Chapter 14B
Model Checking and Deduction (DRAFT)*

Natarajan Shankar¹
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
shankar@csl.sri.com

Abstract. There are two basic approaches to automated verification. In model checking, the system is viewed as a graph representing possible execution steps. Properties are established by exploring or traversing the graph structure. In deduction, both the system and its putative properties are represented by formulas in a logic, and the resulting proof obligations are discharged by decision procedures or by automated or semi-automated proof construction. Model checking sacrifices expressivity for greater automation, and with deduction it is vice-versa. Newer techniques combine deductive and model checking approaches to achieve greater scale, expressivity, and automation. We examine the logical foundations of the two approaches and explore their similarities, differences, and complementarities. The presentation is directed at students and researchers who are interested in understanding the research challenges at the intersection of deduction and model checking.

1 Introduction

Verification establishes properties of all the possible executions of a program. There are two basic approaches to verification. In model checking [CGP99], the system is described as a model $M$ which is a graph in which the vertices are the states of the computation and the edges are possible execution steps. A property is a formula $P$ in a logic that characterizes a class of computations that can be generated from the graph. For example, a mutual exclusion property might state that no two processes are simultaneously in their critical section, while another property might assert that any process that is trying to enter its critical section, eventually succeeds. The satisfaction relation $M \models P$ is used to check if the system $M$ satisfies the property $P$, i.e., no computation that can be generated from $M$ violates $P$.

In the deductive approach [BM07,Sha09], both the system $M$ and the property $P$ are interpreted as formulas in a logic expressing constraints on the possible executions. The goal is to prove an assertion of the form $M \vdash P$ which captures the judgement that all possible executions of $M$ must satisfy the property $P$. A property might assert that a binary search procedure for a key $k$ in a sorted array $a$ terminates by returning an index $i$ such that $a[i] = k$ exactly when the element $k$ occurs in the array.

¹ This research was supported by NSF Grants CSR-EHCS(CPS)-0834810, and CNS-0917375, and by NASA Cooperative Agreements NNA10DE73C and NNA13AC55C.
In terms of automation, model checking verifies $M \models P$ using algorithms that explore the graph structure of $M$, whereas deduction operates on the syntactic structure of the formulas $M$ and $P$. When model checking fails, it produces a concrete counterexample explaining how the model $M$ violates property $P$. Deductive methods can also produce counterexamples, but typically human insight is required to discriminate between an incorrect system $M$, an invalid property $P$, and a misdirected proof attempt. In the latter case, human interaction is needed to redirect the proof attempt. When deduction succeeds, it yields a proof explaining why property $P$ holds of the system $M$.

The trend in model checking is toward greater expressivity in systems and properties while preserving the level of automation, while the trend in deduction is toward greater automation. In recent years, there has been a convergence of ideas and techniques in these two approaches to verification. We present the logical foundations of deduction and model checking, and examine their similarities, differences, and complementarities.

Early approaches to verification were based on deductive techniques. Examples of assertional program proofs were given by Goldstine and von Neumann [NG48] and by Turing [Tur92,MJ84,Jon03], but these were not presented as formal proof systems. McCarthy’s method [McC63] for reasoning about recursive Lisp functions used equational logic and an inference technique called recursion–induction. Floyd’s method [Flo67] is applied to programs represented as flowcharts annotated with assertions. A program flowchart can be seen as a directed graph with statements, statement blocks, or decision conditions on the vertices, and assertions on the edges. The flowchart has a source (start) vertex with a single outgoing edge, the precondition, and a sink (halt) vertex with a single incoming edge, the post-condition. Verification conditions are generated to verify that whenever the assertion for an incoming edge holds and a vertex is executed, the assertion for the corresponding outgoing edge holds. Proving these verification conditions is an effective way of establishing the partial correctness of the program: every execution of the program starting in the start vertex satisfying the precondition that terminates in the halt vertex, satisfies the post-condition.

Since the flowchart might have loops, it is possible to have valid infinite executions of the flowchart program that do not reach the halt vertex. Total correctness requires a proof of termination showing that every execution starting at the start vertex with a program state satisfying the precondition terminates in the halt vertex. Termination is verified using a ranking or variant function on the state that associates an ordinal with each state. The verification condition can be augmented to ensure that the ranking function strictly decreases with the execution of each program block. Since there are no infinite strictly decreasing sequences of ordinals, this eliminates the possibility of a valid infinite execution on the flowchart.

Hoare [Hoa69] transformed Floyd’s method into a set of inference rules for Hoare triples of the form $\{P\} S \{Q\}$ for a well-structured program. The triple expresses the claim that any terminating execution of program $S$ from a state satisfying the assertion $P$ yields a state satisfying the assertion $Q$. The Floyd-Hoare approach to deductive verification can be illustrated with the simple example shown in Figure 1 of a program that finds the maximal element in an array of non-negative integers.

The program in Figure 1 consists of a precondition true, a loop invariant

$$i \leq N \land \forall (j < i) : a[j] \leq \text{max}$$
max = 0;
i = 0;
\{i ≤ N \land \forall(j < i): a[j] ≤ max\}
while (i < N){
    if (a[i] > max){
        max = a[i];
    }
i++;
}\{\forall(j < N): a[j] ≤ max\}

Fig. 1. Program and flowchart for finding the maximal element in a non-negative array.

and a post-condition \(\forall(j < N): a[j] ≤ max\). The loop invariant, which we abbreviate as \(P(i)\), asserts that the value of the index variable \(i\) is at most \(N\), and all the elements preceding the \(i\)’th element in the array \(a\) are at most \(max\).

Informally, termination for the program in Figure 1 can be established by associating a ranking function \(N - i\) with the while loop. If \(S\) abbreviates the loop body \(if(a[i] > max)\{\max = a[i];i++;\}\), then the triple \(\{r = N - i\}S\{r > N - 1\}\) is valid since \(i\) is incremented in \(S\). Note that \(r\) is a logic variable in the triple that is implicitly universally quantified. Since this loop satisfies the invariant \(N - i ≥ 0\), the value of the ranking function is always a non-negative integer and the ordering is well-founded.

Next, we want to establish the post-condition that \(\forall(j < N): a[j] ≤ max\). This can be derived from a combination of the loop exit condition \(\neg(i < N)\) and the loop invariant \(P(i)\). The loop invariant holds trivially following the initialization, as expressed by the triple

\(\{true\}\max = 0; i = 0\{P(i)\}\).

The preservation of the invariant by the loop body is expressed by the triple

\(\{P(i) \land i < N\}S\{P(i)\}\).

The latter triple generates the proof obligations below that are both easily proved.

1. \(i < N \land a[i] > max \land P(i) \Rightarrow i + 1 ≤ N \land \forall(j < i + 1): a[j] ≤ a[i]\), and
2. \(i < N \land a[i] ≤ max \land P(i) \Rightarrow P(i + 1)\).

In the above example, the invariant is a straightforward generalization of the post-condition. To see where the deductive approach is less handy, we examine a simple variant of Peterson’s mutual exclusion algorithm [Pet81] shown in Figure 2. The computation state consists of the control state encoded by two Boolean variables per process: \(critical[i]\) and \(trying[i]\) with \(i = 1, 2\), and a shared Boolean variable \(turn\). Each step of the computation applies a transition rule of one of the processes, where each
Fig. 2. A two-process mutual exclusion protocol

rule is a guarded command. Thus, each computation step is either a step of process 1 according to one of its transition rules (leaving the values of \(\text{try}[2]\) and \(\text{critical}[2]\) unchanged), or a step of process 2 according to one of its transition rules (leaving the values of \(\text{try}[1]\) and \(\text{critical}[1]\) unchanged).

The mutual exclusion property \(\neg(\text{critical}[1] \wedge \text{critical}[2])\) is an invariant in that it holds in every reachable state, but it is not an inductive invariant since it is not preserved by the second transition of either process. A stronger invariant asserting that

\[
(\text{critical}[1] \Rightarrow \text{turn}) \land (\text{critical}[2] \Rightarrow \neg\text{turn})
\]

does turn out to be inductive. Finding such invariant strengthenings is not always simple, and is an active area of research.

Algorithms like those for mutual exclusion need not terminate, but they are required to make progress. For instance, in any computation, once \(\text{try}[i]\) is true, then \(\text{critical}[i] = \text{true}\) must eventually hold. Such an eventuality would not follow if, for example, the other process remains in its critical section by never executing the third transition even when the guard condition holds, i.e., the transition is enabled. Fairness conditions can be used to ensure that enabled transitions are eventually executed, for example, by requiring the condition \(\text{critical}[i] = \text{false}\) to hold infinitely often for each \(i\) along an execution. Deductive rules based on ranking functions can be given to ensure that \(\text{try}[i] = \text{true}\) leads to \(\text{critical}[i] = \text{true}\), but such proofs can be tedious since several eventuality and invariant claims might need to be composed [CM88,MP92].

Model checking [EC82,QS82] was introduced in the early 1980s as an approach for analyzing systems with a lot of control complexity. Here, the flowcharts and programs are less structured and deductive approaches become unwieldy. Examples of such system include hardware, network protocols, and concurrent systems. In the model checking approach, the transition system is viewed as a model for the property. Such a model can be seen as a directed graph where the states, which assign values to variables, are vertices, and the edges are possible transitions between states given by the program. The verification of properties can be performed by graph exploration to determine if some class of states or if a certain kind of cycle is reachable. For the case of the mutual exclusion algorithm in Figure 2, it is possible to scan the entire reachable portion of the graph in the five Boolean variables to check that there are no violations of the mutual exclusion property. If this check fails, model checking can produce a counterexample in the form of a computation path that leads to a violation of mutual exclusion. Conversely, one
could start with the set of “bad” states, i.e., those where \((\text{critical}_1 \land \text{critical}_2)\) holds, and compute the backward reachable states, i.e., those that have computations leading to bad state, to see that no initial state has a computation to a bad state. The progress property can also be verified by showing that there are no fair execution paths in the graph where \(\text{try}_i = \text{true}\) does not eventually lead to \(\text{critical}_i = \text{true}\).

Since this is a finite state system, the only way this eventuality could fail is if the graph contains a state where for some \(i\), \(\text{try}_i = \text{true}\) holds and from which it is possible to reach a cycle in which \(\text{critical}_i = \text{false}\) in each state. Instead of a cycle, it is sufficient (and necessary) to find a strongly connected component, that is, a subset of states in which there is a path between any two states. Such a strongly connected component is fair if each fairness condition holds for some state in it.

Model checking problems are decidable for finite-state systems, namely those with state spaces of bounded cardinality, and also for certain extensions to systems with unbounded state spaces. Examples of such extensions are covered elsewhere in this Handbook, and include

1. Timed automata [AD94], covered in Chapter ??
2. Certain limited classes of hybrid automata, covered in Chapter ??
3. Parametric systems, covered in Chapter ??

Abstraction techniques can be used to approximate large or possibly unbounded state spaces by models with small, finite state spaces, and these are covered in Chapters ?? and ??.

Compositional techniques for decomposing the verification of composite systems with multiple modules into properties associated with the individual modules are covered in Chapter ??.

In model checking, an explicit representation of the state space is one where a state is represented by a specific assignment of values to variables. Model checking with explicit representations is covered in Chapter ??.

In contrast, symbolic representations of the state space use compact data structures to represent sets of states as formulas. Model checking based on symbolic representations is covered in Chapter ?? of this Handbook. Techniques based on model checking can also be used to synthesize systems for a given property, and this topic is covered in Chapter ??.

In this chapter, we focus on the relationship between deduction and model checking. We first outline the shared background of logic for both approaches. We then examine specific ways in which these approaches can complement each other.

## 2 Logic Background

Logic is a system of notations and rules for making statements, and for proving or refuting these statements. The syntax of the logic provides rules for forming well-formed statements. The semantics defines the intended meaning of the language primitives by circumscribing their possible interpretations. The inference rules of the logic specify how valid statements, i.e., those that hold in all possible interpretations, are derived.

Different logics correspond to variations in the syntax, semantics, and the inference rules. We use propositional logic to illustrate some of the key concepts that are relevant to formal verification.
2.1 Propositional Logic

Syntax and Semantics. In propositional logic, statements $P$ and $Q$ are built as formulas from propositional atoms such as $p$ and $q$ drawn from a signature $\Sigma$. A $\Sigma$-formula is constructed using propositional connectives such as negation $\neg P$, conjunction $P \land Q$, disjunction $P \lor Q$, implication $P \Rightarrow Q$, and equivalence $P \Leftrightarrow Q$. The classical semantics is defined by a truth assignment $M$ that maps propositional atoms in $\Sigma$ to truth values, either $\bot$ for logical falsity or $\top$ for logical truth. The interpretation $M[P]$ of a formula $P$ is constructed from the interpretation of its constituents. For a propositional atom $p$ in $\Sigma$, the interpretation $M[p]$ is its truth assignment $M(p)$. For compound formulas $P \phi Q$, where $\phi$ is either $\land$, $\lor$, $\Rightarrow$, or $\Leftrightarrow$, the interpretation $M[P \phi Q]$ is computed from that of $M[P]$ and $M[Q]$ from the truth table interpretation of $\phi$. A formula $P$ is satisfiable if there is some interpretation $M$ such that $M[P] = \top$. We then say that $M \models P$ or $M$ is a model of $P$. If $M \models P$ for all interpretations $M$, then $P$ is said to be valid. The negation of an unsatisfiable formula is valid. A set of formulas $\Gamma$ is satisfiable if there is a model $M$ such that $M \models P$ for each $P \in \Gamma$, or $M \models \Gamma$, for short.

The model checking problem for propositional logic is that of checking $M \models P$ for a given $M$ and $P$. This problem is complete for alternating logarithmic time (ALOGTIME) [Bus87]. The satisfiability problem is that of determining if there is a model $M$ for a given $P$. By the celebrated Cook–Levin theorem, this problem is NP-complete [AB09].

Proof Systems. The inference rules for classical propositional logic can be presented in a number of formats: Hilbert system, natural deduction, or sequent calculus. A proof system for propositional logic based on one-sided sequents is shown in Figure 3. Each sequent has the form $\Gamma \vdash \Delta$, where $\Gamma$ is a finite set of formulas, and represents a judgement that under any interpretation, one of the sequent formulas is logically true. The set obtained from adding $P$ to $\Gamma$ is $P, \Gamma$. The axiom rule $Ax$ asserts that any sequent containing a positive and negative atom is provable. Each rule asserts that the conclusion sequent is valid if the premise sequents are. The implication $P \Rightarrow Q$ can be written as $\neg P \lor Q$ and the conjunction $P \land Q$ as $\neg(\neg P \lor \neg Q)$. Note that the provability of
$P \Rightarrow Q$ can be represented by the sequent $\vdash \neg P, Q$. A formula $P$ is provable if $\vdash P$ can be derived from the axioms given by the $Ax$ rule by applying the rules $\neg \neg, \lor, \neg \lor$, or $Cut$. For example, the sequent $\vdash \neg \neg p \lor \neg p$ can be shown to be provable.

Algorithms for Boolean satisfiability are discussed in Chapter ?? of this Handbook. Propositional logic is also expressive enough to capture constraints over bounded size domains. Such domains can be encoded in binary form. Bit vectors of length $n$ can be written as $n$ Boolean variables. An element of a subrange of integers of size $n$ can be encoded by a bit-vector of length $\lceil \log_2 n \rceil$. Arrays of at most $m$ elements drawn from a set of cardinality at most $n$ can be represented by $m \lceil \log_2 n \rceil$ bits. Bounded length lists from a bounded set of elements can be represented by arrays. Sets, functions, and relations over bounded domains can also be represented by Boolean formulas, as can images of functions and relations with respect to sets and compositions of functions and relations. Computations of bounded length over bounded state spaces can also be represented as Boolean constraints. Such an encoding can be used for bounded model checking, that is, to determine whether computations of a bounded length violate a specific property.

Propositional logic has a number of useful properties. It is possible to provide proof systems for it that are sound — all provable statements are valid, and complete — all valid statements are provable. Propositional logic is also compact: if a set of formulas is unsatisfiable, there is some finite subset that is already unsatisfiable. For a set of propositional $\Sigma$-formulas $\Gamma$ over a signature $\Sigma$, we say that a truth assignment $M$ is a model of $\Gamma$ if it is a model of each formula in $\Gamma$. Then $\Gamma$ is said to entail a propositional formula $P$ if every model of $\Gamma$ is a model of $P$.

**Soundness.** A proof system such as the one in Figure 3 is sound if it proves only valid statements. This means that the sequent $\vdash P$ is provable only when $P$ is valid. More generally, $\vdash \Delta$ is provable only when $\vdash \Delta$ is valid, i.e., for each interpretation $M$, $M \models P$ for some $P \in \Delta$. Soundness can be established by induction on derivations: the $Ax$ rule is valid, and the conclusion of each application of an inference rule is valid if its premises are.

**Completeness.** The sequent calculus in Figure 3 is complete: every valid sequent is provable. Conversely, if $\vdash P$ is not provable, then we can construct a model $M$ of $\neg P$. A set of formulas $\Gamma$ is consistent, i.e., $\text{Con}(\Gamma)$ iff there is no formula $P$ in $\Gamma$ such that $\vdash T, \neg P$ is provable, where $T$ is the set $\{\neg Q \mid Q \in \Gamma\}$. By soundness, when $\vdash T, \neg P$ is provable, then $\neg P$ is entailed by $\Gamma$ since in every interpretation, either $\Gamma$ is falsified or $P$ is. Note that consistency is a property of the proof system and not a semantic property. If $\Gamma$ is consistent, then $\Gamma \cup \{P\}$ is consistent iff $\vdash T, \neg P$ is not provable. If $\Gamma$ is consistent, then at least one of $\Gamma \cup \{P\}$ or $\Gamma \cup \{\neg P\}$ must be consistent. A set of formulas $\Gamma$ is complete if for each formula $P$, it contains $P$ or $\neg P$. It is not difficult to construct an enumeration $\langle Q_i \mid i \geq 0 \rangle$ of all the $\Sigma$-formulas. With this enumeration, if $\vdash P$ is not provable, we construct a complete set $\Gamma$, where $\Gamma = \bigcup_i \Gamma_i$, $\Gamma_0 = \{\neg P\}$, and $\Gamma_{i+1} = \Gamma_i \cup \{Q_i\}$ if $\text{Con}(\Gamma \cup \{Q_i\})$, and $\Gamma_{i+1} = \Gamma_i \cup \{\neg Q_i\}$, otherwise. For any atom $p$, define $M_\Gamma(p) = \top$, if $p \in \Gamma$, and is $M_\Gamma(p) = \bot$, otherwise. Then, $M_\Gamma \models Q$ for each $Q \in \Gamma$. Hence, $M_\Gamma \models \neg P$. For example, $\vdash p \lor q$ is not provable, and we can
construct a model $M_\Gamma$ starting with $\Gamma_0 = \{\neg(p \lor q)\}$ and checking that $\Gamma$ contains $\neg p$ and $\neg q$.

**Compactness.** Propositional logic is compact in the sense that a (possibly infinite) set $\Gamma$ of $\Sigma$-formulas is satisfiable iff it is finitely satisfiable, i.e., each finite subset $\Delta$ of $\Gamma$ is satisfiable. Since any subset of a satisfiable set of formulas is satisfiable, we only have to show that finite satisfiability implies satisfiability. Any finitely satisfiable set $\Gamma$ can be extended to a maximal finitely satisfiable set $\hat{\Gamma}$. This is because by Zorn’s lemma, any partially ordered set $\Theta$ in which every linearly ordered subset has an upper bound, contains a maximal element. The set of finitely satisfiable extensions of $\Gamma$ ordered by inclusion satisfies the conditions of Zorn’s lemma: the union $\bigcup F$ of a linearly ordered subset $F$ of $\Theta$ of finitely satisfiable sets is itself finitely satisfiable — any finite subset of $\bigcup F$ must be a subset of an element of $F$.

Given a maximal finitely satisfiable extension $\hat{\Gamma}$ of a finitely satisfiable set $\Gamma$ of $\Sigma$-formulas, then, by maximality, for any $\Sigma$-atom $p$, either $\hat{\Gamma} \cup \{p\}$ or $\hat{\Gamma} \cup \{\neg p\}$ must be finitely satisfiable. As in the completeness argument, we can define $M_\Gamma$ so that $M_\Gamma(p) = \top$, if $\hat{\Gamma} \cup \{p\}$ is finitely satisfiable, and $M_\Gamma(p) = \perp$, otherwise. By maximality, if $M_\Gamma(p) = \top$, the $p \in \Gamma$, and if $M_\Gamma(p) = \perp$, then $\neg p \in \Gamma$. Then $M_\Gamma$ must be a model for $\Gamma$ since there is a formula $P \in \Gamma$ such that $M_\Gamma(P) = \perp$, then $\hat{\Gamma}$ would not be finitely satisfiable. Hence, $\Gamma$ is satisfiable.

By compactness, if a set $\Gamma$ is unsatisfiable, it has a finite subset $\Delta$ that is unsatisfiable. Also, if $P$ is entailed by $\Gamma$, then there is a finite subset $\Delta$ of $\Gamma$ such that $P$ is entailed by $\Delta$.

**Interpolation.** Given a set $\Gamma$, a formula $P$ is incompatible with $\Gamma$ if $\Gamma \cup \{P\}$ is unsatisfiable. Given sets $\Gamma_1$ and $\Gamma_2$ of $\Sigma_1$ and $\Sigma_2$ formulas, respectively, $\Gamma_1 \cup \Gamma_2$ is unsatisfiable iff there is a $\Sigma_0$-formula $P$ that is entailed by $\Gamma_1$ and incompatible with $\Gamma_2$, where $\Sigma_0 = \Sigma_1 \cap \Sigma_2$. Such a formula $P$ is said to be an interpolant [Cra57]. To see why such an interpolant must exist, let $\Pi$ be the set of $\Sigma_0$-formulas $P$ entailed by $\Gamma_1$. The argument for the existence of an interpolant has the following steps:

1. If all formulas in $\Pi$ are compatible with $\Gamma_2$, then there is a $\Sigma_2$-interpretation $M_2$ such that $M_2 \models \Pi \cup \Gamma_2$. Let $M$ be the interpretation $M_2$ restricted to $\Sigma_0$, and define $\Gamma_M$ to be the set of $\Sigma_0$-formulas $Q$ such that $M \models Q$. Note that $M$ is the unique $\Sigma_0$-model for $\Gamma_M$.
2. If $\Gamma_1 \cup \Gamma_M$ is satisfiable, there must be some $\Sigma_1$-interpretation $M_1$ extending $M$ such that $M_1 \models \Gamma_1$. We then have interpretations $M_1$ and $M_2$ such that $M_1 \models \Gamma_1$, $M_2 \models \Gamma_2$, and $M_1$ and $M_2$ restricted to $\Sigma_0$ are both equal to $M$. We can let the interpretation $N(p)$ be $M_1(p)$, if $p \in \Sigma_1$, and $M_2(p)$, otherwise. Then $N \models \Gamma_1 \cup \Gamma_2$ since $N$, the amalgamation of $M_1$ and $M_2$, is identical to $M$, when restricted to $\Sigma_1$, for $i = 1, 2$. This contradicts the assumption that $\Gamma_1 \cup \Gamma_2$ is unsatisfiable.
3. If $\Gamma_1 \cup \Gamma_M$ is unsatisfiable, then by compactness, there must be a minimal finite subset $\Delta$ of $\Gamma_M$ such that $\Gamma \cup \Delta$ is unsatisfiable. Since $\Delta$ is finite, we can construct a formula $\bigwedge \Delta$ that is the conjunction of the formulas in $\Delta$. Then $\neg \bigwedge \Delta$ is a $\Sigma_0$-formula entailed by $\Gamma_1$, and hence $M \models \neg \bigwedge \Delta$. But, by construction, $M$ is a model
of all the formulas in $\Gamma_M$, and hence all the formulas in $\Delta$. Hence $\Gamma_1 \cup \Gamma_M$ must be satisfiable.

4. Hence, some formula in $\Pi$ must be incompatible with $\Gamma_2$, and is therefore an interpolant.

The above argument can be extended to first-order logic [GRS04]. However, it merely demonstrates the existence of interpolants without giving a procedure for constructing one. As an example of such a procedure, we show how interpolants can also be constructed from proofs. For example, consider the one-sided sequent calculus for propositional logic containing just negation and disjunction shown in Figure 3. Here, if we prove a sequent of the form $\vdash \Gamma, \Delta$, then this essentially says that the set of formulas $\Gamma \cup \Delta$ is unsatisfiable, where $\Gamma$ is the set obtained by complementing each of the formulas in $\Gamma$. This means there must be some formula $P$ such that $\vdash \Gamma, P$ and $\vdash \neg P, \Delta$.

For this, we represent the sequent as $\vdash \Gamma, \Delta$ to separate the two parts of the sequent. Then, the rules in Figure 4 shows how the interpolant can be constructed for cut-free proofs.

Propositional satisfiability can be made more expressive along several dimensions. Adding quantification over Boolean variables allows first-order logic formulas over bounded domains to be expressed in the logic. The resulting logic fragment, quantified Boolean formulas (QBF), has a PSPACE-complete satisfiability problem. With quantification, there is no distinction between satisfiability and model checking — for a closed formula, i.e., one with no free variables, these are equivalent. Quantified Boolean formulas can be expanded into Boolean formulas with an exponential increase in size. With the added expressiveness of quantification, we can capture image construction, bounded length games, and inductive relations over bounded domains.

Modal logics [BdRV02] offer another dimension of expressibility where the satisfiability of a formula is evaluated relative to a frame or Kripke model which is a graph $\langle W, R, L \rangle$ of worlds $W$ that are related by the accessibility relation $R$ and a labeling

\begin{table}
\begin{tabular}{|c|l|}
\hline
Ax$_1$ & $\bot \vdash \Gamma, P, P; \Delta$
\hline
Ax$_2$ & $\top \vdash \Gamma, P; P; \Delta$
\hline
Ax$_3$ & $P \vdash \Gamma, P; P; \Delta$
\hline
$\neg$ & $[I] \vdash P, \Delta$
\hline
$\neg$ & $[I] \vdash \neg P, \Delta$
\hline
$\lor$ & $[I] \vdash A, B, \Delta$
\hline
$\lor$ & $[I] \vdash A \lor B, \Delta$
\hline
$\neg \lor$ & $[I_1] \vdash \Gamma, \neg A; \Delta$
\hline
$\lor$ & $[I_2] \vdash \Gamma, \neg A, \Delta$
\hline
$\neg \lor$ & $[I_1 \lor I_2] \vdash \Gamma, \neg (A \lor B), \Delta$
\hline
$\neg \lor$ & $[I_1 \land I_2] \vdash \Gamma, \neg (A \lor B), \Delta$
\hline
\end{tabular}
\end{table

\textbf{Fig. 4.} Interpolants from cut-free proofs
mapping worlds to truth assignments for the atomic propositions. The worlds adjacent to or accessible from a given one are the possible alternate truth assignments. The modality $\square A$ holds in a world if it holds in all worlds that are accessible from it. A statement is valid in a frame if it holds in all of the possible worlds. Many different modal logics can be defined by varying the properties of the accessibility relation. These yield different informal interpretations for the modalities, including necessity, knowledge, belief, and normativity, among others [BdRV02,Min92]. The model checking problem for modal logics is typically P-complete, whereas the corresponding satisfiability problem is typically PSPACE-complete.

In the context of formal verification, the accessibility relation corresponding to the progress of time is important. Here the possible worlds correspond to points of time in a computation. One world is accessible from another when it is in the future of the second world. This yields temporal logics [Eme90] where $\Diamond P$ holds in a world when there is some future world where $P$ holds, and $\square P$ holds if there is no possible future world where $\neg P$ holds. Temporal logic can be further decomposed into linear-time and branching time logics. In linear time, the possible worlds consist of paths, or sequences of states. These paths are related by a suffix relation. A property $\square P$ holds of a path when $P$ holds of all suffix paths, and a property $\Diamond P$ holds when $P$ holds of some suffix. In contrast, in branching-time temporal logics, to each path representing the future, there are alternate possible paths. The formula $\square P$ holds of a path if $P$ holds along every suffix of the path, and $\forall P$ holds of a path if $P$ holds of every alternate path. Temporal logics are covered in Chapter ?? of this Handbook.

The modal operators can also be indexed by the computations or computations steps. For example, in dynamic logic, the modal operators are indexed by programs so that $[\alpha]P$ holds in a state when all computations of the program $\alpha$ from this state terminate in states satisfying $P$. Henessey–Milner logic indexes the modal operators $[\alpha]P$ and $\langle\alpha\rangle P$, with the labels $\alpha$ corresponding to the actions in a labeled transition system. We can add fixpoint operators to this calculus to arrive at the modal $\mu$-calculus which is covered in Chapter ?? of this Handbook. With these operators, we have formulas of the form $\mu X. F[X]$ and $\nu X. F[X]$, where $F[X]$ is a modal formula in which the propositional variable $X$ occurs with positive parity in $F[X]$, i.e., under an even number of negations.

### 2.2 First-Order Logic

Propositional logic is too limited in its expressivity for many applications. It cannot encode problems over unbounded and infinite domains. Even with problems that can be encoded in propositional logic, one loses the structure and uniformity of the original problem. First-order logic (FOL) [Bar78] refines propositional atoms so that instead of $p$, we also admit propositions $p(x_1,\ldots,x_n)$, where $p$ is a predicate symbol and $x_1,\ldots,x_n$ are variables. So, instead of an opaque proposition of the form: *Birmingham is the capital of England*, we have a more refined predicate *IsTheCapitalOf*, and we can write formulas containing *IsTheCapitalOf*(x, y). This expressivity is naturally well-suited for mathematical formalization where the domains are typically unbounded and relations abound. FOL can also include function symbols with their associated arity, so that a term is either a variable $x$ or a compound term $f(a_1,\ldots,a_n)$ where $f$ is a function
Fig. 5. Proof rules for quantification: $c$ must not occur in the conclusion of $\neg \exists$ symbol of arity $n$, with $n \geq 0$. FOL can also contain a primitive equality symbol so that $a = b$ is a formula whenever $a$ and $b$ are two terms. If we have a function $\text{Capital}$ that maps each country to its capital city, then the above proposition can be written as $\text{Birmingham} = \text{Capital(England)}$. First-order logic is then defined relative to a signature $\Sigma$ which is a set containing the allowed function and predicate symbols. FOL formulas are built from equalities $a = b$, atoms $p(a_1, \ldots, a_n)$ for predicate symbol $p$ of arity $n$, the propositional combinations of negation, disjunction, conjunction, implication, and equivalence, and existential quantification $\exists x. P$ and universal quantification $\forall x. P$. The formula $P$ is the scope of the quantifier $\exists x. P$ and $\forall x. P$. An occurrence of a variable $x$ in a formula $P$ is free if it does not occur in the scope of a quantifier binding the variable $x$. The operation of substituting a term $a$ for a free variable $x$ in a formula $P$ is written as $P[a/x]$. Care is required to ensure that such a substitution does not capture free variables in $a$, so that if $y$ is a free variable in $a$, then $x$ must not occur within the scope of a quantifier binding $y$ in $P$. A sentence is a closed formula, i.e., one without any free variables. Propositional logic is the fragment of first-order logic where the signature $\Sigma$ contains only 0-ary predicate symbols.

The semantics for first-order logic over a signature $\Sigma$ is given by a $\Sigma$-structure $M$ with a non-empty set $\text{dom}(M)$ (the domain) and an interpretation of the function and predicate symbols of $\Sigma$ as functions and predicates over $\text{dom}(M)$. We also need an assignment $\rho$ that maps the variables to elements of $\text{dom}(M)$ so that the meaning of each term $a$ can be defined as $M[\[a\]]\rho$. The satisfaction relation $M, \rho \models P$ is defined in the expected manner.

The sequent calculus in Figure 3 can be extended with a couple of rules for reasoning about existential quantification shown in Figure 5. The $\neg \exists$ rule allows a sequent formula $\neg P[c/x], \Delta$ to be generalized to $\neg \exists P$, provided the constant $c$ does not appear in $\neg (\exists x. P), \Delta$. Since such a constant $c$ can bound to any element in $\text{dom}(M)$ in the premise, the conclusion sequent is justified. The $\exists$ rule allows a formula $\exists x. P$ to be derived from $P[a/x]$ since the term $a$ serves as a witness for the existential quantifier. The rules for universal quantification can be defined from these rules.

First-order logic is a natural formalism for representing mathematical theories. For this purpose, one adds non-logical axioms and axiom schemes to the logic. Mathematical theories that can be represented in this manner include various algebras, Peano arithmetic, and set theory. First-order logic can be given a sound and complete formalization so that all and only the valid statements have proofs. It also has a number of interesting metatheoretic properties like compactness, amalgamation, and interpolation. FOL formulas can be placed in prenex normal form where all the quantifiers appear as a prefix.
at the top of the formula. For example, the sentence $\forall x.(p(x) \land (\exists y.\neg p(y)))$ has the equivalent prenex form $(\forall x.\exists y.p(x) \land \neg p(y))$. The universal quantifier can be replaced by a Skolem function to get the equisatisfiable formula $(\forall x.p(x) \land \neg p(f(x)))$. The resulting formula is unsatisfiable since it has an Herbrand expansion $(p(c) \land \neg p(f(c))) \land (p(f(c)) \land \neg p(f(f(c))))$ which is propositionally unsatisfiable. The Herbrand expansion is a conjunction of instances of the Skolemized formula with terms from the term universe obtained by adding a constant $c$, if needed, to the Skolem functions. Many proof search methods are strategies for constructing unsatisfiable Herbrand expansions.

Certain fragments of first-order logic are decidable [BGG97]. For example, monadic predicate calculus, where there are no function symbols and only monadic predicate symbols, is decidable. The Bernays-Skolem fragment consists of formulas whose prenex form has a block of existential quantifiers followed by a block of universal quantifiers. The satisfiability of sentences in this fragment is decidable. One simple decidable fragment is the first-order theory of pure equality. This theory has an empty signature so that all the atomic formulas are equations or disequations between variables. This fragment is expressive enough for writing formulas that constrain the minimal or maximal number of elements in a model. If a formula in prenex form in this fragment has $n$ distinct variables and is satisfiable, then it is satisfiable in a model with at most $n$ elements. The fragment therefore has the finite model property.

A first-order theory $T$ is the set of sentences that are valid over a given set of interpretations $K$ so that $T = \{ P \mid \forall M \in K. M \models P \}$. Such a theory could also be given by a set of sentences closed under consequence that has at least one model. A theory is stably infinite if whenever a formula has a model, it has one with a countably infinite domain. The first-order theory of pure equality over stably infinite models supports quantifier elimination: from any formula $P$, one can construct an equisatisfiable formula $\hat{P}$ that has no quantifiers. As noted earlier, it is easy to write a sentence that asserts that there are at most $k$ elements, and this sentence has no quantifier-free counterpart. The theory of arithmetic over $0$, $1$, and $+$ (but without multiplication), known as Presburger arithmetic, also supports quantifier elimination. The validity of a sentence in this theory can therefore be decided by constructing its quantifier-free counterpart and evaluating its validity directly. Other first-order theories that support quantifier elimination include dense linear orders, algebraically closed fields, and real closed fields.

Within theories, one can look at whether it is decidable to check for the validity of formulas of a specific form. For a formula $P$, let $\forall P$ represent the universal closure $\forall \pi. P$ of $P$, where $\pi$ is a sequence of the free variables in $P$. The word problem for a theory is that of deciding the validity of a formula $\forall P$, where $P$ is atomic. The uniform word problem is that of deciding the validity of a formula $\forall (P_1 \land \ldots \land P_n \Rightarrow P)$, where $P$ and the formulas $P_i$, for $1 \leq i \leq n$, are atomic. The clausal validity problem is that of deciding the validity of formulas of the form $P_1 \lor \ldots \lor P_n$, where each $P_i$ is either an atomic formula or the negation of an atomic formula. A theory is convex when a clause $\neg p_1 \lor \ldots \lor \neg p_m \lor q_1 \lor \ldots \lor q_n$ is valid iff the uniform word problem $\neg p_1 \lor \ldots \lor \neg p_m \lor q_i$ is valid, for some $i$, $1 \leq i \leq n$. For example, the theory of linear arithmetic over the reals is convex, whereas the theory of integer linear arithmetic is non-convex since $x > 1 \land x < 5 \Rightarrow (x = 2 \lor x = 3 \lor x = 4)$ without the antecedent implying any one of the consequent disjunctions. Since any quantifier-free formula $P$ can be expressed as a
conjunction of clauses $K_1 \land \ldots \land K_n$, the validity of $\forall P$ can be reduced to the validity of $\forall K_i$ for each $i, 1 \leq i \leq n$. However, a more efficient approach to the quantifier-free validity problem is to use clausal validity within a Boolean satisfiability (SAT) procedure to detect that a partial assignment computed by the SAT solver is unsatisfiable in a theory. Theory solvers (for clausal validity) can also be used to strengthen Boolean constraint propagation to exploit theory propagation so that, for example, when $f(x) \neq f(y)$ is in the partial assignment, the literal $x = y$ is immediately implied as being false. The combination of Boolean satisfiability checking with theory solving, namely Satisfiability Modulo Theories (SMT), is covered in Chapter ?? of this Handbook.

A formula is satisfiable in a combination of theories $T_1$ over a signature $\Sigma_1$ and $T_2$ over a signature $\Sigma_2$ if it has a model $M$ such that $M$ restricted to $\Sigma_i$ is a $T_i$-model, for $i = 1, 2$. The Nelson–Oppen method [NO79] can be used to demonstrate clausal validity in a combination of theories with disjoint signatures. A clause $K$ is first purified to an equisatisfiable form $K_1 \lor K_2$, where each $K_i$ is a $\Sigma_i$-clause, for $i = 1, 2$. If the negation $\overline{K_1} \land \overline{K_2}$ is unsatisfiable, then there is an interpolant formula $Q$ in $\Sigma_1 \cap \Sigma_2$ such that $K_1$ entails $Q$ and $Q \land \overline{K_2}$ is unsatisfiable. Since $Q$ is in the empty theory and over the shared variables in $K_1$ and $K_2$, it is in the theory of pure equality. If the theories $T_1$ and $T_2$ are stably infinite, then by quantifier elimination, there must be a quantifier-free interpolant formula $Q$ in the free variables shared by $K_1$ and $K_2$. Such a $Q$ can constructed by testing each possible arrangement of the free variables into equivalence classes. Each such arrangement can be represented as a conjunction of equalities and disequalities. For example, the conjunction $x = y \land y = z \land x \neq u$ represents two equivalence classes, $\{x, y, z\}$ and $\{u\}$. Then $\overline{K_1} \land \overline{K_2}$ is unsatisfiable if for each arrangement $C$, either $K_1 \land C$ or $K_2 \land C$ is unsatisfiable. Each of the latter checks for unsatisfiability uses the satisfiability procedure for the individual theories. The interpolant $Q$ is $\bigvee\{C \mid K_1 \land C$ is satisfiable$\}$ since every model of $\overline{K_1}$ satisfies some disjunct in $Q$.

Model checking first-order logic formulas relative to finite models is a PSPACE-complete problem. Note that it subsumes QBF satisfiability. The complexity of checking a model for a fixed formula with respect to the size of the model, i.e., the model complexity, corresponds to $\text{AC}^0$, the class of bounded depth circuits of polynomial size. This problem is related to database query evaluation and constraint solving, and plays a central role in descriptive complexity theory [Imm99]. Modal logics can be translated to first-order logic by introducing an explicit accessibility relation. However, first-order logic is not expressive enough to capture concepts such as finiteness, the transitive closure of a relation, or fixpoints of monotone predicate transformers. Adding least and greatest fixpoints (see Chapter ?? of this Handbook) to first-order logic enhances the expressiveness, but is still not enough to capture finiteness [Par76].

2.3 Higher-Order Logic

Some of the limitations in the expressiveness of first-order logic can be overcome in higher-order logic (HOL) [And86,Lei94] which extends quantification to functions and predicates. This kind of quantification is already present in first-order logic since the predicate and function symbols in a formula are assumed to be implicitly universally quantified. To avoid inconsistencies arising from paradoxes due to the self-application
of predicates to predicates, higher-order logic is typed according to a strict hierarchy of types. Otherwise, we could have a predicate \( R \) that is defined so that \( R(X) = \neg X(X) \), and we have an inconsistency when \( X \) is instantiated with \( R \) itself. Modern higher-order logics are based on Church’s simple theory of types [Chu40]. The types are built from the base types of individuals \( \iota \) and propositions \( o \) using the function type constructor \( \tau_1 \rightarrow \tau_2 \) which is the type of maps from elements of type \( \tau_1 \) to type \( \tau_2 \). Higher-order logic formulas can be constructed from equality operators of type \( \tau \rightarrow \tau \rightarrow o \) for each type \( \tau \), using application \( s \, t \) and lambda abstraction \( \lambda(x : \tau) : s \). A quantified formula \( \forall(x : \tau).P \) can be defined as \( (\lambda(x : \tau).P) = (\lambda(x : \tau).\top) \), where \( \top \) is itself defined using prefix notation as \( = (\_) (\_) \), where the second and third occurrences of ‘=’ have type \( \iota \rightarrow \iota \rightarrow o \), and the first occurrence has type \( [\iota \rightarrow \iota \rightarrow o] \rightarrow [\iota \rightarrow \iota \rightarrow o] \rightarrow o \). A type \( \tau \) has order 1 if it is either \( \iota \) or \( o \), and it has order \( i + 1 \) if it is of the form \( \tau_1 \rightarrow \tau_2 \), where \( \tau_1 \) has order at most \( i \) and \( \tau_2 \) has order at most \( i + 1 \). An \( i \)’th order logic admits quantification over variables with types of \( i \)’th order. Thus second-order logic allows quantification over variables of type \( \iota \rightarrow o \) and \( \iota \rightarrow \iota \rightarrow \iota \) in addition to quantification over first-order types.

Sets with elements of type \( \tau \) in higher-order logic are just predicates on \( \tau \) which have the type \( \tau \rightarrow o \). The subset ordering on sets can also be easily defined. With this, the set of natural numbers can already be defined in second-order logic as the least set containing \( 0 \) that is closed under the successor operation. Finiteness is easily expressed since a set is finite iff every injective map on it is surjective, or equivalently, if there is no bijection between the set and any proper subset. The definition of natural numbers is an instance of a fixpoint definition. The least and greatest fixpoint operators \( \mu \) and \( \nu \) can be defined in higher-order logic using the Knaster–Tarski theorem. Compared to first-order logic, modal logics can be semantically embedded in a higher-order logic by directly defining the modal operators. Model checking queries can be expressed with this embedding and model checkers can be used as decision procedures for such queries [RSS95].

Higher-order logic lacks many of the metatheoretic properties of first-order logic. Since it can be used to define natural numbers, higher-order logic is not complete with respect to the standard semantics of function types \( \tau_1 \rightarrow \tau_2 \) as the set of all maps from the set interpreting \( \tau_1 \) to the one interpreting \( \tau_2 \). It is complete with respect to the more liberal Henkin interpretations. Properties like compactness and interpolation fail even in second-order logic. However, higher-order logic offers greater convenience for formalizing and manipulating concepts in mathematics and computing, and for embedding various logics and theories. Furthermore, the useful properties of first-order logic can always be invoked for the first-order fragment of higher-order logic.

2.4 PVS: Computation and Deduction

Logics are typically designed as idealized objects of mathematical study than for actual use in formal modeling and reasoning. Further features are needed to support facile expression and efficient proof construction. To illustrate some of pragmatic aspects of using logic, we examine a some of the language and deductive features of the Prototype Verification System (PVS), a specification and verification framework based on higher-order logic [ORSvH95]. The specification language extends higher-order logic with
polymorphic equality, conditionals, and updates, as well as with predicate subtypes, dependent types, parametric theories, and theory interpretations. These features, several of which were originally introduced in the EHDM specification language [RvHO91], bring the language closer to a mathematical vernacular than the textbook presentations of first-order and higher-order logic.

The language and deductive features in PVS can be illustrated by the example in Figure 6. The theory `bubble` takes a parameter `N` that is declared to be a natural number which is itself a subtype of the integers defined as `{i : int | i >= 0}`. The subtype `below(N)` is a possibly empty parametric predicate subtype consisting of the natural numbers in the subrange from 0 to `N-1`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from 0 to `m`.
tions into theories. In PVS, parametric theories are used to package axioms, definitions, and theorems in a reusable form. Examples of such packages include algebras such as groups, parametric datatypes such as lists over an element type, and polymorphic algorithms. Theory interpretations allow types and symbols in abstract theories to be instantiated by concrete interpretations that satisfy the axioms. Such interpretations can be used to demonstrate consistency or as a way of reusing an abstract development, such as a theory of groups, on a concrete group. Predicate subtypes are quite customary in mathematical vernacular. For example, the even numbers are a subtype of the integers, which are in turn a subtype of the rational numbers, and the latter are a subtype of the real numbers. Subtyping not only includes first-order subtypes such as nonzero numbers (needed to ascribe a type to the denominator of the division operation), subranges, prime numbers, and Mersenne primes, but also extends to higher-order concepts like monotone maps, injections, continuous functions, and group homomorphisms. Type-checking with subtypes detects many significant errors and even gaps such as, for example, claiming that \( \frac{n(n-k)!}{k!} \) is an integer without proper justification.

Proofs in PVS are constructed interactively by invoking proof commands on a goal to generate subgoals. These commands can either be primitive steps, including those that invoke external simplifiers such as SAT/SMT solvers, a Boolean simplifier using Binary Decision Diagrams (BDDs), a mu-calculus model checker, and a predicate abstractor, or they are compound proof strategies that can be defined by the end-user. Other proof assistants that offer practical support for proof construction include ACL2 [KMM00], Coq [BC04], HOL [GM93], Isabelle [NPW02], and Nuprl [CAB+86]. Liquid type systems [RKJ08] explore type inference with predicate subtypes using predicate abstraction (see Section 3.3).

### 2.5 Hoare Logic

As mentioned in Section 1, a Hoare triple has the form \( \{P\} S \{Q\} \), where \( P \) and \( Q \) are assertions containing logical variables and program variables, and \( S \) is a program statement. If the logic variables are drawn from a set \( X \) and the program variables from a finite set \( Y \), \( \Sigma \) is a first-order signature, \( e \) ranges over \( \Sigma[Y]-\)terms, and \( S \) ranges over program statements, where a program statement is one of

1. A skip statement \( \text{skip} \).
2. A simultaneous assignment \( \overline{y} := \overline{e} \) where \( \overline{y} \) is a sequence of \( n \) distinct program variables, \( e \) is a sequence of \( n \) \( \Sigma[Y]-\)terms.
3. A conditional statement \( e? S_1 : S_2 \), where \( C \) is a \( \Sigma[Y]-\)formula.
4. A loop while \( e \) do \( S \).
5. A sequential composition \( S_1 ; S_2 \).

If \( P, Q, R \) range over program assertions which are \( \Sigma[X \cup Y]-\)formulas, then the inference rules of the Hoare calculus are as shown in Figure 7.

The semantics can be given relative to a \( \Sigma \)-structure \( M \) which provided the interpretation of the symbols in \( \Sigma \). A state \( \sigma \) maps each program variable \( y \) in \( Y \) to a value in \( \text{dom}(M) \). A \( \Sigma[Y]-\)expression \( e \) has the value \( M[\overline{e}]\sigma \) where \( M[\overline{y}]\sigma = \sigma(y) \) and \( M[\overline{f}(e_1, \ldots, e_n)]\sigma = M(f)(M[e_1]\sigma, \ldots, M[e_n]\sigma) \). The meaning \( M[S] \) of a statement \( S \) is given by a set of sequences (of length at least 2) of states as shown below.
Here, \( \psi \) range over such a sequence of states of the form \( \psi[0], \ldots, \psi[n-1] \). The singleton sequence consisting of state \( \sigma \) is written as \( \sigma \) itself. The sequence \( \psi_1 \circ \psi_2 \) is the concatenation of \( \psi_1 \) and \( \psi_2 \).

1. \( \sigma \circ \sigma \in M[\text{skip}] \), for any state \( \sigma \).
2. \( \sigma \circ \sigma \in M[\text{skip}] \), for any state \( \sigma \).
3. \( \psi_1 \circ \sigma \circ \psi_2 \in M[S_1 \cup S_2] \) if \( \psi_1 \circ \sigma \in M[S_1] \) and \( \sigma \circ \psi_2 \in M[S_2] \).
4. \( \psi \in M[C \land S_1 : S_2] \) if either \( M[C]\psi[0] = \top \) and \( \psi \in M[S_1] \), or \( M[C]\psi[0] = \bot \) and \( \psi \in M[S_2] \).
5. \( \sigma \circ \sigma \in M[\text{while } C \text{ do } S] \) if \( M[C]\sigma = \bot \).
6. \( \psi_1 \circ \sigma \circ \psi_2 \in M[\text{while } C \text{ do } S] \) if \( M[C](\psi_1[0]) = \top, \psi_1 \circ \sigma \in M[S], \) and \( \sigma \circ \psi_2 \in M[\text{while } C \text{ do } S] \).

**Soundness and Completeness.** If \( \sigma \) is an \( M \)-assignment for the variables in \( Y \) and \( \rho \) is an \( M \)-assignment for variables in \( X \), then a \( \Sigma[X \cup Y] \)-formula \( P \) is interpreted as \( M[P] \), where \( M(y) = \sigma(y) \) for \( y \in Y \), and \( M(x) = \rho(x) \) for \( x \in X \). A Hoare triple \( \{ P \} S \{ Q \} \) is **valid** in a \( \Sigma \)-structure \( M \) if for every sequence \( \sigma \circ \psi \circ \sigma' \in M[S] \) and any assignment \( \rho \) of values in \( \text{dom}(M) \) to logical variables in \( X \), either \( M[Q]_{\sigma'} = \top \) or \( M[Q]_{\sigma} = \bot \). The Hoare calculus is sound relative to any \( \Sigma \)-structure \( M \): every derivable triple is valid in \( M \).

The next step is to show that any valid Hoare triple is derivable. For this, we need the language \( \Sigma \) and its models to be expressible enough to capture the assertions needed to verify triples. The proof of a valid triple \( \{ P \} S \{ Q \} \) can be decomposed into the valid triple \( \{ wlp(S)(Q) \} S \{ Q \} \) and the valid assertion \( P \Rightarrow wlp(S)(Q) \), where \( wlp(S)(Q) \)
(the weakest liberal precondition) is an assertion such that for any $\psi \in M[S]$ with $|\psi| = n + 1$ and $\rho$, either $M[Q]_{\psi,\rho} = \bot$ or $M\downarrow wlp(S)(Q)]_{\psi,\rho} = \top$. If for each $S$ and $Q$, there is an $R$ such that $R = wlp(S)(Q)$, then the triple $\{R\}S\{Q\}$ can be verified by the Hoare calculus. This follows by induction on the structure of $S$. It can be checked, for example, that if $Q = wlp(while\ C\ do\ S)(P \land \neg C)$, then the triple $\{Q \land C\}S\{Q\}$ is valid, because $Q \land C \Rightarrow wlp(S)(Q)$, and $Q \land \neg C \Rightarrow P$. The valid implication $P \Rightarrow wlp(S)(Q)$ above might be unprovable because the underlying assertion logic is incomplete. The Hoare logic is relatively complete since it reduces the provability of a valid triple to a set of valid proof obligations in the assertion logic [Coo78,Apt81].

If we pick the assertion logic to be Presburger arithmetic (see Section 2.2), then one can write programs in this language for which the needed assertions cannot be expressed. Rules like Composition and Consequence require new assertions to be invented, and these might not be in the same language. For example, Figure 8 shows a Hoare triple of the form $\{P\}S\{Q\}$, where the program $S$ consists of two consecutive loops where the first loop multiplies a non-negative integer $k$ and a positive integer $m$ by successive addition to compute $j$, and the second loop divides $j$ by $m$ through successive subtraction to obtain $i$. The precondition uses two logic variables $x$ and $y$ to bind the initial values of $k$ and $m$, respectively. The post-condition asserts that the final value of $l$ is the same as the initial value of $k$ which is bound to the logic variable $x$. The correctness argument for the triple requires intermediate assertions involving multiplication and therefore cannot be conducted in Presburger arithmetic.

If we use the first-order theory of arithmetic over the signature consisting of the constants 0 and 1 and the operations $+$ and $\times$, the first source of incompleteness, namely the inexpressiveness of the assertion language, vanishes. This is not merely because we can express multiplication but because we can capture computation. The inductive definition of the set of computation sequences needed for defining $wlp(S)(Q)$ above can be codified in a number of theories, e.g., first-order arithmetic through Gödel numbering. If we assume that there are $n$ program variables $y_0, \ldots, y_{n-1}$ from $Y$, then each state can be represented as a sequence of numerals $k_0, \ldots, k_{n-1}$ representing the state $\sigma$ when $\sigma(y_i) = k_i$ for $0 \leq i < n$, where $k_i$ is the numeral in first-order arithmetic representing the number $k_i$. The sequence $k_0, \ldots, k_{n-1}$ representing $\sigma$ can itself be encoded as a number, as can the sequence of states $\psi$. Let $\psi$ represent the numeral corresponding to the arithmetic encoding of the sequence $\psi$, so that the operation $first(\psi)$ and $last(\psi)$ represent the numerals for the encoding of the first and last states, respectively. For any statement $S$, we can define an arithmetic predicate $p_S$ such that $M \models p_S(\psi)$ iff $\psi \in M[S]$. With such a predicate, we can formalize the weakest liberal precondition as

$$wlp(S)(Q) = \exists z. \overline{\sigma} = first(z) \land p_S(z) \land Q[\text{last}(z)/\overline{\sigma}],$$

where $\overline{\sigma}$ captures the assertion $\bigwedge_{i=0}^{n-1} y_i = k_i$ when $\sigma$ encodes the sequence $k_1, \ldots, k_{n-1}$, and $Q[\text{last}(z)/\overline{\sigma}]$ is the result of substituting $k_i$ for $y_i$ in $Q$ when $\text{last}(z)$ encodes the sequence $k_0, \ldots, k_{n-1}$. A theory such as first-order arithmetic is said to be expressively complete since it can express $wlp(S)(Q)$ for any first-order arithmetic assertion $Q$.

The Hoare calculus above features a simple programming language defined over a state consisting of a finite set of variables. Many extensions have been developed to account for a range of language features. Separation logic is especially interesting since
it supports local assertions about heap-allocated structures like arrays, linked lists, and trees [Rey02].

3 Deduction and Model Checking

We have outlined the relationship between deduction and model checking from propositional to higher-order logic. We now examine the interaction between the two approaches on specific verification problems and techniques.

3.1 Abstract interpretation

Abstract interpretation [CC77,NNH01] is used to compute program properties such as the possible signs or intervals of values assigned to a variable and shapes of data structures, by using an abstract lattice for approximating program behavior. A lattice is a partially ordered set that is closed under the meet operation \( x \sqcap y \) and join operations \( x \sqcup y \) which are respectively the greatest lower bound and least upper bound of \( x \) and \( y \).

A complete lattice is a partially ordered set that contains greatest lower bounds and least upper bounds of arbitrary subsets. From the Knaster–Tarski theorem, we know that any monotone operator on a complete lattice has a complete lattice of fixpoints, including a least and greatest fixpoint.

Any program semantics can be seen as a concrete lattice, a Boolean lattice where the partial order is the subset ordering. For example, the trace semantics for while-programs presented in Section 2.5 can be viewed as a concrete lattice. The trace semantics could itself be abstracted by the set of states that precede each statement in the program. A further abstraction would be to simply consider the set of values that a variable can take. For example, in the program in Figure 8, the variable \( k \) takes only a single value indicated by the logic variable \( x \), the variable \( i \) ranges between 0 and \( x \), and the variable \( j \) takes on values that are multiples of \( y \) between 0 and \( x \times y \). The program actions can be mapped to a transfer function \( F \) on the lattice. The range of values of a variable, say \( i \), can be computed as the least set \( I \) such that \( F_i(I) \subseteq I \), where \( F_i \) is the transfer function for the variable \( i \). As an approximation, we could also just compute a post-fixpoint by finding some \( I \), not necessarily the least one, such that \( F_i(I) \subseteq I \). For example, the initial range of values for the variable \( k \) is \( \{x\} \), and this remains the range of values for \( k \) after each execution. For \( i \), initial set of values is the empty set, and there are statements that add 0 to it, that add \( z + 1 \) if \( z \) is in the range and \( z < x \) (given that the range of \( k \) is \( \{x\} \)) so that we can say that the value of \( i \) is always in the range \( [0, x] \). The latter range could be computed by using the lattice of intervals as an abstract domain. We can also use the sign lattice \( \{\ominus, 0, \oplus, \top\} \) discussed below to interpret the program in order to demonstrate that the possible values of \( j \) are non-negative.

The computation of an approximation to a fixpoint on a concrete lattice can be mapped to the computation of fixpoints on an abstract lattice through a Galois connection. Let \( \langle C, \leq, \sqcap \rangle \) be a concrete lattice and \( \langle A, \sqsubseteq, \sqcup \rangle \) be an abstract lattice. A Galois connection is a pair \( (\alpha, \gamma) \) of maps: \( \alpha \) from \( C \) to \( A \), and \( \gamma \) from \( A \) to \( C \), such that for any \( a \in A \) and \( c \in C \), \( \alpha(c) \sqsubseteq a \iff c \leq \gamma(a) \). Intuitively, \( \gamma(a) \) is the (largest) concretization of \( a \), and \( \alpha(c) \) is the (strongest) abstraction for \( c \), on the respective partial
orders. Note that \( c \leq \gamma(\alpha(c)) \) and \( \alpha(\gamma(a)) \subseteq a \) for any \( c \in C \) and \( a \in A \). The maps \( \alpha \) and \( \gamma \) are order-preserving. If \( F \) is a monotone operator on \( C \), then \( \mu F \) is the least fixpoint of \( F \) in \( C \) and can be defined as \( \bigcap \{ X | F(X) \leq X \} \). It is possible to define an abstract operator \( \hat{F} = \alpha \circ F \circ \gamma \) from a concrete operator \( F \). It is easy to see that \( \mu F \leq \gamma(\mu \hat{F}) \). Furthermore, if \( \hat{F}(a) \subseteq G(a) \) for all \( a \in A \), then \( \mu F \leq \gamma(\mu G) \). Also, if \( Y \) is a post-fixpoint of \( \hat{F} \), then \( \gamma(Y) \) is a post-fixpoint of \( F \) and an inductively valid property of the concrete computation.

Deductive techniques for optimization can be used to compute the abstraction \( \alpha \) as a step in the construction of the abstract fixpoint \([SSM05,LAK+14]\) or to directly pre-compute the abstract transfer function \([CDH00,RSY04]\). This kind of precomputation can be illustrated using a sign abstraction for the integer domain given by an abstract domain \( D = \{ 0, \oplus, \ominus, \top \} \), where \( \gamma(0) \) is the set \( \{ 0 \} \), \( \gamma(\ominus) \) is the set of non-negative integers \([0, \infty)\), \( \gamma(\ominus) \) is the set of non-positive integers \((-\infty, 0]\), and \( \gamma(\top) \) is the set of integers \((-\infty, \infty)\). The operations \( + \) and \( - \) on integers can be lifted to the corresponding operations \( \oplus \) and \( \ominus \) on \( D \) as shown in Figure 9. The table of entries in Figure 9 can be precomputed using theorem proving. Each entry is computed by showing that on the concrete lattice, we have a transformer on the sets \( c_x \) and \( c_y \) of the possible values of variables \( x \) and \( y \), define \( c_x + c_y \) as \( \{ u + v | u \in c_x, v \in c_y \} \). Then, we can compute an abstract transfer \( \hat{\oplus} \) by defining \( \hat{x} \oplus \hat{y} \) to be the least element \( a \) in the sign lattice such that \( \gamma(\hat{x}) + \gamma(\hat{y}) \subseteq \gamma(a) \). The latter condition with, for example, \( \ominus \) for \( a, \oplus \) for \( \hat{x} \), and \( \ominus \) for \( \hat{y} \), corresponds to the proof obligation \( \forall x, y. x \geq 0 \land y \leq 0 \Rightarrow x + y \leq 0 \). This proof obligation is clearly not valid, whereas when \( a = \top \), the corresponding proof obligation is valid. Similarly, if the concrete operation is a condition, e.g., \( x > 3 \), then the transfer function can be computed as \( \hat{x} \cap \oplus \). The transfer function computation can be done in order of increasing precision to compute \( \ominus \hat{\oplus} \) before \( 0 \oplus \top \) so that the latter value already has a useful upper bound. We can also make use of strictness to compute entire rows or columns in one step so that \( \hat{x} \oplus \top = \top \oplus \hat{x} = \top \) for any \( \hat{x} \).

### 3.2 Symbolic Model Checking

We have already seen that modal and temporal formulas can be model checked with respect to a Kripke model. Such Kripke models represent transition systems that are described in symbolic form as the triple \( \langle W, I, N \rangle \), where \( W \) is the type of states, \( I \) is the initialization predicate on states, and \( N \) is a binary transition relation on states. It is often infeasible to compute the concrete Kripke model from the symbolic transition system. For this reason, model checking is carried out with a symbolic representation of

\[
\begin{array}{cccc}
+ & 0 & \oplus & \top \\
0 & 0 & \oplus & \top \\
\ominus & \ominus & \top & \top \\
\ominus & \ominus & \top & \top \\
\top & \top & \top & \top \\
\end{array}
\]

**Fig. 9.** Abstract versions of \( + \) and \( - \)
the model. The use of a symbolic representation somewhat blurs the distinction between
deduction and model checking. Reachability is a canonical model checking query and it has the
\( \mu \)-calculus form \( \mu X.I \sqcup post(N)(X) \). In order to compute the symbolic
representation of the set of reachable states, it is enough to have methods for computing

1. The least upper bound \( X \sqcup Y \) of two formulas
2. The image \( post(N)(X) \) of a formula \( X \), and
3. The equivalence operator \( X \equiv Y \) to check when the fixpoint has been reached

When the symbolic representation is propositional logic, these operations can be
efficiently implemented with BDDs, which are covered Chapters ?? and ?? of this
Handbook. SAT solvers (covered in Chapter ??) can also be used to compute these
operations – for example, the image computation can be carried out using a variant of
AllSAT which generates a representation of all possible feasible solutions for a subset of
the variables. When the initialization predicate, transition relations are expressible with
Boolean combinations of difference constraints, the image computation can be com-
puted using quantifier elimination. Other symbolic representations for model checking
are covered in Chapter ?? of this Handbook.

3.3 Predicate Abstraction

In abstract interpretation, fixpoints are computed dynamically on an abstract lattice
by pre-computed transfer functions on this lattice. Fixpoints such as reachability on
infinite-state systems as well as large finite-state systems can be computed by approx-
imating these systems with a smaller system on which we can apply explicit-state or
symbolic model checking. In predicate abstraction [SG97,SS99] (covered in Chap-
ter ??), we have a concrete transition system \( \langle W_C, I_C, N_C \rangle \) and this is approximated by
an abstract transition system \( \langle W_A, I_A, N_A \rangle \), where bits in the abstract system state \( W_A \)
represent sets of concrete states, or equivalently, predicates over the concrete states.
For a simple example, we use an instance of Lamport’s Bakery algorithm shown in
Figure 10. This is actually an infinite-state system since the ticket value is potentially
unbounded if each process exiting the critical section races around to grab a ticket while
the other process is either in its critical section or waiting to enter it.

Despite being an infinite state system, the mutual exclusion property can be ver-
ified by observing specific predicates that already occur in the transition system, i.e.,
If we label these five predicates by Boolean variables \( c_1, c_2, z_1, z_2, p \), we can build an
abstract transition system in these Boolean variables that tracks the behavior of these
predicates in the concrete transition system. With this abstraction, the initialization
\( I_C(1) \) for process 1 becomes \( I_A(1) = c_1 = false; z_1 = true; p = true \), and that
for process 2 becomes \( I_A(2) = c_2 = false; z_2 = true \). The transition relation \( N_C \)
can also be abstracted as \( N_A \) with respect to the predicates above. For example, the first
transition of process 1 can be tracked by \( z_1 \rightarrow z_1 := false; p := false \). Once the
abstract finite-state transition system has been constructed, it can be model checked. Such
abstractions can also be defined to reduce a finite-state system to a smaller one in order
to make the search space more manageable.
Initially
critical[1] = false; ticket[1] = 0

Transition
critical[1] \implies ticket[1] := 0;

critical[1] = false; ticket[1] = 0

Transition

Fig. 10. A two-process Bakery mutual exclusion protocol

More generally, any concrete formula \( P \) can be abstracted by a formula \( \alpha(P) \). For an abstract formula \( \bar{P} \), let \( \gamma(\bar{P}) \) be the concrete formula that results from substituting the concrete predicate for the abstract one. For example, the abstract formula \( z_1 \land \neg z'_1 \land \neg p \land c_1 \land c_2 = c_2 \land z'_2 = z_2 \), which corresponds to the abstract transition above and abbreviated as \( \bar{P} \), can be concretized by substituting the predicates corresponding to the Boolean variables, to get

\[
\gamma(\bar{P}) = \begin{align*}
critical[1] & \\
\land ticket[1] \neq 0 & \\
\land ticket[1] \leq ticket[2] & \\
\land critical'[1] = critical[1] & \\
\end{align*}
\]

We can then check that \( \vdash \gamma(\bar{P}) \Rightarrow P \), where \( P \) is the formula corresponding to the concrete transition. We can use deduction to construct \( \alpha(P) \) from \( P \) such that \( \vdash \gamma(\alpha(P)) \Rightarrow P \). This can be done by using an SMT solver with an AllSMT capability for constructing all solutions to the formula \( \exists \pi . P \land \bigwedge_{i=0}^{n} p_i = P_i \), where \( \pi \) is a sequence of the concrete variables and \( p_i \) is the Boolean variable abstracting the concrete predicate \( P_i \) [LNO06]. Then \( \alpha(P) \) can be defined as \( \neg \text{AllSMT}(\exists \pi . \neg P \land \bigwedge_{i=0}^{n} p_i = P_i) \). We can then argue that \( \vdash \gamma(\alpha(P)) \Rightarrow P \), since no truth assignment for \( \alpha(P) \) can be extended to a truth assignment for \( \neg P \) satisfying the abstraction map \( \bigwedge_{i=0}^{n} p_i = P_i \). Hence, it must be the case that \( \gamma(\alpha(P)) \), which is equivalent to \( \alpha(P) \land \bigwedge_{i=0}^{n} p_i = P_i \), entails \( P \).

A weaker abstraction can be constructed by using an abstract lattice consisting of monomials, i.e., conjunctions of literals, instead of the Boolean lattice of formulas [SG97, BMMR01]. As with abstract interpretation, we can also compute the abstract transfer function for each individual Boolean variable. Data abstraction, where a datatype is partitioned into a finite set of regions, as for example, the partitioning of...
the integers into positive numbers, negative numbers, and zero, can also be viewed as a form of predicate abstraction.

Predicate abstraction builds an abstract transition system that over-approximates the concrete one. Each abstract state \( a \) represents a set of concrete states \( \{ c | c \models \gamma(a) \} \). The abstract transition system is defined by an \( \exists \exists \) abstraction such that there is a transition \( N_A(a, a') \) between two abstract states \( a \) and \( a' \) whenever there exist concrete states \( c \) and \( c' \) such that \( c \models \gamma(a) \), \( c' \models \gamma(a') \), and \( N_C(c, c') \) holds. Thus, if the set of abstract reachable states \( \mu X.I \subseteq post(N)(X) \) contains a state \( a' \) that is reachable from an abstract initial state \( a \), then there is a path \( \langle a_0, \ldots, a_n \rangle \) with \( a = a_0 \) and \( a' = a_n \). Since we have an \( \exists \exists \) abstraction, for each adjacent pair \( \langle a_i, a_{i+1} \rangle \) with \( 0 \leq i < n \), there is a pair of concrete states \( \langle c, c' \rangle \) such that \( N(c, c') \) holds. However, this does not imply the existence of a concrete path of the form \( \langle c_0, \ldots, c_n \rangle \) such that \( c_i \models \gamma(a_i) \) since it is possible that for some \( a_{i+1} \), the set of states \( \{ c' | \exists c.c \models \gamma(a_{i+1}), c' \models \gamma(a_{i+1}), N(c, c') \} \) has an empty intersection with the set of states \( \{ c'' | \exists c.c \models \gamma(a_{i+1}), c'' \models \gamma(a_{i+1}), N(c, c'') \} \). The absence of a concrete path corresponding to an abstract path can be established by techniques similar to bounded model checking (see below). This means that abstraction must be refined so that the two sets of states are distinguishable. Symbolic methods for computing strongest post-conditions and weakest preconditions can be used for computing predicates needed for an abstraction that prunes spurious abstract counterexamples. The technique of interpolation can also be used for this purpose. Abstraction-based approaches to model checking are covered in Chapter ??.

The technique of \( \exists \exists \) abstraction can yield abstractions that over-approximate concrete behavior. In particular, every concrete computation can be simulated by an abstract one. Thus, for any \( \mu \)-calculus property \( P \) with universal path quantification, the concrete transition system satisfies \( P \) if the abstract transition system satisfies the abstraction \( \hat{P} \).

In particular, when proving an invariant \( P \) by showing that \( \mu X.I \sqcup post(N)(X) \Rightarrow P \), we can see that the latter formula has an existential path-quantified \( \mu \)-calculus formula occurring negatively, which is hence a universal path-quantified formula. The dual problem is that of under-approximating a \( \mu \)-calculus formula so as to preserve the validity of universal path-quantified \( \mu \)-calculus formulas. This requires a \( \forall \exists \) abstraction of the transition relation where we admit a transition between abstract states \( a \) and \( a' \) only when for every state \( c \models \gamma(a) \), there is a state \( c' \models \gamma(a') \) such that \( N(c, c') \). This means that whenever there is an abstract path \( \langle a_0, \ldots, a_n \rangle \) in the abstract transition system, there is a corresponding concrete path \( \langle c_0, \ldots, c_n \rangle \) through the state sets \( \langle \gamma(a_0), \ldots, \gamma(a_n) \rangle \).

### 3.4 Bounded Model Checking and \( k \)-Induction

Invariants are the most common class of properties about transition systems. An invariant is an assertion \( P \) that holds in every reachable state of the system, and the set of reachable states is indeed the strongest system invariant. Bounded model checking uses satisfiability to check a sequence of states of a transition system for states violating the invariant \( P \). For a given transition system \( \langle W, I, N \rangle \), the \( k \)-expansion of the system is a formula of the formula \( I(s_0) \land \bigwedge_{i=0}^k N(s_i, s_{i+1}) \), characterizes the sequences \( \langle s_0, \ldots, s_{k+1} \rangle \) representing an initial segment of \( k + 1 \) steps in a computation. The violation of an invariant can be represented by the formula \( I(s_0) \land N^k \land \bigvee_{i=0}^{k+1} \neg P(s_i) \). If the latter formula is satisfiable, then there is a reachable state where \( P \) does not hold.
As an exercise, the reader can verify the mutual exclusion property of the Peterson’s algorithm in Section 1. Recall that the transition system only had five Boolean variables, and hence at most $2^5$ distinct states. Therefore every state should be reachable in at most 31 steps. In practice, the maximal loop-free paths are significantly smaller. The bound can be expanded by iterative deepening by incrementing the parameter $k$ until a property violation is found. Since the bounded model checking proof obligation of $k + 1$ steps includes the constraints for $k + 1$ steps, the iterative deepening can retain the clauses learned in prior steps.

Bounded model checking can also be applied for checking liveness properties by specifying lasso-like counterexample traces that consist of a path from an initial state $s_0$ to a state $s_k$ and then a nontrivial loop about $s_k$ of length at least $m$, where a property $P$ fails to hold. A counterexample to $\diamond P$ can be constructed with a query of the form $I(s_0) \land \neg P(s_0) \land \bigwedge_{i=0}^{k-1} (\neg P(s_i) \land N(s_i, s_{i+1})) \land \bigwedge_{i=0}^{m} (\neg P(s_{i+k}) \land N(s_{i+k}, s_{i+k+1}) \land s_{k+1} = s_{k+m+1})$.

Satisfiability checking can also be applied to induction arguments. The customary technique of verifying an invariant $P$ for a transition system $\langle W, I, N \rangle$ is by showing that $P$ holds initially, i.e., $I(s_0) \land \neg P(s_0)$ is unsatisfiable, and that it is inductive, i.e., $P(s) \land N(s, s') \land \neg P(s')$ is unsatisfiable. As is well-known, the proof obligations can fail not because the invariant is not valid, but because it is not inductive. The inductivity proof obligation can fail because of a counterexample where the state $s$ is unreachable. In $k$-induction, the base case is identical to the $k$-step bounded model checking query, and the induction step checks the unsatisfiability of $P(s_0) \land \bigwedge_{i=0}^{k-1} (P(s_0) \land N(s_i, s_{i+1})) \land \neg P(s_{k+1})$. Since the state $s_k$ must be reachable in at least $k - 1$ steps, this rules a number of unreachable steps. The usual technique for proving invariance is then just 1-induction. As with bounded model checking, the bound $k$ can be iteratively increased. For example, the mutual exclusion property of the algorithm in Figure 2 can be proved by 5-induction without the need for any invariant strengthening.

Bounded model checking and $k$-induction can also be applied to infinite-state systems by using SMT solvers instead of SAT solvers for checking satisfiability. This approach can be used to handle systems with timed behavior and datatypes such as integers, rationals, reals, sets, arrays, and recursively defined structures such as lists and trees. Iterative deepening with SMT solvers can exploit the retention of learned clauses as well as theory lemmas. Brown and Pike’s verification [PB06] of the biphasic mark and 8N1 communication protocols using the SAL model checker [dMOR+04] illustrates the power of infinite-state $k$-induction. Bounded model checking is covered in Chapter ?? of this Handbook.

3.5 Symbolic Execution and Test Generation

Bounded model checking demonstrates the use of satisfiability solvers for generating counterexamples traces. The same technique can also be used for the symbolic execution of paths through a program as a mechanism for generating test cases. In this case, the goal is to find a test case that reaches a particular control point in a program. For example, with the program in Figure 1, we may want a test case where the program terminates with one unrolling of the loop. We wish to find an input such that the assertion
max = 0 holds at termination. By depth-first search, we can find a path of the form
\[ \max_0 = 0; i_0 = 0; (i_0 \leq N); \neg (a[1] > \max_0); i_1 = i_0 + 1; (i_1 \leq N); (\max_0 = 0). \]

An SMT solver can be used to determine that \( N = 0, a[0] = 0 \) is a possible input. Unlike bounded model checking where the entire transition system is represented as a formula, symbolic execution generates the constraints corresponding to specific paths through the control flow graph of the program. It can be combined with random testing to generate inputs that find inputs for paths not covered by the given inputs. One can also use concrete inputs in conjunction with symbolic execution to, for example, convert nonlinear constraints into linear ones to simplify constraint solving. This connection to test generation is explored in depth in Chapter ?? of this Handbook.

### 3.6 Interpolation-Based Model Checking

We have already seen that whenever we have two sets of formulas that are jointly unsatisfiable, interpolation can be used to construct a formula in the shared language that captures the essence of the unsatisfiability. With bounded model checking, we see that the failed attempts to find a counterexample essentially amount to the BMC query being unsatisfiable. A BMC query has the form \( I(s_0) \land \bigwedge_{i=0}^{k} N(s_i, s_{i+1}) \land \bigvee_{i=0}^{k} \neg P(s_i). \)

For any given \( j < k \), we can split the query into the form \( I(s_0) \land \bigwedge_{i=0}^{j} N(s_i, s_{i+1}) \land \bigvee_{i=0}^{j} \neg P(s_i) \) and \( \bigwedge_{i=j+1}^{k} N(s_i, s_{i+1}) \land \bigvee_{i=j+1}^{k+1} \neg P(s_i). \) The shared language consists of the variables of state \( j + 1 \) so that an interpolant yields an assertion about state \( j + 1 \) that is implied by the computation up to that point. Specifically, if we let \( j \) be 0, then we get an assertion \( Q(s_1) \) that is implied by \( I(s_0) \land N(s_0, s_1) \land \neg P(s_0). \) This means that instead of increasing the bound, we replace the initialization constraint by \( I^1(s_0) = I(s_0) \lor Q(s_0). \) If by progressively weakening the initialization predicate in this proof-directed manner, one reaches a fixpoint where \( I^{i+1}(s_0) = I^i(s_0) \), then this predicate is an invariant which implies the desired property \( P. \)

Interpolation is covered in Chapter ?? of this Handbook. It can also be used for abstraction refinement in conjunction with predicate abstraction.

### 3.7 Property Directed Reachability

We have already seen that inductive invariants are critical for deductive verification. The key to constructing inductive invariants is to find an over-approximation to the set of reachable states that is stronger than the desired property. Bradley’s Property Directed Reachability (PDR) [Bra11,Bra12,EMB11] method searches for an inductive invariant \( Q \) for a transition system \( (W, I, N) \) that is stronger than the desired invariant \( P. \) It constructs a sequence of state predicates \( Q_0, \ldots, Q_n \), where the \( i \)'th predicate over-approximates the set of states reachable in \( i \) or fewer steps. Here, \( Q_0 \) is just the initialization predicate \( I. \) Each predicate \( Q_i \) in this sequence except \( Q_n \) is stronger than \( P, \) that is, for each \( s, Q_n(s) \Rightarrow P(s) \). Furthermore, \( Q_i \) is stronger than \( Q_{i+1}. \) and the image \( N[Q_i] \) of \( Q_i \) under \( N \) is also stronger than \( Q_{i+1} \) for \( 0 \leq i < n. \) The PDR algorithm also maintains for each \( i, 0 \leq i \leq n, \) a set of counterexamples \( C_i, \) a set of state predicates such that for each predicate \( R \) in \( C_i, \) and each \( s \) such that \( R(s), \neg P(s) \)
such that 

holds. Also, for each \( R \) in \( C_i, 0 \leq i < n \), there is an \( R' \) in \( C_{i+1} \), where for each \( s \) such that \( R(s) \), there is a \( s' \) such that \( Q_i(s) \land N(s, s') \land R'(s') \) holds. Each \( R \) in \( C_i \), for some \( i \), is a counterexample to inductivity (CTI) that is part of a symbolic trace \( R_i, \ldots, R_n \) where each \( R_j \) is in \( C_j \), for \( i \leq j \leq n \), and \( N[R_k] \) is stronger than \( R_{k+1} \), for \( i \leq k < n \).

Initially, the sequence consists of just one predicate \( Q_0 \). In each subsequent step, the sequence \( Q_0, \ldots, Q_n \) is progressively strengthened to eliminate the counterexamples to inductivity in one of the following ways:

1. **Fail**: If \( C_0 \) is nonempty, then we have found a counterexample trace leading from an initial state to a state satisfying \( \neg P \).
2. **Succeed**: If \( C_{i+1} \) is stronger than \( C_i \) for some \( i \), then we have found the inductive invariant \( Q = Q_i \), strengthening \( P \).
3. **Extend**: Otherwise, if each \( C_i \) is empty, then \( Q_n \) must strengthen \( P \), and since we have not yet found an inductive invariant, we extend the sequence by letting the new state \( Q_{n+1} \) be an over-approximation of \( N[Q_n] \).
4. **Refine**: Otherwise, for some \( i + 1 \), \( C_{i+1} \) contains a counterexample \( R' \), and we check if the query \( Q_i(s) \land \neg R'(s) \land N(s, s') \land R'(s') \) is satisfiable. This query is equivalent to checking the validity of \( Q_i(s) \land \neg R'(s) \land N(s, s') \Rightarrow \neg R'(s') \), that is, checking if \( \neg R' \) is inductive relative to \( Q_i \). It is safe to assume \( \neg R'(s) \) since if \( \neg R'(s) \land Q_i(s) \) were satisfiable, then the corresponding source counterexample \( R'_0 \) in the symbolic trace would already be reachable at \( Q_{n-1} \), in which case \( Q_n \) would not have been added by the **Extend** rule.
   
   (a) **Strengthen**: If the query is unsatisfiable, then we find some weakening \( \tilde{R} \) of \( R' \), such that \( Q_i(s) \land \neg \tilde{R}(s) \land N(s, s') \land \tilde{R}(s') \) is unsatisfiable, and strengthen each \( Q_j \) for \( 0 < j < i + 1 \) with \( \neg \tilde{R} \). When we do this, we can also remove any counterexample \( R \) from \( C_j \) if \( Q_j(s) \land R(s) \) is unsatisfiable.
   
   (b) **Reverse**: If the query is satisfiable, then we construct a counterexample \( R \) where for each \( s \) such that \( R(s), Q_i(s) \land \neg R'(s) \land N(s, s') \land R'(s') \) is satisfiable, and add \( R \) to \( C_i \). The counterexample \( R \) must not overlap with other counterexamples already in \( C_i \).

Each step of the algorithm either fails with a counterexample trace (**Fail**), finds an inductive invariant (**Succeed**), extends the sequence when there are no counterexamples (**Extend**), strengthens some \( Q_i \ (**Strengthen**), or adds a new counterexample to some \( C_i (**Reverse**. The algorithm converges if we can place an upper bound on the number of times these steps can be executed.

For a transition system where the state variables \( b_1, \ldots, b_m \) are Boolean, each \( Q_i \) can be represented as a set of clauses in these variables, and \( Q_{i+1} \subseteq Q_i \), for \( 0 < i < n \). Here, \( Q_i \) represents the state predicate mapping the state \( s \) consisting of an assignment of truth values to \( b_1, \ldots, b_m \) to the truth value of the conjunction of the clauses in \( Q_i \). The transition relation \( N \) is a Boolean formula in the variables \( b_1, \ldots, b_m, b'_1, \ldots, b'_m \). The primed version \( K' \) of a clause \( K \) is the result of replacing each \( b_i \) in \( K \) by \( b'_i \) for \( 1 \leq i \leq m \). The image \( N[Q_i] \) consists of the clauses \( K \) in \( Q_i \) such that \( Q_i \land N \land \neg K' \) is unsatisfiable. Each counterexample \( R \) is a cube, a conjunction of literals, in the variables \( b_1, \ldots, b_m \). The PDR algorithm converges in this case because we cannot have
a monotonically weakening sequence of predicates without finding a duplicate pair of adjacent predicates in the sequence, namely an inductive invariant.

4 Conclusions

Deduction and model checking are complementary approaches to verification. The former relies on assertions and program structure to construct proofs, whereas the latter is based on actually computing exact or approximate fixpoints. Model checking is effective at computing small properties of large systems, whereas deduction is needed for establishing correctness properties that require nontrivial mathematical reasoning. There is considerable synergy in terms of the reasoning tools: SAT and SMT solving, interpolants, quantifier elimination procedures, and decision procedures. We have outlined the foundations of deductive verification and explored only a few of the rich range of connections to model checking. It should be clear from the examples shown here that deduction is already a key element of many modern verification algorithms. In the algorithms that we have covered, deductive methods are used for symbolic execution, bounded state exploration, predicate abstraction, abstract transfer function construction, abstraction refinement, interpolant generation, model construction, and proof construction. It is reasonable to expect that many of the significant advances in automated verification will be sparked by the complementarity and synergy between deduction and model checking.

Acknowledgment. The article has been significantly improved by the valuable feedback from the anonymous reviewers, and by the thorough, insightful, and constructive corrections and suggestions from Sriram Sankaranarayanan. The author is deeply grateful to them, but takes full responsibility for any remaining mistakes or deficiencies.

References


