Tutorial on Model Checking
Modelling and Verification in Computer Science

Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

Abstract. This paper serves as background material for an invited tutorial on model checking given at the 3rd international conference on algebraic biology (AB'08'). The intended audience of the tutorial were researchers in natural science, particularly life science, but this paper may also serve as a light-weight introduction into model checking techniques in general.

Introduction

In that part of computer science which is concerned with constructing systems, modelling usually has a different flavor than modelling in natural science. The artifacts resp. systems engineered by computer scientists usually do have precise mathematical semantics and work according to these abstract semantics. Therefore computer science allows to use precise models, which are conservative abstractions:

> If a model of a computer science system has a certain property,
> then the real system has this property as well.

This statement is of course incorrect if the system is not interpreted on an abstract level. For instance, if a processor on which a program runs has a defect or even just the compiler that produced the machine code from the original program, then the system as a whole does not have to be correct and it may violate the desired property, even though the program, e.g. the computer science artifact, is correct.

Nevertheless, computer science is able to *prove* resp. *check* properties of real systems, because these systems, in the form of programs or circuit designs, are still abstract. Using automatic techniques for checking properties of computer science systems is the purpose of *model checking.*

Research in model checking is centered around algorithmic aspects, particularly on how to implement model checkers, or related questions, such as which specification and modelling languages can be model checked efficiently. In this paper we focus on those approaches that turn out be successful in practice.

Modelling

An important question is where the models come from. There are two radically different ways to obtain models. In the first scenario a system to be built is modelled using some high-level and abstract modelling language. Usually only one aspect of the final not yet existing system is modelled, such as synchronization of parallel components. Then the model can be analyzed through simulation, i.e. by testing it, or by automatically checking certain properties with the help of a model checker.

After the designer has a good understanding of all the aspects of the model, he makes sure that the model is not overly simplistic and also does not contain any fundamental flaws. Then the model is typically thrown away and the system reimplemented in detail from scratch, usually in a totally different but more concrete language. This is a proven technique in industry, particular since the idea of exploring the design space through an executable model, which can be simulated, is very useful even without using model checking techniques.

In the second scenario, model checkers are applied to concrete systems, such as hardware designs, device drivers, or in general software, described in concrete implementation respectively system description languages. The point is that the description of the system in this scenario is detailed enough, even though it is still a model of the real system, that automatic techniques, particularly compilers, can be used to generate the final product.

Originally, models were finite state. This restriction, at least in principle, allows model checkers to be fully automatic. Then model checking terminates, and it either determines that a property holds on the model or the model checker provides a witness in form of a trace that shows that the property is violated. But due to the state-explosion-problem, which says that the number of states in a model is exponential in the size of its description, it may just take too much time to explore all these states and typically also too much space. Much progress has been made over the years to ameliorate this situation and improve scalability of model checking.

Some of the research in model checking also went into the other direction, and lifted the finiteness restriction. There are various forms of infinite systems, for which theoretical results are available and practical applications exist. One direction is to allow continuous variables in the data domain, another to model continuous time. A third direction is to add probability, and a fourth to parameterize the size or the number of components. Another extension is to replace finite state automata by push down automata. All of these extension have in common that model checking only remains decidable, and thus an automatic almost push-button technology, if the class of models allow finite abstractions in some way or another.

History

Model checking was invented more than 25 years ago in the early 80'ties by E. Clarke and A. Emerson [5] and independently by J. Queille and J. Sifakis [14].

There was a workshop [16] affiliated to the Federated Conference on Logic in Computer Science (FLOC'06) dedicated to this anniversary. Beside the proceedings [16] of this workshop, another reference for model checking research is *the* model checking book [7]. More recently Clarke, Emerson and Sifakis won the 2007 Turing Award for their pioneering work on model checking.

From a historical perspective it is probably important to mention that initially theses ideas were not immediately embraced by the formal verification community eagerly, which at that time was still mainly focused on theorem proving techniques. The main argument was that model checking, as it was described initially would only work on tiny models and thus would not scale.

On one hand this argument is still valid, particularly if the goal is to produce a fully verified concrete system. Without additional manual and automatic abstraction techniques, model checking alone will fail in such an endeavor due to the large number of system states, that have to be explored.

On the other hand model checking has been very successful in providing complementary techniques to simulation and testing in order to *partially* verify concrete systems. Particularly in circuit design, where testing costs and also costs for defects that slip through testing are very high, model checking is applied routinely nowadays. It was shown recently, that hybrid techniques that combine model checking with automated theorem proving, can check much larger systems than each technique alone, even in checking properties of device drivers in an industrial operating system [1], for which this hybrid technique is actually used routinely now as well.

Finally, using model checking checking for pure models, e.g. the first scenario discussed in the previous section, will always be beneficial for systems with complex interactions, such as communication protocols etc.

Temporal Logic

Another aspect where model checking differs from other formal approaches is the choice of the specification languages, which are used to describe properties. In essence model checking is concerned with sequential or temporal behavior of systems. This kind of properties are particularly important for reactive, distributed or parallel systems and typically are described in temporal logic. A. Pnueli [12] is considered to be the first who noticed that specifications of such concurrent systems would benefit from using temporal logic.

In its simplest form temporal logic allows to specify two kinds of temporal behavior. A *safety* property states that a certain error or catastrophic state is not reachable. A dual formulation is, that a safety property holds, if all reachable states fulfill a certain invariant, which is valid initially and remains valid no matter how the system evolves. In terms of programming languages an assertion is a typical simple safety property.

More complex temporal specifications are *liveness* properties, also often referred to as *progress* properties. They are used to specify reactiveness, progress or non-starvation etc. A typical example are request/acknowledge properties, for instance calling the elevator (request), will eventually lead to the elevator doors to open (acknowledge). Another example is that a system after powering up will eventually end up in a properly "initialized" state, no matter in what configuration it started. Liveness usually only makes sense in combination with additional fairness assumptions, for instance, that each component is allowed to have its turn infinitely often.

There are various flavors of temporal logic, most notably computation tree logic (CTL) and linear temporal logic (LTL). Related formalisms, such as omega-regular languages and μ-calculus are also used quite frequently. More information on these formalisms can be found in [7]. There are also standardized temporal logics in industry, e.g. the property specification logic (PSL).

Technology

At the core of model checking are algorithms that implement state space traversal. The reachable state space is traversed to find error states that violate safety properties, or to find cyclic paths on which no progress is made as counter example for liveness properties. In most cases state space traversal can be reformulated as fix-point computation, which for certain temporal logics is the only way to describe model checking algorithms.

Initially, model checkers worked on an explicit state space representation. Each state of the system is represented in the computer explicitly and typically stored in a large hash table. The size of the hash table is closely related to the number of reachable states of the system and thus computer memory became the bottle neck.

There are various techniques to improve space consumption in explicit state model checkers. The most important one is partial-order reduction, which is particularly useful for asynchronous models such as loosely coupled software components, petri-net models etc. Also, if the model is symmetric, these symmetries can be factored out during state space traversal. Finally, if the number of states is still too large to be handled, the idea of bit-state hashing trades scalability for completeness, e.g. model checking becomes a falsification technique, similar to traditional testing, but unable to prove correctness.

The most successful explicit model checker is the SPIN model checker. Its main author, G. Holzmann, received the ACM Software System Award in 2002 for his work on SPIN. His latest book [9] on SPIN is probably the best reference to start learning more about explicit state model checking and its optimizations mentioned above. There is a yearly workshop series on SPIN as well, which has more recent results.

For infinite models there are no explicit methods. States have to be represented symbolically. But even for finite models it is possible to represent states, more precisely set of states, symbolically. The goal is to overcome the state-explosion-problem.

In the mid 80'ties Randy Bryant presented reduced ordered binary decision diagrams (BDDs) as a new data structure for symbolically representing and manipulating boolean functions efficiently [4]. This paper has turned out to be one of the most cited papers in computer science.

Also at CMU a couple of years later E. Clarke and his students picked up this idea and showed that BDDs can also be used to represent set of states with BDDs and more importantly, how state space traversal techniques based on fix-point computations can be implemented efficiently using BDD operations. This break through in the early 90'ties is documented in K. McMillan's thesis [10] which also contains a detailed description of his model checker, the symbolic model verifier SMV. One can even argue that the event of symbolic model checking started a renaissance of formal verification in general now with a focus on real applications.

The research in model checking of the 90'ties produced many refinements of symbolic model checking, both in the core algorithms for finite systems, but also in applying similar techniques to infinite systems. The literature is too large to be listed here explicitly. Please refer to the model checking book [7] and in general to the proceedings of the main conference in model checking research, the conference of computer-aided verification (CAV).

In the late 90'ties it was observed that techniques for boolean satisfiability checking, i.e. SAT solvers, could handle much larger formulas than BDDs, and again researchers at CMU came up with the idea of bounded model checking (BMC) [2], which applies SAT solvers to the model checking problem. BMC in its basic form is a falsification technique, at least in practice, i.e. it trades completeness for scalability. However, follow-up work on BMC improved this situation, particular the work on k-induction [15] and interpolation [11]. A survey on these and other related techniques based on using SAT for model checking can be found in [13].

The improvement in SAT solver technology even accelerated in the last 8 years after the introduction of BMC and made model checking much more useful in industry. SAT and its extension of satisfiability modulo theories (SMT) are *the* working horse in almost all state-of-the-art applications of formal methods in industry. Again as example consider [1], which uses SMT solvers to automatically generate abstractions [8] for actual device drivers, which are then checked by a symbolic model checker. If the abstraction is too coarse the abstraction is refined [6], again with the help of SMT solvers. For more information on SAT see the Handbook of Satisfiability [3] and recent proceedings of the yearly SAT conference.

Conclusion

This short note represents a personal interpretation of the first 27 years of model checking from a computer science perspective. It summarize the historical development and gives pointers to recent important results. We hope that we contributed to spread these ideas further across the boundaries of computer science.

References

1. T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *Workshop on Model Checking of Software (SPIN)*, volume 2057 of *LNCS*. Springer, 2001.
2. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *LNCS*. Springer, 1999.
3. A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability.* IOS Press, 2008. To be published.
4. R. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Computers*, C(35), 1986.
5. E. Clarke and E. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Proc. of the Workshop on Logic of Programs*, volume 131 of *LNCS*. Springer, 1982.
6. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5), 2003.
7. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT press, 1999.
8. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. Intl. Conf. on Computer-Aided Verification (CAV)*, volume 1254. Springer, 1997.
9. G. Holzmann. *The SPIN Model Checker.* Addison Wesley, 2004.
10. K. McMillan. *Symbolic Model Checking: An approach to the State Explosion Problem.* Kluwer, 1993.
11. K. McMillan. Interpolation and SAT-based Model Checking. In *Proc. Conf. on Computer-Aided Verification (CAV)*, volume 2725 of *LNCS*. Springer, 2003.
12. A. Pnueli. The temporal logic of programs. In *Proc. IEEE Symp. on Found. of Computer Science*, 1977.
13. M. Prasad, A. Biere, and A. Gupta. A survey on recent advances in SAT-based formal verification. *Software Tools for Technology Transfer (STTT)*, 7(2), 2005.
14. J. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Intl. Symp. on Programming*, volume 137 of *LNCS*. Springer, 1982.
15. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proc. Intl. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, volume 1954 of *LNCS*. Springer, 2000.
16. H. Veith and O. Grumberg, editors. *25 Years of Model Checking*, volume 5000 of *LNCS*. Springer, 2008. To be published.