

Chapter 0

Introduction

Objective

Hardware and software systems are growing rapidly in scale and functionality. From smartcards to air-traffic controllers, computers are being deployed everywhere. As the complexity of a design grows, so does the likelihood of subtle errors it contains. While the complexity of designs has been growing rapidly, the underlying design methodology has evolved only marginally. Consequently, assuring reliability has become one of the key engineering challenges to sustain the ongoing computer revolution. While reliability has always been a concern in system design, some current trends are worth noting:

- Computers are used increasingly in safety-critical applications such as medical treatment and mission control. Presence of bugs in such applications has unacceptable consequences.
- Bugs found at later stages of design can be very expensive, an extreme case of which is illustrated by the notorious floating-point division error in Intel's Pentium processor.
- There is an ever-increasing pressure to reduce time-to-market. This calls for maximum automation of all stages of the design process, including debugging.
- The current trend in design of embedded systems is towards greater use of programmable components. This shifts the focus from low-level optimizations to high-level designs.

Motivated by these concerns, computer-aided verification aims to provide tools that help in detecting logical errors in early stages of design.

Formal verification

Formal methods seek to establish a mathematical proof that a system works correctly. A formal approach provides (1) a modeling language to describe the system, (2) a specification language to describe the correctness requirements, and (3) an analysis technique to verify that the system meets its specification. The model describes the possible behaviors of the system, and the specification describes the desired behaviors of the system. The statement the model P satisfies the specification φ is now a mathematical statement, to be proved or disproved using the analysis technique. The following two are distinguishing hallmarks of formal verification, as opposed to traditional techniques such as testing and simulation:

1. Formal: The intuitive correctness claim is made mathematically precise.
2. Verification: The goal of the analysis is to prove or disprove the correctness claim. It is not adequate to check a representative sample of possible behaviors as in simulation, rather a guarantee that all behaviors satisfy the specification is required.

Automated verification

Primary focus of this book is on analysis that can be performed algorithmically. Typically, such analysis is performed by an exhaustive simulation of the model on all possible inputs. The analysis is performed by a software tool, called a verifier, which determines whether the model satisfies the specification, and returns the answer Yes or No along with relevant diagnostic information. While automation has been central to the rising industrial interest in formal verification, automated analysis is not always possible. First, the analysis problem is typically undecidable. For instance, given a C program, determining whether it terminates or not, is a classical undecidable problem. On the other hand, if all the variables of a model are known to be of finite types, then the number of possible states of a system is finite, and the typical analysis problem is decidable. Second, decidability, in itself, does not imply feasibility of analysis. As we shall study, even the simplest analysis problem is computationally hard. In spite of the above mentioned difficulties, there is still a great deal of interesting analysis that can be performed by a verifier. Many recent advances have led to heuristics that make analysis feasible for interesting classes of systems. Furthermore, there is a great deal of flexibility in setting up the analysis problem, and thus, when the original analysis problem is infeasible, it can be simplified so that a verifier can solve it, and still provide useful feedback.

An alternative to automated verification is interactive verification. In interactive verification, the analysis problem is formulated as proving a theorem in a mathematical proof system, and the designer attempts to construct the proof of the theorem using a theorem prover as an aid. For instance, one can formulate the correctness of a sorting program using Floyd-Hoare logic, and use a theorem prover such as HOL to prove it. While this approach is very general, it requires considerable expertise, and the involved manual effort has proved to be an obstacle so far.

Reactive systems

The more conventional view of a computer program is the functional view: a program computes a function from inputs to outputs. For example, given a list of numbers as the input, a sorting program computes another list of numbers as the output. A more general view is the reactive view: a system is interacting with its environment accepting requests and producing responses. For instance, a computer network may be viewed as a reactive system that is accepting packets from different sources and delivering them at different destinations. In the reactive view, the computation of a system may not terminate. Such a reactive view is more appropriate to understand a variety of systems such as an operating system, a microprocessor, a telecommunications switch, and the world-wide web. In this book, we are concerned with modeling, specification, and analysis of reactive systems.

Modeling

The first step in formal verification is to describe the system in the chosen modeling language. A modeling language can be thought as a very high-level concurrent programming language. At this point, it is worthwhile to note the following features of modeling which distinguish it from programming:

- The purpose of a model, in the context of formal verification, is to describe control and interaction. In modeling, we usually avoid describing complex data structures and data manipulation. A modeling language provides a simple set of operations to build complex modules from simpler ones. This simplicity makes analysis more feasible, and helps to focus attention on how the components interact.

- A modeling language typically supports nondeterminism explicitly. For example, a model of a lossy buffer can be described using a choice: when a message is received by the buffer, the message is either discarded or stored. Consequently, unlike a program, a model may exhibit many possible behaviors even after all the inputs are specified.

- In modeling, the designer describes not only the system, but also the environment in which the system is supposed to operate. For instance, in a model of a traffic controller, the designer describes the controller together with its environment, namely, cars approaching the intersection from different directions. Explicit modeling of the environment is essential for meaningful analysis of the system. The model of the environment captures the assumptions about the manner in which the system is to be used.

- A system can be modeled at many different levels of abstraction. Thus, the same system may have different models. A model may be incomplete, and can specify only some of the components. Unlike programming, implementability is not the central issue in modeling.

Finite-state machines are widely used to describe control flow. In practice, many extensions of finite-state machines are used. Extended finite-state machines support variables. To describe a system consisting of interacting components, finite-state machines need to be equipped with a communication mechanism. For instance, in Mealy machines the edges are annotated with input and output symbols. Our modeling language is reactive modules which allows description of extended finite-state machines with communication mechanism rich enough to describe different types of interactions.

Requirements

For formal analysis, a designer needs to specify the correctness requirements separately from the model of the system. Requirements capture what the system is intended to do. Typical requirements are of two kinds. The first kind stipulates that the system should always operate within a safe set, or something bad never happens. For instance, for a traffic controller, the lights for cross-traffic in perpendicular directions should never be green simultaneously. The second kind stipulates that the system discharges its obligations, or something good will eventually happen. For instance, for a traffic controller, it is essential that a particular light does not stay red forever, and turns green eventually.

Requirements can be expressed formally in a mathematical logic. Pnueli proposed the use of temporal logic to specify requirements concerning behavior of a reactive system over time. Just as a formula of a propositional logic is interpreted over a valuation that assigns truth values to boolean variables, a formula of a temporal logic is interpreted over a sequence (or more generally, a tree) of valuations that capture the truth values of boolean variables at different instances of time. Temporal logic can express many requirements succinctly and naturally, and is well-suited for algorithmic analysis.

For writing specifications, an alternative to temporal logic is automata. In the automata-theoretic approach to verification, the system is viewed as a generator of a formal language, the specification is an acceptor defining a formal language, and the verification problem reduces to a language inclusion question: whether every behavior generated by the system automaton is accepted by the specification automaton. The theory of ω-automata—automata over infinite sequences—provides an elegant conceptual framework to study verification.

Model checking

Model checking means checking that the model satisfies the specification. The analysis is performed algorithmically by searching the state-space of the model. The term was coined by Clarke and Emerson in 1981 in the context of checking a finite model against requirements specified in a temporal logic called Computation Tree Logic (CTL). Today the term applies more generally: models need not be finite-state, and requirements can be written in a variety of other languages.

Model checking is computationally expensive even if we restrict attention to simple requirements. The problem is rooted in the fact that the number of states of a system grows exponentially with the number of variables used to describe it. This is the so-called state-space explosion problem. We will study a variety of techniques that alleviate this problem. In particular, symbolic model checking has proved to be quite effective in analyzing systems of practical interest.

Given a model and a requirement as input, a model checker does not simply answer Yes or No, rather, when the model does not satisfy the requirement, it produces a counter-example, an evidence for the failure. This diagnostic information is extremely useful for debugging purposes. Indeed, model checking is typically an iterative process. The designer starts with a model of the system, checks a variety of requirements, and uses the feedback to modify the model. This methodology is illustrated in Figure 1.

Hierarchical verification

In an ideal approach to design, the design begins with a very simple initial model. Model checking is used to debug the model. As the designer gains confidence in its logical correctness, the model is made more complex by adding details, and more requirements can be analyzed. During the refinement step, the designer would like to ensure some consistency between the models before and after adding the details. The relationship between two different models can be made mathematically precise by defining a refinement preorder over the set of models. Different formal methods advocate different views regarding when one model should be considered to be a refinement of another, that is, they differ in the semantics of the refinement relation. The refinement checking problem, then, is to verify that a detailed model is a refinement of the abstract model. Observe that refinement checking is like model checking where the same language is used to describe the model as well as the requirement. A hierarchical approach to design and analysis corresponds to constructing models at several

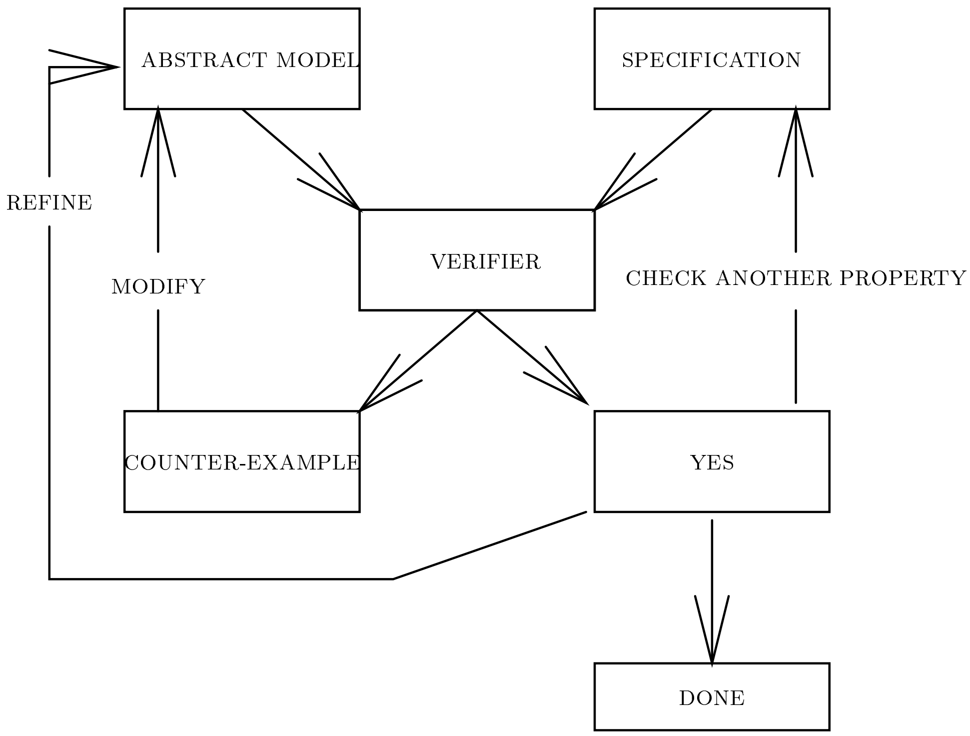

Figure 1: Model checking methodology

different levels of abstraction, and establishing each level to be a refinement of the next higher level.

The problem of refinement checking, while decidable for finite-state systems, is computationally intractable. However, the designer's intuition regarding the correspondence between the abstract and detailed model can be exploited to establish the refinement claim. Of particular interest to us will be the compositional methods in which the structure of the descriptions of the two models is used to decompose the required refinement claim into subclaims regarding refinement relationships among components of the two models.

Hardware verification

Model checking has been most successful in hardware verification. In 1992, the model checker SMV was used to pinpoint logical errors in the cache coherence protocol described in the IEEE Futurebus+ Standard. This, and numerous subsequent, case-studies attracted attention of hardware industry eager to enhance capabilities of design automation tools. Model checking seems suitable to debug intricate aspects of microprocessor designs. Today semiconductor companies such as Lucent, IBM, Intel, Motorola, and Siemens, have internal verification groups aimed at integrating formal verification in design flow, while CAD-tool vendors such as Cadence and Synopsis are exploring ways to add verification capability to design tools. Some verification tools are already commercially available, for instance, FormalCheck of Lucent Technologies.

An important reason for the success of model checking in hardware verification is the ease with which it fits into the existing design methodology. Hardware description languages such as VHDL and Verilog are extremely popular, and designers routinely use simulation and synthesis tools available for these languages. While these languages have not been designed with computer-aided verification in mind, a significant subset can be subjected to analysis.

Software verification

High-level design is somewhat uncommon in software development, and consequently, model checking has had limited influence in this domain. An important application has been design of protocols used in safety-critical applications such as aviation. Unlike in hardware design, there is no commonly accepted standard language that has precise mathematical semantics and is amenable to analysis. Increasing popularity of synchronous languages such as Esterel, and of Statecharts, a graphical formalism to describe hierarchically structured state machines, suggest a promising future.

Limitations

Computer-aided verification is only one of the many weapons available to a designer to ensure reliability. Let us note a few limitations to understand what it can do and what it cannot do.

- The application domain of computer-aided verification is control-intensive concurrent systems. It seems unsuitable to analyze, for instance, a database query manager, or a word processor.
- A model checker analyzes a model, and not the system itself. Even when the modeling language coincides with the implementation language (e.g. as in hardware design using VHDL), to make analysis feasible, a variety of simplifications are used. Consequently, the gap between the model and its implementation remains. This implies that the greatest strength of computer-aided verification is detecting bugs, rather than certifying absence of bugs.
- Formalizing requirements is a challenging task common to all formal methods. The designer can enumerate several requirements that the system is

supposed to satisfy, but usually cannot be sure that the list is complete. The gap between the intuitive understanding of correctness and its formalization in the specification language reasserts the applicability of model checking as a falsification, rather than certification, tool.

- With new heuristics, and with increasing speed and available memory on modern computers, we can hope to apply computer-aided verification to analyze systems beyond the scope of today's tools. However, the high computational complexity of the analysis problem is, and will remain, a hurdle.