Verifying Sequential Behavior with Model Checking

Armin Biere
Computer Systems Institute, ETH Zürich
ETH Zentrum, RZ H, CH-8092 Zürich, Switzerland
biere@inf.ethz.ch

Abstract

The design of state-of-the-art digital circuits often involves interacting state machines with very complex control flow. As consequence functional verification of sequential designs is becoming a major bottleneck in the design process. Model checking techniques, the topic of this tutorial, promise to speed up verification time by checking high level temporal properties. Model checking is best used in early design phases where it may help to catch fundamental design flaws and errors as early as possible.

Introduction

Digital systems are becoming ubiquitous in our daily lives. Mainly driven by consumer products and embedded systems, such as cellular phones, the number of produced circuits is steadily increasing. Additionally, more functions are integrated on each chip using smaller physical structures exploiting Moore's law. This exponential decrease in feature size will probably continue for at least a couple of decades.

These changes in production, larger demand and shorter development cycles result in increasing design and verification complexity which can not be handled by traditional means only. Manual inspection is too cumbersome and traditional testing can not provide enough coverage within reasonable time. The industry is facing a testing cost problem with testing often contributing more than 70% to development costs.

Several solutions to this problem have been proposed. The first step which has seen widespread acceptance in industry during the last decade is the adoption of synthesis tools for automatic generation and optimization of gate-level net lists from high level RTL descriptions. In combination with automatic place-and-route tools, this approach allows to concentrate on RTL designs with only a small penalty in space and time utilization.

Recently this idea has been taken a step further by using even higher level building blocks, e.g. assembling IP in the context of system on chip design. However when moving up in the abstraction hierarchy the approach relies on the ability to prove functional correctness of basic building blocks in an efficient way. Additionally, the tools for refining abstract specifications, such as compilers, can not be totally relied upon. The verification of implementation versus specification becomes necessary and may itself be costly using traditional techniques.

The use of Formal Methods is another orthogonal way to ensure functional correctness. They are all based on rigorous mathematical reasoning and will often be able to completely, i.e. with 100% coverage, verify certain aspects of functional behavior. However some formal methods may also be used in bug-finding-mode in addition or as alternative to traditional testing and simulation techniques. This tutorial concentrates on model checking, a technique for verifying sequential or temporal properties of state machines.

Other prominent formal methods are equivalence checking, e.g. [1], which is a restricted version of propositional property checking, and theorem proving in general. Propositional property checking uses efficient satisfiability procedures (SAT) [2] or compact representations for boolean functions such as binary decision diagrams (BDDs) [3]. SAT can be applied to large designs. Particularly, equivalence checking is mature enough to handle industrial sized circuits.

The main reason for the success of propositional property checking is its full automation. Commercial equivalence checking is a push-button technology. However, SAT works reasonably well for low-level properties only, such as checking that two similar circuits have the same combinational behavior. On the other side of the spectrum, theorem proving, e.g. [4], is able to prove arbitrary complex properties on a much higher abstraction level, but relies on human guidance, to find a proof. Frequent interaction with the user and

the requirement for highly specialized proof engineers with considerable mathematical training prevents the acceptance of theorem proving in industry so far.

Model checking [5] lies between both extremes. Its main objective is to check sequential behavior, such as protocol conformance. But alone it is not powerful enough to prove for instance generic structural properties. Model checking is fully automatic and automatically produces counterexamples, i.e. a simulation trace, if a property does not hold. On the other side model checking does not scale up to the same circuit size as propositional property checking. However recent years have seen various combinations of model checking with SAT and also theorem proving, in order to obtain the best of all approaches: scalability and expressiveness.

Property Specification

An important feature of model checking are various specification formalisms for sequential behavior. These formalisms fall into two categories, automata and temporal logic. From a practical point of view both can be translated into each other.

Commercial model checkers often come with predefined properties containing place holders, that can be filled in by the user to match the design to be checked. Such libraries of property templates make it possible to quickly adapt a model checker in practice. However, to master larger designs and more involved properties, it is necessary that the user of a model checker has some understanding of temporal logic or automata based specification.

Automata

The idea of automata based specification [6, 7] is similar to the usage of *monitor* state machines in conventional testing. The monitor, respectively the automata, observes the inputs and outputs of the design under test (DUT) and checks whether certain states can be reached, for instance using the *assert* statement in Verilog. In this scenario model checking can replace random simulation of the circuit, consisting of the monitor and the DUT, by a complete search through the whole state space. Monitors can be described in the same language as the design and the user does not have to learn a new language.

Simple properties that can be checked this way are called *safety* properties, i.e. something bad does not happen, and the monitor essentially is a bad state detector. Thus the intended property the user wants to check can not often be described directly but has to be encoded by its negation. Combining and reasoning about properties within this approach, for example when modeling the environment of a basic block, is difficult. Even worse, it may easily lead to wrong assumptions or inconsistent results.

In the context of model checking more expressive specification mechanisms for monitors exist. In addition to assertions for checking safety properties the automata based approach also allows specifications of the form that certain states have to occur infinitely often. This may be necessary to produce a counterexample trace for *liveness* properties. A typical liveness property is the absence of livelocks, i.e. it is not possible to stay in a certain set of states forever. Another example is the liveness property that a *request* is always followed by an *acknowledge*.

A counterexample trace for liveness properties will actually end in a loop, thus representing an infinite execution sequence. Again it is important to realize that monitors for liveness properties represent the negation of desired system properties. More involved liveness properties can be hard to encode as automata.

Simulation is restricted to safety properties. Thus, in principle, model checking is able to formulate and check more powerful properties than simulation. Note however, that liveness properties can often be bounded, in the sense that the designer knows a bound or time limit after which for instance the livelock has to be resolved. Bounded liveness can always be reformulated as safety, and thus makes bounded liveness properties accessible to simulation as well.

On higher abstraction levels, often liveness can not be bounded. Therefore support for general liveness is necessary. However, language support for unbounded liveness is missing in common hardware description languages. Thus the model checker has to work with program annotations by the user, and the user essentially has to learn a new language.

Temporal Logic

Another way to specify sequential behavior is to use *temporal logic* [8]. Beside the common operators for boolean expressions, a formula of temporal logic may contain temporal operators to relate the truth of certain subformulae with each other as time evolves. These temporal operators can be classified as safety and liveness operators. The standard safety operator is *globally*, written **G**. The formula $\mathbf{G}p$ holds along a trace, if p holds in each state of the trace. The dual formula is $\mathbf{F}p$ which holds along a path if *eventually* a state is reached where p holds. Various temporal logic systems

also contain a *next-time* operator. A formula $\mathbf{X}p$ holds at the beginning of a trace if p holds at the second state, or more precisely along the same trace with its first state chopped off.

The subformula p can be any signal assignment or it may be defined recursively as a boolean expression of temporal formulae. The request/acknowledge template discussed above can be formulated as follows

$$\mathbf{G}(req \rightarrow \mathbf{F}ack)$$

where $\rightarrow$ denotes implication. This possibility of explicit nesting of properties is the main difference to the automata based approach.

Temporal properties describe properties in a positive way without negation, though negation, if needed, is a language construct as well. Reasoning about temporal formulae, and thus environment modeling, becomes much easier. In the automata based approach the environment has to be modeled by a circuit stub, restricting the behavior of the DUT. It is not clear how these assumptions can be discharged correctly by checking other parts of the system.

Recently there has also been much interest in test automation tools. In essence, these tools use temporal logic formulae to specify test cases. The goal is to speed up the process for writing test cases. From the model checking perspective and the arguments discussed above in favor for temporal logic versus automata this view is well supported.

The oral presentation will also describe various variants of temporal logic and give an introduction to subtle differences between linear time (LTL), and branching time (CTL) temporal logic. We will also briefly discuss the μ-calculus, a fixpoint based logic for specifying temporal properties.

Model Checking Algorithms

Model checking is supposed to be automatic. Thus the user would not need to know how the algorithms work behind the scenes. However, since the capacity of model checkers is rather restricted compared to equivalence checkers, the user still has to understand various aspects of model checking algorithms to be able to check realistic designs.

Explicit Model Checking

Currently there are three established techniques for model checking, explicit, BDD and SAT based model checking. Originally model checking [9] was developped with an explicit representation of states in mind, i.e. states are stored as bit vectors. This type of model checking is called *explicit model checking*. The state space graph is explored with depth first search, looking for violation of safety properties, or for loops in strongly connected components, violating liveness properties. The algorithms are linear in the size of the model.

In practice explicit model checking is the most efficient model checking technique if the number of reachable states is small, e.g. below several million states. Additionally, several explicit specific techniques exist to prune the search considerably. For instance partial order reduction, as implemented in the SPIN model checker [6], can lead to large reductions in time and space requirements, especially for asynchronous circuits or software.

However, since standard algorithms for explicit model checking require all reachable states to be stored in a large hash table, systems with 10^{20} reachable states or more are out of reach for explicit model checkers. Another problem arises from a large number of primary inputs, which result in many transitions from each single state. This leads to a large branching factor in depth first search, and the explicit model checker will run out of time.

Symbolic Model Checking with BDDs

Symbolic model checking [10] with binary decision diagrams (BDDs) [3] became the dominant technique for model checking synchronous circuits. It can be applied to circuits with hundreds of state bits, and thus potentially many more reachable states than with explicit model checking.

Binary decision diagrams are a compact representation for boolean functions. In the context of model checking they are used to represent not only next state functions of the circuit, but also for the represention of set of states via their characteristic function. For many practical circuits an exponential reduction can be achieved. This reduction allows model checking to be applied to systems with several hundred state bits and more than 10^{20} reachable states.

The main problem with BDDs is to come up with a good variable order for a particular circuit, since the BDD algorithms require a linear order between all boolean variables of the circuit. There are techniques for dynamic reordering variables. However, dynamic reordering is very costly and sometimes a good order does not even exist. As consequence for designs of typical block size, i.e. more than 1000 state bits, symbolic model checking may need more memory than available.

Image computation is used in breadth first search of symbolic model checking to calculate the set of states reachable from a given set of states in one step. Beside dynamic variable reordering, image computation is the most expensive operation in BDD based model checking. There has been considerable efforts to make this operation as fast as possible. But often enough, even for moderate sized circuits, a single image computation step may take too much time.

Symbolic Model Checking with SAT

Procedures for checking propositional formulae (SAT) can handle millions of variables and do not require a global linear variable order. Efficient splitting heuristics exists. In combination with conflict diagnosis and relevance learning, good partial orders of variables can be found easily for many practical problems.

Based on the efficiency of SAT, recently the technique of bounded model checking (BMC) [11] replaced BDDs with SAT in symbolic model checking. Much larger designs can be handled by bounded model checking. The drawback is, that in practice one has to give up completeness. For most practical problems, BMC is not complete, and can only be used to find bugs. It often fails to prove for example safety properties to hold in all reachable states.

If the size of the system is too big for explicit model checking, industrial experience suggests to apply BMC as the first model checking engine, looking for bugs. After BMC does not find any more bugs in a reasonable amount of time, BDD based model checking may be the next technique to try.

Conclusion

This tutorial contains an overview of model checking techniques. We started with a comparison of model checking with other formal methods, and motivated why formal methods for digital designs are important. Then we explained different specification formalism for model checking. Finally we discussed various model checking algorithms. More details can be found in [5] or in recent CAV and DAC proceedings.

Model checking has the potential to reduce testing and thus overall development time considerably. Currently the capacity of complete model checking does not match the size of basic blocks. However, bounded model checking can handle realistic cuircit sizes, if completeness is not required.

Future directions are tighter integration of model checking with testing and better adaption of the design process to formal methods in general. Faster and more robust model checking algorithms will be the most convincing argument for its widespread adoption in industry.

References

[1] A. Kühlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *DAC*, 1997.

[2] J. Silva, *Search Algorithms for Satisfiability Problems in Combinational Switching Circuits*. PhD Thesis, University of Michigan, 1995.

[3] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[4] M. Kaufmann, P. Manolios, and J. Moore, *Computer-Aided Reasoning: An Approach*. Kluwer, 2000.

[5] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[6] G. Holzmann, *The Design and Validation of Computer Protocols*. Prentice Hall, 1997.

[7] R. Kurshan, *Computer-Aided Verification of Coordinating Processes: the automata theoretic approach*. Princeton University Press, 1994.

[8] A. Emerson, "Temporal and modal logic," in *Handbook Theoretical Computer Science: Volume B, Formal Methods and Semantics*, North-Holland, 1995.

[9] E. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Proceedings of the IBM Workshop on Logics of Programs*, 1981.

[10] K. L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[11] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*, 1999.