Model checking

In computer science, **model checking** or **property checking** is a method for checking whether a finite-state model of a system meets a given specification (a.k.a. correctness). This is typically associated with hardware or software systems, where the specification contains liveness requirements (such as avoidance of livelock) as well as safety requirements (such as avoidance of states representing a system crash).

In order to solve such a problem algorithmically, both the model of the system and its specification are formulated in some precise mathematical language. To this end, the problem is formulated as a task in logic, namely to check whether a structure satisfies a given logical formula. This general concept applies to many kinds of logic as well as suitable structures. A simple model-checking problem consists of verifying whether a formula in the propositional logic is satisfied by a given structure.

Elevator control software can be model-checked to verify both safety properties, like *"The cabin never moves with its door open"*,[1] and liveness properties, like *"Whenever the n^{th} floor's* call *button is pressed, the cabin will eventually stop at the n^{th} floor and open the door"*.

Contents

Overview

Property checking is used for verification when two descriptions are not equivalent. During refinement, the specification is complemented with details that are unnecessary in the higher-level specification. There is no need to verify the newly introduced properties against the original specification since this is not possible. Therefore, the strict bi-directional equivalence check is relaxed to a one-way property check. The implementation or design is regarded as a model of the circuit, whereas the specifications are properties that the model must satisfy.[2]

An important class of model-checking methods has been developed for checking models of hardware and software designs where the specification is given by a temporal logic formula. Pioneering work in temporal logic specification was done by Amir Pnueli, who received the 1996 Turing award for "seminal work introducing temporal logic into computing science".[3] Model checking began with the pioneering work of E. M. Clarke, E. A. Emerson,[4][5][6], by J. P. Queille, and J. Sifakis.[7] Clarke, Emerson, and Sifakis shared the 2007 Turing Award for their seminal work founding and developing the field of model checking.[8][9]

Model checking is most often applied to hardware designs. For software, because of undecidability (see computability theory) the approach cannot be fully algorithmic; typically it may fail to prove or disprove a given property. In embedded-systems hardware, it is possible to validate a specification delivered, i.e., by means of UML activity diagrams[10] or control interpreted Petri nets.[11]

The structure is usually given as a source code description in an industrial hardware description language or a special-purpose language. Such a program corresponds to a finite state machine (FSM), i.e., a directed graph consisting of nodes (or vertices) and edges. A set of atomic propositions is associated with each node, typically stating which memory elements are one. The nodes represent states of a system, the edges represent possible transitions which may alter the state, while the atomic propositions represent the basic properties that hold at a point of execution.

Formally, the problem can be stated as follows: given a desired property, expressed as a temporal logic formula *p*, and a structure *M* with initial state *s*, decide if $M, s \models p$. If M is finite, as it is in hardware, model checking reduces to a graph search.

Symbolic model checking

Instead of enumerating reachable states one at a time, the state space can sometimes be traversed more efficiently by considering large numbers of states at a single step. When such state space traversal is based on representations of a set of states and transition relations as logical formulas, binary decision diagrams (BDD) or other related data structures, the model-checking method is symbolic.

Historically, the first symbolic methods used BDDs. After the success of propositional satisfiability in solving the planning problem in artificial intelligence (see satplan) in 1996, the same approach was generalized to model checking for the Linear Temporal Logic LTL (the planning problem corresponds to model checking for safety properties). This method is known as bounded model checking.[12] The success of Boolean satisfiability solvers in bounded model checking led to the widespread use of satisfiability solvers in symbolic model checking.[13]

Techniques

Model-checking tools face a combinatorial blow up of the state-space, commonly known as the state explosion problem, that must be addressed to solve most real-world problems. There are several approaches to combat this problem.

1. Symbolic algorithms avoid ever explicitly constructing the graph for the finite state machines (FSM); instead, they represent the graph implicitly using a formula in quantified propositional logic. The use of binary decision diagrams (BDDs) was made popular by the work of Ken McMillan[14] and the development of open-source BDD manipulation libraries such as CUDD[15] and BuDDy.[16]

2. Bounded model checking algorithms unroll the FSM for a fixed number of steps, k, and check whether a property violation can occur in k or fewer steps. This typically involves encoding the restricted model as an instance of SAT. The process can be repeated with larger and larger values of k until all possible violations have been ruled out (cf. Iterative deepening depth-first search).
3. Abstraction attempts to prove properties of a system by first simplifying it. The simplified system usually does not satisfy exactly the same properties as the original one so that a process of refinement may be necessary. Generally, one requires the abstraction to be *sound* (the properties proved on the abstraction are true of the original system); however, sometimes the abstraction is not *complete* (not all true properties of the original system are true of the abstraction). An example of abstraction is to ignore the values of non-boolean variables and to only consider boolean variables and the control flow of the program; such an abstraction, though it may appear coarse, may, in fact, be sufficient to prove e.g. properties of mutual exclusion.
4. Counterexample guided abstraction refinement (CEGAR) begins checking with a coarse (i.e. imprecise) abstraction and iteratively refines it. When a violation (i.e. counterexample) is found, the tool analyzes it for feasibility (i.e., is the violation genuine or the result of an incomplete abstraction?). If the violation is feasible, it is reported to the user. If it is not, the proof of infeasibility is used to refine the abstraction and checking begins again.[17]

Model-checking tools were initially developed to reason about the logical correctness of discrete state systems, but have since been extended to deal with real-time and limited forms of hybrid systems.

First-order logic

Model checking is also studied in the field of computational complexity theory. Specifically, a first-order logical formula is fixed without free variables and the following decision problem is considered:

Given a finite interpretation, for instance, one described as a relational database, decide whether the interpretation is a model of the formula.

This problem is in the circuit class **AC0**. It is tractable when imposing some restrictions on the input structure: for instance, requiring that it has treewidth bounded by a constant (which more generally implies the tractability of model checking for monadic second-order logic), bounding the degree of every domain element, and more general conditions such as bounded expansion, locally bounded expansion, and nowhere-dense structures.[18] These results have been extended to the task of enumerating all solutions to a first-order formula with free variables.

Tools

Here is a partial list of model-checking tools that have a Wikipedia page:

- Alloy (Alloy Analyzer)
- BLAST (Berkeley Lazy Abstraction Software Verification Tool)
- CADP (Construction and Analysis of Distributed Processes) a toolbox for the design of communication protocols and distributed systems
- CPAchecker: an open-source software model checker for C programs, based on the CPA framework
- ECLAIR: a platform for the automatic analysis, verification, testing, and transformation of C and C++ programs

- FDR2: a model checker for verifying real-time systems modelled and specified as CSP Processes
- ISP code level verifier for MPI programs
- Java Pathfinder: a open-source model checker for Java programs
- Libdmc: a framework for distributed model checking
- mCRL2 Toolset, Boost Software License, Based on ACP
- NuSMV: a new symbolic model checker
- PAT: an enhanced simulator, model checker and refinement checker for concurrent and real-time systems
- Prism: a probabilistic symbolic model checker
- Roméo: an integrated tool environment for modelling, simulation, and verification of real-time systems modelled as parametric, time, and stopwatch Petri nets
- SPIN: a general tool for verifying the correctness of distributed software models in a rigorous and mostly automated fashion
- TAPAs: a tool for the analysis of process algebra
- TAPAAL: an integrated tool environment for modelling, validation, and verification of Timed-Arc Petri Nets
- TLA+ model checker by Leslie Lamport
- UPPAAL: an integrated tool environment for modelling, validation, and verification of real-time systems modelled as networks of timed automata
- Zing[19] – experimental tool from Microsoft to validate state models of software at various levels: high-level protocol descriptions, work-flow specifications, web services, device drivers, and protocols in the core of the operating system. Zing is currently being used for developing drivers for Windows.

See also

- List of model checking tools
- Binary decision diagram
- Büchi automaton
- Computation tree logic
- Formal verification
- Linear temporal logic
- Partial order reduction
- Program analysis (computer science)
- Abstract interpretation
- Automated theorem proving
- Static code analysis

References

Citations

1. For convenience, the example properties are paraphrased in natural language here. Model-checkers require them to be expressed in some formal logic, like LTL.

2. Lam K., William (2005). "Chapter 1.1: What Is Design Verification?" (http://my.safaribooksonline.com/book/electrical-engineering/semiconductor-technology/0131433474/an-invitation-to-design-verification/ch01lev1sec1#X2ludGVybmFsX0h0bWxWaWV3P3htbGlkPTAxMzE0MzM0NzQlMkZjaDAxbGV2MXNlYzEmcXVlcnk9). *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Retrieved December 12, 2012.
3. "Amir Pnueli - A.M. Turing Award Laureate" (http://amturing.acm.org/award_winners/pnueli_4725172.cfm/).
4. Allen Emerson, E.; Clarke, Edmund M. (1980), "Characterizing correctness properties of parallel programs using fixpoints", *Automata, Languages and Programming*, Lecture Notes in Computer Science, **85**: 169–181, doi:10.1007/3-540-10003-2_69 (https://doi.org/10.1007%2F3-540-10003-2_69), ISBN 978-3-540-10003-4
5. Edmund M. Clarke, E. Allen Emerson: "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic" (http://portal.acm.org/citation.cfm?id=747438&dl=). Logic of Programs 1981: 52-71.
6. Clarke, E. M.; Emerson, E. A.; Sistla, A. P. (1986), "Automatic verification of finite-state concurrent systems using temporal logic specifications", *ACM Transactions on Programming Languages and Systems*, **8** (2): 244, doi:10.1145/5397.5399 (https://doi.org/10.1145%2F5397.5399)
7. Queille, J. P.; Sifakis, J. (1982), "Specification and verification of concurrent systems in CESAR", *International Symposium on Programming*, Lecture Notes in Computer Science, **137**: 337–351, doi:10.1007/3-540-11494-7_22 (https://doi.org/10.1007%2F3-540-11494-7_22), ISBN 978-3-540-11494-9
8. Press Release: ACM Turing Award Honors Founders of Automatic Verification Technology (http://www.acm.org/press-room/news-releases/turing-award-07/)
9. *USACM*: 2007 Turing Award Winners Announced (http://usacm.acm.org/usacm/weblog/index.php?p=572)
10. I. Grobelna, M. Grobelny, M. Adamski, "Model Checking of UML Activity Diagrams in Logic Controllers Design (https://link.springer.com/chapter/10.1007/978-3-319-07013-1_22)", Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX, Advances in Intelligent Systems and Computing Volume 286, Springer International Publishing Switzerland, pp. 233–242, 2014
11. I. Grobelna, "Formal verification of embedded logic controller specification with computer deduction in temporal logic (https://www.researchgate.net/profile/Jan_Sikora3/publication/267037615_Advanced_Numerical_Modelling/links/5442adc40cf2e6f0c0f9366b/Advanced-Numerical-Modelling.pdf#page=63)", Przeglad Elektrotechniczny, Vol.87, Issue 12a, pp.47–50, 2011
12. Clarke, E.; Biere, A.; Raimi, R.; Zhu, Y. (2001). "Bounded Model Checking Using Satisfiability Solving". *Formal Methods in System Design*. **19**: 7–34. doi:10.1023/A:1011276507260 (https://doi.org/10.1023%2FA%3A1011276507260).
13. Vizel, Y.; Weissenbacher, G.; Malik, S. (2015). "Boolean Satisfiability Solvers and Their Applications in Model Checking". *Proceedings of the IEEE*. **103** (11): 2021–2035. doi:10.1109/JPROC.2015.2455034 (https://doi.org/10.1109%2FJPROC.2015.2455034).
14. * *Symbolic Model Checking*, Kenneth L. McMillan, Kluwer, ISBN 0-7923-9380-5, also online (http://www.kenmcmil.com/thesis.html).
15. "CUDD: CU Decision Diagram Package" (https://www.cs.rice.edu/~lm30/RSynth/CUDD/cudd/doc/).
16. "BuDDy – A Binary Decision Diagram Package" (http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.itu.dk/research/buddy/).

17. Clarke, Edmund; Grumberg, Orna; Jha, Somesh; Lu, Yuan; Veith, Helmut (2000), "Counterexample-Guided Abstraction Refinement" (http://www.aladdin.cs.cmu.edu/papers/pdfs/y2000/counterex.pdf) (PDF), *Computer Aided Verification*, Lecture Notes in Computer Science, **1855**: 154–169, doi:10.1007/10722167_15 (https://doi.org/10.1007%2F10722167_15), ISBN 978-3-540-67770-3
18. Dawar, A; Kreutzer, S (2009). "Parameterized complexity of first-order logic" (https://pdfs.semanticscholar.org/ac54/505a6c9b843259727dba98fad1a02af2a567.pdf) (PDF). *ECCC*.
19. Zing (https://www.microsoft.com/en-us/research/project/zing)

Sources

- This article is based on material taken from the *Free On-line Dictionary of Computing* prior to 1 November 2008 and incorporated under the "relicensing" terms of the GFDL, version 1.3 or later.

Further reading

- *Model Checking* (http://mrw.interscience.wiley.com/emrw/9780470050118/ecse/article/ecse247/current/abstract), Doron Peled, Patrizio Pelliccione, Paola Spoletini, Wiley Encyclopedia of Computer Science and Engineering, 2009.
- *Model Checking*, Edmund M. Clarke, Orna Grumberg and Doron A. Peled, MIT Press, 1999, ISBN 0-262-03270-8.
- *Systems and Software Verification: Model-Checking Techniques and Tools*, B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, ISBN 3-540-41523-8
- *Logic in Computer Science: Modelling and Reasoning About Systems*, Michael Huth and Mark Ryan, Cambridge University Press, 2004. doi:10.2277/052154310X (https://doi.org/10.2277%2F052154310X).
- *The Spin Model Checker: Primer and Reference Manual* (http://spinroot.com/spin/Doc/Book_extras/), Gerard J. Holzmann, Addison-Wesley, ISBN 0-321-22862-6.
- Julian Bradfield and Colin Stirling, Modal logics and mu-calculi, Inf.ed.ac.uk (http://homepages.inf.ed.ac.uk/jcb/Research/bradfield-stirling-HPA-mu-intro.ps.gz)
- Specification Patterns KSU.edu (http://patterns.projects.cis.ksu.edu/documentation/patterns.shtml)
- Property Pattern Mappings for RAFMC Inria.fr (http://cadp.inria.fr/resources/evaluator/rafmc.html)
- Radu Mateescu and Mihaela Sighireanu Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus (http://vasy.inria.fr/publications/Mateescu-Sighireanu-03.html), page 6, Science of Computer Programming 46(3):255–281, 2003
- Müller-Olm, M., Schmidt, D.A. and Steffen, B. *Model checking: a tutorial introduction.* (http://people.cis.ksu.edu/~schmidt/papers/sas99.ps.gz) Proc. 6th Static Analysis Symposium, G. File and A. Cortesi, eds., Springer LNCS 1694, 1999, pp. 330–354.
- Baier, C., Katoen, J.: Principles of Model Checking. 2008.
- E.M. Clarke: "The birth of model checking" doi:10.1007/978-3-540-69850-0_1 (https://doi.org/10.1007%2F978-3-540-69850-0_1)
- E. Allen Emerson: "The Beginning of Model Checking: A Personal Perspective" (http://www.model.in.tum.de/um/25/pdf/Emerson.pdf) (this is also a very good introduction and overview of model checking)