m be a method. For the first conjunct in the theorem we argue as follows:

$meaning(execute(guardm(g, m))) \sqsupseteq meaning(guardm(g, m))$

$\Leftrightarrow meaning(execute(m)) \sqsupseteq meaning(guardm(g, m))$ definition of *execute*

$\Leftrightarrow meaning(execute(m)) \sqsupseteq \mathcal{C}(g) \lhd meaning(m)$ definition of *meaning*

$\Leftarrow meaning(execute(m)) \sqsupseteq meaning(m)$ weaken precondition

$\Leftarrow true$ induction hypothesis.

The next combinator we investigate is the `AND` combinator. Remember that the type rules of Booster insist that the methods in an `AND` combinator are independent.

$meaning(execute(conj(m_1, m_2))) \sqsupseteq meaning(conj(m_1, m_2))$

$\Leftrightarrow meaning(execute(m_1))\ ;\ meaning(execute(m_2)) \sqsupseteq meaning(m_1) \cap meaning(m_2)$

definition of *meaning*

$\Leftarrow meaning(execute(m_1)) \sqsupseteq meaning(m_1) \wedge meaning(execute(m_2)) \sqsupseteq meaning(m_2)$

since m_1 and m_2 are independent

$\Leftarrow true$ induction hypothesis

Now let use examine the `OR` combinator:

$meaning(execute(disj(m_1, m_2))) \sqsupseteq meaning(disj(m_1, m_2))$

$\Leftrightarrow meaning(pre\ m_1 \rightarrow (execute\ m_1)\ \square$
$\quad \neg\ (pre\ m_1) \wedge pre\ m_2 \rightarrow (execute\ m_2)) \sqsupseteq meaning(disj(m_1, m_2))$

definition of *execute*

$\Leftrightarrow \mathcal{C}(pre\,m_1) \lhd meaning(execute\,m_1) \cup$
$\quad \mathcal{C}(\neg\,(pre\,m_1) \wedge pre\,m_2) \lhd meaning(execute\,m_2)) \sqsupseteq$
$\quad\quad meaning\,m_1 \cup \mathrm{dom}\,(meaning\,m_1) \ntriangleleft (meaning\,m_2)$

definition of *meaning*

$\Leftarrow \mathcal{C}(pre\,m_1) \lhd meaning(execute\,m_1) \sqsupseteq meaning\,m_1\ \wedge$
$\quad \mathcal{C}(\neg\,(pre\,m_1) \wedge pre\,m_2) \lhd meaning(execute\,m_2)) \sqsupseteq$
$\quad\quad \mathrm{dom}\,(meaning\,m_1) \ntriangleleft (meaning\,m_2)$ lemma 7.5.5

$\Leftrightarrow meaning(execute\,m_1) \sqsupseteq meaning\,m_1\ \wedge$
$\quad \mathcal{C}(\neg\,(pre\,m_1) \wedge pre\,m_2) \lhd meaning(execute\,m_2)) \sqsupseteq$
$\quad\quad \mathrm{dom}\,(meaning\,m_1) \ntriangleleft (meaning\,m_2)$ theorem 7.5.1

$\Leftrightarrow \mathcal{C}(\neg\,(pre\,m_1) \wedge pre\,m_2) \lhd meaning(execute\,m_2)) \sqsupseteq$
$\quad \mathrm{dom}\,(meaning\,m_1) \ntriangleleft (meaning\,m_2)$ induction hypothesis

$\Leftrightarrow \mathcal{C}(\neg\,(pre\,m_1)) \cap \mathcal{C}(pre\,m_2) \lhd meaning(execute\,m_2)) \sqsupseteq$
$\quad \mathrm{dom}\,(meaning\,m_1) \ntriangleleft (meaning\,m_2)$ definition of $\mathcal{C}$

$\Leftrightarrow \mathcal{C}(\neg\,(pre\,m_1)) \lhd (\mathcal{C}(pre\,m_2) \lhd meaning(execute\,m_2)) \sqsupseteq$
$\quad \mathrm{dom}\,(meaning\,m_1) \ntriangleleft (meaning\,m_2)$ set theory

$\Leftrightarrow \mathcal{C}(\neg\,(pre\,m_1)) \lhd (meaning(execute\,m_2)) \sqsupseteq \mathrm{dom}\,(meaning\,m_1) \ntriangleleft (meaning\,m_2)$

theorem 7.5.1

$\Leftrightarrow \overline{\mathcal{C}(pre\,m_1})\lhd (meaning(execute\,m_2)) \sqsupseteq \mathrm{dom}\,(meaning\,m_1) \ntriangleleft (meaning\,m_2)$

lemma D.0.5

$\Leftrightarrow \overline{\mathcal{C}(pre\,m_1})\lhd (meaning(execute\,m_2)) \sqsupseteq \overline{\mathrm{dom}\,(meaning\,m_1})\lhd (meaning\,m_2)$

set theory

$\Leftarrow true$ induction hypothesis and lemma 7.5.4

Finally, we show that the theorem holds for the `THEN` combinator.

$meaning(execute(then(m_1, m_2))) \sqsupseteq meaning(then(m_1, m_2))$
$\Leftrightarrow meaning(execute\,m_1\ ;\ execute\,m_2) \sqsupseteq meaning(then(m_1, m_2))$

definition of *execute*

$\Leftrightarrow meaning\,(execute\,m_1)\ ;meaning\,(execute\,m_2) \sqsupseteq meaning\,m_1\ ; meaning\,m_2$

definition of *meaning*

$\Leftarrow meaning\,(execute\,m_1) \sqsupseteq meaning\,m_1 \wedge meaning\,(execute\,m_2) \sqsupseteq meaning\,m_2$

theorem 3.1.17

$\Leftarrow true$ induction hypothesis

□

LEMMA 7.5.3. *Let p and q be programs and let a and b be sets. Then,*

$$p \sqsupseteq q \wedge b \subseteq a \Rightarrow a \lhd p \sqsupseteq b \lhd q$$

PROOF. Let p and q be programs such that p refines q, and let a and b be sets such that b is a subset of a. We argue as follows:

$a \lhd p \sqsupseteq b \lhd q$
$\Leftrightarrow \mathrm{dom}(b \lhd q) \subseteq \mathrm{dom}(a \lhd p) \wedge \mathrm{dom}(b \lhd q) \lhd (a \lhd p) \subseteq b \lhd q$ definition of $\sqsupseteq$

$\Leftrightarrow b \cap \operatorname{dom} q \subseteq a \cap \operatorname{dom} p \wedge (b \cap \operatorname{dom} q) \lhd (a \lhd p) \subseteq b \lhd q$ set theory

$\Leftarrow b \subseteq a \wedge \operatorname{dom} q \subseteq \operatorname{dom} p \wedge (b \cap \operatorname{dom} q \cap a) \lhd p \subseteq b \lhd q$ set theory

$\Leftarrow (b \cap \operatorname{dom} q) \lhd p \subseteq b \lhd q$ $b \subseteq a, p \sqsupseteq q$

$\Leftrightarrow b \lhd (\operatorname{dom} q \lhd p) \subseteq b \lhd q$ set theory

$\Leftarrow (\operatorname{dom} q \lhd p) \subseteq q$ set theory

$\Leftarrow \mathit{true}$ $p \sqsupseteq q$

□

LEMMA 7.5.4. *Let p and q be programs, and let c be a constraint. Then,*

$$p \sqsupseteq q \Rightarrow c \lhd p \sqsupseteq c \lhd q$$

PROOF. Let p and q be programs, and let c be a constraint. We argue as follows:

$c \lhd p \sqsupseteq c \lhd q$

$\Leftrightarrow \operatorname{dom}(c \lhd q) \subseteq \operatorname{dom}(c \lhd p) \wedge \operatorname{dom}(c \lhd q) \lhd (c \lhd p) \subseteq c \lhd q$

definition of refinement

$\Leftrightarrow c \cap \operatorname{dom} q \subseteq c \cap \operatorname{dom} p \wedge \operatorname{dom}(c \lhd q) \lhd (c \lhd p) \subseteq c \lhd q$ set theory

$\Leftarrow \operatorname{dom} q \subseteq \operatorname{dom} p \wedge (c \cap \operatorname{dom} q) \lhd p \subseteq c \lhd q$ set theory

$\Leftrightarrow \mathit{true} \wedge (c \cap \operatorname{dom} q) \lhd p \subseteq c \lhd q$ assumption

$\Leftrightarrow c \lhd (\operatorname{dom} q \lhd p) \subseteq c \lhd q$ set theory

$\Leftarrow \operatorname{dom} q \lhd p \subseteq c \lhd q$ set theory

$\Leftarrow \mathit{true}$ assumption

□

LEMMA 7.5.5. *Let p, q, r, and s be programs such that $p \sqsupseteq q$ and $r \sqsupseteq s$. If the domains of q and r are disjoint, and the domains of s and p are disjoint then $p \cup r$ refines $q \cup s$.*

PROOF. Let p, q, r, and s be programs. Assume that $p \sqsupseteq q$, $r \sqsupseteq s$, $\operatorname{dom} q \cap \operatorname{dom} r = \{\}$ and $\operatorname{dom} s \cap \operatorname{dom} p = \{\}$. We argue as follows:

$p \cup r \sqsupseteq q \cup s$

$\Leftrightarrow \operatorname{dom}(q \cup s) \subseteq \operatorname{dom}(p \cup r) \wedge \operatorname{dom}(q \cup s) \lhd (p \cup r) \subseteq q \cup s$

definition of refinement

$\Leftrightarrow \operatorname{dom} q \cup \operatorname{dom} s \subseteq \operatorname{dom} q \cup \operatorname{dom} r \wedge \operatorname{dom}(q \cup s) \lhd (p \cup r) \subseteq q \cup s$ set theory

$\Leftarrow \mathit{true} \wedge \operatorname{dom} q \cup s \lhd (p \cup r) \subseteq q \cup s$ since $p \sqsupseteq q$ and $r \sqsupseteq s$

$\Leftrightarrow \operatorname{dom} q \lhd (p \cup r) \cup \operatorname{dom} s \lhd (p \cup r) \subseteq q \cup s$ lemma D.0.7

$\Leftrightarrow \operatorname{dom} q \lhd p \cup \operatorname{dom} s \lhd (p \cup r) \subseteq q \cup s$ since $\operatorname{dom} q \cap \operatorname{dom} r = \{\}$

$\Leftrightarrow \operatorname{dom} q \lhd p \cup \operatorname{dom} s \lhd p \subseteq q \cup s$ since $\operatorname{dom} s \cap \operatorname{dom} p = \{\}$

$\Leftarrow \operatorname{dom} q \lhd p \subseteq q \wedge \operatorname{dom} s \lhd r \subseteq s$ set theory

$\Leftarrow \mathit{true}$ since $p \sqsupseteq q$ and $r \sqsupseteq s$

□

7.6. Modular implementation of conjoined specifications

Method combinators construct the specification of a large system from specifications of smaller subsystems in such a way that the implementation of the system is constructed from the implementations of the subsystems.

However, in the case of the AND combinator it is not always possible to do so. For example, the program $x = 0 \rightarrow y := 1$ refines the specification $(x = 0 \mid y = 1)$ and the program $y = 0 \rightarrow x := 1$ refines the specification $(y = 0 \mid x = 1)$, but we cannot combine them to refine the specification $(x = 0 \mid y = 1)$ AND $(y = 0 \mid x = 1)$.

We can describe the problem as follows: given the specification A AND B, a program P_A that refines A and a program P_B that refines B, when can we say that the program $P_A \,;P_B$ refines A AND B? In other words, under which conditions

$$P_A \sqsupseteq A \wedge P_B \sqsupseteq B \Rightarrow P_A \,; P_B \sqsupseteq A \text{ AND } B$$

We may argue as follows:

$P_A \,; P_B \sqsupseteq A \text{ AND } B$

$\Leftrightarrow wp(P_A \,; P_B)(A \text{ AND } B) = \text{dom}(A \text{ AND } B)$ lemma 3.2.10

$\Leftrightarrow wp(P_A \,; P_B)(A \cap B) = \text{dom}(A \cap B)$ meaning of AND

$\Leftrightarrow wp(P_A \,; P_B)\, A \cap wp(P_A \,; P_B)\, B = \text{dom}(A \cap B)$ wp distributes through $\cap$

One way to establish this equality is to require the following three conditions to hold:

$$wp(P_A \,; P_B)\, A = \text{dom}\, A$$
$$wp(P_A \,; P_B)\, B = \text{dom}\, B$$
$$\text{dom}(A \cap B) = \text{dom}\, A \cap \text{dom}\, B$$

We can reformulate the first two conditions above as:

$$wp(P_A \,; P_B)\, A = wp\, P_A\, A$$
$$wp(P_A \,; P_B)\, B = wp\, P_B\, B$$

because since P_A refines A, we know that $wp\, P_A\, A = \text{dom}\, A$ and similarly $wp\, P_B\, B = \text{dom}\, B$.

It is almost as if the program P_B is *skip* in the first equation and the program P_A is *skip* in the second equation. Of course we cannot assume that P_A and P_B are always *skip*. Instead we would like to find when one program does not affect another program in the context of a particular goal.

Formally, given two programs p and q and a specification r we would like to know when the following two equations hold:

$$wp(p \,; q)\, r = wp\, p\, r$$
$$wp(p \,; q)\, r = wp\, q\, r$$

Let us say that when the first equation holds p *skips over* q with respect to r, and when the second equation holds q *ignores* p with respect to r.

Let us begin by looking for a sufficient condition for p to skip over q. We expand the first equation to get:

$wp(p \,; q)\, r = wp\, p\, r$

$\Leftrightarrow wp\, p\, (\text{dom}\, q) \setminus \text{dom}((p \fatsemi q) \cap \overline{r}) = wp\, p\, r$ lemma 7.6.1

$\Leftrightarrow wp\, p\, (\text{dom}\, q) \setminus \text{dom}((p \fatsemi q) \cap \overline{r}) = \text{dom}\, p \setminus \text{dom}(p \cap \overline{r})$ definition of wp

$\Leftarrow wp\, p\, (\text{dom}\, q) = \text{dom}\, p \wedge \text{dom}((p \fatsemi q) \cap \overline{r}) = \text{dom}(p \cap \overline{r})$

Expanding the second equation is similar:

$wp(p\,;q)\,r = wp\,q\,r$
$\Leftrightarrow wp\,p\,(\mathrm{dom}\,q) \setminus \mathrm{dom}((p \fatsemi q) \cap \overline{r}) = wp\,q\,r$ — lemma 7.6.1
$\Leftrightarrow wp\,p\,(\mathrm{dom}\,q) \setminus \mathrm{dom}((p \fatsemi q) \cap \overline{r}) = \mathrm{dom}\,q \setminus \mathrm{dom}(q \cap \overline{r})$ — definition of wp
$\Leftarrow wp\,p\,(\mathrm{dom}\,q) = \mathrm{dom}\,q \wedge \mathrm{dom}((p \fatsemi q) \cap \overline{r}) = \mathrm{dom}(q \cap \overline{r})$

LEMMA 7.6.1. *Let p, q and r be programs. Then,*

$$wp\,(p\,;q)\,r = (wp\,p\,(\mathrm{dom}\,q)) \setminus \mathrm{dom}((p \fatsemi q) \cap \overline{r})$$

PROOF. Let p, q and r be programs. We argue as follows:

$wp\,(p\,;q)\,r$
$= \mathrm{dom}(p\,;q) \setminus \mathrm{dom}((p\,;q) \cap \overline{r})$ — definition of wp
$= (wp\,p\,(\mathrm{dom}\,q)) \setminus \mathrm{dom}((p\,;q) \cap \overline{r})$ — lemma 3.1.15
$= (wp\,p\,(\mathrm{dom}\,q)) \setminus ((wp\,p(\mathrm{dom}\,q)) \cap \mathrm{dom}((p \fatsemi q) \cap \overline{r}))$ — lemma 7.6.2
$= ((wp\,p\,(\mathrm{dom}\,q)) \setminus (wp\,p(\mathrm{dom}\,q))) \cup ((wp\,p\,(\mathrm{dom}\,q)) \setminus \mathrm{dom}((p \fatsemi q) \cap \overline{r}))$
— $A \setminus (B \cap C) = (A \setminus B) \cup (A \setminus C)$
$= (wp\,p\,(\mathrm{dom}\,q)) \setminus \mathrm{dom}((p \fatsemi q) \cap \overline{r})$ — set theory

□

LEMMA 7.6.2. *Let p,q and t be programs. Then,*

$$\mathrm{dom}((p\,;q) \cap t) = (wp\,p\,(\mathrm{dom}\,q)) \cap \mathrm{dom}((p \fatsemi q) \cap t)$$

PROOF. Let p, q and t be programs. We argue as follows:

$\mathrm{dom}((p\,;q) \cap t)$
$= \mathrm{dom}(((wp\,p\,(\mathrm{dom}\,q)) \lhd (p \fatsemi q)) \cap t)$ — definition of ;
$= \mathrm{dom}((wp\,p\,(\mathrm{dom}\,q)) \lhd ((p \fatsemi q) \cap t))$ — $(A \lhd B) \cap C = A \lhd (B \cap C)$
$= (wp\,p\,(\mathrm{dom}\,q)) \cap \mathrm{dom}((p \fatsemi q) \cap t)$ — $\mathrm{dom}(A \lhd B) = A \cap (\mathrm{dom}\,B)$

□

The following list summarizes the conditions necessary for a program to skip over or ignore another program:

- For p to skip over q with respect to r it is sufficient that

$$wp\,p\,(\mathrm{dom}\,q) = \mathrm{dom}\,p$$
$$\mathrm{dom}((p \fatsemi q) \cap \overline{r}) = \mathrm{dom}(p \cap \overline{r})$$

- For q to ignore p with respect to r it is sufficient that

$$wp\,p\,(\mathrm{dom}\,q) = \mathrm{dom}\,q$$
$$\mathrm{dom}((p \fatsemi q) \cap \overline{r}) = \mathrm{dom}(q \cap \overline{r})$$

These equations are still not clear enough to reveal to us a simple criterion that we can use in order to determine if two methods can be combined with an `AND` combinator. However, we can use our intuition to come up with more specific criterion to achieve these two equations and then show that the result is sound by proving that our criterion implies the equations above.

It seems reasonable to require that the skipped program must not change any of the variables of the first program that appear in the postcondition (it may change variables that do not appear in the postcondition because such a change will not be detected by the postcondition). In addition, the postcondition must not constrain any of the variables that the skipped program changes (otherwise such changes will reveal a difference when the skipped program is removed). To express these concepts formally we first define the following equivalence relation between states:

DEFINITION 7.6.1. *Any two states s and σ are equivalent with respect to an alphabet α if the values of all the variables in α are equal in both states:*

$$\sim : \mathbb{P}\,Name \rightarrow (State \leftrightarrow State)$$

$$\forall\,\alpha : \mathbb{P}\,Name \bullet \\ \quad \forall\,s, \sigma : State \bullet \\ \quad\quad s \sim_{\alpha} \sigma \Leftrightarrow \alpha \lhd s = \alpha \lhd \sigma$$

It is easy to see that this is indeed an equivalence relation because reflexivity, transitivity and symmetry all follow immediately from the equality used in the definition.

Here, for example, are two equivalent states in a state space that consists of two binary variables x and y. The equivalence is with respect to the alphabet $\{x\}$:

$$00 \sim 01$$
$$10 \sim 11$$

Here and in the following examples we will use the first binary digit to represent the value of x and the second digit to represent the value of y.

Another concept that we will find useful is the complement of an alphabet.

DEFINITION 7.6.2. *The complement of an alphabet α with respect to some program, denoted $\overline{\alpha}$, is the set of all the variables that appear in the alphabet of the program but are not in the alphabet α.*

We do not include the program in the definition above to avoid cluttering the notation. This is not a problem because in the rest of the discussion we always use programs with the same alphabet.

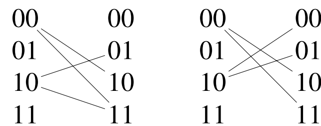
Now we can define formally what it means for a program not to constrain an alphabet.

DEFINITION 7.6.3. *A program* does not constrain the alphabet α *if whenever it maps a state s to a state s' it also maps s to all the states that are equivalent to s' with respect to $\overline{\alpha}$:*

$$unconstrains : \mathbb{P}\,Name \rightarrow \mathbb{P}\,Program$$

$$\forall\,\alpha : \mathbb{P}\,Name;\ prog : Program \bullet \\ \quad prog \in unconstrains\,\alpha \Leftrightarrow \\ \quad\quad \forall\,s, s' : State \mid s \mapsto s' \in prog \bullet \\ \quad\quad\quad \forall\,\sigma : State \mid s' \sim_{\overline{\alpha}} \sigma \bullet s \mapsto \sigma \in prog$$

For example, the left program below constrains the alphabet $\{y\}$ while the right program does not constrain it:

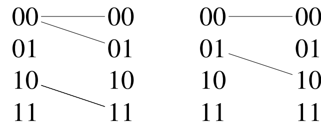


the left program constrains y because from the state $x = 1, y = 0$ we can move only to states where the value of y is 1.

DEFINITION 7.6.4. *A program* does not change the alphabet α *if the variables in* α *in the before and after states are the same. In other words, if all the before and after states of the program are equivalent with respect to* α*:*

$$\begin{array}{l} nochange : \mathbb{P}\,Name \rightarrow \mathbb{P}\,Program \\ \hline \forall\,\alpha : \mathbb{P}\,Name;\ prog : Program \bullet \\ \quad prog \in nochange\,\alpha \Leftrightarrow \\ \qquad \forall\,s, s' : State \mid s \mapsto s' \in prog \bullet s \sim_{\alpha} s' \end{array}$$

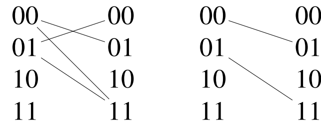
For example, the left program below does not change the alphabet $\{x\}$ while the right program does change it:



DEFINITION 7.6.5. *A program* does not depend on the alphabet α *if it maps equivalent states to equivalent states (with respect to* $\overline{\alpha}$*):*

$$\begin{array}{l} notdepends : \mathbb{P}\,Name \rightarrow \mathbb{P}\,Program \\ \hline \forall\,\alpha : \mathbb{P}\,Name;\ prog : Program \bullet \\ \quad prog \in notdepends\,\alpha \Leftrightarrow \\ \qquad \forall\,s, s', \sigma : State \mid s \mapsto s' \wedge s \sim_{\overline{\alpha}} \sigma \bullet \\ \qquad\quad \exists\,\sigma' : State \bullet \sigma \mapsto \sigma' \in prog \wedge s' \sim_{\overline{\alpha}} \sigma' \end{array}$$

For example, the left program below does not depend on $\{y\}$ while the right program does depend on $\{y\}$:



the right program depends on y because the initial value of y decides if we end in a state in which x is 0 or 1, whereas in the left program for any initial value of y we can find a transition to any value of x.

We can now prove the following theorem:

THEOREM 7.6.3. *Let* p*,* q *and* r *be programs with the same alphabet, and let* α_1 *and* α_2 *bet two sets of names that partition the alphabet such that* r *does not constrain* α_2*, and* q *does not change* α_1*. In addition, assume that* $\operatorname{ran} p \subseteq \operatorname{dom} q$*. Then,*

$$\operatorname{dom}((p \fatsemi q) \cap r) = \operatorname{dom}(p \cap r)$$

PROOF. Assume that p, q, r, α_1 and α_2 satisfy the preamble of the theorem, and let x be a state. First we show that the left hand side is a subset of the right hand side:

$x \in \operatorname{dom}((p \fatsemi q) \cap r)$

$\Leftrightarrow \exists\,z : State \bullet x \mapsto z \in (p \fatsemi q) \wedge x \mapsto z \in r$ — definition of dom

$\Leftrightarrow \exists\,y, z : State \bullet x \mapsto y \in p \wedge y \mapsto z \in q \wedge x \mapsto z \in r$ — definition of $\fatsemi$

$\Rightarrow \exists y, z : State \bullet x \mapsto y \in p \wedge y \mapsto z \in q \wedge x \mapsto z \in r \wedge y \sim_{\alpha_1} z$ — q does not change α_1

$\Rightarrow \exists y, z : State \bullet x \mapsto y \in p \wedge y \mapsto z \in q \wedge x \mapsto z \in r \wedge y \sim_{\alpha_1} z \wedge x \mapsto y \in r$

r does not constrain α_2 and $\alpha_2 = \overline{\alpha_1}$

$\Rightarrow \exists y : State \bullet x \mapsto y \in p \wedge x \mapsto y \in r$ — predicate logic

$\Leftrightarrow \exists y : State \bullet x \mapsto y \in (p \cap r)$ — set theory

$\Leftrightarrow x \in \mathrm{dom}(p \cap r)$ — set theory

Since x is an arbitrary state we have shown that the left hand side is a subset of the right hand side. Now we show that the right hand side is a subset of the left hand side:

$x \in \mathrm{dom}(p \cap r)$

$\Leftrightarrow \exists y : State \bullet x \mapsto y \in p \wedge x \mapsto y \in r$ — definition of dom

$\Rightarrow \exists y, z : State \bullet x \mapsto y \in p \wedge x \mapsto y \in r \wedge y \mapsto z \in q$ — the assumption $\mathrm{ran}\, p \subseteq \mathrm{dom}\, q$

$\Rightarrow \exists y, z : State \bullet x \mapsto y \in p \wedge x \mapsto y \in r \wedge y \mapsto z \in q \wedge y \sim_{\alpha_1} z$ — q does not change α_1

$\Rightarrow \exists y, z : State \bullet x \mapsto y \in p \wedge x \mapsto y \in r \wedge y \mapsto z \in q \wedge y \sim_{\alpha_1} z \wedge x \mapsto z \in r$

r does not constrain α_2 and $\alpha_2 = \overline{\alpha_1}$

$\Rightarrow \exists z : State \bullet x \mapsto z \in (p \fatsemi q) \wedge x \mapsto z \in r$ — definition of $\fatsemi$

$\Leftrightarrow \exists z : State \bullet x \mapsto z \in ((p \fatsemi q) \cap r)$ — set theory

$\Leftrightarrow x \in \mathrm{dom}((p \fatsemi q) \cap r)$ — definition of dom

Since x is an arbitrary state we have shown that the right hand side is a subset of the left hand side. From both arguments it now follows that both sides are equal and therefore we have proven the theorem. □

Now let us consider what are sufficient conditions for q to ignore p. First, we may note that because q appears after p it can change any variable it likes. However, q must not depend on any initial variable that is changed by p

We can prove the following theorem:

THEOREM 7.6.4. *Let p, q and r be programs with the same alphabet, and let α_1 and α_2 be two sets of names that partition the alphabet such that r does not constrain α_1, p changes only α_1 and q does not depend on α_1. In addition, assume that* $\mathrm{dom}\, q \subseteq \mathrm{dom}\, p$. *Then,*

$$\mathrm{dom}((p \fatsemi q) \cap r) = \mathrm{dom}(q \cap r)$$

PROOF. Let p, q and r be as they are defined in the preamble of the theorem, and let x be a state. First we show that the left hand side is a subset of the right hand side:

$x \in \mathrm{dom}((p \fatsemi q) \cap r)$

$\Leftrightarrow \exists z : State \bullet x \mapsto z \in (p \fatsemi q) \wedge x \mapsto z \in r$ — definition of dom

$\Leftrightarrow \exists y, z : State \bullet x \mapsto y \in p \wedge y \mapsto z \in q \wedge x \mapsto z \in r$ — definition of $\fatsemi$

$\Rightarrow \exists y, z : State \bullet x \mapsto y \in p \wedge y \mapsto z \in q \wedge x \mapsto z \in r \wedge y \sim_{\alpha_2} x$ — p changes only α_1

$\Rightarrow \exists y, z, \sigma : State \bullet x \mapsto y \in p \wedge y \mapsto z \in q \wedge x \mapsto z \in r \wedge y \sim_{\alpha_2} x \wedge x \mapsto \sigma \wedge \sigma \sim_{\alpha_2} z$

q does not depend on α_1 and $\alpha_2 = \overline{\alpha_1}$

$\Rightarrow \exists \sigma : State \bullet x \mapsto \sigma \in q \wedge x \mapsto \sigma \in r$ — r does not constrain α_1

$\Leftrightarrow \exists \sigma : State \bullet x \mapsto \sigma \in (q \cap r)$ — set theory

$\Leftrightarrow x \in \mathrm{dom}(q \cap r)$

This shows that the left hand side is a subset of the right hand side. Now we show the other direction:

$x \in \mathrm{dom}(q \cap r)$

$\Leftrightarrow \exists z : \mathit{State} \bullet x \mapsto z \in q \wedge x \mapsto z \in r$ — definition of dom

$\Rightarrow \exists y, z : \mathit{State} \bullet x \mapsto z \in q \wedge x \mapsto z \in r \wedge x \sim_{\alpha_2} y \wedge x \mapsto y \in p$

the assumption $\mathrm{dom}\, q \subseteq \mathrm{dom}\, p$ and p does not change α_2

$\Rightarrow \exists y, z, \sigma : \mathit{State} \bullet x \mapsto z \in q \wedge x \mapsto z \in r \wedge x \sim_{\alpha_2} y \wedge x \mapsto y \in p \wedge y \mapsto \sigma \in q \wedge \sigma \sim_{\alpha_2} z$

q does not depend on α_1

$\Rightarrow \exists y, z, \sigma : \mathit{State} \bullet x \mapsto z \in q \wedge x \mapsto z \in r \wedge x \sim_{\alpha_2} y \wedge$
$\quad x \mapsto y \in p \wedge y \mapsto \sigma \in q \wedge \sigma \sim_{\alpha_2} z \wedge x \mapsto \sigma \in r$ — r does not constrain α_1

$\Rightarrow \exists \sigma : \mathit{State} \bullet x \mapsto \sigma \in (p \fatsemi q) \wedge x \mapsto \sigma \in r$ — definition of $\fatsemi$

$\Leftrightarrow x \in \mathrm{dom}((p \fatsemi q) \cap r)$ — set theory

This shows that the right hand side is a subset of the left hand side. From both arguments we can deduce that the left hand side is equal to the right hand side and therefore the proof is complete. □

DEFINITION 7.6.6. *We say that the specifications A and B are* independent *if A does not constrain the alphabet of B and B does not constrain the alphabet of A.*

LEMMA 7.6.5. *If A and B are independent then*

$$\mathrm{dom}\, A \cap \mathrm{dom}\, B = \mathrm{dom}(A \cap B)$$

PROOF. We prove the lemma by showing that each side of the equality is a subset of the other side. First we show that every member of $\mathrm{dom}(A \cap B)$ is a member of $\mathrm{dom}\, A \cap \mathrm{dom}\, B$:

$x \in \mathrm{dom}(A \cap B)$

$\Leftrightarrow \exists y : \mathit{State} \bullet x \mapsto y \in A \cap B$

$\Leftrightarrow \exists y : \mathit{State} \bullet x \mapsto y \in A \wedge x \mapsto y \in B$

$\Rightarrow (\exists y : \mathit{State} \bullet x \mapsto y \in A) \wedge (\exists y : \mathit{State} \bullet x \mapsto y \in B)$

$\Leftrightarrow x \in \mathrm{dom}\, A \wedge x \in \mathrm{dom}\, B$

$\Leftrightarrow x \in \mathrm{dom}(A \cap B)$

Now we show that every member of $\mathrm{dom}\, A \cap \mathrm{dom}\, B$ is a member of $\mathrm{dom}(A \cap B)$. Let x be a member of $\mathrm{dom}\, A \cap \mathrm{dom}\, B$, and assume that $x \notin \mathrm{dom}(A \cap B)$. This means that

$$\forall y, z : \mathit{State} \bullet x \mapsto y \in A \wedge x \mapsto z \in B \Rightarrow y \neq z$$

Let y and z be states such that $x \mapsto y \in A$ and $x \mapsto z \in B$. Because B does not constrain the alphabet of A we can deduce that

$$x \mapsto z \in B \Rightarrow x \mapsto ((\alpha B \lhd z) \cup (\alpha A \lhd y)) \in B$$

Similarly, because A does not constrain the alphabet of B we can deduce that

$$x \mapsto y \in A \Rightarrow x \mapsto ((\alpha A \lhd y) \cup (\alpha B \lhd z)) \in A$$

But this contradicts the assumption that x is not in the domain of $A \cap B$. This contradiction means that our assumption is false and therefore x must be a member of $\mathrm{dom}(A \cap B)$. □

Finally, by taking theorems 7.6.4 and 7.6.3 together with lemma 7.6.5 we can enumerate the conditions that are sufficient to ensure that the sequential composition of two specifications refines their conjunction:

THEOREM 7.6.6. *Let A and B be two independent specifications. Let P_A be a program that does not change α_B, and P_B be a program that does not change α_A and does not depend on α_A. Then,*

$$P_A \sqsupseteq A \wedge P_B \sqsupseteq B \Rightarrow P_A \,;\, P_B \sqsupseteq A \; AND \; B$$

PROOF. To prove the theorem we show that

$$\mathrm{dom}(A \cap B) = \mathrm{dom}\, A \cap \mathrm{dom}\, B$$
$$wp(P_A \,;\, P_B)\, A = \mathrm{dom}\, A$$
$$wp(P_A \,;\, P_B)\, B = \mathrm{dom}\, B$$

The first equation follows from lemma 7.6.5 and the assumption that A and B are independent. The second equation follows from theorem 7.6.3 and the assumptions: A does not constrain α_B, P_B does not change α_A. The third equation follows from theorem 7.6.4 and the assumptions: B does not constraint α_A, P_A does not change α_B and P_B does not depend on α_A. □

It is easy to see that if the syntactic description of A does not refer to any variables in B and vice versa then they are independent. In addition, because the code that implements each specification uses only the variables that appear in the specification, the other conditions are satisfied as well. Therefore, to ensure that we can create the conjunction of two specifications A and B it is enough to check that they are syntactically independent (see definition 4.6.1).

7.7. Discussion

We have described a technique for automatically generating a program from a state based specification. We base the code of a primitive specification on the structure of its postcondition and then add the weakest precondition of the generated code to the original precondition. This ensures that the code refines the (modified) specification. To generate the code of a combinator we combine the code of the combinator's arguments. In particular, we have investigated the conditions that allow us to conjoin two specifications using an `AND` combinator in such a way that it could be implemented by the sequential composition of their implementations.

One may say that because we compile Booster specifications into executable programs, the Booster language is just another kind of programming language. After all there is a strong connection between equality and assignment, implication and conditional statements, conjunction and sequential composition. However, there is in fact an important difference between Booster and a programming language.

In a programming language the rule 'garbage in — garbage out' always applies. Anything we write will get executed. In contrast, if we write an inconsistent specification in Booster it will either be modified to become consistent, or it will be prevented from running at all.

The ideas in this chapter first appeared in the paper [**22**]. However they were not formally related to the semantics of Booster and their explanation did not contain the detailed semantics of the guarded command language and it did not contain the detailed proofs that demonstrate the soundness of the approach.

Updates in the context of aliasing. We have described a simple weakest precondition rule for attribute assignment which deals correctly with object references. Given a reference to some object *obj* with a field *field*, the weakest precondition of assignment to *obj.field* is

$$wp\,(obj.field := expr, p) = p[field := field \oplus \{obj \mapsto expr\}]$$

The same rule was defined by Leino in [**37**] using an axiomatic definition of the update function which he calls *store*:

$$wp\,(obj.field := expr, p) = p[field := store(field, obj, expr)]$$

where the meaning of *store* is axiomatically defined to be

$$\begin{array}{l} store : (ObjectID \mapsto Value) \times ObjectID \times Value \mapsto (ObjectID \mapsto Value) \\ \hline \forall\, obj : ObjectID;\ field : ObjectID \mapsto Value;\ v : Value \bullet \\ \quad store(field, obj, v)(obj) = v \\ \forall\, obj, obj' : ObjectID;\ field : ObjectID \mapsto Value;\ v : Value \mid obj' \neq obj \bullet \\ \quad store(field, obj, v)(obj') = obj.field \end{array}$$

Deductive program synthesis. Deductive program synthesis [**39, 47**] is a technique in which a program is extracted from a constructive proof of the specification. The proof demonstrates that the specification is implementable, and the program is a constructive witness of the demonstration. The work of Poernomo, Crossley and Wirsing [**47**] provides a thorough introduction to the various approaches for implementing deductive program synthesis. Here we demonstrate the technique using the work of Manna and Waldinger [**39**], which is the technique used by the Amphion automatic program synthesiser[3]:

In the approach of Manna and Waldinger specifications are logical formulas that can be written in the following pattern:

$$\forall\, input \bullet P(input) \Rightarrow \exists\, output \bullet R(input, output)$$

Many common algorithms can be written as instances of this pattern. For example, finding an element in a sequence can be described as:

$$\forall\, list : \mathrm{seq}\, Item;\ item : Item \bullet item \in \mathrm{ran}\, list \Rightarrow \exists\, i : \mathrm{dom}\, list \bullet list\, i = item$$

And sorting a list can be described as:

$$\forall\, in : \mathrm{seq}\, Item \bullet true \Rightarrow \exists\, out : \mathrm{seq}\, Item \bullet ordered(out) \wedge permutation(in, out)$$

Because such specifications are logical formulas, we can try to prove them. If we manage to construct the proof in a way that tells us for each input what is the corresponding output, then we can extract a program from the proof.

In order to perform this task we use a structure, called a *sequent*, that consists of three sections: a list of assertions $A_i(a,x)$, a list of goals $G_i(a,x)$ and a list of outputs $t_i(a,x)$. We use a to denote the constants in the formula and x to denote the free variables of the formula. The meaning of a sequent is that if all the assertions hold for every x then there is at least one x that satisfies at least one goal:

$$\forall\, x \bullet A_1 \wedge \ldots \wedge \forall\, x \bullet A_n \Rightarrow \exists\, x \bullet G_1 \vee \ldots \exists\, x \bullet G_m$$

The output $t_i(a,x)$ plays no role in the proof but it records the manipulations we perform during the proof, and we arrange it in such a way that when the proof completes it describes the requested program.

To make it easier to manipulate the sequent we write it in a tabular form. Each line in the table contains either an assertion or a goal but never both:

assertions	goals	outputs
$A_i(a,x)$		$t_i(a,x)$

or

assertions	goals	outputs
	$G_i(a,x)$	$t_i(a,x)$

To generate a program from a specification we begin with an assertion that contains the assumption, a goal that contains the constraint, and an output that contains the output variable:

assertions	goals	outputs
$P(input)$	$R(input, output)$	$output$

We now begin to manipulate the formulas and terms in the sequent by applying various rewriting rules and transformations. For example, we can split a goal of the form $P \vee Q$ into two goals, the first contains P and the second Q, or we can split an assertion of the form $P \wedge Q$ into two assertions P and Q. We can also take the negation of an assertion as a goal, or take the negation of a goal as an assertion. In addition,

[3] The Amphion system generates models of planetary motion from domain specific specifications.

we can use pattern matching algorithms to instantiate generic rules with the particular instances in the sequent. For example, if we have the rule

$$u + 0 \Rightarrow u \quad \text{if } u \in \mathit{integer}$$

then from the sequent

assertions	goals	outputs
	$x + z < a$	z

we can derive the sequent

assertions	goals	outputs
	$x < a$	0

In other words, if we want to find a z such that $x + z < a$ then provided we can show that $x < a$ we can use $z = 0$ as the output.

The paper describes many more rules, and even shows how it is possible to use inductive proofs in order to derive recursive algorithms. Eventually if we manage to reach a sequent of the form

assertions	goals	outputs
	true	t

or

assertions	goals	outputs
false		t

then we have proved the theorem and t is the desired program.

In principle this approach is more powerful then the technique we use in Booster, especially because it can (again in principle) generate iterative and recursive programs that implement inductive specifications. However in practice the automatic program generators that use this technique avoid dealing with such specifications and instead adopt the same solution that is used by Booster, namely to encapsulate the low level code that requires explicit iterations and recursion in domain specific libraries:

> Another practical insight concerns the choice of the composition mechanisms - such as conditions and recursion - used during synthesis. Although constructive synthesis can generate them all, recursion formation is by far the most difficult composition. If sufficiently many library routines performing sophisticated calculations are provided, then synthesis need not really "lift" recursion from them but may rather amount to generating an adequate straight-line program — with just sequential and conditional composition — from the specification.
>
> Achievements and Prospects of Program Synthesis [**23**].

Because deductive program synthesis relies on an automatic theorem prover, it shares the same problems from which automatic theorem proving suffers. Theorem proving is undecidable and therefore we cannot be sure that the theorem prover will always find a proof (and therefore no program will be generated). In such a case it is not clear if the reason is that the prover is not powerful enough or because the specification contains a mistake. In practice this problem proves to be a barrier for the adoption of such systems because the engineer does not know if it is worthwhile to invest more time to prove the theorem, or instead to look for a counter example that demonstrates that the theorem is wrong.

CHAPTER 8

Discussion

We have described the theory that underlies the Booster system. We have seen how it is possible to transform, in the context of a particular domain, an object model and the specification of its behavior into a working program. It is now time to consider the ideas in this work in a wider context. To discuss their limitations, to suggest directions for future research and to see what we can learn from them not only about the implementation of a particular domain specific code generator, but also about the wider context of software engineering.

8.1. The development of the theory

First however, we provide a brief outline of the development of the ideas in this work. This is also an opportunity to highlight my own contributions and to recognize the contributions of the other members of the Booster research team.

8.1.1. Model completion. We have begun the theoretical work on Booster by trying to justify the completion rules. I thought of modeling associations as relations in order to theoretically justify the completion rules of Booster, and James Welch suggested to model each end of the association as a separate relation and to tie them together by asserting that they are inverses of each other. However, when we first published these ideas in [**58**] we did not have a well defined semantics for Booster so it was not clear in what way the completions affect the semantics of a Booster model. My idea, which is the one I have described in chapter 6, is to insist that the completions of each method are legal only if they are implied by the invariants of the model and the postcondition, and therefore adding them does not change the semantics of the model. This means that my completions can never strengthen the preconditions and this limits the kinds of completions the Booster compiler is allowed to perform. We describe a more flexible approach in the journal paper [**59**]. It allows to strengthen preconditions, but the result is a different model. This may not be so important in practice, but it makes it more difficult to formally express the relation of the change to the original model.

8.1.2. Semantics. We have developed the semantics of Booster after the work on association invariants. The idea of treating association invariants (and later all attributes) as relations led me to represent the state of a Booster model using a single schema that contained the state of all the relations in the model. This has prompted Jim Davies to suggest that an entire Booster model is in fact a single abstract data type, and that consequently classes are not abstract data types. This idea was the essence of [**15**]. But the semantics of this paper is very abstract, essentially ignoring the details of the Booster language and of the code generation process. This makes it difficult to show that the completion rules and the code generation process (which were published separately in other papers [**58, 22**]) agrees with the semantics described in the paper. My contribution in the thesis was to refine the abstract model to include the explicit structure of the Booster language, and to show that all the different components of the compiler behave according to this refined model.

In addition, the Booster semantics of [**15**] is too strong because it depends on the implementation strategy (represented in the paper as the function P and used to define the semantics of Booster operations). In contrast, the semantics I have described in the thesis gives postconditions a relational meaning that is independent of any implementation strategy.

8.1.3. Consistency enforcement. The original Booster compiler was strengthening the preconditions of methods by catching patterns of postconditions that would otherwise yield contradictory specifications. However these patterns were inserted manually, which meant that the Booster compiler had to be extended every time a new combination of contradictory formulas would appear in a postcondition. My idea was to use the generated code to automatically create the necessary protection by calculating the weakest precondition of the generated code on the postcondition, and adding the result to the original precondition. This idea was published in [**22**] but the theory of that paper was using predicate transformers, which made it difficult to connect with the work on association invariants and with the abstract relational semantics of an entire Booster model. By interpreting weakest preconditions as relational operators I was able to use the same mathematical framework to integrate all three subjects.

8.2. Contribution to Booster in practice

The Booster system was in existence some two years before we have begun to study its theory. During this time it has been used to successfully implement the online database of the software engineering school at the university of Oxford. It may have been argued that because Booster works in practice there is no need for a theoretical study. After all, Booster clearly works in practice so why waste time struggling with mathematical models to prove that it works. Nevertheless, the theoretical investigation proved very useful in practice because it has revealed important areas where the current implementation of Booster was wrong, simplified the semantics of method combinators, eliminated unecessary constructs (e.g. change lists), provided clear rules for the correct strengthening of method preconditions and showed how the postcondition language can be extended to support a richer set of logical operators.

8.2.1. Clarifying the rules of model completion. When we first formalized the rules for model completion we had to systematically enumerate all the different combinations of associations and to devise a rule for each combination. We then tested each combination using the current Booster implementation to see if it agrees with our rules. In some cases, particularly in combinations that were not used in practice until then, we have found that the implementation was actually wrong. For example the completion of an operation that removes an object x from the many end of a one-to-many association was also destroying x (removing it from its extent). This rule was not taking into account the possibility that there may be other objects referencing x.

The mathematical analysis not only gave us a framework in which we can develop the completion rules but also provided a systematic way to ensure that all the allowed combinations work as expected.

8.2.2. Simplifying the combinators. Originally, Booster had no `AND` and `THEN` combinators but a single combinator called `ALSO`. After spending considerable time trying to understand the behaviour of `ALSO` we came to the conclusion that it behaves as following (here we describe it in terms of `AND` and `THEN` but at the time we did not have these combinators so the description was even emore complicated):

$$M_1 \text{ ALSO } M_2 = \begin{cases} M_1 \text{ THEN } M_2 & \text{when } wp(M_1, pre(M_2)) \text{ holds} \\ M_1 \text{ AND } M_2 & \text{when } pre(M_1) \wedge pre(M_2) \text{ holds} \\ M_1 & \text{when only } pre(M_1) \text{ holds} \\ M_2 & \text{when only } pre(M_2) \text{ holds} \end{cases}$$

Once we have formulated the behaviour of `ALSO` it was clear that even though it is very complicated, we never use its full power because in practice we use `ALSO` either to put two combinators in parallel or to put them in sequence. The complicated mixing of these two behaviours is almost never needed and

in the cases where a more complicated behaviour is required it can be achieved with the aid of the `OR` combinator. Therefor we have decided to decompose `ALSO` into two more simple combinators: one is a combinator that puts the two methods in parallel (to achieve both their goals at once) and another that puts them in sequence. We then called the fist combinator `AND` and the second `THEN`.

8.2.3. Clarifying the meaning of old variables. Informally, the meaning of an old variable in a method is the value of the variable immediately before the method begins executing. This was our understanding of old variables prior to the development of the theory. It was sufficient to understand specifications and programs where a simple variable was used. But it was more difficult to understand statements such as

$b_0.c := e$

What is the meaning of assignment when an old variable is involved? is this statement even legal? it appears that we are changing the attribute of an old object, but how can we do that? without a clear semantics it is very difficult to understand. However with a relational model of attributes we suddenly get a very simple answer to all these questions:

$b_0.c := e$

$= c := c \oplus \{b_0 \mapsto e\}$

First it is clear that the entity that is changed is the c attribute/relation and that the b_0 variable is not affected. Second, there is no problem updating the attribute because it is the variable b_0 that is old, not the object it points to.

8.2.4. Eliminating the change list. The relational model of attributes also contributed to the understanding and eventual removal of the change-list from method specifications. Originally a method specification contained in addition to the pre- and postconditions, also a change-list: a list of variable names that specified which variables the method is allowed to change. Thus a primitive method specification was essentially identical to a specification statement in the refinement calculus. For example, the method specification

```
M(true | a.b | a.b = e)
```

means that we can only change the `b` attribute of the object `a`.

This feature looks very reasonable, but it has several problems. First, as a consequence of the relational model, it is clear that the variable `a` should not be in the change list because is not changed and should not be changed. Only the relation `b` may change.

Second, if we interpret the change list as defining the *only* variables that may be changed then this means that most of the completion rules do not refine the specification because they must modify the other (unmentioned) end of the association.

Finally, in the presence of combinators the change-list is too crude a device to keep track of which variables change. For example the combinator expression

```
(x > 0 | y | y  = 1) OR (x <= 0 | z | z = 1)
```

is not equal to the primitive method

```
(true | y,z | (x_0 > 0 => y = 1) & (x_0 <= 0 => z = 1))
```

but to the more complicated specification

```
(true | z,y | (x_0 > 0 => (z = z_0 & y = 1)) &
              (x_0 <= 0 => (y = y_0 & z = 1)))
```

because we must keep track of the fact that `y` changes only when `x` is positive and that `z` changes only when `x` is not positive.

These arguments were an important factor in convincing us that the change-list should be removed from the language.

8.2.5. Defining the correct strengthening of preconditions. In the original Booster implementation preconditions are strengthened by a set of pattern matching rules that identify 'problematic' combinations of predicates in the postcondition and generate predicates in the precondition that protect against these problematic combinations.

The theoretical investigation has led us to devise a precise and simple law that defines the exact nature of strengthening preconditions: the precondition must be strengthened by the weakest precondition of the generated code on the postcondition.

This law can be used either as an implementation (as suggested in this thesis) of predicate strengthening, or as a precise specification if predicate strengthening is implemented using some other (perhaps more efficient) set of transformations.

As in the case of the association invariant rules we have used the weakest precondition rule as an oracle to test (and correct) the precondition strengthening algorithm of the current Booster implementation.

8.2.6. Extending the postcondition language. The original Booster language restricts the postcondition language into a small subset of predicate logic that corresponds to the predicates from which we can generate programs. For example negations and disjunctions are forbidden.

However, the use of weakest preconditions to strengthen the precondition makes it possible to support all the logical operators in postconditions. Predicates such as negation and disjunction, from which we do not generate code, can now play a role of passively achieving the postcondition because they are propagated back to the precondition by the weakest precondition calculator.

In addition, several times during the development of Booster the implication operator has been removed and then reinserted to the postcondition, without any final decision about its fate. The theoretical investigation has revealed that it is important to keep implication but in order for it to be useful as a code generation pattern its antecedent must be constrained to refer only to old variables.

8.2.7. Summary. To conclude, we have seen that the theoretical investigation has contributed to the development of Booster in practice by serving as a precise oracle against which the existing implementation can be tested, by highlighting and removing troublesome features (such as change lists and the `ALSO` combinator) and replacing them with simple and sound alternatives, and by suggesting directions where the language can safely be extended to become more expressive (such as relaxing the restrictions on the postcondition language).

8.3. Booster in practice

Although Booster is a research project, it has originated from a concrete need — to provide an information management system to support the administration of the software engineering programme at the university of Oxford. Not only has Booster been able to successfully provide such a system but also in the last few years it has been used to implement other systems in the university, all of which are operating to the full satisfaction of their users. In this section we briefly describe each of these systems.

8.3.1. The Software Engineering school. This is the oldest system in use, in operation since 2003. We have already described it in other sections of the thesis (see for example section 2.4.2). It is currently the largest and most complicated Booster system. It consists of about 3000 lines of Booster code and manages about 1000 users. Since it has been put to use it has never failed. [**14**]

8.3.2. Graduate Studies. The graduate studies system [**46**] is used to keep track of the progress of all the graduate students at the university of Oxford. It was commissioned by the university as a working prototype for the purpose of gathering the system's requirements. The Booster system currently manages and serves about 11,000 people. It is in operation since Hilary Term 2007.

8.3.3. IFM 2007. This system was used to manage the submission and review process of the papers for the IFM 2007 conference [**2**]. Authors submitted their papers for review through the system and the reviewers downloaded the papers and uploaded their reviews. The system was also used to manage the registration process and to produce invoices and receipts. However partly because of time constraints and partly because it was felt that users will be reluctant to store their credit card numbers in a research system, the interface to the financial operations was available only to the operators of the system. The system supported about 70 authors and 150 registrations.

8.3.4. Kellogg college. This system was introduced in 2007 to replace a few Excel spreadsheets and a large set of scripts that where used to store the details of students and their accommodation, to keep track of and calculate college fees and to store information about fellows and alumni of Kellog College in Oxford. The system has three users and stores information on about 300 people.

8.3.5. The Computing Laboratory web site. This system is used to store the content of the Oxford University Computing Laboratory web site [**1**]. Currently the system stores the public details of the researchers and students in the Computing laboratory, including their lists of publications. However it is evolving to support more information. This system is in operation since May 2008. It is currently used by about 300 people and contains around 3000 publications.

8.4. Model completion

The specification of a system that consists of a large number of relationships is often very tedious. The reason is that the relationships force us to explicitly specify the consequences of the operation on the referenced entities. Some languages, for example Z, allows the consequences to remain implicit, but while such a feature makes it possible to specify very compact models it also makes their implementation more difficult, because in order to implement the system we must extract these implications ourselves.

In contrast, with Booster we can specify only the intention of the operation, and then let Booster automatically deduce their consequences by analyzing the invariants of the model.

This technique makes it easier to specify operations because we do not have to deal with all their consequences, but instead focus on stating their main purpose — their intention. In addition, it makes the specification easier to read because it contains only the essential purpose of the operation. Of course, we should always look at the completed version in order to ensure that the interaction between the intention and the invariants did not give rise to unexpected consequences.

Invariants are commonly thought of as a device for maintaining the structural integrity of systems. But as we have shown, they can also play an important role in directing and simplifying the specification of operations.

8.5. Programming with specifications

The Booster language encourages a bottom up approach to the construction of specifications. Instead of breaking up a large specification into smaller parts, as stepwise refinement advocates, we construct the complete specification out of smaller parts.

This technique is easier to automate because it eliminates the search problem that is a fundamental difficulty with traditional approaches to program generation — how to select in a reasonable amount of time, a correct and efficient implementation from the huge space of possible programs. The reason we eliminate this search is that the implementation of the combinators is given by the combination of the implementations.

But this bottom up style is often a better way to construct large systems than the traditional top down approach that is advocated by many formal methods.

The essential problem of stepwise refinement is that it assumes that we can keep refining until we get to the lowest level (programming language) building blocks. This may be right for algorithm development, but in system development we are often forced to use existing subsystems. The final refinement must be mapped to the capabilities of these subsystems.

Software engineers never write all the code themselves, but instead choose (and often this choice is dictated by external forces) a particular operating system, database management system, user interface library, programming language and so on and so forth. These choices create a toolkit that defines and restricts the way they can implement their systems.

The engineers, therefore, must create a design that can be mapped to the capabilities of their toolkit. In fact, the engineers have more flexibility to change the specification of the system than to change the capabilities of their toolkit. Therefore, to develop a large system it makes much more sense to begin with the specifications of the available building blocks, and to compose them into a specification that pleases the customer (of which there are many), rather than beginning with the specification of the complete system and searching for a decomposition (that may not exist) that will fit the existing toolkit.

8.6. Domain specific formal methods

The problem with the refinement approach to formal development is that it requires the engineer to use a very big hammer (theorem proving) to crack open many small nuts. A large system often contains tens of thousands of methods. Formally proving their proof obligations, even if eighty percent of them are discharged automatically, is a lot of intellectual grunt work because the subject of the proofs is almost always trivial.

In contrast, the approach that we have describe here focuses on eliminating as much as possible the tedious generation of simple operations. Booster focuses on a specific domain. It assumes that within this particular domain it is possible to use the specification of an operation as a description for the actions that are required to implement it. Therefore, we can reasonably expect that a simple code generation strategy suffices for implementing the operations.

Not only that, but if a generated operation fails to achieve its specification then there is a very good chance that the mistake is actually in the specification.

This is why we can get away with strengthening the precondition. It is not refinement, but it formalizes something that always happens in practice. Whether the reason is lack of time, skill or knowledge, programmers must often make their specifications easier to implement. There is nothing sacred about the specification. We may change it in whatever way we like, as long as the version we implement satisfies the customer.

Indeed, the specification of a large system often contains many mistakes. Of course, most of the mistakes are technical and quite simple to correct. Nevertheless they are mistakes, and if they are not corrected they usually find their way into the implementation.[1]

8.7. Solving problems at the right level

An important point that is made explicit in this work is that specifications are not ideal products, and their development suffers in practice from the same problems as the development of programs. Since humans are not very good at creating perfect formal structures, there is no reason to assume that they will create correct specifications.

However, many problems that are difficult to handle at the level of the programming language are easier to handle at the level of the specification language. In this work we have seen two important examples. The first is the problem of how to maintain the consistency of operations in the presence of associations. As we have already discussed in chapter 6, solving the problem at the level of the

[1] Unfortunately there is no correlation between the simplicity of a mistake and the extent of the damage that it may cause if not fixed.

programming language is complicated because after the invariant is already broken we cannot rely on the behavior of the rest of the system (which we use to fix the problem).

The second problem is how to deal with undefined expressions. The only way to deal with this problem at the level of the programming language is to throw an exception when for example a computation is about to divide by zero. The problem with this approach is that we often cannot know in which state the computation was left when the exception was thrown. Indeed, it is very difficult to reconcile exceptions with the transactional Hoare logic view of operations that is used by many specification languages. It is much better to change the specification, as we do in this work, to ensure that no undefined expression will be evaluated.

8.8. Object oriented specifications

Most approaches to the theory of objects have been focused upon the features of programming language implementations: the models are seen as a way of organizing and abstracting object-oriented programs, rather than as specifications in their own right. But there is a difference between implementation and specification, and this implies a difference between the tools we use in each case, even if both the implementation and the specification are object oriented.

For example, encapsulation hides the internal (implementation) details of an object from its external environment, exposing only the abstract interface of the object. However, in a specification everything is already (by definition) abstract so there is no need to hide anything. On the contrary, encapsulating parts of the specification makes it impossible to reason about and analyze them. This defeats the purpose of a specification which is to provide a model that we can reason about and analyze. Once we encapsulate a specification detail it becomes unavailable for analysis, and therefore loses the only reason for including it in the model.

Another feature of object oriented programming whose role in a specification is dubious, is dynamic binding. Dynamic binding makes it possible to create more flexible programs and to reuse code (since whenever we create a new implementation we reuse all the code that uses the interface) but it has no role as a specification device since in a specification everything is an interface. There is no implementation and nothing is executed.

The only feature of objects that requires an essential difference at the specification level is the notion of object identity and its associated reference semantics.

References are one of the most powerful ideas of object orientation, but they also greatly complicate our ability to reason about systems that use them. References are powerful because they make it easy to create models that capture correctly the interdependency between information structures — if the contact details of an employee are changed then all the departments that refer to this employee will automatically see the changes. Information automatically propagates through the references into every part of the system that uses the information. But this is also the source of the difficulty, because it means that operations no longer have a local effect.

Traditional formal specification notations (like Z,B, VDM, and others) have no direct support for objects with reference semantics. Attempts to develop general conventions that deal with object references and with their implications produce a great deal of visible machinery that is often more complicated than the systems we are trying to model [**26**].

In contrast, Booster has built in support for describing objects with reference semantics, and its underlying model provides a simple way to deduce the effects of changing the attributes of objects in the presence of references and aliases.

8.9. Classes are not abstract data types

As we have seen in chapter 5 classes are not abstract data types. Whenever classes refer to each other we cannot understand them in isolation but must instead consider the network that they create as a single component (or abstract data type).

Just because classes capture entities in the problem domain does not mean that they are independent. On the contrary, classes that represent entities that depend on each other must also depend on each other in the model. There is no meaning to an employee class without the corresponding company class.

But because classes are not abstract data types it does not mean that abstract data types are unnecessary. On the contrary, abstract data types are essential tools for the construction of large systems. They provide the encapsulation that makes it possible to build a large system out of smaller components that were developed and verified independently. We could not build today's systems without the lists, queues, balanced-trees and hash-tables that are encapsulated in the standard libraries of nearly every programming language.

But because abstract data types have no explicit representation in today's languages they are implemented using classes. This causes a great deal of confusion since the same concept is used to implement different things [**60**].

The role of classes is to structure a program as a model of its problem domain. The classes represent entities of the problem domain at a particular level of abstraction. But the model that they implement provides a higher level of abstraction. We may create an on-line bookshop out of entities such as books, suppliers, storage depots, and so on, but the bookshop itself is a component that abstracts the details of the shipment and delivery of the books.

8.10. Limitations and future work

Automating the generation of software is effective only if it is specific. This means that any automatic code generator will have limitations. Some of the limitations are fundamental, others are not and may be relaxed after further research. Unfortunately it is not always possible to know in advance which limitations fall into which category. Therefore in the rest of the discussion we discuss the current limitations of Booster and describe possible avenues of future work which may, we hope, overcome some of the limitations or show more clearly why they are fundamental to the approach.

8.10.1. Supporting new operations and domains. At the beginning of the thesis we have said that the domain of Booster is information management systems. This is a very large domain. It consists of many sub-domains that have their own particular needs. A management system that keeps track of patients in a hospital is very different from a management system that keeps track of paper reviews for a conference. For example we may be required to encrypt patient records to keep them confidential, or we may be required to send or receive information from a proprietary system. In such cases we must develop this functionality using other means (preferably using formal methods) and embed it into Booster as primitive operators.

Currently this process requires changing the Booster engine itself. However, a better way is to connect Booster with a refinement based toolkit such that sophisticated operations for new domains could be rigorously developed and verified and then automatically introduced to Booster.

It may also be possible to apply Booster to domains other than information management systems. For example, data structures such as lists and tress are very similar to the structures that we create in a typical information system. A bidirectional list for example can be concisely specified in Booster by creating a bidirectional association between the left and right attributes of the nodes in the list. So we may be able to use Booster as a model driven platform for the implementation of data structures. However more complicated data structures may require capabilities that go beyond what Booster can offer at the moment. For example, we often need recursion to perform such things as to balance a search tree. It may still be possible to use Booster to generate a part of the implementation and leave the programmer to focus on the hard parts. However this remains open to further research.

8.10.2. Generating code from universal quantifiers. The code generation strategy we have described in chapter 7 does not deal with universal or existential quantifiers. Existential quantifiers are a generalization of disjunction which rules them out as candidates for code generation (since we do not

generate code from disjunctions). However in certain cases we may be able to generate a useful program from universal quantifiers, by considering the quantified predicate as an equality upon the corresponding set expressions.

We first observe that if the body of the quantification is a conjunction, then the quantified expression may be considered as a conjunction:

```
forall i : I . conj1 & conj2
   ≡ (forall i : I . conj1) & (forall i : I . conj2)
```

Thus we need only consider quantifications in which the body is a possibly-guarded primitive postcondition.

If the body is a guarded set-membership predicate (or its negation), then it is equivalent to the statement that the characteristic set is a subset of (or does not intersect with) the set-valued attribute in question; in set-theoretic terms, we observe that

$$\forall\, i : I \bullet P \Rightarrow e \in s \Leftrightarrow \{i : I \mid P \bullet e\} \subseteq s$$
$$\forall\, i : I \bullet P \Rightarrow e \notin s \Leftrightarrow \{i : I \mid P \bullet e\} \cap s = \varnothing$$

where indexing variable i may appear free in expression e, but not in set-valued expression s.

Thus these conditions can be safely implemented as simple assignments in our language of guarded commands:

assign `forall i : I . P => e : s` $= add(s, \{i : I \mid P \bullet e\})$
assign `forall i : I . P => e /: s` $= remove(s, \{i : I \mid P \bullet e\})$

The set expressions on the right-hand side of each assignment can in turn be implemented as iterations over the set in question. In either case, the weakest precondition is easily calculated, as a simple substitution.

8.10.3. Justifying the transformations. Another set of limitations concerns the process of strengthening the precondition. As we have described in chapter 7, a part of the code generation process involves calculating the weakest precondition of the generated code and adding it to the precondition of the original specification.

This precondition is used by Booster in several places. First, it is used at runtime to prevent the application of a method when its precondition is false. Second, it is used in the implementation of the guard and the `OR` combinators. Therefore it is important to implement the preconditions as efficiently as possible.

In addition, because the weakest precondition is appended to the original precondition it must be made as readable as possible since otherwise we might not be able to understand the reasons for its introduction.

To overcome this problem we use a term rewriting system with a set of rules that simplify the precondition. Simplification using term rewriting is used by many theorem provers and formal development tools (well known examples are the B toolkit, the ACL2 theorem prover and the PVS theorem prover, but there are many others).

However, the effectiveness of simplification depends on the depth of the domain. If the domain is deep, for example if we try to capture the rules of number theory, then we need a very extensive set of rules and even then there is a good chance that many formulas will not be adequately simplified. Fortunately the domain of information management systems is not as deep as number theory. It consists of simple manipulations of sets and sequences and, occasionally, simple applications of arithmetic. It is therefore possible to come up with a set of rules that effectively simplify most of the terms that arise out of such manipulations.

However currently there is no way to judge in advance how effective is a set of term rewriting rules for a particular domain. The only way is to try the system on a large number of cases and observe how it behaves. This may become an obstacle for adopting this technique to new domains.

In addition, calculating the weakest preconditions might become expensive because of duplications of substitutions that occur for example in the case of conditional statements.

Finally, successful simplification is itself a problem because it may leave the engineer without any clue as to why a particular precondition became, for example, false. Indeed, we currently do not provide adequate means to explain to the engineer the results of the transformations. The engineer may compare the original model with the transformed model, but the intermediate steps are not explained.

8.10.4. Generating code from non deterministic specifications. Currently Booster does not attempt to generate code from non deterministic constructs such as disjunctions and relational operators (other than equality). When encountering such a formula in a postcondition the compiler simply generates *skip*.

It is possible to think of ways to generate code from such constructs, for example we may generate code from a disjunction $p \vee q$ by generating code for the first disjunct.

Alternatively we may generate code from both disjuncts and combine them in an asymmetrical choice operator similarly to how we implement the *OR* combinator:

$$
\begin{aligned}
& program(p \vee q) = \\
& \qquad guard(pp) \rightarrow pp \\
& \qquad \Box \\
& \qquad (\neg\, guard(pp)) \wedge guard(pq) \rightarrow pq
\end{aligned}
$$

where $pp = program(p)$ and $pq = program(q)$ and the function *guard* returns a formula that describes the program's domain.

This program will achieve the postcondition in more states than the previous program, because if *pp* cannot achieve the goal there is still the possibility that *pq* will achieve the goal.

We may even use dynamic programming techniques to generate code from systems of linear inequalities.

All of these techniques should be considered in the context of the domain of application, to see if they make sense for the particular models that are being constructed for the domain.

The approach itself is flexible and will admit many other kinds of implementation patterns. Any implementation pattern can be accommodated as long as the compiler can calculate the weakest precondition of the generated code.

8.10.5. Model evolution. Information systems frequently change throughout their lifetime. This may occur because external changes cause the requirements to change (the law may change or the way taxes are calculated may change, hardware vendors may go out of business, operating systems may change, and so on and so forth), or because once the customers begin to use the system they understand better what it is they want the system to provide in the first place. Indeed, changes in the requirements of a large system are often inevitable because the act of deploying the system is itself a major change to the environment in which it operates.

Such changes can sometimes be accommodated by changing the behavior of the system — the way its methods are implemented — without affecting its structure. However, in many cases the changes also affect the structure of the system, they affect the model. New associations may have to be introduced and existing associations may have to be removed. New attributes are often added, a single class may be split into several classes and so on.

The problem is that such systems often accumulate a large amount of data that fits the structure of the existing system, but when we change this structure it cannot accommodate the old information. This is a common problem that affects any piece of software that persists information, including Booster.

However the techniques that we have described in this thesis suggest an approach that may help to automate this process.

We may use a language like Booster to describe the relationships between the old and the new information and to specify a program that performs the upgrade, then let Booster generate the upgrade program automatically.

It may even be possible in some (perhaps most?) cases to generate the upgrade program just from the description of the relationship between the old and the new structure, without having to manually specify the upgrade.

8.10.6. Communicating Booster systems. Currently a Booster model defines a single component. However large systems consist of many components that communicate with each other. It is therefore worth while to extend the theory of Booster to support multiple communicating Booster components. Such a theory will not only allow Booster to scale beyond the relatively small systems that can be described using a single component, but will also make it possible to connect Booster systems to other systems. We may even be able to use Booster models as the specification of other systems. Of course the implementation of such components is no longer automatic and must be verified using other means, but it can offer many different implementations a way to describe their services in a common notation and provides a single language in which their composition can be described and analyzed.

This may even lead to the development of a formal language for the description (both of the interfaces and the composition) of the kinds of service oriented architectures that are now becoming popular in the industry.

Cooperation and communication is the domain of process algebras like CSP [**28**] and of temporal logic [**32**]. However while temporal logic can be used to specify abstract properties of communicating components it is not well suited for the descriptions of models of such systems. Therefore it appears that a theory of communicating Booster components should be based on a process algebra.

APPENDIX A

The concrete syntax of Booster

A.1. Specifications

⟨*model*⟩ ::= ‘SYSTEM’ ::= ⟨*name*⟩ ⟨*class*⟩+ ‘END’

⟨*class*⟩ ::= ‘CLASS’ ⟨*name*⟩ ‘ATTRIBUTES’ ⟨*attr*⟩+ ‘METHODS’ ⟨*method*⟩ ‘END’

⟨*attr*⟩ ::= ⟨*name*⟩ ‘:’ ⟨*type*⟩

⟨*type*⟩ ::= ⟨*name*⟩ | ⟨*name*⟩ ‘.’ ⟨*name*⟩ | ‘[’ ⟨*type*⟩ ‘]’
| ‘SET’ ‘(’ ⟨*type*⟩ ‘)’

⟨*method*⟩ ::= ⟨*name*⟩ ‘(’ ⟨*methodkind*⟩ ‘)’

⟨*methodkind*⟩ ::= ⟨*primitive method*⟩ | ⟨*guarded method*⟩ | ⟨*disj combinator*⟩

⟨*primitive method*⟩ ::= ⟨*formula*⟩ ‘|’ ⟨*formula*⟩

⟨*guarded method*⟩ ::= ⟨*formula*⟩ ‘|’ ⟨*methodkind*⟩

⟨*disj combinator*⟩ ::= ⟨*disj combinator*⟩ ‘OR’ ⟨*conj combinator*⟩
| ⟨*conj combinator*⟩

⟨*conj combinator*⟩ ::= ⟨*conj combinator*⟩ ‘AND’ ⟨*seq combinator*⟩
| ⟨*seq combinator*⟩

⟨*seq combinator*⟩ ::= ⟨*seq combinator*⟩ ‘THEN’ ⟨*method ref*⟩
| ⟨*method ref*⟩

⟨*method ref*⟩ ::= ⟨*term*⟩ | ‘(’ ⟨*methodkind*⟩ ‘)’

⟨*formula*⟩ ::= ⟨*implication*⟩

⟨*implication*⟩ ::= ⟨*implication*⟩ ‘=>’ ⟨*disjunction*⟩ | ⟨*disjunction*⟩

⟨*disjunction*⟩ ::= ⟨*disjunction*⟩ ‘or’ ⟨*conjunction*⟩ | ⟨*conjunction*⟩

⟨*conjunction*⟩ ::= ⟨*conjunction*⟩ ‘&’ ⟨*negation*⟩ | ⟨*negation*⟩

⟨*negation*⟩ ::= ‘not’ ⟨*basic formula*⟩ | ⟨*basic formula*⟩

⟨*basic formula*⟩ ::= ‘true’ | ‘false’ | ‘(’ ⟨*formula*⟩ ‘)’
| ⟨*term*⟩ ⟨*relop*⟩ ⟨*term*⟩

⟨*relop*⟩ ::= ‘=’ | ‘!=’ | ‘:’ | ‘/:’ | ...

⟨*term*⟩ ::= ‘this’ | ⟨*literal*⟩ | ⟨*name*⟩ | ⟨*name*⟩ ‘_0’
| ⟨*term*⟩ ‘.’ ⟨*term*⟩ | ⟨*name*⟩ ‘(’ ⟨*term*⟩* ‘)’

A.2. Programs

Even though the users of the Booster system do not write programs manually, it is convenient to describe programming fragments using a concrete syntax. Here is the formal definition of this syntax:

⟨*statement*⟩ ::= skip | ⟨*lhs*⟩ ‘:=’ ⟨*term*⟩
  | ⟨*statement*⟩ ‘;’ ⟨*statement*⟩
  | ⟨*formula*⟩ ‘->’ ⟨*statement*⟩
  | ⟨*statement*⟩ ‘[]’ ⟨*statement*⟩
  | ⟨*block*⟩

⟨*lhs*⟩ ::= ⟨*name*⟩ | ⟨*term*⟩ ‘.’ ⟨*name*⟩

⟨*block*⟩ ::= ‘local’ ⟨*lvars*⟩ ‘in’ ⟨*statement*⟩ ‘end’

⟨*lvars*⟩ ::= ⟨*name*⟩ ’=’ ⟨*term*⟩ ⟨*rest*⟩

⟨*rest*⟩ ::= | ‘,’ ⟨*lvars*⟩

APPENDIX B

Formal Booster operators

$\{\} == literal(set\ \{\})$

$\neg\ == negation$

$_\wedge_ : Formula \times Formula \rightarrow Formula$

$\forall f, g : Formula \bullet f \wedge g = conjunction(f, g)$

$_\vee_ : Formula \times Formula \rightarrow Formula$

$\forall f, g : Formula \bullet f \vee g = disjunction(f, g)$

$_\Rightarrow_ : Formula \times Formula \rightarrow Formula$

$\forall f, g : Formula \bullet f \Rightarrow g = implication(f, g)$

$_\sqsupseteq_ : Formula \times Formula \rightarrow Formula$

$\forall f, g : Formula \bullet f \sqsupseteq g = refinement(f, g)$

$_=_ : Term \times Term \rightarrow Formula$

$\forall s, t : Term \bullet s = t = basic(isEqual, s, t)$

$_\neq_ : Term \times Term \rightarrow Formula$

$\forall s, t : Term \bullet s = t = basic(isNotEqual, s, t)$

$_\in_ : Term \times Term \rightarrow Formula$

$\forall s, t : Term \bullet s \in t = basic(isMember, s, t)$

$_\notin_ : Term \times Term \rightarrow Formula$

$\forall s, t : Term \bullet s \notin t = basic(isNonMember, s, t)$

$_in : Name \rightarrow Term$

$\forall n : Name \bullet n_in = inputvar\ n$

$_0 : Name \rightarrow Term$

$\forall n : Name \bullet n_0 = before\ n$

$_ : Name \rightarrow Term$

$\forall n : Name \bullet n = after(association\ n)$

$$_._ : Term \times Term \rightarrow Term$$
$$\forall s, t : Term \bullet s.t = path(s, t)$$

$$_.\mathrm{the} : Term \rightarrow Term$$
$$\forall t : Term \bullet t.\mathrm{the} = the(t)$$

$$_ \cup _ : Term \times Term \rightarrow Term$$
$$\forall s, t : Term \bullet s \cup t = union(s, t)$$

$$_ \setminus _ : Term \times Term \rightarrow Term$$
$$\forall s, t : Term \bullet s \setminus t = setminus(s, t)$$

$$_ \oplus _ : Term \times Term \rightarrow Term$$
$$\forall s, t : Term \bullet s \oplus t = update(s, t)$$

$$_ := _ : Name \times Term \rightarrow Statement$$
$$\forall n : Name;\ t : Term \bullet \quad n := t = assignment(n, t)$$

$$_ \Box _ : Statement \times Statement \rightarrow Statement$$
$$\forall s_1, s_2 : Statement \bullet \quad s_1 \Box s_2 = choice(s_1, s_2)$$

$$_ ;\ _ : Statement \times Statement \rightarrow Statement$$
$$\forall s, t : Statement \bullet \quad s ;\ t = sequence(s, t)$$

$$_ \Box _ : Statement \times Statement \rightarrow Statement$$
$$\forall s, t : Statement \bullet \quad s \Box t = choice(s, t)$$

$$_ \rightarrow _ : Statement \times Statement \rightarrow Statement$$
$$\forall s, t : Statement \bullet \quad s \rightarrow t = guard(s, t)$$

APPENDIX C

Term rewriting

In order to make syntactical transformations easy to follow we describe them as rewrite rules [**34**]. A rewrite rule consists of two patterns separated by an arrow. When we apply the rule to a term it replaces every sub-term that matches the left hand side pattern with the right hand side. As a simple example consider the following abstract syntax tree:

$exp ::= lit\langle\langle\mathbb{N}\rangle\rangle \mid plus\langle\langle exp \times exp \mid var\langle\langle Name\rangle\rangle$

We may use the rewrite rule $plus(x,0) \longrightarrow x$ to simplify addition with zero. When we apply this rule to the term

$$plus(plus(var(a),0), plus(var(b),0)),$$

the rule replaces $plus(var(a),0)$ with $var(a)$ and $plus(var(b),0)$ with $var(b)$, which results in the term $plus(var(a), var(b))$.

The red variable, x, is a pattern variable. Its different color separates it from variables and literals that belong to the language of *exp*.

The right hand side of rules is also a pattern, but its pattern variables are always a subset of the variables of the left hand side. As a result, whenever the left hand side matches a term, the variables of the right hand side are bound to specific terms and the *instantiated* right hand side becomes a term.

We sometimes use a collection of rules, called a *a term rewriting system*, to describe the effect of a syntactical transformation. For example, the following rewrite system eliminates zeroes that appear in any side of a *plus* term:

$$plus(x,0) \longrightarrow x$$
$$plus(0,x) \longrightarrow x$$

Informally, a rewrite system operates by repeatedly picking a rule that matches the term and applying it to the term, until no rule matches.

Another useful thing we can do with patterns is to use them to *query* a term and collect all the sub-terms that match the pattern. We write $term \upharpoonright pattern$ to denote the set of all sub-terms of *term* that match the pattern *pattern*.

For example, here is what the results of applying the pattern $plus(var(a),x)$ to the term

$$plus(var(a), plus(var(a), lit(1)))$$

looks like:

$$plus(var(a), plus(var(a), lit(1))) \upharpoonright plus(var(a),x) = \{plus(var(a), plus(var(a), lit(1))), plus(var(a), lit(1))\}$$

Modeling rewrite rules

In order to explain the semantics of rewrite rules as clearly as possible we define a model which uses the simple abstract syntax tree in the previous section. It should not be difficult to see how the model applies to any other syntax tree.

A rule consists of two patterns:

Rule

$lhs, rhs : Pattern$

We model patterns as functions. The formal parameters of the function represent the pattern variables. For example, we can model the pattern $plus(x, 0)$ using the function $\lambda x : exp \bullet plus(x, 0)$.

$$Pattern == exp \rightarrow exp$$

Patterns can have more than one free variable, however it is more simple to explain the ideas using a single variable.

To model the application of a rule we first describe how it performs a single step, in which it substitutes at least one sub-term. We can then describe the application of the rule as the fixed point operator of this step function.

In a single step, when the *lhs* pattern matches the term, the rule replaces the term by its *rhs*. When the *lhs* does not match the term, the effect depends on the structure of the term. If the term is a ground term, that is it has no terms as children, then the term is left unmodified. If the term has children, then the rule is applied recursively to each child.

StepMatch

$Rule$
$step : exp \rightarrow exp$

$\forall e : exp \bullet \exists x : exp \bullet lhs\, x = e \Rightarrow step\, e = rhs\, x$

StepNoMatch

$Rule$
$step : exp \rightarrow exp$

$\forall e : \mathrm{ran}\, lit \mid \forall x : exp \bullet lhs\, x \neq e \bullet step\, e = e$
$\forall s, t : exp \mid \forall x : exp \bullet lhs\, x \neq plus(s, t) \bullet step\, plus(s, t) = plus(step\, s, step\, t)$

$$Step == StepMatch \land StepNoMatch$$

To apply the rule until there are no more instances to replace, we define the application function as the fixed point of the *step* function:

ApplyRule

$Step$
$apply : exp \rightarrow exp$

$apply = fix\, step$

$[T]$

$fix : (T \leftrightarrow T) \rightarrow\!\!\!\!\!\!\!+ \; (T \leftrightarrow T)$

$\forall f : T \leftrightarrow T;\; n : \mathbb{N} \bullet fix f = f^n \Leftrightarrow f^n = f^{n+1}$

Note that according to this definition, the rule always matches the biggest term first. For example, when we apply the rule

$$plus(x, 0) \longrightarrow x$$

to the term

$$plus(plus(var(a), 0), 0)$$

the rule substitutes the outer occurrence of the pattern first, yielding the intermediate result $plus(var(a), 0)$ then substitutes this intermediate term with $var(a)$ which is the final result.

A rewrite system is a set of rules:

RewriteSystem

$rules : \mathbb{P}\, Rule$

Because a rewrite system contains many rules, the same term may be rewritten to different terms depending on the order in which we apply the rules. This means that unlike a single rule which has a functional behavior, a rewrite system is best described as a relation.

SystemStep

$RewriteSystem$
$step : exp \leftrightarrow exp$

$$\forall\, e, e' : exp \bullet e \mapsto e' \in step \Leftrightarrow$$
$$(\exists\, Rule \mid \theta Rule \in rules \bullet Step \wedge step\, e = e')$$

ApplySystem

$SystemStep$
$apply : exp \leftrightarrow exp$

$apply = fix\, step$

We now describe the meaning of the filter operator, $term \upharpoonright pattern$. A term tt belongs to the set $t \upharpoonright p$ if it is equal to t and matches p, or if t is compound and tt is in the filter set of any of t's children:

$$_ \upharpoonright _ : exp \times Pattern \rightarrow \mathbb{P}\, exp$$

$$\forall\, t, tt : exp;\; p : Pattern \bullet$$
$$tt \in t \upharpoonright p \Leftrightarrow tt = t \wedge \exists\, x : exp \bullet p\, x = t \vee$$
$$\exists\, t_1, t_2 : exp \bullet t = plus(t_1, t_2) \wedge tt \in t_1 \upharpoonright p \vee tt \in t_2 \upharpoonright p$$

Confluence

We have defined the meaning of a rewrite system as a function. But sometimes a rewrite system may produce different results for the same term, depending on the order in which we apply the rules. For example, consider the following rewrite system in which all the identifiers are literals:

$$a + b \longrightarrow c$$
$$a \longrightarrow d$$
$$d + b \longrightarrow e$$

If we apply this simple system to the term $a + b$ using the first rule we get c. But we can also apply the second rule, followed by the third rule and get e.

Such non deterministic rule systems are sometimes undesirable. However, in other cases we may not care, as long as the rule performed the requested transformation.

Termination

Not every rule has a fixed point. For example when we apply the rule $x \longrightarrow plus(x, 0)$ to the term $lit(1)$ it will add an additional term in each step, and so there is no integer n for which the nth step is equal to the $n + 1$th step.

There is no general way to decide if a given set of rules always terminates. However, if every in every rule in the system the right hand side has fewer constructors than the left hand side, then it is easy to see

by a simple induction on the number of constructors in the term, that such a rewriting system always terminates.

In the more general case, we may choose any quantity that is a function of the structure of the terms and if we can show that each rule strictly reduces this quantity then the rule system always terminates. For example, we may choose to count the number of occurrences of a particular constructor *c*, and if every rule reduces this number then no matter how many other constructors are added to the term, eventually there will be no more occurrences of *c* and the system will stop.

APPENDIX D

An overview of Z

The Z language is used throughout this thesis both as a meta-language that describes the syntax and semantics of Booster, and as a specification language which is used as a source of inspiration for Booster. This chapter provides a short introduction to Z. We do not present all the language, only a subset that is relevant to the thesis. For a complete introduction see [**61**].

The Z language consists of three major components: a notation for describing data types using sets, relations, and functions; a predicate logic notation for describing the desired properties of these data types; and a schema language for structuring the details of large specifications.

Describing data types

Z has only one built in data type, $\mathbb{Z}$, the set of integers. We can define subsets of $\mathbb{Z}$ by explicitly enumerating their members, for example, here is the set that consists of the first three prime numbers: $\{2, 3, 5\}$.

Another way to describe sets is to use a *set comprehension*

$$\{x : S \mid P(x)\}$$

which describes the set of all elements in S that satisfy the predicate $P(x)$. The extended version of a set comprehension uses an expression to describe the elements in the set:

$$\{x : S \mid P(x) \bullet E(x)\}$$

which describes the set that we get by taking all the members in S that satisfy P and applying the expression E to each one. For example, the following set comprehension describes all the positive odd numbers:

$$\{x : \mathbb{Z} \mid x \geq 0 \bullet 2 * x + 1\}$$

The axiomatic definition

$$\begin{array}{|l} n : \mathbb{Z} \\ \hline n > 10 \end{array}$$

introduces the name n to the document and asserts that it denotes an integer number that is bigger than 10. In general for any set S, the axiomatic definition

$$\begin{array}{|l} x : S \\ \hline P \end{array}$$

introduces a new name x to the document and asserts that it is a member of the set S and that it satisfies the predicate P.

To introduce a name that describes a subset of S, we declare it as a member of the *power set* of S. For example the axiomatic definition

$$\begin{array}{|l} xs : \mathbb{P}\,\mathbb{Z} \\ \hline 1 \in xs \end{array}$$

introduces x as a subset of integers that contain the number 1.

We may describe pairs of elements by making using the Cartesian product operator. For example:

$$
\begin{array}{|l}
xx : \mathbb{Z} \times \mathbb{Z} \\
\hline
xx.1 = xx.2
\end{array}
$$

introduces the name xx as a pair of integers whose members are equal to each other. We may use either the syntax (u, v) or the syntax $u \mapsto v$ to describe specific pairs.

When we apply the power set operator to a Cartesian product we get a subset of pairs — a binary relation between the elements of the two sets. For example:

$$
\begin{array}{|l}
Less : \mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \\
\hline
\forall x, y : \mathbb{Z} \bullet x \mapsto y \in Less \Leftrightarrow x < y
\end{array}
$$

defines *Less* to be the set of all the pairs of integers in which the first element is smaller than the second. We often use the shortcut notation $A \leftrightarrow B$ to describe binary relations.

If a pair $x \mapsto y$ is a member of a relation r, we say that r maps the first member x to the second member y. The *domain* of a relation, written $\operatorname{dom} r$, consists of all the first members of each pair in r. The range of a relation, written $\operatorname{ran} r$, consists of all the second members of each pair in r. Therefore, r maps every element in its domain to at least one element in its range.

Because relations are sets of pairs we can use all the standard set operators on relations. Moreover, relations are closed under intersection, union, and set difference.

The various restriction operators offer a convenient way to restrict our attention to a particular part of the domain or range of a relation:

$$\{1\} \lhd \{1 \mapsto 3, 2 \mapsto 5\} = \{1 \mapsto 3\}$$
$$\{1\} ⩤ \{1 \mapsto 3, 2 \mapsto 5\} = \{2 \mapsto 5\}$$
$$\{1 \mapsto 3, 2 \mapsto 5\} \rhd \{5\} = \{2 \mapsto 5\}$$
$$\{1 \mapsto 3, 2 \mapsto 5\} ⩥ \{5\} = \{1 \mapsto 3\}$$

When a relation never maps the same element in its domain to more than one element in its range we say that it is a *function*. Because functions are such an important modeling device, Z provides a special syntax for the different kinds of functions:

$A \pfun B$	partial function
$A \rightarrow B$	total function
$A \pinj B$	partial injection
$A \rightarrowtail B$	total injection
$A \psurj B$	partial surjection

However this syntax is, like the syntax for defining relations, a shorthand notation, and every kind of function can be described equivalently as a relation between sets or indeed as a set of pairs, if we accompany the definition with the appropriate predicate.

We often use functions to model the state of some component. For example the function might map seats in an airplane to the passenger that was allocated for the set. When the state of the component changes, for example, when a new passenger is cancels a flight, we need to update the function. For such a purpose it is convenient to use the *update* operator, written $f \oplus g$, which describes a function that is identical to f in any element of its domain, except that for those elements that appear in the domain of g it is identical to g. For example

$$\{1 \mapsto 2, 2 \mapsto 5\} \oplus \{1 \mapsto 3\} = \{1 \mapsto 3, 2 \mapsto 5\}$$

Sequences are a special kind of function that maps the location of each element in the sequence to the element. Z provides a special notation for sequences, but like in the case of functions and relations,

this is only a convenient shorthand. For example, the sequence $\langle 5, 3, 8\rangle$ may be also represented as the set of pairs $\{1 \mapsto 5, 2 \mapsto 3, 3 \mapsto 8\}$.

Using only integers to model systems can quickly become tedious and confusing. We can define new types using Z's *given type* construct. For example the following definition

$[Person]$

introduces a new type, called *Person*. We know nothing about the structure of a *Person*, only that there is an unbounded number of them. Like anything else in Z, types are sets.

Finally, we may form new datatypes using a notation that is similar to BNF. Such notation is very helpful when we define recursive data types like for example, abstract syntax trees.

$Expr ::= Literal\langle\langle \mathbb{Z} \rangle\rangle \mid Plus\langle\langle Expr \times Expr\rangle\rangle \mid Times\langle\langle Expr \times Expr\rangle\rangle$

The different alternatives to construct the data type are separated using vertical bars. Each alternative is a *construct*, a function that takes its argument and returns a corresponding, unique *Expr*. Like the other notational devices we have described previously, free types can be reduced to simple sets, but we will not describe the details here. The interested reader may find them in the literature [**61**].

Useful properties of sets and relations

In this section we collect a set of properties that we use in proofs throughout the thesis.

LEMMA D.0.1. *Let a and b be subsets of the set s. Then*

$$a \cap \overline{b} = \{\} \Leftrightarrow a \subseteq b$$

where $\overline{b} = s \setminus b$.

PROOF. Let a and b be subsets of the set s. We argue as follows:

$a \cap \overline{b} = \{\}$
$\Leftrightarrow \{x : s \mid x \in a \wedge x \in \overline{b}\} = \{\}$
$\Leftrightarrow \{x : s \mid x \in a \wedge x \notin b\} = \{\}$ — definition of $\overline{b}$
$\Leftrightarrow \forall x : s \bullet x \notin a \vee x \in b$ — set comprehension
$\Leftrightarrow \forall x : s \bullet x \in a \Rightarrow x \in b$ — predicate calculus
$\Leftrightarrow a \subseteq b$

□

LEMMA D.0.2. *Let p and w be relations. Then,*

$$\operatorname{dom}(p \,⨾\, w) = \operatorname{dom}(p \rhd \operatorname{dom} w)$$

PROOF. Let p and w be relations, and let x be a member of $\operatorname{dom} p \,⨾\, w$. We argue as follows:

$x \in \operatorname{dom}(p \,⨾\, w)$
$\Leftrightarrow \exists z : State \bullet x \mapsto z \in p \,⨾\, w$ — definition of dom
$\Leftrightarrow \exists z : State \bullet \exists y : State \bullet x \mapsto y \in p \wedge y \mapsto z \in w$ — definition of ⨾
$\Leftrightarrow \exists y : State \bullet \exists z : State \bullet x \mapsto y \in p \wedge y \mapsto z \in w$ — predicate logic
$\Leftrightarrow \exists y : State \bullet x \mapsto y \in p \wedge \exists z : State \bullet y \mapsto z \in w$ — predicate logic
$\Leftrightarrow \exists y : State \bullet x \mapsto y \in p \wedge y \in \operatorname{dom} w$ — definition of dom
$\Leftrightarrow \exists y : State \bullet x \mapsto y \in p \rhd \operatorname{dom} w$ — definition of $\rhd$
$\Leftrightarrow x \in \operatorname{dom}(p \rhd \operatorname{dom} w)$ — definition of dom

□

LEMMA D.0.3.

$$k ⩤ s = k ⩤ s'' \wedge l ⩤ s'' = l ⩤ s' \Rightarrow l \cup k ⩤ s = l \cup k ⩤ s'$$

PROOF.

$l \cup k ⩤ s$	
$= l ⩤ (k ⩤ s)$	lemma D.0.4
$= l ⩤ (k ⩤ s'')$	assumption
$= (l \cup k) ⩤ s''$	lemma D.0.4
$= (k \cup l) ⩤ s''$	set theory
$= k ⩤ (l ⩤ s'')$	lemma D.0.4
$= k ⩤ (l ⩤ s')$	assumption
$= k \cup l ⩤ s'$	lemma D.0.4

□

LEMMA D.0.4.

$$k \cup l ⩤ s = k ⩤ (l ⩤ s)$$

PROOF. Let $x \mapsto y$ be an arbitrary member of $k \cup l ⩤$. We argue as follows:

$x \mapsto y \in k \cup l ⩤ s$	definition of $⩤$
$\Leftrightarrow x \notin (k \cup l) \wedge x \mapsto y \in s$	set theory
$\Leftrightarrow x \notin k \wedge x \notin l \wedge x \mapsto y \in s$	definition of $⩤$
$\Leftrightarrow x \notin k \wedge x \mapsto y \in l ⩤ s)$	definition of $⩤$
$\Leftrightarrow x \mapsto y \in k ⩤ (l ⩤ s)$	

□

LEMMA D.0.5. *Let p be a relation from A to B and a be a set of elements from A. Then,*

$$a ⩤ p = \overline{a} \lhd p$$

PROOF. Let p be a relation from A to B and a be a set of elements from A. Let x be an element of A and y be an element of B. We argue as follows:

$x \mapsto y \in a ⩤ p$	
$\Leftrightarrow x \mapsto y \in p \wedge x \notin a$	definition of $⩤$
$\Leftrightarrow x \mapsto y \in p \wedge x \in \overline{a}$	definition of $\overline{a}$
$\Leftrightarrow x \mapsto y \in \overline{a} \lhd p$	definition of $\lhd$

□

LEMMA D.0.6. *Let A and B be two relations from X to X, and let id be the identity relation over X. Then*

$$\mathrm{dom}(A \cap id) \cap \mathrm{dom}(B \cap id) = \mathrm{dom}(A \cap B \cap id)$$

PROOF. First we show that $\mathrm{dom}(A \cap B \cap id)$ is a subset of $\mathrm{dom}(A \cap id) \cap \mathrm{dom}(B \cap id)$:

$\mathrm{dom}(A \cap B \cap id)$
$= \mathrm{dom}(A \cap id \cap B \cap id)$ set theory
$\subseteq \mathrm{dom}(A \cap id) \cap \mathrm{dom}(B \cap id)$ set theory

Now we show that $\mathrm{dom}(A \cap id) \cap \mathrm{dom}(B \cap id)$ is a subset of $\mathrm{dom}(A \cap B \cap id)$:

$x \in \mathrm{dom}(A \cap id) \cap \mathrm{dom}(B \cap id)$
$\Leftrightarrow \exists u : X \bullet x \mapsto u \in A \cap id \wedge \exists v : X \bullet x \mapsto v \in B \cap id$ definition of dom
$\Leftrightarrow \exists u : X \bullet x \mapsto u \in A \wedge x = u \wedge \exists v : X \bullet x \mapsto v \in B \wedge x = v$ definition of *id*
$\Leftrightarrow x \mapsto x \in A \wedge x \mapsto x \in B$ one point rule
$\Leftrightarrow x \mapsto x \in A \cap B \cap id$ set theory
$\Leftrightarrow x \in \mathrm{dom}(A \cap B \cap id)$

Since each side of the equation is a subset of the other side they must be equal. □

LEMMA D.0.7. *Let a and b be sets and let p be a binary relation. Then,*

$$a \cup b \lhd p = a \lhd p \cup b \lhd p$$

PROOF. Let $x \mapsto y$ be an arbitrary member of $a \cup b \lhd p$. We argue as follows:

$x \mapsto y \in a \cup b \lhd p$
$\Leftrightarrow x \in a \vee x \in b \wedge x \mapsto y \in p$ definition of $\lhd$
$\Leftrightarrow (x \in a \wedge x \mapsto y \in p) \vee (x \in b \wedge x \mapsto y \in p)$ predicate logic
$\Leftrightarrow x \mapsto y \in a \lhd p \vee x \mapsto y \in b \lhd p$ definition of $\lhd$

We have shown that the pair $x \mapsto y$ is a member of the left hand relation if and only if it is a member of the right hand relation. Since $x \mapsto y$ is an arbitrary pair, both relations must be equal. □

Schemas

The mathematical toolkit we have described so far can be used to model small fragments of software systems. However, like programs, models of software consist of a large number of details and without a proper way to control the amount of details it quickly becomes too confusing to understand what is going on.

A schema is a structuring device that binds together declarations and predicates to form a higher level unit that we can use to structure our models.

A schema consists of two sections separated by a horizontal bar. The section above the bar contains named definitions, and the section below the bar contains predicates that constrain the defined entities and (often) relate them to each other.

For example the following schema models the state of a component that consists of several parts, together with constraints between these parts:

$$
\begin{array}{|l}
\hline
S \\
as : \mathbb{P} A \\
ab : A \nrightarrow B \\
bs : \operatorname{seq} B \\
\hline
\operatorname{dom} ab \subseteq \operatorname{dom} as \\
bs \neq \langle\rangle \Leftrightarrow \operatorname{dom} ab = as \\
\operatorname{ran} bs \cap \operatorname{ran} ab = \{\} \\
\hline
\end{array}
$$

When the description becomes too big we can split it into separate schemas and then combine them together to form the complete description. For example we can split the schema above into several parts:

$$
\begin{array}{|l}
\hline
S1 \\
ab : A \nrightarrow B \\
bs : \operatorname{seq} B \\
\hline
\operatorname{ran} bs \cap \operatorname{ran} ab = \{\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
S2 \\
as : \mathbb{P} A \\
ab : A \nrightarrow B \\
\hline
\operatorname{dom} ab \subseteq \operatorname{dom} as \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
S3 \\
as : \mathbb{P} A \\
ab : A \nrightarrow B \\
bs : \operatorname{seq} B \\
\hline
bs \neq \langle\rangle \Leftrightarrow \operatorname{dom} ab = as \\
\hline
\end{array}
$$

The complete system S can now be described as

$S \mathrel{\widehat=} S1 \wedge S2 \wedge S3$

The conjunction of two schemas is itself a schema whose definition part consists from all the definitions in both schemas, and whose constraint part is the conjunction of the constraints of both schemas. The operation is defined only when equal names in the two schemas have the same type.

We can use all the standard operators of predicate logic to combine schemas. In every case the operation is well defined only if equal names have the same type in both schemas. The constraint part is combined according to the nature of the operator: when we use disjunction the constraints are disjoined, and when we use negation the constraints are negated[1].

We can also embed one schema inside another. If we place the schema in the declaration part then the declarations of the inner schema are added to those of the outer schema and their constraints are conjoined. Therefore we can also write S as follows:

$$
\begin{array}{|l}
\hline
S \\
S1 \\
S2 \\
S3 \\
\hline
\end{array}
$$

[1] to perform negation correctly we must be careful and separate the constraints that are embedded in declarations. For a discussion of this topic see [**61**]

Finally, we can also use universal and existential quantifiers to create schemas. In this case the definitions of the resulting schema are the definitions of the original schema except for those whose name is bound by the quantifier, and the constraint section of the schema consists of the original constraint embedded in the quantifier:

$\exists as : \mathbb{P} A \bullet S2 = [ab : A \pfun B \mid \forall as : \mathbb{P} A \bullet \operatorname{dom} ab \subseteq \operatorname{dom} as]$

Here we have used the second form of writing schemas, where the definition section appears to the left of the vertical bar and the constraint appears to the right of the bar.

Finally, we can decorate a schema with a prime symbol. This operation adds a prime symbol to every definition in the schema. For example,

$S2' = [as' : \mathbb{P} A;\ ab' : A \pfun B \mid \operatorname{dom} ab' \subseteq \operatorname{dom} as']$

Modeling behavior

To describe the behavior of an operation we use a schema that contains two instances that describe the state of the system, one is not decorated and represents the current state of the system, and the other is decorated and represents the new state of the system. We then specify the behavior of the operation by describing its effect as a relation between initial and final states.

Input and output are represented by additional definitions in the operation schema. We decorate inputs with the symbol ? and outputs with the symbol !. These decorations however have no special meaning as far as Z is concerned, they are simply useful conventions.

For example the following schema describes an operation that updates the function *ab* by replacing an existing binding with a new value that comes as an input to the operation:

```
┌─ Op ─────────────────────────────
│ S
│ S′
│ a? : A
├──────────────
│ a? ∈ as
│ as′ = as
│ ab ⊕ {a? ↦ head bs} = ab′
│ bs′ = tail bs
└──────────────────────────────────
```

Bibliography

[1] Oxford University Computing Laboratory. http://www.comlab.ox.ac.uk/.

[2] iFM 2007: integrated Formal Methods Oxford UK. http://www.softeng.ox.ac.uk/ifm2007/programme.html, 2nd–5th July 2007.

[3] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In *Verification: Theory and Practice*, pages 11–41, 2003.

[4] Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[5] Jean-Raymond Abrial and Louis Mussat. On using conditional definitions in formal theories. In *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 242–269, London, UK, 2002. Springer-Verlag.

[6] G. Antoniou and M. Arief. Executable declarative business rules and their use in electronic commerce. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 6–10, New York, NY, USA, 2002. ACM Press.

[7] Ralph-Johan J. Back, Abo Akademi, and J. Von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[8] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments revisited: a ten-year perspective on the industrial application of formal methods. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 8–16, New York, NY, USA, 2005. ACM Press.

[9] Philippa J. Broadfoot and A. W. Roscoe. Tutorial on FDR and its applications. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, page 322, London, UK, 2000. Springer-Verlag.

[10] Ana Cavalcanti and Jim Woodcock. A Weakest Precondition Semantics for Z. *The Computer Journal*, 41(1), 1998.

[11] Stefano Ceri, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. Automatic generation of production rules for integrity maintenance. *ACM Trans. Database Syst.*, 19(3):367–422, 1994.

[12] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[13] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[14] J. Davies, C. Crichton, E. Crichton, D. Neilson, and Ib H. Sørensen. Formality, evolution, and model-driven software engineering. In Alexandre Mota and Arnaldo Moura, editors, *Proceedings of SBMF 2004*, ENTCS, 2005.

[15] Jim Davies, David Faitelson, and James Welch. Domain-specific semantics and data refinement of object models. In *Proceedings of SBMF 2006*. Eletronic Notes in Theoretical Computer Science, 2006.

[16] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: model-oriented proof methods and their comparison*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

[17] Birgit Demuth, Heinrich Hussmann, and Sten Loecher. OCL as a specification language for business rules in database applications. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 104–117, London, UK, 2001. Springer-Verlag.

[18] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[19] Moshe Deutsch, Martin C. Henson, and Steve Reeves. An Analysis of Total Correctness Refinement Models for Partial Relation Semantics I. *Logic Jnl IGPL*, 11(3):285–315, 2003.

[20] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[21] Roger Duke and Gordon Rose. *Formal Object-Oriented Specification Using Object-Z*. Macmillan Press, 2000.

[22] David Faitelson, James Welch, and Jim Davies. From predicates to programs: the semantics of a method language. In *Proceedings of SBMF 2005*. Eletronic Notes in Theoretical Computer Science, 2005.

[23] Pierre Flener. Achievements and prospects of program synthesis. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, pages 310–346, London, UK, 2002. Springer-Verlag.

[24] Peter M.D. Gray, Krishnarao G.Kulkarni, and Norman W.Paton. *Object-Oriented Databases A Semantic Data Model Approach*. Prentice Hall, 1992.

[25] Alena Griffiths. An Extended Semantic Foundation For Object-Z. Technical report, SVRC, University of Queensland, 1995.

[26] Anthony Hall. Using Z as a Specification Calculus for Object-Oriented Systems. In *Proceedings of VDM '90*, pages 290–318. Springer-Verlag, 1990.

[27] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[28] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[29] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Commun. ACM*, 30(8):672–686, 1987.

[30] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[31] John G Hughes. *Object-Oriented Databases*. Prentice Hall, 1991.

[32] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England, 2000.

[33] H. V. Jagadish and Xiaolei Qian. Integrity maintenance in object-oriented databases. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 469–480. Morgan Kaufmann Publishers Inc., 1992.

[34] Stefan Kahrs. Term rewriting systems by "Terese", cambridge university press, 2003. *J. Funct. Program.*, 15(4):651–651, 2005.

[35] Viktor Kuncak and Daniel Jackson. Relational analysis of algebraic datatypes. *SIGSOFT Softw. Eng. Notes*, 30(5):207–216, 2005.

[36] Kevin Lano, David Clark, and Kelly Androutsopoulos. UML to B: Formal verification of object-oriented models. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *IFM*, volume 2999 of *Lecture Notes in Computer Science*, pages 187–206. Springer, 2004.

[37] K Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. *The Fourth International Workshop on Foundations of Object-Oriented Languages*, Jan 1997.

[38] Udo W. Lipeck and Bernhard Thalheim, editors. *Modelling Database Dynamics, Selected Papers from the Fourth International Workshop on Foundations of Models and Languages for Data and Objects, Volkse (near Braunschweig), Germany, 19-23 October 1992*, Workshops in Computing. Springer, 1993.

[39] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

[40] Enric Mayol and Ernest Teniente. A survey of current methods for integrity constraint maintenance and view updating. In Peter P. Chen, David W. Embley, Jacques Kouloumdjian, Stephen W. Liddle, and John F. Roddick, editors, *ER (Workshops)*, volume 1727 of *Lecture Notes in Computer Science*, pages 62–73. Springer, 1999.

[41] Tim McComb and Graeme Smith. Compositional class refinement in Object-Z. In *FM 2006*, volume 4085 of *LNCS*. Springer, 2006.

[42] C. Metayer, J. Abrial, and L. Voisin. Event-B language. Ist-511599, RODIN Project, May 2005.

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

[44] C. Carroll Morgan. *Programming From Specifications (second edition)*. Prentice Hall, 1998.

[45] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Trans. Program. Lang. Syst.*, 11(4):517–561, 1989.

[46] University of Oxford. Graduate studies. `http://www.admin.ox.ac.uk/gso/`.

[47] Iman Hafiz Poernomo, John Newsome Crossley, and Martin Wirsing. *Adapting Proofs-as-Programs: the Curry–Howard Protocol*. Springer, 2005.

[48] N. S. Prywes, Amir, and S. Shastry. Use of a nonprocedural specification language and associated program generator in software development. *ACM Trans. Program. Lang. Syst.*, 1(2):196–217, 1979.

[49] Charles Rich and Richard C. Waters. Approaches to automatic programming. *Advances in Computers*, 37:1–57, 1993.

[50] Gregory R. Ruth. Automatic programming: Automating the software system development process. In *ACM '77: Proceedings of the 1977 annual conference*, pages 174–180, New York, NY, USA, 1977. ACM Press.

[51] Klaus-Dieter Schewe, Bernhard Thalheim, Joachim W. Schmidt, and Ingrid Wetzel. Integrity enforcement in object-oriented databases. In Lipeck and Thalheim [**38**], pages 174–195.

[52] Rolf Schwitter and Norbert E. Fuchs. Attempto - from specifications in controlled natural language towards executable specifications. *CoRR*, cmp-lg/9603004, 1996.

[53] Graeme Smith. *The Object-Z specification language*, volume 1. Kluwer Academic, Boston, 2000.

[54] C. Snook and M. Butler. UML-B: Formal modelling and design aided by UML. Technical report, Electronics and Computer Science, University of Southampton, 2004.

[55] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[56] Bill Stoddart, Steve Dunne, and Andy Galloway. Undefined expressions and logic in Z and B. *Form. Methods Syst. Des.*, 15(3):201–215, 1999.

[57] J. van den Berg, C.-B. Breunesse, B. Jacobs, and E. Poll. On the role of invariants in reasoning about object-oriented languages. In *Proceedings of ECOOP'2001*, 2001.

[58] James Welch, David Faitelson, and Jim Davies. Automatic maintenance of association invariants. In *Proceedings of SEFM 2005*. IEEE Computer Society Press, 2005.

[59] James Welch, David Faitelson, and Jim Davies. Automatic maintenance of association invariants. *Software and Systems Modeling*, To appear.

[60] Niklaus Wirth. Records, modules, objects, classes, components. In J. Davies, A. W. Roscoe, and J. Woodcock, editors, *Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare*. Palgrave, 2000.

[61] Jim Woodcock and Jim Davies. *Using Z*. Prentice Hall, 1996.

[62] J. B Wordsworth. *Software engineering with B*. Addison-Wesley, Harlow, England, 1996.