Chapter 9
Formal Methods

Mathematical reasoning is the foundation of most engineering disciplines. It would be unthinkable to construct a ship, bridge, or building without first making a mathematical model of the design and calculating that the design satisfies relevant requirements. Such models are used in the exploration of the design space, in quality assurance processes during construction, and in certification processes.

Computers and systems' counterpart to traditional engineering mathematics is called *formal methods*. The goal of this discipline is to capture the essence of computer code in formal mathematical theory and then draw conclusions about the code through mathematical analysis. In this chapter, we give an overview of the field, with the aim of understanding how formal methods can help us defend against untrusted equipment vendors.

9.1 Overview

The ultimate goal of a formal method is to prove the properties of the behaviour of a piece of executable code or electronic circuit. In our case, we are interested in proving that the piece of equipment at hand does what it is supposed to do and nothing else. Five basic notions are important to grasp on how such a proof is actually possible:

Specification If we want to prove a property of some code, we first have to state what this property is. Such statements are written in a formal language that is often an enriched version of some mathematical theory.

Implementation This is the object whose properties are to be proved. It can consist of code in some programming language or a specification of an entire integrated circuit or parts of one.

Semantic translation Given the implementation and specification of the wanted properties, one must synthesize the proof obligations, that is, determine which mathematical proofs need to be performed before we can conclude that the implementation fulfils the specification.

© The Author(s) 2018

O. Lysne, *The Huawei and Snowden Questions*, Simula SpringerBriefs on Computing 4, https://doi.org/10.1007/978-3-319-74950-1_9

Model In some cases, the detailed complexity of a complete implementation is such that a full formal proof is practically impossible. In that case, one can make a simplified model of what an implementation of the specification could be, prove that the model fulfils the specification, and finally generate the implementation automatically from the model. The idea is still to prove that the implementation fulfils the specification but this is now done by a detour through a model. In this case, it is the specification and the model that are passed on to the semantic translation.

Logic propositions The result of semantic translation is a set of propositions expressed in some mathematical logic. If all of the propositions are true in the given logic, we can conclude that the implementation fulfils the specification.

Theorem proving This is the process of proving the logic propositions above.

Figure 9.1 illustrates the structure of the full formal proof process of a piece of code. It is important to note that this figure intends to capture ways of finding a complete and watertight proof of correctness. Formal methods have been developed to help minimize the number of unintended programming faults. If that were our goal, there would be many variations of Fig. 9.1 that would be more enlightening. We are, however, not interested in improving the quality of programming practice. Rather, we are looking for ways to ensure that no malicious code has deliberately been inserted by the software makers. For that purpose, our description is appropriate.

In a perfect world, all the phases of a full formal proof would be performed automatically. If that were doable, we would only need to provide the specification and the correctness of the implementation could be checked by a machine. Unfortunately, as we saw in Chap. 5, there are very tight limits to what can be decided automatically. This means that all complete sets of formal methods that are able to prove interesting properties of a piece of code will require the heavy use of human expertise, in either semantic translation, theorem proofs, or both.

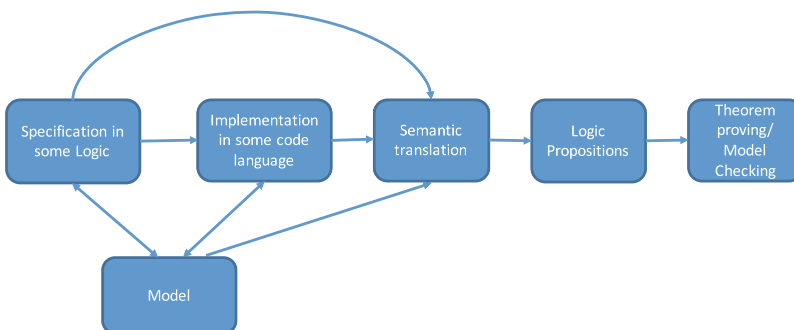

Fig. 9.1 These are the elements that constitute proof of a property using formal methods

9.2 Specification

An early attempt to build a logic for reasoning on the behaviour of a program was made via the notion of *assertions*. An assertion is a statement that is to be evaluated as **true** at a given place in a program [14]. Assume, for example, that we have the statement

```
x := y+1
```

Then, clearly, after the statement's execution, the assertion that `x>y` would be true.

This argument can be extended to the notions of *preconditions* and *postconditions*, where a precondition is an assertion that is supposed to be true before the start of a program and a postcondition is supposed to be true at the end of a program's execution. Preconditions and postconditions can be used to specify a program's behaviour. For instance, if we want to specify that a program is to sort an array `A` of `n` distinct integers in increasing order, we would require a postcondition that stated that, for all integers `i` between `1` and `n-1`, `A(i-1) < A(i)` should be true.[1] Furthermore, the postcondition would state that all integers present in `A` before the execution should also be present after the execution. The precondition should simply state that no integers are represented in `A` more than once.

Pre- and postconditions are important because they can be used to specify what a program should do without specifying how it should be done. Note that all of the different sorting algorithms one has ever learned would be valid according to the pre- and postconditions above, even though they differ vastly in their approach.

Pre- and postconditions were involved in an early approach to specify the outcome of a program's execution. It had several shortcomings, though, in that several important aspects of programming could not be easily captured. Some, such as input/output and user interaction, can be overcome by introducing sequences of inputs and outputs, respectively, as different variables into the machine. Others, such as the fact that, for instance, an operating system is assumed to run indefinitely and is therefore not supposed to have a final state, are more of a stretch. Several important notions emerged from these observations. One is the notion of an *invariant*, which is an assertion that is supposed to be true at a point of the code that can be reached infinitely many times. Another is the development of the *temporal logic of programs* [23], which allows one to express properties about time, such as *when A happens, then eventually B will happen*. Many other notions could be mentioned, but these two stand out as the most important and they form the basis of most of the others.

Significant amounts of training are required to master the rigour and precise formal semantics of provable specifications. This has been a barrier for the uptake of formal methods. Because of this, the specification of software is generally done in languages that are not based in firm formal semantics. An example of such a language is the *Unified Modeling Language* (UML), which is widely used for the modelling and

---

[1] We follow the informatics convention of starting the enumeration at zero. An array of `n` integers is thus enumerated from `0` to `n-1`.

specification of systems [25]. Bridging the gap between formal rigour and intuitively appealing specification methods has therefore been one of the key challenges to making fully formal specification more widespread [11].

9.3 Programming Languages

Historically, programming developed from bit-level programming to assembly programming, to structured programming with loops and procedure calls, and then to object-oriented programming. All of these programming paradigms are said to be *imperative*, in that they require the programmer to specify sequences of actions that change the state of memory. Imperative programming languages, including C, C++, and Java, form the basis of the overwhelming majority of software development today. They have therefore also been the subject of a great deal of research trying to capture programs' properties through formal methods.

There are, however, many different paradigms that have the strength of expression needed to form the basis for software development. In the formal methods community, some are of particular interest, since their mode of operation closely parallels existing mathematical theories. Two such programming paradigms that have gained industrial importance are *functional programming* and *logic programming*. Functional programming languages use the recursive definition of mathematical functions as their main building block. Execution of a program corresponds to evaluation of the mathematical function. In its pure form, a functional program has no explicit internal state; thus, there is no concept of state change. Most functional languages in industrial use can, however, be deemed to have a hybrid nature, where some notion of program state is supported. Examples of functional languages that have industrial use are Lisp [26], Scheme [27], and Erlang [2].

Logic programming languages are based on formal logic and a programs in these languages can therefore be viewed as logical clauses. Execution of a program consists of finding solutions to a logical clause, in the sense of finding instantiations of the variables for which the clause is evaluated as being true. The industrial impact of logic programming languages has been relatively limited, but Prolog [8] and Datalog [5] are worth mentioning.

The definition of integrated circuits was for a very long time quite distinct from that of programming. These days, the distinctions have diminished, in that the definition of circuits is mostly carried out in high-level languages such as those described above. Synthesis tools transform these high-level descriptions into descriptions in terms of logic gates (see Sect. 4.2).

9.4 Hybrid Programming and Specification Languages

In Sect. 9.2, we concentrated on the pure specification of programs, in the sense that we only considered methods that did not say anything about how the execution should be carried out. Then, in Sect. 9.3, we turned our attention to programming languages that specify the execution in itself. There is, however, a class of languages that are somewhere in the middle, in that they are mostly used for specification purposes but where the specifications are, to some extent, executable. For efficiency reasons, these languages are most often used for modelling only. Programs or specifications in these languages are translated, manually or automatically, into other languages before they are executed.

The earliest example of this involves state machines [12]. A state machine consists of a (usually finite) set of states that a system can be in, together with rules for what events make the system move from one state to the other. Recall that, in Chap. 5, we learned that a Turing machine is believed to be able to simulate any computing device. When we study a Turing machine, we see that it can easily be represented by a set of states and some rules for transitioning between the states. Therefore, most reasonable definitions of what a state machine is will be Turing equivalent. Even though this means that a state machine can be seen as a complete programming language, state machines are rarely used to model a computer system in full detail. A computer with, for example, 1 GB of memory will have $2^{10^{12}}$ potential states, which is far beyond what can be used in a formal method proof chain. The states are therefore usually abstracted and aggregated into logical *extended* states.

One particular string of hybrid languages came out of the observation that the distinction between specification and programming caused problems when the systems became parallel and distributed. Whereas a sequential program will execute the same way for the same input every time, a parallel program can behave differently for the same input, depending to how race conditions play out. This issue made it clear that, when proving the properties of a parallel program, it would be highly beneficial if the specification itself contained notions of the different parallel processes in the program, as well as how they were supposed to interact. This led to the development of *process algebras*, which, in the context of specification, have the commonalities that they describe the processes of a parallel program and that these processes communicate through explicit message passing rather than through the manipulation of shared variables. The first two developments that fall into the category of process algebras are communicating sequential processes (CSP) [15] and the calculus of communicating systems (CCS) [17]. These appeared more or less at the same time and were developed in parallel for a long time. Later, several variants of process algebras were introduced. The most acknowledged of these is the π-calculus [18] that still forms the basis for an active area of research.

A third strand worth mentioning consists of languages that specify programs as algebraic transformations. The elegance of this approach is that it unifies a mathematical theory of algebraic equivalence with the operational aspect provided by algebraic

simplifications [13]. This strand of research had its early days in the late 1970 s and is still vibrant today, through the executable specification language Maude [7].

9.5 Semantic Translation

When we have a program and a specification of what the program should do, the task of semantic translation is to extract from the two a set of *proof obligations*. These proof obligations are a set of mathematical expressions that, if proven to be true, imply that the program behaves according to the specification. Our entry point into this issue is *Hoare logic* [14], which is an axiomatic basis for reasoning about imperative programs.

The core of Hoare logic is the Hoare triple, denoted $\{P\}S\{Q\}$. In this triple, S is a program and P and Q are logic expressions on the state of the program. Intuitively, the triple means that if P is true before the execution of S, then Q will be true after the execution. An example of a valid triple is

$$\{x + 1 < y\} \;\; x := x + 1; \;\; \{x < y\}$$

Hoare logic devises deduction rules for reasoning on such triplets over constructs that occur in imperative languages, such as assignments, if statements, loops, and statement composition. Using these deduction rules, it is possible to establish a triple $\{P\}S'\{Q\}$ for programs of any complexity. Finally, instantiation of the consequence rule below relates this to the specified precondition and postcondition (see Sect. 9.2):

$$\frac{Precondition \Rightarrow P \quad \{P\}S\{Q\} \quad Q \Rightarrow Postcondition}{\{Precondition\}S\{Postcondition\}}$$

This rule states that, if a program is specified by a precondition and a postcondition and Hoare logic is used to prove the triple $\{P\}S\{Q\}$, then the resulting mathematical proof obligations are that the precondition implies that P holds and that Q implies that the postcondition holds.

Unfortunately, proving a Hoare triple by the deduction rules given by Hoare was soon shown to be highly nontrivial. Later developments therefore attempted to reformulate the deduction rules to form the basis of algorithmic support for building this part of the proof. This approach led to the notion of *weakest precondition*, introduced by Dijkstra [9], which reformulated the ideas of Hoare logic into a system where the formation of a weakest possible precondition from a given postcondition could partly be determined automatically. Later, the work of Hoare and Dijkstra was extended to parallel programs [16] and object-oriented programs, to mention just two [22].

As stated in the previous section, process algebra was borne out of the observation that the distinction between specification and implementation seen in Hoare logic creates difficulties when it comes to parallel and distributed systems. Since

a specification in process algebra is partially executable in its own right, the proof obligation for these formalisms is to establish equivalence between the execution of the specification and its implementation. Such equivalence is defined as capturing a situation where the two simulate each other. *Bisimulation* is thus a relation between two executables where, intuitively, an external observer is not able to identify which one is executing. Execution time is usually not deemed to be observable in these definitions. The reason is that improved execution time is the reason for replacing the specification with the implementation in the first place.

9.6 Logics

Several preexisting mathematical theories have been used as the fixed frame for proving the properties of programs. Initial approaches were based on some predicate logic – usually first-order logic – enriched with algebraic laws for integers or rational numbers. Others were developed specifically for the purpose. Two examples are temporal logic [23], which was specifically developed to capture the notion of time, which is important in infinite processes and real-time systems, and logics to handle the finite resources available in a physical computer [24].

9.7 Theorem Proving and Model Checking

After the relation between the specification and the program has been transformed into propositions in formal logic, these propositions need to be proven. The implication is that most processes for proving theorems are interactive. A great deal of research has gone into the development of interactive theorem provers and a few of the most important ones are the Prototype Verification System (PVS) [20], Isabelle [21], and Coq [3]. Going into the details of their different properties is outside the scope of this book, so we refer to Boldo et al. [4] for a good overview.

Ideally, one would hope that these theorem provers could work automatically but, unfortunately, all the relevant problems are NP-hard. Furthermore, a majority of them are super-exponential or even undecidable. This should come as no surprise, because we saw in Chap. 5 that whether a piece of code performs malicious actions is generally undecidable. This undecidability would have to show up in the case of formal proofs as well, either in semantic translation or in theorem proving.

The undecidability problem did limit the applicability of formal methods to prove the correctness of more than very small systems for a long time. The full complexity of the proof obligation for large systems simply became intractable. *Model checking* then came to the rescue and showed its value through the 1990s. The idea in model checking is to abstract a simplified model of the state space of the system and then check the entire state space or parts of it for execution traces that would be a counterexample to the specification.

Model checking has proven itself to be very useful for both finding bugs and proving the properties of protocols [6]. Still, for the problem we study in this book, model checking comes at a high price, because building a model of a system creates a level of indirection in the proof systems. Even if we can prove a model's properties and can generate the code directly from it, the generation of code is in itself a possible point of insertion of unwanted behaviour. This fact is discussed in detail in Chap. 4.

9.8 Proof-Carrying Code

We have seen that it is very hard to find a full formal proof for a finished piece of code. Semantic translation of the specification and the code is highly nontrivial and proving the logic propositions that come out of this translation is an NP-complete problem at best and most often undecidable. One of the main obstacles is that it is generally very hard to analyse code that one has not participated in writing. We return to this topic in Sect. 10.10.

Providing proof of correctness while one develops the code should, however, be easier. Clearly, when one programs, one has a mental understanding of why the code will work. Constructing proof of correctness while one programs should therefore amount to formalizing an understanding one already has. Furthermore, while finding a proof for a piece of code is very hard, checking the correctness of an existing proof is trivially simple. The idea behind proof-carrying code [19] is to exploit these two facts. The burden of providing proof is moved to the producer of the equipment and the buyer only has to check that the proofs hold water. Proof-carrying code was first described in application to software but the idea was later transferred to hardware [10]. The concept should therefore apply to the entire technology stack.

Although the idea of proof-carrying code holds great promise, it has not seen much practical use. The reasons for this are that, even though it is easier for a programmer to provide proof while programming, proof-carrying code still requires a great deal of effort and expertise. The general sentiment is that the extra cost that proof-carrying code would incur in a development project will seldom be covered by increased sales. Still, this remains one of the promising paths to follow when looking for solutions to the problem of trust in untrusted vendors.

9.9 Conclusion

Much effort has been expended for many decades in finding the foundations of programming and developing formal methods for reasoning about programs and integrated circuits. This effort has had a profound impact on the development of programming languages and language constructs. Furthermore, it has contributed notions such as *invariant* and *precondition* to the everyday language of most programmers. Finally, formal methods has been successfully applied to finding mistakes in specifications and rooting out bugs in computer code.

Unfortunately, hopes that formal methods will one day provide the full mathematical proof of the total correctness of an entire system, as described in Chap. 3, have dwindled. The scalability of the approaches has in no way kept up with the rapid increase in size and complexity of the systems that one needs to consider. There are some noteworthy successes of full formal proofs of software, but they are confined to select application areas and most often to critical subparts of the system itself. For example, in a survey on the engineering of security into distributed systems, Uzunov and his co-authors [28] stated that 'the application of formal methods is almost invariably localised to critical aspects of a system: no single formal technique is capable of formally verifying a complete complex system'. The picture is somewhat different for integrated circuits, where formal methods in many development projects have become a natural and integral part of the development process [1].

If we look at formal methods from our viewpoint of wanting to build trust in equipment from untrusted vendors, we could choose to conclude that there is no help to be obtained from this particular research area. As long as we cannot build complete formal proofs of an entire system, we cannot guarantee that inserted malicious behaviour will be detected. We should, however, not be discouraged. At this point in the book, it is clear that a guarantee cannot be given by any methodology. This means that we should look for approaches that increase the risk of the acts of a malicious vendor being detected. Clearly, if the risk of being detected is either high, unpredictable, or both, this would be strong deterrent against inserting hidden malicious functionality into a system.

While whether the application of formal methods to a system will result in a high probability of finding unwanted functionality is debatable, formal methods do have properties that make them unpredictable. Formal methods have shown their worth in finding very subtle bugs in computer code, bugs that were exceptionally challenging to detect just through testing or code review. This property is interesting for our problem. It partly means that techniques that can be used to hide functionality, such as code obfuscation, are of limited value in the face of formal methods. Using computer-aided means of formal reasoning on the correctness of computer code therefore has the potential to make the risk of being detected impossible to estimate.

The focus of the research in formal methods has been to provide formal verification, find bugs, or generate cases for testing. Finding deliberately inserted but possibly obfuscated functionality has not been addressed specifically. Still, in the quest for a solution to our problem, further research in this direction stands out as one of the most promising paths forward. It is up to the research community to rise to the challenge.

References

1. Alur, R., Henzinger, T.A., Vardi, M.Y.: Theory in practice for system design and verification. ACM Siglog News **2**(1), 46–51 (2015)
2. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang. Prentice Hall, Englewood Cliffs (1993)

3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Springer Science and Business Media, Berlin (2013)
4. Boldo, S., Lelay, C., Melquiond, G.: Formalization of real analysis: a survey of proof assistants and libraries. Math. Struct. Comput. Sci. 1–38 (2014)
5. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Trans. Knowl. Data Eng. **1**(1), 146–166 (1989)
6. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
7. Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., Talcott, C.: Maude manual (version 2.7) (2015)
8. Clocksin, W., Mellish, C.S.: Programming in Prolog. Springer Science and Business Media, Springer (2003)
9. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
10. Drzevitzky, S., Kastens, U., Platzner, M.: Proof-carrying hardware: towards runtime verification of reconfigurable modules. In: 2009 International Conference on Reconfigurable Computing and FPGAs, pp. 189–194. IEEE (2009)
11. Evans, A., France, R., Lano, K., Rumpe, B.: The UML as a formal modeling notation. The Unified Modeling Language. UML'98: Beyond the Notation, pp. 336–348. Springer, Berlin (1998)
12. Gill, A., et al.: Introduction to the Theory of Finite-State Machines. McGraw-Hill, Maidenheach (1962)
13. Guttag, J.V., Horning, J.J.: The algebraic specification of abstract data types. Acta informatica **10**(1), 27–52 (1978)
14. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
15. Hoare, C.A.R.: Communicating Sequential Processes. Springer, Berlin (1978)
16. Lamport, L.: The 'hoare logic' of concurrent programs. Acta Informatica **14**(1), 21–37 (1980)
17. Milner, R.: A Calculus of Communicating Systems. Springer, Berlin (1980)
18. Milner, R.: Communicating and Mobile Systems: The Pi Calculus. Cambridge university press, Cambridge (1999)
19. Necula, G., Lee, P.: Proof-carrying code. Technical report CMU-CS-96-165, School of Computer Science, Carnegie Mellon University (1996)
20. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. Automated Deduction—CADE-11, pp. 748–752. Springer, Berlin (1992)
21. Paulson, L.C.: Isabelle: A Generic Theorem Prover, vol. 828. Springer Science and Business Media, Berlin (1994)
22. Pierik, C., De Boer, F.S.: A syntax-directed hoare logic for object-oriented programming concepts. Formal Methods for Open Object-Based Distributed Systems, pp. 64–78. Springer, Berlin (2003)
23. Pnueli, A.: The temporal logic of programs. In: Proccedings of 18th Annual Symposium on Foundations of Computer Science, 1977, pp. 46–57. IEEE (1977)
24. Pym, D., Tofts, C.: A calculus and logic of resources and processes. Form. Asp. Comput. **18**(4), 495–517 (2006)
25. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual. The, Pearson Higher Education (2004)
26. Steele, G.: Common LISP: The Language. Elsevier, London (1990)
27. Steele Jr., G.L., Sussman, G.J.: The revised report on scheme: a dialect of lisp. Technical report, DTIC Document (1978)
28. Uzunov, A.V., Fernandez, E.B., Falkner, K.: Engineering security into distributed systems: a survey of methodologies. J. UCS **18**(20), 2920–3006 (2012)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.