Chapter 2
Overview of Formal Verification

Formal verification of a computing system entails a mathematical proof showing that the system satisfies its desired property or specification. To do this, we must use some mathematical structure to model the system of interest and derive its desired properties as theorems about the structure. The principal distinction between the different formal verification approaches stems from the choice of the mathematical formalism used in the reasoning process. In this chapter, we will survey some of the key formal verification techniques and understand their strengths and limitation.

Formal verification is a very active area of research, and numerous promising techniques and methodologies have been invented in recent years for streamlining and scaling its application on large-scale computing systems. Given the vastness of the subject, it is impossible to do justice to any of the techniques involved in this short review; we only focus on the aspects of the different methods that are directly relevant to the techniques discussed in the subsequent chapters. Section 2.4 lists some of the books and papers that provide a more comprehensive treatment of the materials discussed here.

2.1 Theorem Proving

Theorem proving is one of the key approaches to formal verification. A theorem prover is a computer program for constructing and checking derivations in a formal logic. A formal logic comprises the following three components.

1. A *formal language* to express formulas
2. A collection of formulas called *axioms*
3. A collection of *inference rules* for deriving new formulas from existing ones

To use formal logic for reasoning about a mathematical artifact, one considers a logic such that the formulas representing the axioms are *valid*, *i.e.*, can be interpreted as self-evident truths about the artifact, and the inference rules are validity preserving. Consequently, the formulas derived by applying a sequence of inference rules from the axioms must also be valid. Such formulas are referred to as *formal theorems* (or simply *theorems*). A sequence σ of formulas such that each member

S. Ray, *Scalable Techniques for Formal Verification*, DOI 10.1007/978-1-4419-5998-0_2,

of σ is either an axiom or obtained from a collection of previous members of σ by applying an inference rule is called a *derivation* or *deduction*. In the context of theorem proving, "verifying a formula" means showing that there is a deduction of the formula in the logic of the theorem prover.

There are many theorem provers in active use today. The popular ones include ACL2 [122], Coq [70], HOL [88], Isabelle [196], NuPrl [59], and PVS [194]. The underlying logics of theorem provers vary considerably. There are theorem provers for set theory, constructive type theory, first-order logic, higher order logic, etc. There is also substantial diversity in the amount of automation provided by the different theorem provers; some are essentially proof checkers while others can do a considerable amount of unassisted reasoning. In the next chapter, we will see some details of one theorem prover, ACL2. In spite of diversity of features, however, a common aspect of all theorem provers is that they support logics that are rich and expressive. The expressiveness allows one to use the technology for mechanically checking deep and interesting theorems in a variety of mathematical domains. For instance, Gödel's incompleteness theorem and Gauss' Law of Quadratic Reciprocity have been mechanically verified in the Nqthm theorem prover [220, 234], and Ramsey's Theorem and Cantor's Theorem have been proven in Isabelle [197, 198].

However, the expressive power comes at a cost. Foundational research during the first half of the last century showed that any sufficiently expressive logic that is consistent must be undecidable [81, 250]. That means that there is no automatic procedure (or algorithm) that, given a formula, can always determine if there exists a derivation of the formula in the logic. Thus, the successful use of theorem proving for deriving nontrivial theorems typically involves interaction with a trained user. Nevertheless, theorem provers are still invaluable tools for formal verification. Some of the things that a theorem prover can do are the following:

- A theorem prover can mechanically *check* a proof, that is, verify that a sequence of formulas does correspond to a legal derivation in the proof system. This is usually an easy matter; the theorem prover needs to merely check that each derivation in the sequence is either an axiom or follows from the previous ones by a legal application of the rules of inference.
- A theorem prover can assist the user in the construction of a proof. Most theorem provers in practice implement several heuristics for proof search. Such heuristics include generalizing the formula for applying mathematical induction, using appropriate instantiation of previously proven theorems, judicious application of term rewriting, and so on.
- If the formula to be proved as a theorem is expressible in some well-identified decidable fragment of the logic, then the theorem prover can invoke a decision procedure for the fragment to determine whether it is a theorem. Most theorem provers integrate decision procedures for several logic fragments. For instance, PVS has procedures for deciding formulas in Presburger arithmetic, and formulas over finite lists [187, 237], and ACL2 has procedures for deciding linear inequalities over rationals [24, 111].

In spite of success in approaches designed to automate the search of proofs, it must be admitted that undecidability poses an insurmountable barrier in developing automation. In practice, the construction of a nontrivial derivation via theorem proving is a creative process requiring substantial interaction between the theorem prover and a trained user. The user provides an outline of the derivation and the theorem prover is responsible for determining if the outline can indeed be turned into a formal proof. At what level of detail the user needs to give the outline depends on the complexity of the derivation and the implementation and architecture of the theorem prover. In general, when the theorem prover fails to deduce the derivation of some formula given a proof outline, the user needs to refine the outline, possibly by proving some intermediate lemmas. In a certain sense, interacting with a theorem prover for verifying a complex formula "feels" like constructing a *very* careful mathematical argument for its correctness, with the prover checking the correctness of the low level details of the argument.

How do we apply theorem proving to prove the correctness of a computing system? The short answer is that the basic approach is exactly the same as what we would do to prove the correctness of any other mathematical statement in a formal logic. We determine a formula that expresses the correctness of the computing system in the logic of the theorem prover, and derive the formula as a theorem. Throughout this monograph we will see several examples of computing systems verified in this manner. However, the formulas we need to manipulate for reasoning about computing systems are extremely long. A formal specification of even a relatively simple system might involve formulas ranging over 100 pages. Doing formal proof involving formulas at such scale, even with mechanical assistance from a theorem prover, requires a nontrivial amount of discipline and attention to detail. Kaufmann and Moore [125] succinctly mention this issue as follows:

> Minor misjudgments that are tolerable in small proofs are blown out of proportions in big ones. Unnecessarily complicated function definitions or messy, hand-guided proofs are things that can be tolerated in small projects without endangering success; but in large projects, such things can doom the proof effort.

Given that we want to automate proofs of correctness of computing systems as much as possible, why should we apply theorem proving for this purpose? Why not simply formalize the desired properties of systems in a decidable logic and use a decision procedure to check if such properties are theorems? There are indeed several approaches to formal verification based on decidable logics and use of decision procedures; in the next section we will review one such representative approach, *model checking*. Nevertheless, theorem proving, undecidable as it is, has certain distinct advantages over decision procedures. In some cases, one needs an expressive logic simply to state the desired correctness properties of the system of interest, and theorem proving is the only technology that one can resort to for proving such properties. Even when the properties can be expressed in a decidable logic, *e.g.*, when one is reasoning about a finite-state system, theorem proving has the advantage of being both succinct and general. For instance, consider a system $\mathscr{S}_3$ of three processes executing some mutual exclusion protocol. As we will see in the next section, it is possible to state the property of mutual exclusion for the system in a decidable logic,

and indeed, model checking can be used to prove (or disprove) the property for the system. However, if we now implement the *same* protocol for a system $\mathscr{S}_4$ with four processes, the verification of $\mathscr{S}_3$ does not give us any assurance in the correctness of this new system, and we must reverify it. In general, we would probably like to verify the implementation for a system $\mathscr{S}_n$ of n processes. However, the mutual exclusion property for such a parameterized system $\mathscr{S}_n$ might not be expressible in a decidable logic. Some advances made in *Parameterized Model Checking* allow us to apply automated decision procedures to parameterized systems, but there is strong restriction on the kind of systems on which they are applicable [71]. Also, even when decision procedures are applicable in principle (*e.g.*, for the system $\mathscr{S}_{100}$ with 100 processes), they might be computationally intractable. On the other hand, one can easily and succinctly represent mutual exclusion for a parameterized system as a formula in an expressive logic, and use theorem proving to reason about the formula. While this will probably involve some human effort, the result is reusable for any system irrespective of the number of processes, and thereby solves a family of verification problems in one fell swoop.

There is one very practical reason for applying theorem proving in the verification of computing systems. Most theorem provers provide a substantial degree of control in the process of derivation of complex theorems.[1] This can be exploited by the user in different forms, *e.g.*, by proving key intermediate lemmas that assist the theorem prover in its proof search . By manually structuring and decomposing the verification problem, the user can guide the theorem prover into proofs about very complex systems. For instance, using ACL2, Brock and Hunt [29, 31] proved the correctness of an industrial DSP processor. The main theorem was proved by crafting a subtle generalization which was designed from the user understanding of the high-level concepts behind the workings of the system. This is a standard occurrence in practical verification. Paulson [200] shows examples of how verification can be streamlined if one can use an expressive logic to succinctly explain the high-level intuition behind the workings of a system. Throughout this monograph, we will see how user control is effectively used to simplify formal verification.

2.2 Temporal Logic and Model Checking

Theorem proving represents one approach to reasoning about computing systems. In using theorem proving, we trade automation in favor of expressive power of the underlying logic. Nevertheless, automation, when possible, is a key strength. A substantial amount of research in formal verification is aimed at designing decidable

[1] By theorem provers in this monograph, we normally mean the so-called *general-purpose* theorem provers. There are other more automated theorem provers, for instance Otter [162], which afford much less user control and substantially more automation. We do not discuss these theorem provers here since they are not easily applicable in the verification of computing systems.

formalisms in which interesting properties of computing systems can be formulated, so that one can check the truth and falsity of such formulas by decision procedures.

In addition to providing automation, the use of a decidable formalisms has one other significant benefit. When the system has a bug, that is, its desired properties are not theorems, a decision procedure can provide a counterexample. Such counterexamples can be invaluable in tracing the source of such errors in the system implementation.

In this section, we study in some detail one decidable formalism, namely *propositional Linear Temporal Logic* or pLTL (LTL for short). LTL has found several applications for specifying properties of reactive systems, that is, computing systems which are characterized by nonterminating computations. Learning about LTL and decision procedures for checking properties of computing systems specified as LTL formulas will give us some perspective of how decision procedures work and the key difference between them and deductive reasoning. Later, in Chap. 13, we will consider formalizing and embedding LTL into the logic of ACL2.

Before going any further, let us point out one major difference between theorem proving and decision procedures. When using theorem proving, we use derivations in a formal logic, which contains axioms and rules of inferences. If we prove that some formula expressible in the logic is a theorem, then we are asserting that for *any* mathematical artifact such that we can provide an *interpretation* of the formulas in the logic such that the axioms are true facts (or *valid*) about the artifact under the interpretation, and the inference rules are validity preserving, the interpretation of the theorem must also be valid. An interpretation under which the axioms are valid and inference rules are validity preserving is also called a *model* of the logic. Thus, the consequence of theorem proving is often succinctly described as: "Theorems are valid for all models." For our purpose, theorems are the desired properties of executions of computing systems of interest. Thus, a successful verification of a computing system using theorem proving is a proof of a theorem which can be interpreted to be a statement of the form: "All legal executions of the system satisfy a certain property." What is important in this is that the legal executions of the system of interest must be expressed as some set of formulas in the formal language of the logic. Formalisms for decision procedures, on the other hand, typically describe syntactic rules for specifying *properties* of executions of computing systems, but the executions of the systems themselves are not expressed inside the formalism. Rather, one defines the *semantics* of a formula, that is, a description of when a system can be said to satisfy the formula. For instance, we will talk below about the semantics of LTL. The semantics of LTL are given by properties of paths through a *Kripke Structure*. But the semantics themselves, of course, are not expressible in the language of LTL. By analogy from our discussion about theorem proving, we can say that the semantics provides an *interpretation* of the LTL formulas and the Kripke Structure is a model of the formula under that interpretation. In this sense, one often refers to the properties checked by decision procedures as saying that "they need to be valid under one interpretation." Thus, a temporal logic formula representing some property of a computing system, when verified, *does not* correspond to a formal theorem in the sense that a property verified by a theorem prover does. Indeed,

it has been claimed [53] that by restricting the interpretation to one model instead of all as in case of theorem proving, formalisms based on decision procedures succeed in proving interesting properties while still being amenable to algorithmic methods. Of course, it is possible to think of a logic such that the LTL formulas, Kripke Structures, and the semantics of LTL with respect to Kripke Structures can be represented as formulas inside the logic. We can then express the formula: "Kripke Structure κ satisfies formula ψ" as a theorem. Indeed, this is exactly what we will seek to do in Chap. 13 where we choose ACL2 to be the logic. However, in that case the claim of decidability is not for the logic in which such formulas are expressed but only on the *fragment* of formulas which designate temporal logic properties of Kripke Structures.

So what do (propositional) LTL formulas look like? The formulas are described in terms of a set $\mathscr{A}\mathscr{P}$ called the set of *atomic propositions*, standard Boolean operators "$\wedge$," "$\vee$," and "$\neg$," and four *temporal operators* "X," "G," "F," and "U," as specified by the following definition.

Definition 2.1 (LTL Syntax). The syntax of an LTL formula is recursively defined as follows.

- If $\psi \in \mathscr{A}\mathscr{P}$ then ψ is an LTL formula.
- If ψ_1 and ψ_2 are LTL formulas, then so are $\neg\psi_1$, $\psi_1 \vee \psi_2$, $\psi_1 \wedge \psi_2$, $\mathsf{X}\psi_1$, $\mathsf{G}\psi_1$, $\mathsf{F}\psi_1$, and $\psi_1 \mathsf{U} \psi_2$.

As mentioned above, the semantics of LTL is specified in terms of paths through a Kripke Structure, specified below.

Definition 2.2 (Kripke Structure). A Kripke Structure κ is a tuple $\langle S, R, L, s_0 \rangle$, where S is a set of *states*, R is a relation over $S \times S$ that is assumed to be left-total called the *transition relation*, $L : S \to 2^{\mathscr{A}\mathscr{P}}$ is called the *state labeling function*, and $s_0 \in S$ is called the *initial state*.[2]

It is easy to model computing systems in terms of Kripke Structures. A system usually comprises several *components* which can take up values in some range. To represent a system as a Kripke Structure, we take the set of states to be the set of all possible valuations of the different components of the system. A state is simply one such valuation. The system is called *finite state* if the set of states is finite. The *initial state* corresponds to the valuation of the components at the time of system initiation or reset. Two states s_1 and s_2 are related by the transition relation if it is possible for the system to transit from state s_1 to s_2 in one step. Notice that by specifying the transition as a *relation* we can talk about systems that make nondeterministic transitions from a state. The state labeling function (*label* for short) maps a state s to the atomic propositions that are true of s.

Given a Kripke Structure κ, we can talk about *paths* through κ. An *infinite path* (or simply, a path) π is an infinite sequence of states such that any two consecutive states in the sequence are related by the transition relation. Given any infinite

[2] The initial state s_0 is dropped in some expositions on model checking.

sequence π, we will refer to the ith element in π by π^i and the subsequence of π starting from π^i by π_i. Given an LTL formula ψ and a path π we then define what it means for π to satisfy ψ as follows.

Definition 2.3 (LTL Semantics). The notion of a path π satisfying an LTL formula ψ is defined recursively as follows:

1. If $\psi \in \mathscr{AP}$ then π satisfies ψ if and only if $\psi \in L(\pi^0)$.
2. π satisfies $\neg\psi$ if and only if π does not satisfy ψ.
3. π satisfies $\psi_1 \vee \psi_2$ if and only if π satisfies ψ_1 or π satisfies ψ_2.
4. π satisfies $\psi_1 \wedge \psi_2$ if and only if π satisfies ψ_1 and π satisfies ψ_2.
5. π satisfies $\mathsf{X}\psi$ if and only if π_1 satisfies ψ.
6. π satisfies $\mathsf{G}\psi$ if and only if for each i, π_i satisfies ψ.
7. π satisfies $\mathsf{F}\psi$ if and only if there exists some i such that π_i satisfies ψ.
8. π satisfies $\psi_1 \mathsf{U} \psi_2$ if and only if there exists some i such that (i) π_i satisfies ψ_2 and (ii) for each $j < i$, π_j satisfies ψ_1.

Not surprisingly, "F" is called the *eventuality* operator, "G" is called the *always* operator, "X" is called the *next time* operator, and "U" is called the *until* operator. A Kripke Structure $\kappa \doteq \langle S, R, L, s_0 \rangle$ will be said to satisfy formula ψ if and only if every path π of κ such that $\pi^0 \doteq s_0$ satisfies ψ.

Given this semantics, it is easy to specify interesting properties of reactive systems using LTL. Consider the mutual exclusion property for the three-process system we referred to in the previous section. Here a state of the system is the tuple of the *local states* of each of the component processes, together with the valuation of the shared variables, communication channels, etc. A local state of a process is given by the valuation of its local variables, such as program counter, local stack, etc. Let P_1, P_2, and P_3 be atomic propositions that specify that the program counter of processes 1, 2, and 3 are in the critical section, respectively. That is, in the Kripke Structure, the label maps a state s to P_i if and only if the program counter of process i is in the critical section in state s. Then the LTL formula for mutual exclusion is given by

$$\psi \doteq \mathsf{G}(\neg(P_1 \wedge P_2) \wedge \neg(P_2 \wedge P_3) \wedge \neg(P_3 \wedge P_1)).$$

Remark 2.1 (Conventions). Notice the use of "$\doteq$" above. When we write $A \doteq B$, we mean that A is a shorthand for writing B, and we use this notation throughout the monograph. In other treatises, one might often write this as $A = B$. But since much of the work in this monograph is based on a fixed formal logic, namely ACL2, we restrict the use of the symbol "=" to formulas in ACL2.

Clearly, LTL formulas can become big and cumbersome as the number of processes increases. In particular, if the system has an unbounded number of processes then specification of mutual exclusion by the above approach is not possible; we need stronger atomic propositions that can specify quantification over processes.

Given an LTL formula ψ and a Kripke Structure κ how do we decide if κ satisfies ψ? This is possible if κ has a finite number of states, and the method is known

as *model checking*.[3] There are several interesting model checking algorithms and a complete treatment of such is beyond the scope of this monograph. We merely sketch one algorithm that is based on the construction of a Büchi automaton, since we will use properties of this algorithm in Chap. 13.

Definition 2.4 (Büchi Automaton). A Büchi automaton $\mathscr{A}$ is given by a 5-tuple $\langle \Sigma, Q, \Delta, q_0, F \rangle$, where Σ is called the *alphabet* of the automaton, Q is the set of *states* of the automaton, $\Delta \subseteq Q \times \Sigma \times Q$ is called the *transition relation*, $q_0 \in Q$ is called the *initial state*, and $F \subseteq Q$ is called the *set of accepting states*.[4]

Definition 2.5 (Language of Büchi Automaton). An infinite sequence of symbols from Σ constitutes a *word*. We say that $\mathscr{A}$ *accepts* a word σ if there exists an infinite sequence ρ of states of $\mathscr{A}$ with the following properties:

- ρ^0 is the initial state of $\mathscr{A}$
- For each i, $\langle \rho^i, \sigma^i, \rho^{i+1} \rangle \in \Delta$
- Some accepting state occurs in ρ infinitely often

The *language* $\mathscr{L}(\mathscr{A})$ of $\mathscr{A}$ is the set of words accepted by $\mathscr{A}$.

What has all this got to do with model checking? Both LTL formulas and Kripke Structures can be translated to Büchi automata. Here is the construction of a Büchi automaton $\mathscr{A}_\kappa$ for a Kripke Structure $\kappa \doteq \langle S, R, L, s_0 \rangle$, such that every word σ accepted by $\mathscr{A}_\kappa$ corresponds to paths in κ. That is, for every word σ accepted by $\mathscr{S}$, $\sigma^i \subseteq \mathscr{AP}$ and there is a path π in κ such that σ^i is equal to $L(\pi^i)$.

- The set of states of $\mathscr{A}$ is the set S. Each state is an accepting state.
- $\Sigma \doteq 2^{\mathscr{AP}}$ where $\mathscr{AP}$ is the set of atomic propositions.
- $\langle s, \alpha, s' \rangle \in \Delta$ if and only if $\langle s, s' \rangle \in R$ and $L(s)$ is equal to α.

Similarly, given an LTL formula ψ we can construct a Büchi automaton $\mathscr{A}_\psi$ such that every word accepted by this automaton satisfies ψ. This construction, often referred to as *tableau construction*, is complicated but well known [51, 53]. Checking if κ satisfies ψ now reduces to checking $\mathscr{L}(\mathscr{A}_\kappa) \subseteq \mathscr{L}(\mathscr{A}_\psi)$. It is well known that the languages recognized by a Büchi automaton are closed under complementation and intersection. It is also known that given an automaton $\mathscr{A}$, there is an algorithm to check if $\mathscr{L}(\mathscr{A})$ is empty. Thus, we can check the language containment question above as follows. Create a Büchi automaton $\mathscr{A}_{\kappa,\psi}$ such that $\mathscr{L}(\mathscr{A}_{\kappa,\psi}) \doteq \mathscr{L}(\mathscr{A}_\kappa) \cap \overline{\mathscr{L}(\mathscr{A}_\psi)}$. Then check if $\mathscr{L}(\mathscr{A}_{\kappa,\psi})$ is empty.

[3] Model checking is the generic name for decision procedures for formalisms based on temporal logics, μ-calculus, etc. We only talk about LTL model checking in this monograph.

[4] The terms *state*, *transition relation*, etc. are used with different meanings when talking about automata and Kripke Structures, principally because they are used to model the same artifacts, *e.g.*, the executions of a computing system. In discussing Kripke Structures and automata together, this "abuse" of notation might cause ambiguity. We hope that the structure we are talking about will be clear from the context.

No practical model checker actually constructs the above automaton and checks emptiness explicitly. However, the construction suggests a key computational bottleneck of model checking. If a system contains n Boolean variables then the number of possible states in κ is 2^n. Thus, the number of states in the automaton for κ is exponential in the number of variables in the original system. Practical model checkers perform a number of optimizations to prevent this "blow-up." One key approach is to efficiently represent sets of states using BDDs [35]: this allows model checking to scale up to systems containing thousands of state variables. Nevertheless, in practice, model checking suffers from the well-known *state explosion* problem.

Several algorithmic techniques have been devised to ameliorate the state explosion problem. These are primarily based on the idea that in many cases one can reduce the problem of checking of a temporal formula ψ on a Kripke Structure κ to checking some formula possibly in a smaller structure κ'. We will consider two very simple reductions in Chap. 13. Modern model checkers perform a host of reductions to make the model checking problem tractable. They include identification of *symmetry* [50], reductions based on partial order [119], assume-guarantee reasoning [203], and many others. In addition, one popular approach is based on iterative refinement of abstractions based on counterexamples [52], often called "Counter Example Guided Abstraction Refinement" (CEGAR for short). Since model checking is a decision procedure, if a Kripke Structure does not satisfy a formula, then the procedure can return a counterexample (*i.e.*, a path through the Kripke Structure that does not satisfy the formula). In CEGAR, one starts with a Kripke Structure $\hat{\kappa}$ such that every execution of κ can be appropriately viewed as an execution of $\hat{\kappa}$ (but not necessarily vice versa). Then $\hat{\kappa}$ is referred to as an *abstraction* of κ. It is possible to find abstractions of a Kripke Structure with very small number of states. One then applies model checking to check if $\hat{\kappa}$ satisfies ψ. If model checking succeeds, then κ must satisfy ψ as well. If model checking fails, the counterexample produced might be spurious since $\hat{\kappa}$ has more execution paths than κ. One then uses this counterexample to iteratively *refine* the abstraction. This process concludes when either (a) the counterexample provided by model checking the abstraction is also a counterexample on the "real" system, that is, κ or (b) one finds an abstraction such that the model checking succeeds.

2.3 Program Logics, Axiomatic Semantics, and Verification Conditions

The use of decision procedures is one approach to scaling up formal verification. The idea is to automate verification by expressing the problem in a decidable formalism. The use of *program logics* is another approach. The goal of this approach is to simplify program verification by factoring out the details of the machine executing the program from the verification process.

How do we talk about a program formally? If we use Kripke Structures to model the program executions, then we formalize the executions in terms of *states*.

The states are valuations of the different components of the *machine* executing the program. Thus, in using Kripke Structures as a formalism to talk about the program, we must think in terms of the underlying machine. Specifying the semantics of a program by describing the effect of its instructions on the machine state is termed the *operational approach* to modeling the program and the semantics so defined are called *operational semantics* [161].[5] Thus, operational semantics forms the basis of applying model checking to reason about programs. We will later see that in many theorem proving approaches we use operational semantics as well. Operational models have often been lauded for their clarity and concreteness. Nevertheless, it is cumbersome to reason about the details of the executing machine when proving the correctness of a program. The goal of program logics [67, 103] is to come to grips with treating the program text itself as a mathematical object.

To do this, we will think of each instruction of the program as performing a transformation of predicates. To see how this is done, assume that I is any sequence of instructions in the programming language. The axiomatic semantics of the language are specified by a collection of formulas of the form $\{\mathscr{P}\}I\{\mathscr{Q}\}$, where $\mathscr{P}$ and $\mathscr{Q}$ are (first order) predicates over the program variables. Such a formula can be read as: "If $\mathscr{P}$ holds for the state of the machine when the program is poised to execute I then, after the execution of I, $\mathscr{Q}$ holds." Predicates $\mathscr{P}$ and $\mathscr{Q}$ are called the *precondition* and *postcondition* for I, respectively. For example, if I is a single instruction specifying an assignment statement `x := a`, then its axiomatic semantics is given by the following schema, also known as the "axiom of assignment."

Axiom of Assignment.
$\{\mathscr{P}\}\mathtt{x} := \mathtt{a}\{\mathscr{Q}\}$ holds if $\mathscr{P}$ is obtained by replacing every occurrence of the variable `x` in $\mathscr{Q}$ by `a`.

We can use this schema to derive, for example, $\{\mathtt{a} > 1\}\mathtt{x} := \mathtt{a}\{\mathtt{x} > 1\}$ which is interpreted as saying that if the machine is in a state s in which the value of `a` is greater than 1, then in the state s' reached after executing `x := a` from s, the value of `x` must be greater than 1. Notice that although the axiom is *interpreted* to be a statement about machine states, the schema itself, as well as its application, involve syntactic manipulation of the program constructs without requiring any insight into the operational details of the machine.

Hoare [103] provides five schemas like the above to specify the semantics of a simple programming language. In addition, he provides the following inference rule, often referred to as the *rule of composition*.

[5] We use "operational semantics" to denote the semantics derived from the definition of a formal interpreter in a mathematical logic. This terminology is common in program verification, in particular in mechanical theorem proving. However, this usage is somewhat dated among programming language communities, where "operational semantics" now refers to Plotkin's "Structural Operational Semantics" (SOS) [201], a style of specifying language semantics through inductively defined relations and functions over the program syntax. In current programming language terminology, our approach might more appropriately be referred to as "abstract machine semantics."

Rule of Composition.
Infer $\{\mathscr{P}\}\langle i_1; i_2\rangle\{\mathscr{Q}\}$ from $\{\mathscr{P}\}i_1\{\mathscr{R}\}$ and $\{\mathscr{R}\}i_2\{\mathscr{Q}\}$

Here $\langle i_1; i_2\rangle$ represents the sequential execution of the instructions i_1 and i_2. Another rule, which allows generalization and the use of logical implication, is the following.

Rule of Implication.
Infer $\{\mathscr{P}\}i\{\mathscr{Q}\}$ from $\{\mathscr{R}_1\}i\{\mathscr{R}_2\}$, $\mathscr{P} \Rightarrow \mathscr{R}_1$, and $\mathscr{R}_2 \Rightarrow \mathscr{Q}$

Here, "$\Rightarrow$" is simply logical implication. First-order logic, together with the Hoare axioms and inference rules, forms a proof system in which we can talk about the correctness of programs. This logic is referred to as a *Hoare logic*. Hoare logic is an instance of *program logic*, *i.e.*, logic tailored for reasoning about program correctness. In particular, Hoare logic is the logic for reasoning about programs with loops (but no pointers). To use Hoare logic for reasoning about programs, one maps assertions to predicates on program variables, and views program instructions as transformations of such predicates. From the examples above, it should be clear that one can capture the semantics of a programming language in terms of how predicates change during the execution of the program instructions. The semantics of a programming language specified by describing the effect executing instructions on assertions (or predicates) about states (rather than states themselves) is known as *axiomatic semantics*. Thus, Hoare logic (or in general a program logic) can be viewed as a proof system over the axiomatic semantics of the programming language.

Suppose we are given a program Π and we want to prove that if the program starts from some state satisfying $\mathscr{P}$, then the state reached on termination satisfies $\mathscr{Q}$. This can be succinctly written in Hoare logic as the statement $\{\mathscr{P}\}\Pi\{\mathscr{Q}\}$. $\mathscr{P}$ and $\mathscr{Q}$ are called the precondition and postcondition *of the program*. One then derives this formula as a theorem.

How do we use Hoare logic for program verification? One annotates the program with assertions at certain locations (*i.e.*, for certain values of the program counter). These locations correspond to the entry and exit of the basic blocks of the program such as loop tests and program entry and exit points. These annotated program points are also called *cutpoints*. The entry point of the program is annotated with the precondition, and the exit point is annotated with the postcondition. One then shows that if the program control is in an annotated state satisfying the corresponding assertion, then the next annotated state it reaches will also satisfy the assertion. This is achieved by using the axiomatic semantics for the programming language.

Let us see how all this is done with a simple one-loop program shown in Fig. 2.1. The program consists of two variables `X` and `Y`, and simply loops ten times incrementing `X` in each iteration. In the figure, the number to the left of each instruction is the corresponding program counter value for the loaded program. The cutpoints for this program correspond to program counter values 1 (program entry), 3 (loop test), and 7 (termination). The assertions associated with each cutpoint are shown to the right. Here the precondition `T` is assumed to be the predicate that is universally true. The postcondition says that the variable `X` has the value 10. Notice that in writing

```
1:  X:=0;                                  {T}
2:  Y:=10;
3:  if (Y ≤ 0) goto 7;                     {(X + Y) = 10}
4:  X:=X+1;
5:  Y:=Y-1;
6:  goto 3;
7:  HALT                                   {X = 10}
```

Fig. 2.1 A simple one-loop program

the assertions we have ignored type considerations such as that the variables X and Y store natural numbers.

How do we now show that every time the control reaches a cutpoint the assertion holds? Take for a simple example the cutpoints given by the program counter values 1 and 3. Let us call this fragment of the execution $1 \rightarrow 3$, identifying the beginning and ending values of the program counter along the execution of the basic block. Then we must prove the following formula as a theorem.

Proof Obligation.
$\{T\}\langle X := 0; Y := 10\rangle\{(X + Y) = 10\}$

Applying the axioms of assignment, composition, and implication above, we can now derive the following simplified obligation.

Simplified Proof Obligation.
$T \Rightarrow (0 + 10) = 10$

This proof obligation is trivial. But one thing to observe about it is that by applying the Hoare axioms we have obtained a formula that is free from the constructs of the programming language. Such a formula is called a *verification condition*. To finish the proof of correctness of the program here, we generate such verification conditions for each of the execution paths $1 \rightarrow 3$, $3 \rightarrow 3$, and $3 \rightarrow 7$, and show that they are logical truths.

Note that we have added an assertion at the loop test in addition to the precondition and postcondition. The process of annotating a loop test is often colloquially referred to as "cutting the loop." It is difficult to provide a syntactic characterization of loops in terms of predicates (as was done for the assignment statement); thus, if we omit annotation of a loop, then the Hoare axioms are normally not sufficient to generate verification conditions.

The verification conditions above only guarantee *partial correctness* of the program. That is, it guarantees that if the control ever reaches the exit point then the postcondition holds. It does not guarantee *termination*, that is, the control eventually reaches such a point. *Total correctness* provides both the guarantees of partial correctness and termination. To have total correctness, one needs an argument based on well-foundedness. We will carefully look at such arguments in the context of the ACL2 logic in the next chapter. For program termination, well-foundedness means that there must be a function of the program variables whose value decreases as the

control goes from one cutpoint to the next until it reaches the exit point, and the value of this function cannot decrease indefinitely. Such a function is called a *ranking function*. For total correctness one attaches, in addition to assertions, ranking functions at every cutpoint, and the axiomatic semantics are augmented so as to be able to reason about such ranking functions.

In practice, the verification conditions may be complicated formulas and their proofs might not be trivial. Practical applications of program verification based on axiomatic semantics depend on two tools: (1) a *verification condition generator* (VCG) that takes an annotated program and generates the verification conditions and (2) a theorem prover that proves the verification conditions. In this approach, it is not necessary to formalize the semantics of the program in a theorem prover or reason about the operational details of the machine executing the program. The VCG, on the other hand, is a tool that manipulates assertions based on the axiomatic semantics of the language.

One downside to applying this method is that one requires two trusted tools, namely a VCG and a theorem prover. Furthermore, one has to provide separate axiomatic semantics (and implement VCG) for each different programming language that one is interested in, so that the axioms capture the language constructs as formula manipulators. As new programming constructs are developed, the axiomatic semantics have to be changed to interpret such constructs. For instance, Hoare axioms are insufficient if the language contains pointer arithmetic, and additional axioms are necessary. With the plethora of axioms it is often difficult to see if the proof system itself remains sound. For example, early versions of *separation logic* (*i.e.*, an augmentation of Hoare logic to permit reasoning about pointers) were later found to be unsound [216]. In addition, implementing a VCG for a practical programming language is a substantial enterprise. For example, method invocation in a language like the Java Virtual Machine (JVM) involves complicated nonsyntactic issues like method resolution with respect to the object on which the method is invoked, as well as side effects in many parts of the machine state such as the call frames of the caller and the callee, thread table, heap, and class table. Coding all this in terms of predicate transformation, instead of state transformation as required when reasoning about an operational model of the underlying language, is difficult and error-prone. VCGs also need to do some amount of logical reasoning in order to keep the size of the generated formula reasonable. Finally, the axiomatic semantics of the program are not usually specified as a formula in a logic but rather encoded in the VCG which makes them difficult to inspect. Of course, one answer to all these concerns is that one can model the VCG itself in a logic, verify the VCG with respect to the logic using a theorem prover, and then use such a verified VCG to reason about programs. Some recent research has focused on formally verifying VCGs using a theorem prover with respect to the operational semantics of the corresponding language [80, 106]. However, formal verification of a VCG is a substantial enterprise and most VCGs for practical languages are not verified.

Nevertheless, axiomatic semantics and assertions have been popular both in program verification theory and in its practical application. It forms the basis of several verification projects [11, 66, 101, 185]. A significant benefit of using the approach is

to factor out the *machine details* from the program. One reasons about the program using assertions (and ranking functions), without concern for the machine executing it, except for the axiomatic semantics.

2.4 Bibliographic Notes

The literature on formal verification is vast, with several excellent surveys. Some of the significant surveys of the area include those by Kern and Greenstreet [133] and Gupta [94]. In addition, a detailed overview on model checking techniques is presented in a book on the subject by Clarke et al. [53].

Theorem proving was started arguably with the pioneering *Logic Theorist System* of Newell et al. [188]. One of the key advocates of using theorem proving for verification of computing systems was McCarthy [161] who wrote: "Instead of debugging programs one should prove that it meets its specification and the proof should be checked by a computer program." McCarthy also suggested the use of *operational semantics* for reasoning about programs. Many of the early theorem provers were based on the principle of resolution [218] that forms a sound and complete proof rule for first-order predicate calculus. The focus on resolution was motivated by the goal to implement fully automatic theorem provers. Theorem provers based on resolution are used today in many contexts; for instance, in 1999, EQP, a theorem prover for equational reasoning has recently "found" a proof of the Robbin's problem which has been an open problem in mathematics for about 70 years [163]. In the context of verification of computing systems, however, nonresolution theorem provers have found more applications. Some of the early work on nonresolution theorem proving were done by Wang [255], Bledsoe [16], and Boyer and Moore [21]. The latter, also known as the Boyer–Moore theorem prover or Nqthm, is the precursor of the ACL2 theorem prover that is the basis of the work discussed in this monograph. Nqthm and ACL2 have been used in reasoning about some of the largest computing systems ever verified. The bibliographic notes for the next chapter lists some of their applications. Other significant theorem provers in active use for verification of computing systems include HOL [88], HOL Light [100], Isabelle [189], Nuprl [59], and PVS [194].

The idea of using temporal logics for specifying properties of reactive systems was suggested by Pnueli [202]. Model checking was discovered independently by Clarke and Emerson [49] and Queille and Sifakis [207]. It is one of the most widely used formal verification techniques used in the industry today. Some of the important model checkers include SMV [165], VIS [28], NuSMV [47], SPIN [105], and Murϕ [69]. Most model checkers in practice include several reductions and optimizations [50, 119, 203]. Model checkers have achieved amazing results on industrial problems. While other decision procedures such as *symbolic trajectory evaluation* [44] and its generalization [259] have found applications in specification and verification of computing systems in recent times, they can be shown to be logically equivalent to instances of the model checking algorithms [233].

The notion of axiomatic semantics was made explicit in a classic paper by Floyd [77], although the idea of attaching assertions to program points appears much earlier, for example, in the work of Goldstein and von Neumann [84] and Turing [251]. Program logics were introduced by Hoare [103], Dijkstra [67], and Manna [149]. Assertional reasoning was extended to concurrent programs by Owicki and Gries [193]. Recently, separation logic has been introduced augmenting Hoare logic to reason about languages with pointers [217]. King [134] wrote the first mechanized VCG. Implementations of VCGs abound in the program verification literature. Some of the recent substantial projects involving complicated VCG constructions include ESC/Java [66], proof carrying code [185], and SLAM [11].

http://www.springer.com/978-1-4419-5997-3

Scalable Techniques for Formal Verification

Ray, S.

2010, XIV, 243 p., Hardcover
ISBN: 978-1-4419-5997-3