Chapter 2
An Overview of Formal Methods Tools and Techniques

The goal of this chapter is to give an overview of the different approaches and tools pertaining to formal methods. We do not attempt to be exhaustive, but focus instead on the main approaches. After reading the chapter the reader will be familiar with the terminology of the area, as well as with the most important concepts and techniques. Moreover the chapter will allow the reader to contextualise and put into perspective the topics that are covered in detail in the book.

Why do we need an overview of formal methods? Why not just study *one* rigorous method for software development? This is a very pertinent and legitimate question. The behavioural essence of software is not captured by a unique unified mathematical theory. Such a general foundation is unlikely to exist.

Think for instance about the diversity of programming language paradigms and theories, and the resulting jungle of existing computer programming languages. Is there a definite paradigm (or, even, language) that makes obsolete all the other ones? Clearly not. Different languages will be chosen by different people to solve the same problem, and someone may well use different languages to solve different problems. Similarly, depending on the goals of the software designers and of the verification process, one may prefer a theory over another one, and even use more than one theory (and related formal methods techniques and tools), in the context of the development of a single system.

Even if the theory is fixed, several dialects and related tools may exist for it. Turning back to the programming language analogy, think for instance of the different existing C or Prolog dialects and compilers. A particular compiler may not even be significantly better than another, but its use will be justified for some users and some application scenarios. This is also a common situation with formal methods. Is it desirable? There is an open and vigorous debate about this issue; we simply point out that the potential user of formal methods should be aware of it and understand the different flavours available.

Our goal with the present overview is then to draw a map of this jungle of theories, techniques, and tools, to make some sense of it.

J.B. Almeida et al., *Rigorous Software Development*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-0-85729-018-2_2, © Springer-Verlag London Limited 2011

2.1 The Central Problem

Questions such as "What are formal methods?" or "What added value can be expected from the use of formal methods?" have been largely debated in the Software Engineering community [31, 36, 53, 88, 90]. In the following we sum up and discuss the main ideas.

The central problem of formal methods is to be able to guarantee the behaviour of a given computing system following some rigorous approach. At the heart of formal methods one finds the notion of *specification*. A specification is a model of a system that contains a description of its desired behaviour—*what* is to be implemented, by opposition to *how*. This specification may be totally abstract (in which case the model *is* the description of the behaviour), or it may be more operational, in which case the description is somehow contained, or implied, by the model. In this context, the central problem can be seen as being split in the following two aspects:

1. How to enforce, at the specification level, the desired behaviour? This is called the *model validation* problem.
2. How to obtain, from a specification, an implementation with the same behaviour? Or alternatively, given an implementation, how can it be guaranteed that it has the same behaviour as the specification? This is the *formal relation between specifications and implementations* problem.

The different families of formal methods cover a wide range of approaches to these two questions. This variety encompasses the study of specifications by *animation*, by *transformation*, or by *proving properties*. Analogously, implementations may be *derived* from specifications, or else they may be *guaranteed to be correct* with respect to them, either by *construction*, or by *verification*, i.e. by presenting a *formal proof.*

2.1.1 Some Existing Formal Methods Taxonomies

An interesting characterisation used to classify formal methods is the one that takes into account the ease of use and automation of the processes involved in their application. This gives rise to the taxonomy *lightweight* versus *heavyweight* usually found in the literature. Lightweight formal methods usually do not require deep expertise, by opposition to heavyweight formal methods, which are more complex, less automatic, but also more finely grained and powerful. They are typically confined to very specific application areas where their use and cost are justified.

Another existing taxonomy is related to the application of the Balzer software life cycle, as explained in the previous chapter. One speaks of *formally designed software* when formal methodologies and tools are applied in the horizontal fashion. The vertical application is referred to as the *correct-by-construction* approach or, when formal proof facilities are also provided in addition to the generation of code, as *formal development*.

2.1.2 *This Overview*

This chapter is based on a description of formal methods organised instead by layers of functionalities. We will first take a tour of the approaches to describing and analysing (formal) models; we will then cover the different existing proof mechanisms, and continue with a description of ways of formally relating models with programs, i.e. of approaching the second sub-problem identified above. Finally we will take a look at mechanisms for dealing with scalability issues. For each approach we will discuss in turn the key concepts and foundations involved and the corresponding tools.

The following central notions are common to a vast number of formal methods techniques and tools:

- The operational essence of the modelled systems is usually captured by some form of *transition system* (described either logically, relationally, or algebraically). The different mechanisms discussed below imply particular interpretations of *states*, *transitions*, and *state transformations*.
- The behavioural essence of the modelled systems is usually captured by some *program logic* (such as Hoare logic), a notion that stands at the heart of a large part of formal methods techniques and tools.

These notions are pervasive in the overview that follows.

2.2 Specifying and Analysing

The definition of a specification and the analysis of its behaviour may be carried out formally, i.e. in the context of some mathematical formalism. The advantages of this include the following:

- The formal nature of the specification language employed forces one to reason about and understand all the fine details of the specified system, and thus clarify potential hidden ambiguities. Several published case studies confirm such benefits (see Sect. 2.7). For instance the survey [61] reports on work by Don Syme, that allowed several non trivial errors to be found simply by writing down hand-built specifications in a formal way.
- The possibility to animate, or even execute, a specification—and thus to directly observe its behaviour—if an implementation of the underlying mathematical formalism exists. Such a functionality allows for the system to be *prototyped*. The straightforward advantage of having a prototype is that it is in general simpler to deploy than the system itself. A prototype makes possible the validation of a system without actually implementing it.
- A prototype obtained in this way is still a formal entity, amenable to being mathematically manipulated. One may thus reason about it (either by hand or with the assistance of a computer).

A specification essentially describes the manipulated data and how they evolve, i.e. the operations that transform them. The two main approaches to formal specification differ on the focus given to these two aspects:

- The behaviour of the modelled system can be expressed by focusing on its *operations*, available mechanisms (services), or actions that can be performed. In this view the crucial element is a clear definition of the modifications or changes performed by each operation on the *internal state* of the modelled system. Such specification languages are referred to as *state-based* or *model-based specification* languages.
- The behaviour of the target system can instead be expressed by focusing on the *manipulated data*, how they evolve, or the way in which they are related. This class of specifications includes *algebraic specifications*, sometimes also known as *axiomatic specifications*.

In what follows we will consider in turn the characteristics of the specification languages used for each of these approaches.

2.2.1 Model-Based Specification

The languages used for this class of specifications are characterised by the ability to describe the notion of internal state of the target system, and by their focus on the description of how the operations of the system modify this state. The underlying foundations are in discrete mathematics, set theory, category theory, and logic.

Abstract State Machines Proposed by Gurevich [58], abstract state machines (ASM), also called *evolving algebras*, form a specification language in which the notions of state and state transformation are central. A system is described in this formalism by the definition of states and by a finite set of (possibly non-deterministic) state transition rules, which describe the conditions (also called guards) under which a set of transformations (modifications of the machine's internal state) take place. These transitions are not necessarily deterministic: the formalism takes into account configurations in which several transitions are eligible for a certain state of the machine.

Given that the ASM formalism has the computational power of a Turing machine, it can be used as an executable specification language. The notion of execution of an ASM is the usual notion in the context of transition systems. For instance ASM_Gopher [94] is an implementation of this formalism that has served as the basis for a formalisation of the Java programming language and virtual machine.

Another specification methodology based on the notion of ASM is the B Method, accompanied by its B specification language [2]. The systems modelled in B are seen, as in many other formal methods, as transition systems. The basic unit is called an *abstract machine*, and specifications and programs are represented using a dedicated notation for these abstract machines. In a sense, the B modelling methodology

Fig. 2.1 An example abstract machine in B

```
MACHINE Car_status
SETS
  STATUS = {sold, available}
USES Cars
VARIABLES status
INVARIANT
  status ∈ CARS ⇸ STATUS
INITIALISATION status := ∅
OPERATIONS
set_status(x, m) ≙
   PRE m ∈ STATUS ∧ x ∈ CARS
   THEN
     status(x) := m
   END
 END;
```

is close to object-oriented modelling. Each machine defines the structure of its internal state (values), the properties that this state must always comply with (static properties called *machine invariants*), and the expected operations (the transitions). The expected properties are defined in a suitable first-order logic extended with a particular set theory. An important principle is that each specified operation *must preserve* the machine invariants (a property referred to as *internal consistency*).

Over the years the B method has given rise to more than a dozen implementations, including Atelier B [87] (a commercial product with a freely available version), BRILLANT [40, 41] (an open source platform), ProB [77] (also open source, includes an animator and a model checker), and Rodin [3] (an open source platform dedicated to the implementation of a recent and popular dialect called Event-B).

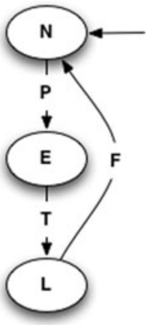
Let us give an example of a B specification. The abstract machine shown in Fig. 2.1 encapsulates the notion, for a car dealer, of a car being sold or available. Using the previously defined machine *Cars* that introduces the notion of *CARS*, the machine keeps track of the previously recorded cars and associated status by the use of a partial function from cars to status. At the level of code, one may expect this to be implemented by some form of database or container datatype.

This machine also illustrates how the internal state may evolve, by providing an operation that allows for the status of a car to be introduced or updated. More precisely, the specification ensures that if the operation *set_status* is executed in a state in which the condition $m \in STATUS \wedge x \in CARS$ holds, then the resulting state records the previous *status* updated, by mapping x to m.

Set and Category Theory These two mathematical theories offer similar expressiveness at the level of specifications. States are described in terms of mathematical structures such as sets, relations, or functions. Transitions are expressed as *invariants* as well as preconditions and postconditions.

Z [93] and VDM [67] are classic examples of formal methods whose specification languages rely on set theory. These methods have been at the origin of many other systems. RAISE [56], for instance, is an evolution of VDM that also includes

Fig. 2.2 One task and one resource



a concurrency specification layer *à la* CSP. The B specification method can also be seen as falling in this category, since it combines the use of ASMs with features inherited from both Z and VDM. We may also cite the emerging methodology Alloy and its related tool Alloy Analyser [65], which adapts and extends declarative languages like Z to bring in fully automatic (but partial) analysis. Specware [68] and Charity [38] on the other hand offer formalisms based on category theory.

Automata-Based Modelling A different class of transition systems used for specification purposes are *automata* [5, 6, 74, 78, 86, 97, 100]. In this case it is the *concurrent behaviour* of the system being specified that stands at the heart of the model. The main idea is to define how the system reacts to a set of *stimuli* or events. A state of the resulting transition system represents a particular configuration of the modelled system. This formalism is particularly adequate for the specification of *reactive*, *concurrent*, or *communicating* systems, and also protocols. It is however less appropriate to model systems where the sets of states and transitions are difficult to express.

Let us consider for instance a system that allows two tasks to access a shared resource. For one task, the process of using this resource is depicted by the Büchi automaton shown in Fig. 2.2. The state N corresponds to the situation in which the task is idle. The access request is done via the event/transition P, after which the automaton reaches the state E, which corresponds to waiting for access to the resource. The transition T represents the authorisation of access, which allows the task to reach the state L, in which the task can perform any operation that requires access to the resource. The event F represents the release of the resource, leading back to state N.

Now if one wants to extend this simple model to two tasks (a and b) that compete for access to the same resource, this can be done with the automaton of Fig. 2.3, which is obtained by calculating the Cartesian product of the previous automaton with itself, and then removing states and transitions that are meaningless or simply unwanted.[1] We remark that we can easily convince ourselves that there is no deadlock in this schema, but more interestingly this can also be proved. One can

[1] The remaining transitions are usually called the *synchronisation set* over the Cartesian product.

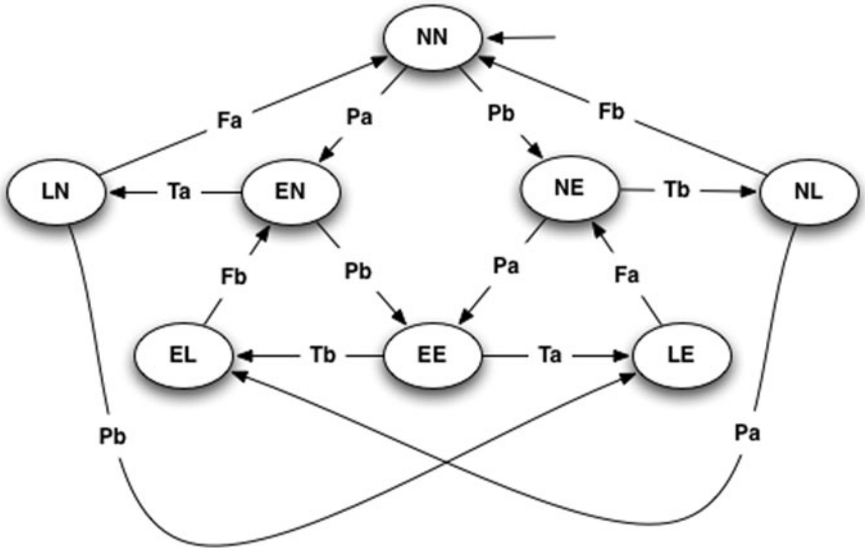


Fig. 2.3 Two tasks and one shared resource

also understand quite easily (and again, also prove) that this sharing policy is not fair. In fact it is possible that a asks for the resource followed by b, that repeatedly asks and is given access to it (this corresponds to the repeated execution of the loop $EN - EE - EL$). More on this simple example can be found in [19].

Modelling Languages for Real-Time Systems Extending the simple automata framework gives rise to several interesting formalisms for the specification of real-time systems. When dealing with such systems, the modelling language must be able to cope with one or more physical concepts like time (or duration), temperature, inclination, altitude, etc. In fact, examples of real-time systems include for instance control systems that react in dynamic environments. Traditionally, time (usually modelled by means of clocks that follow a very specific progression law) is the dimension that has attracted most attention from researchers in the field.

For instance in the context of *synchronous concurrent models* [18, 59], the time flow is partitioned in discrete instants. An integer variable x can be treated, in fact, by considering the sequence of all the values that it takes at each discrete instant (e.g. $x = 1, 2, 3, 4, 5, 6, 7, \ldots$). The modelled systems progress (behave) according to successive atomic reactions upon the previous and actual values of the involved variables. Again, this kind of model of a system can be seen as an automaton where the states correspond to assignments of values to system variables, and where transitions correspond to system reactions.

Lustre [35] is a (textual) synchronous dataflow language, and SCADE [1] is a complete modelling environment that provides a graphical notation based on Lustre. Both provide a notation for expressing synchronous concurrency based on dataflow. Consider the SCADE example of Fig. 2.4. It introduces an operator (the unit of en-

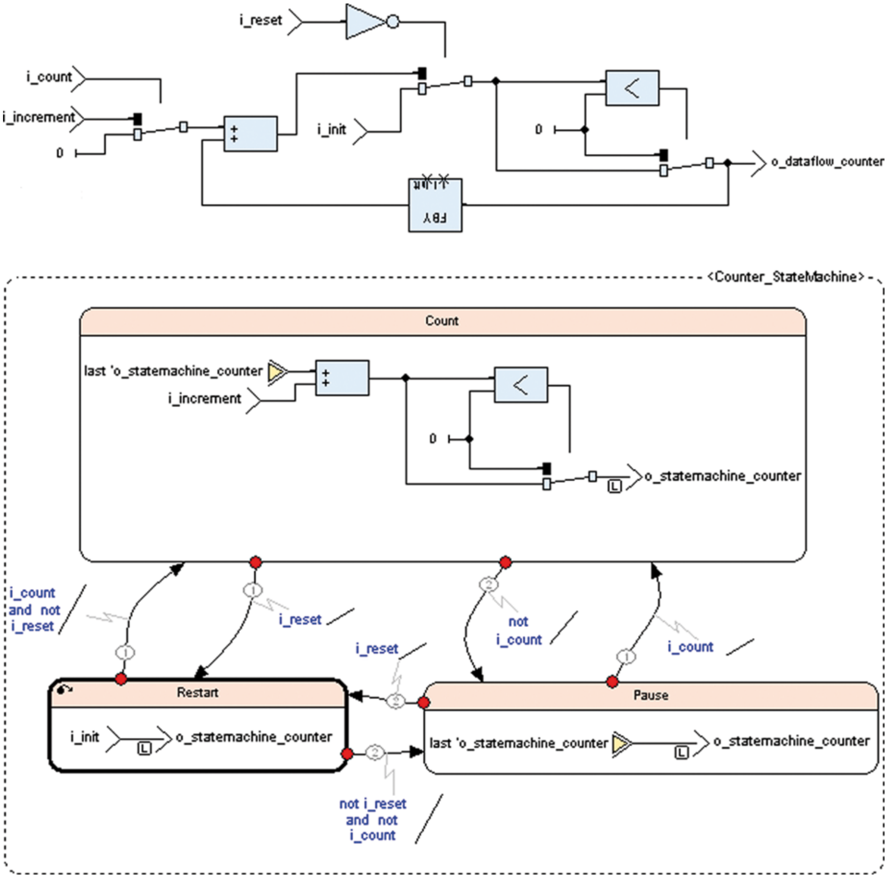


Fig. 2.4 Two counters in SCADE

capsulation in SCADE) that provides two alternative implementations of a counter: a dataflow-based counter and a state machine counter. Both counters run concurrently.

This operator has four input parameters:

- *i_init*: initial value of the counter;
- *i_increment*: value of the increment;
- *i_count*: counter switch;
- *i_reset*: reset the counter to the value of *i_init*;

and two output parameters:

- *O_statemachine_counter*: records the value of the state machine counter;
- *O_dataflow_counter*: records the value of the dataflow counter.

Other graphical formalisms have proved to be suitable for the modelling of real-time systems. One of the most popular is based on networks of *timed automata*.

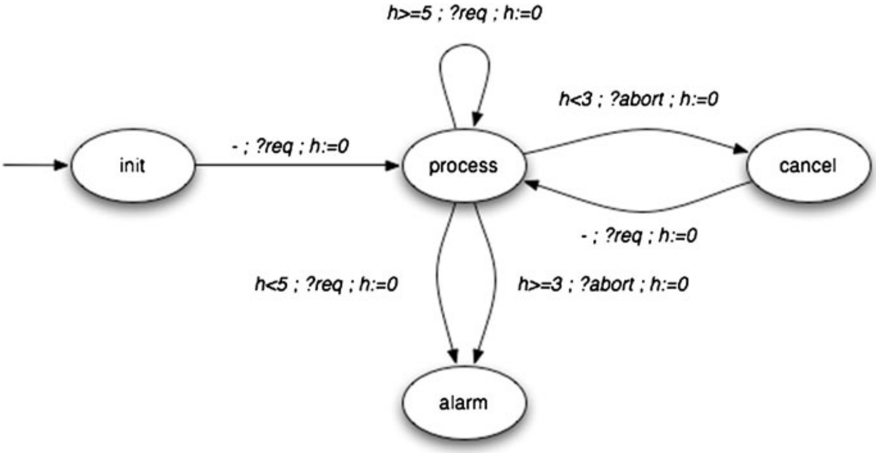


Fig. 2.5 A timed automaton for a requests processor

Basically, timed automata extend classic automata with clock variables (that evolve continuously but can only be compared with discrete values), communication channels, and guarded transitions. A theory of timed automata [7] provides the underlying foundations of this approach, which gave rise to model checking tools (see Sect. 2.3.3) like Uppaal [17] or Kronos [32].

The timed automaton represented in Fig. 2.5 models a control system that handles some processing request in a timely fashion. The control system reacts to two actions, a process request *req* and a cancel request *abort*. A process request is taken into account if a previous request has not taken place less than 5 time units before. Similarly an abort request is processed only if it is received at least three time units after a process request. If either of these two rules is not observed then the control system reaches an alarm state. The control involving time is done via the clock h. Only simple comparisons and a reset operation are possible. The transition $h \geq 5; ?req; h := 0$ stands for "this transition is possible when $h \geq 5$ and a process request is received; if the transition is chosen, then the clock h is reset to 0". This control system is supposed to be executed concurrently with a system (modelled by another timed automaton) that emits the required signals through the adequate communication channel ($!req$ and $!abort$).

Hybrid automata [62] extend timed automata with the possibility to deal with other physical measures beyond time. Although the general theoretical context is very difficult (the problems that arise are easily undecidable), the existence of decidable fragments has enabled the creation of tools, such as Hytech [63]. Although essentially a deductive tool, we should also mention the recent Keymaera platform [85] for modelling and verification of hybrid systems, based on *differential dynamic logic* [84]. This is intended to model dynamic systems with interacting discrete and continuous behaviour, which is classically characterised by differential equations and discrete transitions. One advantage of the underlying theory is that it can handle such characterisations and provide proof mechanisms for them.

Fig. 2.6 Algebraic specification of lists

Spec: $LIST_0\ (ELT)$
Extends: Nat_0
Sorts: *list*
Operations:

$$\begin{array}{lllll}
nil & : & & \rightarrow list & // constructor \\
cons & : elt & \rightarrow\ list & \rightarrow list & // constructor \\
length & : list & & \rightarrow nat & \\
hd & : list & & \rightarrow elt & \\
tl & : list & & \rightarrow list & \\
append & : list & \rightarrow\ list & \rightarrow list & \\
rev & : list & & \rightarrow list &
\end{array}$$

Axioms: $xs, ys : list,\ \ x : elt$

$$\begin{array}{ll}
length(nil) & = 0 \\
length(cons(s, xs)) & = 1 + length(xs) \\
hd(cons(x, xs)) & = x \\
tl(cons(x, xs)) & = xs \\
append(nil, ys) & = ys \\
append(cons(x, xs), ys) & = cons(x, append(xs, ys)) \\
rev(nil) & = nil \\
rev(cons(x, xs)) & = append(rev(xs), cons(x, nil))
\end{array}$$

2.2.2 Algebraic Specification

A second classic approach to specification is based on the use of multi-sorted algebras. A multi-sorted algebra consists of a collection of data grouped into sets (one set for each datatype), a collection of functions on these sets corresponding to functions of the program being modelled, and a set of axioms specifying the basic properties of the functions in the algebra. Such a formalism allows one to abstract away from the algorithms used to encode the desired properties, to concentrate on the representation of data and on the input-output behaviour of functions. In addition to multi-sorted algebras, the foundations of algebraic specification lie in mathematical induction and equational logic.

An algebraic specification then consists in a series of sort declarations, function signatures, and axioms that declare the basic behaviour of each function symbol. CASL [22, 39], OBJ [54], Clear [33], Larch [55], and ACT-ONE [51] are all examples of tools based on algebraic specification languages. LOTOS [24] extends the algebraic framework with CCS primitives, and thus allows for specifying and reasoning about concurrent systems.

As an illustrative example, Fig. 2.6 introduces a container datatype for lists, based on the two usual elementary operations on lists—the constructors *nil* (the empty list) and *cons* (the operation that adds one element at the head of a list). This datatype relies on some existing algebraic specification of natural numbers. The operations over the datatype are declared by stating their names and signatures; their behaviour is expressed equationally in terms of their relation with the two constructors. Any implementation complying with this specification must ensure the stated equational properties hold. Thus, such implementations will also comply with the inferred properties.

An interesting point is the ability to infer additional and rich properties from the stated ones. We will address this issue in the sequel.

2.2.3 Declarative Modelling

An important class of specification languages includes logic-based languages, functional languages, rewriting languages, and languages for defining formal semantics. All of these rely on well-known mathematical foundations.

Logic programming languages, such as Prolog [95], propose an approach to modelling based on the notion of *predicate*. Data are represented with the help of some simple, but sufficiently expressive datatypes, such as lists, and operations are described by their behavioural properties, similarly to axioms in algebraic specifications. Prolog allows for specifications to be executed.

Functional languages on the other hand offer a specification framework in which the notion of *function* is the central element. The core of a functional language is the *λ-calculus* [10, 11], which has the same expressiveness as Turing machines and allows for the formulation of any operation, and even any datum, in terms of higher-order functions.

Languages like Scheme [50], SML [81], Haskell [96] or OCaml [75], and proof assistants such as ACL2 [69], Coq [79], PVS [91], HOL [57], Isabelle [83], and Agda [27], are all based on typed variants or extensions of the λ-calculus. These extensions are easier to use than the original calculus, and all propose a number of basic data types, powerful type construction operators (such as inductive types), and mechanisms for the definition of functions.

Execution in these languages relies on the notion of *reduction*, which resembles the notion of calculation in mathematics. These languages are all higher-order, and some (such as Coq) possess even richer extensions. This allows for great flexibility and expressiveness. The languages underlying proof assistant systems based on type theory and the Curry-Howard isomorphism [11, 92] may also be used as higher-order logical languages.

Rewriting systems [8, 21, 72] like ELAN [25] or SPIKE [26] offer languages that are very close to those used in algebraic specification, with the difference that axioms are replaced by equations that characterise the behaviour of function symbols in calculations.[2] As in the λ-calculus, execution relies on the notion of reduction, which is usually defined by sets of equations.

For illustration purposes, the Coq example shown in Fig. 2.7 introduces the abstract syntax of two simple languages. `expr` denotes simple arithmetic expressions with variables and assignment; `intr` corresponds to expressions of a very simple assembly language. The recursive function `compil` introduces a compilation schema, that is a translation of expressions to programs (lists of instructions). Once these notions have been introduced, one can define the operational semantics via the

[2] See http://rewriting.loria.fr/ for a list of many other rewriting systems.

```
Inductive expr : Set :=  | Var : ident -> expr
                         | Num : nat -> expr
                         | Atrib: ident -> expr -> expr
                         | Sum : expr -> expr -> expr
                        ...

Inductive instr : Set :=  | LOAD : ident -> instr
                          | STORE : ident -> instr
                          | PUSH : nat -> instr
                          | DUP : instr
                          | ADD : instr
                         ...

Definition program := (list instr).

Fixpoint compil (e : expr) {struct e} : program := match e with
            | Var i => (LOAD i)::nil
            | Num n => (PUSH n)::nil
            | Atrib i e2 => app (compil e2) (DUP::(STORE i)::nil)
            | Sum e1 e2 => (app (app (compil e1) (compil e2)) (ADD::nil))
           ...
end.
...

Theorem correctness : forall e:expr,
                     (extract_eval (aval e nil)) =
                     (extract_exec (exec (mkstate nil nil )(compil e))).
Proof.
...
```

Fig. 2.7 Two simple languages in Coq

notion of execution at source (`aval`, that depends on a variable environment) and at machine level (`exec`, that depends on an execution environment). This allows us to elegantly define the notion of compiler correctness in this very simple context: for every expression `e`, executing `e` at source level or executing its compiled form at machine level must always give the same result. This is what is stated by the `correctness` theorem.

2.3 Specifying and Proving

Animation and execution are not sufficient to ensure a certain behaviour for a specification: it is essential to obtain a rigorous demonstration. Rather than simply constructing specifications and models one is interested in proving properties about them. We are thus in the realm of *formal verification*.

Different notions of verification exist; Rushby [88] proposes a three-level classification of proof methods as follows:

1. The first level groups formal frameworks that do not offer any computer-based support for handling proofs. Demonstrations must then be carried out by hand; proofs are validated when reviewers are convinced of their contents.
2. In the second level we have frameworks that additionally offer a formal system allowing for a more rigorous formulation of demonstrations. For instance usage of natural language, which is possible in level 1, is not admissible in level 2. But demonstrations are still carried out by hand.

3. The third level consists of computer-based tools with support for proofs and for carrying out demonstrations. This level offers the highest degree of exactitude and guarantee. The undeniable advantage is that models can be both *expressed* and *reasoned about* in a formal (and possibly mechanical) way.

Whichever method is used, proving properties about specifications presupposes the use of some logical system. In what follows we first give in Sect. 2.3.1 an overview of logical concepts. Propositional logic and first-order logic will be the subject of Chaps. 3 and 4, so our goal here is to give the basic notions necessary for understanding the subsequent Sects. 2.3.2 to 2.3.4, in which we discuss different classes of computer-based proof tools that are often used in formal verification.

2.3.1 Logic in a Nutshell

Logic can be described as the study of the principles of reasoning. Reasoning about situations means constructing arguments about them. We are interested in doing this formally, so that the arguments are valid and can be defended rigorously. A formal logic is a language equipped with rules that allow one to establish when the truth of a given sentence can be concluded from the truth of other sentences.

Symbolic logic is the branch of mathematics devoted to formal logic, i.e. to the study of logical languages, their semantics, and their proof theory, and the way in which these are related.

A logic consists of:

- A *logical language* in which sentences are expressed. A logical language is a formal language having a precise, syntactic characterisation of well-formed sentences. A logical language consists of logical symbols, characterised by having a fixed interpretation, and non-logical ones, whose interpretations are not fixed. These symbols are combined together to compose well-formed formulas.
- A *semantics* that differentiates valid sentences from refutable ones. The semantics is defined in terms of the truth values of sentences. This is done using an interpretation function that assigns meaning to the basic components, given some domain of objects that our reasoning is concerned with.
- An *inference system* (or *proof system*) that supports the formalisation of arguments justifying the validity of sentences. The inference system is composed of a set of axioms (sentences of the logic that are accepted as true) and inference rules (that give ways of deriving the right conclusions from a given set of premises).

Of course, extreme care must be taken regarding the definition of an inference system, since it is expected that all the derived formulas are indeed justified semantically (this is usually called a *soundness criterion*), and that all consequences justified semantically are derivable by the system (a *completeness criterion*).

Propositional Logic, First-Order Logic, and Higher-Order Logic Let us briefly describe these three well-known logics. Each is a richer logic than the previous; all of them are widely used in the context of formal verification, so it is useful to have some understanding of their characteristics.

Propositional logic (also known as the *propositional calculus*) is the simplest of the three. The formulas of propositional logic are built from atomic propositions, which are sentences with no internal structure and which one can classify as being "true" or "false". Propositions are declarative sentences such as "Mary is the mother of John" or "$3 < 5$". Propositions are combined using Boolean operators that capture notions like "not", "and", "or", "implies", etc. In fact, the content of the propositions is not relevant. Propositional logic is not the study of the truth of individual formulas, but rather of the way in which the truth of one statement affects that of another.

First-order logic is a considerably richer logic than propositional logic. In addition to the symbols of propositional logic, a first-order language contains elements that allow us to reason about individuals of a given domain of discourse. These include functions, predicates, and quantification over individuals, dealing with the notions of "there exists" and "for all". There are two sorts of things involved in a first-order logic formula: *terms*, which are interpreted as individuals in the domain of discourse; and*formulas*, which are interpreted as truth values. First-order logic is also known as the *predicate calculus* in the sense that it is a calculus for reasoning about predicates such as "x is the mother of y" or "$x < x + 1$". While propositions are either true or false, predicates evaluate to true or false depending on the values given to their parameters (x and y in the previous examples). Quantification over individuals makes possible to express concepts such as "every person has a mother".

Higher-order logic is distinguished from first-order logic in several ways. It has both individual and relational variables, and both types of variables can be quantified, so quantifiers may apply to variables standing for predicates. There is a typing discipline to distinguish individuals from predicates; predicates can take as arguments both individual symbols and predicate symbols (these are higher-order predicates). Concepts such as "every property that holds for x also holds for y" can be naturally expressed in higher-order logic.

First-order logic is more expressive than propositional logic: it has more rules, that allow one to construct more complex formulas. In turn, higher-order logic is more expressive than first-order logic.

Classical versus Intuitionistic Logic There are two different branches of formal logic: the *classical* (based on the notion of *truth*) and the *intuitionistic* (based on the notion of *proof*). The classical branch of logic is based on the understanding that the truth of a statement is absolute: statements are either true or false. In a classical setting, "false" and "not true" mean the same thing. This is expressed by the *law of the excluded middle*, which states that $A \vee \neg A$ must hold independently of the meaning assigned to A. Classically, in order to prove a proposition A, it is valid to assume $\neg A$ and obtain a contradiction as result. This classic practice of proving a statement *by contradiction* is captured by an inference rule known as *reductio ad absurdum*.

The intuitionistic (or *constructive*) branch of logic rejects the law of the excluded middle. A statement A is "true" if we can prove it, or is "false" if we can show that if we have a proof of A we get a contradiction. If neither of these can be shown, then there exists no justification for the presumed truth of the disjunction $A \vee \neg A$. In an intuitionistic setting, judgements about a statement are based on the existence of a proof (or "construction") of that statement. Mathematicians are typically inclined to resort to classical reasoning, in spite of the fact that most standard mathematics fit within the framework of intuitionistic logic. In some cases the inability to use classical proof methods such as proofs by contradiction make reasoning much more difficult.

The guiding principle of intuitionistic logic is however very attractive to computer scientists, due to the algorithmic nature of constructive proofs. The importance of the relationship between logic and computer science cannot be overstated. Each field is of paramount importance to the other: logic plays a crucial role in computer science since it supplies the tools to formalise many important concepts in this field, in such a way that they can be reasoned about formally. On the other hand computers are useful tools for logic: formal logic makes it possible to calculate consequences at the symbolic level, and computers can be used to automate such symbolic calculations.

One of the most remarkable manifestations of this interplay between logic an computer science is the correspondence between systems of intuitionistic logic and typed lambda calculi, known as the *Curry-Howard isomorphism*. This correspondence establishes a connection between proof theory and type theory that is visible in many aspects: propositions correspond to types, proofs correspond to terms, the provability of a formula corresponds to the inhabitation of a type, proof normalization corresponds to term reduction, and so on. On a practical level, this correspondence give us ways to extract computer programs from constructive proofs, or even to view proofs as programs themselves. Moreover, the Curry-Howard isomorphism is at the core of some proof assistant tools (see also Sect. 2.4.2).

Propositional and first-order logic are essential tools for the verification of programs in the sense explained in Sect. 2.3.4, which is the main theme of this book. As such they will be covered in detail in Chaps. 3 and 4.

Temporal Logic The examples we gave to illustrate the expressive power of the different calculi where chosen because the truth value of the statements are static and cannot vary over time (i.e. they are always true or always false). Consider now the statement: "It is raining". Although the meaning of this statement is stable over time, its truth value can vary, sometimes it is true and sometimes it is false. One can capture this dependence by considering time as an object of discourse and making statements depend on a time variable, but the result is very clumsy.

The expression *temporal logics* is used to refer to logics that implicitly include a notion of time, providing a way to represent temporal information in the logical framework. In temporal logics the truth of a formula is not fixed in the semantics but depends on the point in time in which it its considered. Temporal logics have two kinds of operators: the usual logical connectives (such as "not", "and" or "implies") and temporal connectives (such as "eventually", "always" or "until"), allowing one to express statements like "It will be raining until the end of the race" or "It will eventually rain".

There exist many different sorts of temporal logics. Concerning the way time is viewed they are classified as *linear-time*, when time is represented by a sequence of time instants, or *branching-time*, when time is viewed as a tree of time instants, having the present instant as root, and with each branch corresponding to one possible future evolution. Temporal logics have found important applications in formal methods, in modeling behavioural aspects of the execution of computer systems.

2.3.2 Proof Tools

We have mentioned that computers can be used to calculate logical consequences. Let use discuss in some detail the ways in which this can be done.

Two opposing factors have an impact on the deductive behaviour of proof engines. On one hand, logical *expressiveness* permits studying complex properties and deductions; on the other hand, the *simplicity* of the logical formalism facilitates the automation of deductions. The two families of proof tools presented below represent choices that lead to different trade-offs between expressiveness and automation.

Automated Theorem Provers These tools favour automation of deduction rather than expressiveness. Construction of proofs is automatic, once the proof engine has been adequately parameterised. Naturally these systems must rely on the decidability of at least a large fragment of the underlying theory. This is the case for *Horn clauses* as used in Prolog; for the first-order rewriting used for instance by ELAN; and for the fragment of first-order logic used by ACL2. Satisfiability Modulo Theory (SMT) solvers also fall in this category: tools like Yices [48], CVC3 [13], Z3 [46], or Alt-Ergo[3] [42] provide decision procedures for several theories of real numbers, integers, and of various data structures such as lists, arrays, bit vectors and so on. A historically important theorem prover with an almost unbeatable reputation in the program verification community is Simplify [47], an ancestor of modern SMT solvers.

Unlike model checkers (covered in Sect. 2.3.3), theorem provers may be able to employ techniques that allow for reasoning about infinite sets. Inductive reasoning techniques are an example of these.

Proof Assistants Unlike theorem provers, proof assistants elect highly expressive (and thus undecidable) underlying logics, such as higher-order logic. There exist no decision procedures capable of proving arbitrary properties in such logics. If this

[3] Among others; see http://www.smtlib.org/ for a more complete list.

sounds restrictive, it must be emphasized that many properties need the power and elegance of higher-order logic to be adequately expressed and proved.

Proof assistants typically combine the following two modules:

- a *proof-checker*, responsible for verifying the well-formedness of the theories defined in the modeling process, and for checking the correctness of proofs;
- an interactive *proof development system*, to help (error-prone) users developing proofs. When the construction of a proof is finished, a proof script can be stored, describing that construction.

In most proof assistants proofs are *interactively constructed* by applying high-level proof-manipulation functions, usually known as *tactics*. Each tactic encodes a proof step. The proof state is usually represented as a stack of sequents (a pair of a sequence of hypotheses and a conclusion to be proved from them). A proof of a property ϕ is established by applying (in an appropriate order and with the right parameters) a set of tactics, in order to construct a proof tree linking axioms and theorems to the conclusion ϕ. In its simplest form, a tactic expresses a basic proof step, such as *modus ponens*. Tactics are however not restricted to such atomic steps—there is scope for complex reasoning and even for complete proofs, as is the case with tactics implementing decision procedures on decidable fragments of higher-order logic.

Two approaches are possible in the world of proof assistants. The first consists in giving users the possibility to define the logic in which they desire to express proofs. This logic is usually called the *object logic*. This axiomatic approach has been adopted for instance by the Isabelle system. The other approach is to offer a basic language that is sufficiently expressive to formulate most of mathematics. This integrated approach can be found in systems like Coq. In Coq proofs are first-class citizens of the language, at the same level as propositions and specifications. In these systems, proofs can be directly written by the user—but they are not in general necessarily easy to write. In practice, proof terms are more often generated as the result of a successful proof process, in addition to a proof script. Proof terms can be independently checked, which finds applications notably in the proof-carrying code techniques mentioned in Chap. 1. Other advantages of this approach will be discussed below.

To finish the section, let us remark that it is possible to combine automatic theorem proving with interactive proof facilities—this is the case in implementations of the B Method, whose (first order) proof mechanism allows for the interactive demonstration of lemmas that could not be proved mechanically.

2.3.3 Model Checking

Model checking [9, 19, 37] is a technique for the verification of finite-state (concurrent) systems (typically modelled by automata, see Sect. 2.2.1). It is one of the most widely used families of formal methods tools.

The idea of model checking is that the expected properties of the model are expressed by formulae of a *temporal logic*, and efficient symbolic algorithms are used to traverse the model in its entirety, so as to verify if all possible configurations validate those properties. The set of all states is called the model's *state space*. When a system possesses a finite state space, model-checking algorithms may in theory be used to realise the automatic demonstration of properties. If a property is not valid, a *counterexample* is exhibited.

A serious drawback of this approach is *state space explosion*: the transition graph typically grows exponentially on the size of the system, with the immediate consequence that no matter how efficient the checking algorithms are, the exploration of the state space eventually becomes impracticable.

Different techniques have been proposed that try to solve this problem. *Abstraction* is one such technique: a simplified version of the model is proposed, called an *abstract model*, whose state space may be explored within reasonable time. The abstract model must respect certain properties, in order to ensure that if a property is valid in it, then it is valid in the original model.

2.3.4 Program Logics and Program Annotation

To finish off with proof techniques, let us now consider tools based on *program annotations*. A program annotation is a formula placed together with the code of a program whose behaviour one wants to verify. The annotation of a program function, method, or piece of code in general, is supposed to indicate the conditions that should be met before the said code is executed, as well as describe the logical state of the program after its execution. The logical formalisms underlying this approach are program logics like *Hoare logic*. This family of formalisms is very diverse: even if the basis of the annotation languages is quite standard, the semantics of annotations is specific to the programming language at hand.

Note that this is a somewhat different setting with respect to what we have been considering in Sect. 2.3: we are no longer discussing properties of formal models of systems in general, but instead behavioural properties of *programs* (in particular source code) written as annotations. Roughly speaking, an annotation is a code-aware version of the requirements. The rationalisation of the code annotation methodology, coupled with its integration within the software engineering discipline, gave rise to a software development paradigm based on the notion of *contract* (a specific form of annotation), as pioneered in the Eiffel programming language [80], which implements the notion of *runtime* or *dynamic verification* of contracts.

This paradigm has nowadays become very popular, in fact almost every widespread programming language benefits from a contracts layer. Let us cite for instance the programming languages SPEC# [12] (which can be seen as a superset of C#) or SPARK [34], a carefully chosen subset of ADA targeted to the development of safety critical systems. Like Eiffel, both languages natively support the paradigm, but they additionally support the static verification of contracts. In the context of the Java programming language, different annotation systems exist that are based on the JML annotation language [66], such as Esc/Java [52], KeY [4] and Krakatoa [73].

The last programming language we consider here (and many are left out of this discussion) is C. Along with ADA, C is a popular choice in the safety critical industry, and the contract-based approach fits well the need for the static assurance of safety properties. A complete analysis and validation platform for C that provides a contracts layer is the Frama-C toolset [43], based on the ACSL annotation language [16], which in turn was inspired by JML. ACSL and the program verification functionality of Frama-C will be covered in Chaps. 9 and 10 of this book respectively. Another interesting contracts layer for C is provided by the VCC toolset [45]. The VCC approach allows for the verification of concurrent aspect of C programs.

Tools for statically checking the correspondence between the code and the annotations can be totally proof-based, but they can also associate the use of model-checking with a proof assistant. This is the case of the Loop [20] and Bandera [49] tools.

The undeniable advantage of the annotation-based approach is that it is the source-language implementation, and not some specification, that serves as the basis for the verification. In fact, a model is here constructed by taking as inputs a program and its annotations, together with an underlying model of the programming language. This approach is more and more seen as providing a satisfying alternative to the central problem of formal methods. This book covers the foundations of this approach, from Hoare logic to the generation of verification conditions for programs consisting of annotated routines.

2.4 Specifying and Deriving

We have to this point considered tools and techniques that address the first part of the central problem of formal methods. We now turn to the second part of this problem; given a specification that enjoys the desired properties, how to obtain an implementation whose behaviour matches the specification?

Again, different solutions exist to this problem. The solutions fit in two categories: either the specification is itself a program that can be directly executed (and the problem is immediately solved), or an implementation is produced from the specification, in which case the problem of the *correctness* of derivations must be dealt with.

One approach to dealing with this problem focuses on the derivation mechanisms, which can be restricted in appropriate ways, to ensure that the derived code satisfies the properties of the original specification. A second approach is to make the (not necessarily correct) derivation process generate a set of *proof obligations* such that, if all these obligations can be proved, this guarantees the correctness of the implementation with respect to the specification. Correction may then be ensured either manually, or preferably by machine, using level 3 formal verification tools (see Sect. 2.3). Formal verification is important even if the first approach is followed: in this case, it is the derivation process itself that has to be validated. If level 3 verification can be applied successfully to the derivation mechanism, then a universally valid procedure is obtained for all derivations.

Thus either the derivation mechanism or individual derivations must be subject to formal verification, which is in fact omnipresent at all stages of the central problem of formal methods. Both these approaches are sometimes referred to as *correct-by-construction* software development. We remark that some authors use this expression for the first approach only, when the derivation mechanism has been verified once and for all and specific derivations do not require proof. Other authors use it for development based on successive verified refinement steps, as will be described in Sect. 2.4.1.

We remark that in the program annotation approach discussed in Sect. 2.3.4, an internal model is deduced from the annotated code; the code is correct with respect to the annotations if the proof obligations that arise from this translation process and inspection of the model can be proved. So although the perspective is slightly different, the correctness properties still concern the relation between implementation and model.

2.4.1 Refinement

Refinement is the technique that synthesizes a program from a specification step by step, such that each step increases the degree of precision with respect to the initial specification. Each additional step represents an implementation choice, such as the choice of algorithm for implementing a given function, or the choice of a concrete datatype to implement an abstract type (say the implementation of a set as a linked list) or even the weakening of a precondition of an operation.

Individual refinement steps must be proved correct, i.e. the effect of the concrete specification must not contradict the effect of the abstract/refined specification, in order for the final program to enjoy the same properties as the original specification. Each step thus generates a number of *refinement proof obligations* that must be discharged. The good news is that the correctness of each individual step is in principle much easier to establish than the overall correctness.

This is the technique followed by approaches like Z, VDM, and B. This very special ability to link a high level view to the resulting code via a chain of design choices and proofs is particularly suitable (and has been used) in the context of a vertical application of the Balzer life cycle. Refinement is a very popular and successful example of application of formal methods in industry. It is well supported in terms of tools, and what is more it provides the simplest way to realize Balzer's vision of the software development process. As mentioned before, software developed through a chain of formally verified refinement steps is sometimes referred to as *correct-by-construction*.

In order to illustrate this concept, we consider in Fig. 2.8 the very classic example of a B machine that introduces the notion of a finite subset of the natural numbers with cardinal less than or equal to a given parameter (*maxelem*). Using the set-theoretic foundations of the method, the internal state includes the set in question,

Fig. 2.8 Finite sets of natural numbers in B

```
MACHINE $Set1(maxelem)$
CONSTRAINTS $maxelem \in \mathbb{N}_1$
VARIABLES $set$
INVARIANT
  $set \subseteq \mathbb{N}$
  $\land\ card(set) \leq maxelem$
INITIALISATION $set := \emptyset$
OPERATIONS
$add(n) \mathrel{\widehat{=}}$
   PRE $n \in \mathbb{N} - set \ \land\ card(set) < maxelem$
   THEN $set := set \cup \{n\}$
   END
END;
```

Fig. 2.9 Refinement of finite sets in B

```
REFINEMENT $Set2$
REFINES $Set1$
VARIABLES $tab, index$
INVARIANT
  $index \in 0..maxelem$
  $\land\ tab \in 1..index \rightarrowtail \mathbb{N}$
  $\land\ ran(tab) = set$
INITIALISATION $index, tab := 0, \emptyset$
OPERATIONS
$add(n) \mathrel{\widehat{=}}$
   PRE $n \in \mathbb{N} - ran(tab) \land index < maxelem$
   THEN $index := index + 1 || tab(index + 1) := n$
   END
END;
```

that is initialised to the empty set. The machine has an invariant (a property that must hold before and after the execution of any operation) stating that the variable *set* is indeed a subset of the natural numbers, and has a cardinality that is less than or equal to the parameter. The only provided operation adds a new element n to the set, provided that n is not already an element, and that the set can in fact be augmented (in terms of its cardinality).

This machine can be refined by the following design choice: we opt for a scalar representation of the recorded values instead of a set. The resulting machine *Set*2 is shown in Fig. 2.9. Here, the injective function *tab* assumes the role of the variable *set*. The invariant of the refined machine now states, and this is an important point, the relation between *set* and *tab*. The variable *index* records the cardinal of the set and indeed is used as the index of the last added element, if one sees *tab* as an array. The proof obligations generated by any implementation of B will establish, when discharged, that the behaviour of machine *Set*2 is indistinguishable from the behaviour of machine *Set*1.

2.4.2 Extraction

The *Calculus of Inductive Constructions* (CIC), that stands at the foundation of the Coq system, is an extension of the typed λ-calculus. By the Curry-Howard isomorphism [11, 92], this calculus is also a *constructive* minimal higher-order logic, with the consequence that every logical proposition can also be seen as a specification, and every proof expressible in the CIC can be seen as a program that obeys a specification. Indeed, this strong paradigm allows for the *extraction* of the computational contents of the proof (the program it contains) in a given programming language. As an example, let ϕ be the following theorem

$$\forall x, y \in \mathbb{N}.\ \exists q, r \in \mathbb{N}.\ (y = (q \times x + r)\ \wedge\ 0 \leq r < x)$$

A proof of ϕ is, in this context, a function that takes two integers x and y and computes *the* pair (q, r) that testifies the validity of $(y = (q \times x + r)\ \wedge\ 0 \leq r < x)$. This pair is of course $(y \div x,\ y \,\mathsf{mod}\, x)$. In general the proof of a theorem of the form $\forall x.\exists y.\ (R\ x\ y)$, if it exists, is a function that maps x into a value y such that $(R\ x\ y)$ holds.

Coq is capable of performing extractions into the untyped functional language Scheme or into the typed functional languages Haskell and OCaml. The extraction of the program corresponding to the proof t of a property ϕ is done automatically and in a single step.

2.4.3 Execution

Specifying with the help of (declarative) programming languages is a *de facto* method for obtaining implementations. Logic-based languages like Prolog and functional languages like Scheme, ACL2, Haskell, SML, and OCaml can all be seen as offering scope for both specification and implementation in the same language.

The simple problem faced by these languages is their relevance in terms of industry-strength applications. It is arguable whether, say, Prolog can in fact be regarded by industry as a viable implementation language.

2.5 Specifying and Transforming

When discussing model checking we mentioned that it is sometimes desirable to transform a specification in order to make it amenable to manipulation by verification methods. This is true also outside the context of model checking: it is often useful, and even necessary, to construct variations of a specification in order to hide details that obscure the verification, or conversely to enrich the specification with an extra level of detail to take into account new behaviour. The main and general foundation for such transformations is the theory of *abstract interpretation* [44], which provides a framework for defining sound approximations.

In general, the possibility of constructing variations of the initial model allows for the *modularity* of formal verification. Behaviours may be decomposed in a number of different *views*, each of which concerns a well-defined part of the global model. In principle, studying each individual aspect of the model is easier than studying the global behaviour, and under certain conditions it may be equivalent in terms of the results obtained.

Surprisingly there is no popular and well established tool support for such transformations; in fact, even the definition of "tool-supported model transformation" is still an open issue. This is due in part to the fact that the abstraction problems, in their general form, are easily undecidable, and transformations are deeply tied to the properties to be proved and the modelling language used. Designing appropriate and sound approximations is not an easy task, and is usually undertaken in an ad-hoc fashion.

The JaKarTa toolset [14] for reasoning about JavaCard specifications is an early example of this approach. It provides a (rule-based) language and mechanisms for the specification of ad-hoc model transformations, based on their effects on the data structures of the model under analysis. For instance, when modelling the operational semantics of a virtual machine one may want to focus on the typing policy. In this case, the manipulated values by themselves are not relevant for the analysis. Given the specification of the effect of the transformation to be performed on the data manipulated by the virtual machine (forgetting the values, keeping the types), JaKarTa is able to automatically pass this transformation on to the operations of the machine, and to produce proof obligations (in Coq) that ensure the soundness of the transformation.

2.6 Conclusions

Clarke et al. [36] claim that no single tool seems to solve in a completely satisfying manner the central problem of formal methods. While proof assistants are solid tools for formal verification, they are hard to use and lack automation. Some tools propose a vertical approach, complete from specifications to programs, but they miss proof-support functionality. Hardly any proof tool offers support for transformation of specifications. Some efforts have been made to integrate functionally vertical specification methods with proof capabilities, such as B. Also, model checking modules have been proposed for both PVS and Coq [98], but the results cannot be considered to be entirely satisfying.

The obvious conclusion to draw from these observations is that in the current state of development, resorting to a *combination of methods and tools* is an appealing alternative. Code-oriented platforms like Key or Frama-C propose rich environments that integrate several tools. For instance, Frama-C allows the integration and interaction of several static analyses (slicing, value, interval, dead-code analysis, etc.) with deductive methods. The deductive facility itself allows for formal verification using several proof tools like Coq or SMT-solvers.

We finish the chapter with a discussion of the applicability of formal methods in industry.

2.6.1 Are Formal Methods Tools Ready for Industry?

After the discussion in the previous chapter and the overview of the present chapter we may now attempt to answer this question. We saw how both the horizontal and vertical application of the Balzer life cycle are addressed by formal methods tools—recall for instance the use of the correct by construction paradigm, or tool-supported approaches like SCADE or the B Method. In these last few years there has been a dramatic increase in the maturity of several tools, thus one can reasonably expect an even better context for formal methods in the coming years.

Nevertheless, even if it is now more reasonable, the use and application of formal methods still requires a solid knowledge of basic mathematics, and can still be considered to be challenging to the average software engineer (if not simply frightening or a waste of time). The reasons for this are multiple and complex, and include for instance

- the lack of adequate mathematical training;
- a software development context that is under the strong commitments of a reduced *time to market*; or
- the simple absence of proper planning, due to the development process being subject to constantly changing requirements.

The first argument is a fairly difficult foundational issue, but the variable geometry of the development process can at least in part be addressed by formal methods instruments; think for instance of the contract-based approach to software development. Nevertheless, there is undoubtedly a question of image at the heart of the problem. Any training or dissemination activity is a valuable contribution to the improvement of visibility, understanding, and acceptance of formal methods. This is especially important to demonstrate that, as we have already noted, formal methods are now sufficiently mature and usable.

The adequate use of most formal methods tools in an industrial context requires that the development team contains only one specialist in the field.[4] We refer the reader to [70] for a recent remark in this direction. A notable exception is the use of heavyweight formal methods (involving proof assistants, for instance) that clearly require specialised mathematical skills, but whose application is only justified in very specific contexts.

Nevertheless, while formal methods in general still have to improve their ability to cope with modularity and scalability, we have been seeing with increasing frequency the announcement of several *tours de force* in formal verification[5] which strengthen our belief that formal methods now possess all the arguments to change the state of affairs. As stated by Jim Woodcock in the context of the software verification grand challenge,

[4] In the same way that it takes only a single Linux guru in a team to disseminate and properly use this operating system.

[5] Consider for instance the published results on the formal verification of compilers [76], operating systems [71], avionic control systems [23] or cryptographic software [15], among many others.

> 1000000 of verified lines of code: you can't say any more it can't be done! Here, we've done it!

2.6.2 *Is Industry Ready to Use Formal Methods?*

An important aspect when considering the use of formal methods is that they are not a mere product. Using these methods is not like installing and applying an antivirus. As stated by J.-R. Abrial in several tutorials and documents about the B Method, adopting formal methods in a software company is more a strategical and methodological issue than a technical one. We do not believe or advocate the widespread use of these methods in the software industry in general; their application should instead be considered when reliability, safety or security are a concern. Conscientious industrial applications of formal methods have already been conducted successfully in key areas, that have become flagship application areas.

Nevertheless, every software company has favoured and adopted some particular development process, and is unlikely to renounce it in favour of a completely new development process based on the use of formal methods. In order to adopt these methods, software companies have to reshape and adapt their in-house software design *savoir-faire*. This brings us again to the arguments stated in the previous chapter, and in particular to the Balzer life cycle.

Formal specification and verification are not easy or cheap, but the real cost has to be considered in the long term. One the other hand, their conclusions have to be taken with care: formal methods can only be used to specify or prove what was carefully stated beforehand, and cannot be used to reason about what was not. Formally specifying and verifying a whole system is then unlikely to be feasible or even reasonable. The advisable practice is then to determine the important (or critical) parts of the system do be designed and validated, and to apply formal methods on these parts.

2.7 To Learn More

Formal methods are the subject of numerous books, surveys and technical overviews. Many of them have already been cited in this chapter. We highlight here some general popular references. The most widely cited references [28–30, 60] report on the use of formal methods in the general context of software engineering. More technical surveys can be found in [36, 89] or in the more recent [64], dedicated to software verification. The latter special issue includes the already cited overview [99] that covers an important aspect barely touched in this chapter: the practice and industrial use of formal methods.

Several specialised books are also dedicated to formal methods, for instance [82] provides a nice introduction to the subject. [19] complements the previous reference by giving an overview of model checking tools.

References

1. Abdulla, P.A., Deneux, J.: Designing safe, reliable systems using scade. In: Proc. ISoLA 2004 (2004)
2. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
3. Abrial, J.-R.: Modeling in Event-B System and Software Engineering. Cambridge University Press, Cambridge (2010)
4. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Softw. Syst. Model. **4**, 32–54 (2005)
5. Alur, R., Dill, D.: Automata-theoretic verification of real-time systems. In: Formal Methods for RealTime Computing. Trends in Software Series, pp. 55–82. Wiley, New York (1996)
6. Alur, R., Dill, D.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
7. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
8. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
9. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
10. Barendregt, H.P.: The Lambda Calculus, its Syntax and Semantics. Studies in Logic and the Foundations of Mathematics, vol. 103. North-Holland, Amsterdam (1984)
11. Barendregt, H.P.: Lambda calculi with types. In: Abramsky, S., Gabbay, D., Maibaum, T. (eds.) Handbook of Logic in Computer Science, vol. 2, pp. 117–310. Oxford University Press, New York (1992)
12. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: CASSIS: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, vol. 3362, pp. 49–69. Springer, Berlin (2004)
13. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07). Lecture Notes in Computer Science, vol. 4590, pp. 298–302. Springer, Berlin (2007)
14. Barthe, G., Courtieu, P., Dufay, G., de Sousa, S.M.: Tool-assisted specification and verification of typed low-level languages. J. Autom. Reason. **35**(4), 295–354 (2005)
15. Barthe, G., Grégoire, B., Béguelin, S.Z.: Formal certification of code-based cryptographic proofs. In: Shao, Z., Pierce, B.C. (eds.) POPL, pp. 90–101. ACM, New York (2009)
16. Baudin, P., Fillitre, J.-C., March, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. Preliminary Design (version 1.4). From the Frama-C website, http://frama-c.com (2010)
17. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Berlin (2004)
18. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages 12 years later. Proc. IEEE **91**(1), 64–83 (2003)
19. Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P.: Systems and Software Verification. Model-Checking Techniques and Tools. Springer, Berlin (2001)
20. van den Berg, J., Jacobs, B.: The LOOP compiler for Java and JML. In: Margaria, T., Yi, W. (eds.) Proceedings of TACAS'01. Lecture Notes in Computer Science, vol. 2031, pp. 299–312. Springer, Berlin (2001)
21. Bezem, M., Klop, J.W., de Vrijer, R. (eds.): Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge (2002)
22. Bidoit, M., Mosses, P.D.: CASL User Manual. LNCS (IFIP Series), vol. 2900. Springer, Berlin (2004). With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki
23. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. CoRR, abs/cs/0701193 (2007)

24. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language Lotos. Comput. Netw. ISDN Syst. **14**(1), 25–59 (1987)
25. Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E., Ringeissen, C.: An overview of ELAN. In: Kirchner, C., Kirchner, H. (eds.) Proceedings of the International Workshop on Rewriting Logic and its Applications. Electronic Notes in Theoretical Computer Science, vol. 15. Pont-à-Mousson, France, September 1998. Elsevier, Amsterdam (1998)
26. Bouhoula, A., Kounalis, E., Rusinowitch, M.: SPIKE, an automatic theorem prover. In: Voronkov, A. (ed.) Proceedings of the International Conference on Logic Programming and Automated Reasoning (LPAR'92). Lecture Notes in Artificial Intelligence, vol. 624, pp. 460–462. Springer, Berlin (1992)
27. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda—a functional language with dependent types. In: TPHOLs '09: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, pp. 73–78. Springer, Berlin (2009)
28. Bowen, J.P., Hinchey, M.G.: Seven more myths of formal methods. IEEE Softw. **12**(4), 34–41 (1995)
29. Bowen, J.P., Hinchey, M.G.: Ten commandments of formal methods. Computer **28**(4), 56–63 (1995)
30. Bowen, J.P., Hinchey, M.G.: Ten commandments of formal methods … ten years later. Computer **39**(1), 40–48 (2006)
31. Bowen, J.P., Stavridou, V.: Safety-critical systems, formal methods and standards. IEE/BCS Softw. Eng. J. **8**(4), 189–209 (1993)
32. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: A model-checking tool for real-time systems (1998)
33. Burstall, R.M., Goguen, J.A.: An informal introduction to specification using CLEAR. In: Boyer, R.S., Moore, J.S. (eds.) The Correctness Problem in Computer Science, pp. 185–213. Academic Press, New York (1981)
34. Carré, B., Garnsworthy, J.: Spark—an annotated Ada subset for safety-critical programming. In: TRI-Ada '90: Proceedings of the Conference on TRI-ADA '90, pp. 392–402. ACM, New York (1990)
35. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: Lustre: A declarative language for real-time programming. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 178–188. ACM, New York (1987)
36. Clarke, E.M., Wing, M.J.: Formal methods: State of the art and future directions. ACM Comput. Surv. **28**(4), 626–643 (1996)
37. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
38. Cockett, R., Fukushima, T.: About Charity. Technical Report 92/480/18, University of Calgary (June 1992)
39. CoFI (The Common Framework Initiative): CASL Reference Manual. LNCS (IFIP Series), vol. 2960. Springer, Berlin (2004)
40. Colin, S., Petit, D., Mariano, G., Poirriez, V.: BRILLANT: An open source platform for B. In: Workshop on Tool Building in Formal Methods (held in conjunction with ABZ2010), February 2010
41. Colin, S., Petit, D., Poirriez, V., Rocheteau, J., Marcano, R., Mariano, G.: BRILLANT: An open source and XML-based platform for rigourous software development. In: SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods, Washington, DC, USA, 2005, pp. 373–382. IEEE Computer Society, Los Alamitos (2005)
42. Conchon, S., Contejean, E., Kanig, J.: Ergo: A theorem prover for polymorphic first-order logic modulo theories (2006)
43. Correnson, L., Cuoq, P., Puccetti, A., Signoles, J.: Frama-C user manual. From the Frama-C website, http://frama-c.com (2010)
44. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs. In: Proceedings of the 4th ACM Symposium on Principles of Programming Languages, pp. 238–252. ACM, New York (1977)

45. Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: VCC: Contract-based modular verification of concurrent C
46. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2008)
47. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
48. Dutertre, B., De Moura, L.: The Yices SMT solver. Technical report, SRI (2006)
49. Dwyer, M., Hatcliff, J., Joehanes, R., Laubach, S., Pasareanu, C., Visser, R.W., Zheng, H.: Tool-supported program abstraction for finite-state verification. In: Proceedings of ICSE'01 (2001)
50. Dybvig, R.K.: The Scheme Programming Language: ANSI Scheme, 2nd edn. Prentice-Hall International, Upper Saddle River (1996)
51. Ehrig, H., Fey, W., Hansen, H.: ACT ONE: An algebraic specification language with two levels of semantics. Technical Report 83–03, Technical University of Berlin, Fachbereich Informatik (1983)
52. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: International Symposium on FME 2001: Formal Methods for Increasing Software Productivity. Lecture Notes in Computer Science, vol. 2021, pp. 500–517. Springer, Berlin (2001)
53. Formal methods ressources. http://www.afm.sbu.ac.uk/
54. Futatsugi, K., Goguen, J., Jouannaud, J.-P., Meseguer, J.: Principles of OBJ-2. In: Reid, B. (ed.) Proceedings 12th ACM Symp. on Principles of Programming Languages, pp. 52–66. Association for Computing Machinery, New York (1985)
55. Garland, S.J., Guttag, J.V., Horning, J.: An Overview of Larch. Lecture Notes in Computer Science, vol. 693, pp. 329–348. Springer, Berlin (1993)
56. George, C., Haxthausen, A.E., Hughes, S., Milne, R., Prehn, S., Pedersen, J.S.: The Raise Development Method. Prentice-Hall International, London (1995)
57. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic. Cambridge University Press, Cambridge (1993)
58. Gurevich, Y.: Evolving algebras 1993: Lipari guide. In: Börger, E. (ed.) Specification and Validation Methods, pp. 9–36. Oxford University Press, Oxford (1995)
59. Halbwachs, N.: Synchronous Programming of Reactive Systems. Kluwer Academic, Norwell (1993)
60. Hall, A.: Seven myths of formal methods. IEEE Softw. **7**(5), 11–19 (1990)
61. Hartel, P.H., Moreau, L.: Formalizing the safety of Java, the Java virtual machine, and Java card. ACM Comput. Surv. **33**(4), 517–558 (2001)
62. Henzinger, T.A.: The theory of hybrid automata. In: LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science, Washington, DC, USA, 1996, p. 278. IEEE Computer Society, Los Alamitos (1996)
63. Henzinger, T.A., Ho, P.-H., Wong-toi, H.: Hytech: A model checker for hybrid systems. Softw. Tools Technol. Transf. **1**, 460–463 (1997)
64. Hoare, C.A.R., Misra, J.: Preface to special issue on software verification. ACM Comput. Surv. **41**(4), 1–3 (2009)
65. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
66. JML Specification Language. http://www.jmlspecs.org
67. Jones, C.B.: Software Development. A Rigorous Approach. Prentice-Hall International, Englewood Cliffs (1980)
68. Juellig, R., Srinivas, Y., Liu, J.: SPECWARE: An advanced environment for the formal development of complex software systems. In: Proceedings of AMAST'96. Lecture Notes in Computer Science, vol. 1101, pp. 551–554. Springer, Berlin (1996)
69. Kaufmann, M., Strother Moore, J.: ACL2: An industrial strength version of Nqthm. COMPASS—Proceedings of the Annual Conference on Computer Assurance, pp. 23–34 (1996). IEEE catalog number 96CH35960
70. Klein, G.: Correct os kernel? proof? done! USENIX ;login: **34**(6), 28–34 (2009)

71. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: sel4: Formal verification of an os kernel. In: Matthews, J.N., Anderson, T.E. (eds.) SOSP, pp. 207–220. ACM, New York (2009)
72. Klop, J.W.: Term-rewriting systems. In: Abramsky, S., Gabbay, D., Maibaum, T. (eds.) Handbook of Logic in Computer Science, vol. 2, pp. 1–116. Oxford Science Publications, New York (1992)
73. Krakatoa. http://www.lri.fr/marche/krakatoa/
74. Krauss, K.G.: Petri Nets Applied to the Formal Verification of Parallel and Communicating Processes. Lehigh University, Dissertation, Bethlehem, PA (1987)
75. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system, release 3.06 (2002). http://caml.inria.fr
76. Leroy, Xavier: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
77. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transf. (STTT) **10**(2), 185–203 (2008)
78. Mazzeo, A., Mazzocca, N., Russo, S., Savy, C., Vittorini, V.: Formal specification of concurrent systems: a structured approach. Comput. J. **41**(3), 145–162 (1998)
79. The Coq development team. The Coq proof assistant reference manual. LogiCal Project (2008). Version 8.2
80. Meyer, B.: Eiffel: The Language. Prentice Hall, Hemel Hempstead (1992)
81. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). MIT Press, Cambridge (1997)
82. Monin, J.F.: Understanding Formal Methods. Springer, New York (2001)
83. Paulson, L.: Isabelle: A Generic Theorem Prover. Lecture Notes in Computer Science, vol. 828. Springer, Berlin (1994)
84. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer, Heidelberg (2010)
85. Platzer, A., Quesel, J.-D.: KeYmaera: A hybrid theorem prover for hybrid systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR. LNCS, vol. 5195, pp. 171–178. Springer, Berlin (2008)
86. Reisig, W.: Petri nets and algebraic specifications. Theor. Comput. Sci. **80**, 1–34 (1991)
87. Requet, A.: An overview of Atelier B 4.0. In: Proceedings of the Conference The B Formal Method: From Research to Teaching'2008, Nantes (June 2008)
88. Rushby, J.: Formal methods and their role in the certification of critical systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International, Menlo Park, CA (March 1995)
89. Rushby, J.: Formal specification and verification for critical systems: Tools, achievements, and prospects. In: Suri, N., Walter, C.J., Hugue, M.M. (eds.) Advances in Ultra-Dependable Distributed Systems, pp. 282–296. IEEE Computer Society, Los Alamitos (1995)
90. Sannella, D.: A survey of formal software development methods. Technical Report ECS-LFCS-88-56, University of Edinburgh (July 1988)
91. Shankar, N., Owre, S., Rushby, J.M.: The PVS Proof Checker: A Reference Manual. Computer Science Laboratory, SRI International (February 1993)
92. Sørensen, M.H., Urzyczyn, P.: Lectures on the Curry-Howard Isomorphism. Studies in Logic and the Foundations of Mathematics, vol. 149. Elsevier, Amsterdam (2006)
93. Spivey, J.: An introduction to Z and formal specification. IEEE Softw. Eng. J. **4**(1), 40–50 (1989)
94. Stärk, R., Schmid, J., Börger, E.: Java and the Java Virtual Machine—Definition, Verification, Validation. Springer, Berlin (2001)
95. Sterling, L., Shapiro, E.: The Art of Prolog, 2nd edn. MIT Press, Cambridge (1994)
96. Thompson, S.: Haskell: The Craft of Functional Programming. Int. Comupt. Sci. Pearson Edn (1999)

97. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Symposium on Logic in Computer Science (LICS'86), pp. 332–345. IEEE Computer Society Press, Los Alamitos (1986)
98. Verma, K.N., Goubault-Larrecq, J.: Reflecting BDDs in Coq. Technical Report RR3859, INRIA projet Coq (January 2000)
99. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: Practice and experience. ACM Comput. Surv. **41**(4), 1–36 (2009)
100. Wu, W., Saeki, M.: Specifying software architectures based on colored petri nets. IEICE Trans. Inf. Syst. **E83-D**(4), 701–712 (2000)

Springer

http://www.springer.com/978-0-85729-017-5

Rigorous Software Development
An Introduction to Program Verification
Almeida, J.B.; Frade, M.J.; Pinto, J.S.; Melo de Sousa, S.
2011, XIII, 307 p. 52 illus., Softcover
ISBN: 978-0-85729-017-5