HOW EFFECTIVE IS MODEL CHECKING IN PRACTICE?

TheAnh Do, A. C. M. Fong and Russel Pears
Auckland University of Technology, Auckland, New Zealand

Keywords: Formal methods, Static analysis, Model checking, Hardware verification, Software verification.

Abstract: Software and hardware systems are becoming increasingly large, complex, and can change rapidly. Ensuring reliability of these systems can therefore be a problem. Traditional techniques such as testing and simulation are completely infeasible to cope. Model checking offers an alternative, but its use is still limited. We identify the disadvantages of model checking in practical usages and research directions to tackle these. We clearly define the context for each disadvantage and concretely describe difficulties for which verification users may face when applying the model checking technique to verifying certain systems. We also provide a comprehensive picture of research works in this context and emphasize outcomes and shortcomings of each work by means of others'. The paper would be therefore the useful user manual for verification users in practical usages and the helpful guidance for doing research in model checking.

1 INTRODUCTION

Software and hardware systems are becoming increasingly large, complex and often evolve rapidly. Traditional techniques such as testing and simulation (Myers, 1979) are inadequate because exhaustively checking all possible execution paths of such systems is practically infeasible. Also, these techniques can only show the presence of bugs, but not their absence. Verification technique using theorem proving (Bledsoe and Loveland, 1984) requires hand constructed axioms and proof rules. It is thus difficult to use, and unscalable to practical systems with large size and high complexity.

In contrast, model checking (Clarke et al., 1999) is an automated verification technique that, given a finite-state model of the system under consideration and a property of interest, exhaustively explores all states of the system to check whether this property holds for (a given state in) that model. Hence if an execution terminates correctly, the system may be considered "bug-free". Often, model checking only takes a few minutes, which is much faster than manually constructing axioms and proof rules that can last days or months. Model checking has been successfully applied to verify some practical systems (Holzmann and Smith, 2000), (Clarke et al., 1993). It has become a protocol design and validation tool (Holzmann, 1990), and a verification toolkit (Ball et al., 2004), (Fix, 2008) of many software and hardware companies. Moreover, it is taught in universities (Clarke, 2011), (Holzmann, 2011) and is recognized by standards-developing organizations (Eisner and Fisman, 2006).

However, successful application of model checking is predicated on the availability of an accurate model. It has been reported that the constructed system model is judged to be correct, but the real system itself still exposes severe bugs (Havelund et al., 2000), (Havelund et al., 2001). The task of manually modeling systems to obtain checkable finite-state models can be difficult for those that lack expertise in model checking. We therefore believe that understanding the capability of the model checking technique as well as its disadvantages is necessary for verification users to apply it correctly, effectively and efficiently.

In this paper, we aim to provide a comprehensive picture about the applicability of model checking in practice. We approach the technique in terms of its disadvantages and identify obstacles of its practical usages from the point of view of verification users. We define clearly the context for each disadvantage and describe difficulties for which verification users may face when applying the technique to their specific problems. We also provide an extensive perspective of research works in this context and emphasize outcomes and shortcomings of each work. The paper would be therefore the useful user manual for verification users in practical usage and the helpful guidance for doing research in model checking. In the former case, it could help

Do T., Fong A. and Pears R..
HOW EFFECTIVE IS MODEL CHECKING IN PRACTICE?.
DOI: 10.5220/0003467402390244
In *Proceedings of the 6th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2011), pages 239-244
ISBN: 978-989-8425-57-7

verification users to realize “to what extent, the technique model checking is applicable” and to apply the technique sufficiently to their specific problems. In the latter case, it could help researchers capture the current progress of model checking research as well as its future challenges.

The paper is organized as follows. Section 2 introduces basic concepts of model checking. Section 3 highlights a number of disadvantages of model checking. State space explosion, which is the major disadvantage of modelling checking, is separately described in Section 4 together with state-of-the-art techniques to treat it for large complex systems. Finally, we conclude the paper in Section 5.

2 MODEL CHECKING

Model checking originated from the need to verify circuit designs and protocols for which checking all of the possible interactions and subtle bugs in the systems is extremely difficult (Clarke and Emerson, 1981), (Queille and Sifakis, 1982).

One prerequisite input to model checking is a *formal model*, but not the actual system itself. Model checking is hence an instance of model-based verification techniques which carry out the verification on a high-level description of the system under consideration. As a result, any obtained result is only as good as the system model. In addition, the size of the formal model is required to be *finite*. The other input is a *formal property* that represents the behaviour of interest of the system.

During verification, it performs a search algorithm to *systematically* (and exhaustively) explore all system states to determine if the property is violated or not. In the former case, a *counterexample* is provided to indicate the falsification of the property and is used for debugging purposes. Otherwise, the property holds.

Model checking entails four main steps: modelling systems, formalizing system requirements, execution and analyzing the results.

2.1 Formal Model

A formal model is a high-level description of the system under consideration which consists of information about the system at a certain moment of its behaviour and how the system can evolve from one state to another. In other words, it describes how the system behaves using the model description language of a chosen model checking engine such as the Process Meta Language (or PROMELA) in SPIN (Holzmann, 2004) or the SMV language in SMV (McMillan, 1993).

2.2 Property Specification

A property specification prescribes what the system should and should not do. Property specifications are expressed using temporal logics that allow the specification of the relative order of events in the behaviour of interest of the system. For example, “Once a process has requested the token, it continues to request the token until the token is received”. The underlying nature of time in temporal logics can be either linear i.e. LTL (Pnueli, 1977) or branching i.e. CTL (Clarke and Emerson, 1981).

2.3 Model Checking Algorithm

In principle, the problem of model checking is given a formal model M and a temporal formula f, find all states s of M such that $M, s \models f$. A model checking engine often performs a search algorithm that systematically explores all states of the formal model and checks in each state whether the temporal formula is true or not. In practice, depending on the temporal logic supported (either LTL or CTL) as well as specific techniques used to combat with high computation complexities, the search algorithm can be carried out in various fashions, e.g. automata-based LTL model checking (Vardi and Wolper, 1986), CTL model checking (Clarke and Emerson, 1981), symbolic model checking (McMillan, 1993).

3 MODEL CHECKING: DISADVANTAGES AND RESEARCH DIRECTIONS

Model checking is increasingly gaining recognition in hardware and software industry. Its applicability is still a problem chiefly due to its intrinsic nature and the complexity of real systems.

3.1 Model-based Verification

Model checking is a model-based verification technique. This means to apply any model checking engine, a formal model of the system under consideration must first be constructed. However, constructing manually the verification model for any non-trivial systems is often a laborious task. Second, the constructed model may not truly reflect the behaviour of the real system. Third, there may be a relatively far semantic distinction between correctness at the formal model level and at the actual system implementation (or source code) level. Last, whenever a property is falsified, much effort is required to find real bugs in the actual system.

To address these issues, early research works tend to build translators to convert program text literally into the input language of a model checking engine. As the translation is done without abstraction, restrictions must be imposed on the input language to keep the verification problem decidable (Havelund and Pressburger, 2000). Later works leverage program slicing techniques together with abstraction to extract a checkable finite-state formal model for model checking (Hatcliff et al., 2000), (Corbett et al., 2000). The work of Holzmann (2001) offers a means to promote better formal models, but still constructed manually. In fact, all of these works rely heavily on the capability of existing model checking engines e.g. SPIN (Holzmann, 2004), and lack important features such as dynamic memory allocation that are not supported by those engines. Besides, model checking engines which conduct the verification on source code like Verisoft (Godefroid, 1997) are restricted to only basic safety properties and not scalable to large systems.

3.2 Coverage

Given a system model and a property of interest, model checking is able to determine whether the model satisfies the property or not. Provided all specified properties have been checked successfully, are we sure that the system model is indeed correct? Unfortunately, model checking cannot answer this question due to the following reasons. First, model checking checks only stated properties; validity of properties that are not checked cannot be judged. Second, if complete model checking algorithms run out of essential available resources, e.g. memory, before completion, the validity of the property being checked will remain unknown. This is the well known state space explosion problem and will be discussed in detail in Section 4. Last, incomplete model checking algorithms (Biere et al., 2003) offer sound and termination checking, but evidently the unexercised state space may still harbour errors. This problem is called *Coverage* on model checking and has been studied together with the state explosion phenomenon for decades.

3.3 Control-intensive Applications

One more disadvantage of model checking is that the technique is less suited for verifying data-intensive systems. In contrast, control-intensive systems often expose features that are very natural for applying model checking. First, these systems, especially hardware systems, tend to be well-structured and often have finite-state spaces. Second, the separation of control flow and data flow in the systems is relatively clear, abstraction techniques can remove substantial parts of the data flow from the systems and significantly reduce the state space. Furthermore, efficient tools with effective techniques have been implemented to check properties on the abstracted systems.

3.4 Generalization Verification

In the design of reactive systems in both software and hardware, systems are often described schematically in terms of a parameter *n*, representing the arbitrary number of components, or are parameterized. For instance, the token-ring design of a distributed mutual exclusion algorithm consists of an *unknown* number of processes in which mutual exclusion is guaranteed by means of a token that is passed around the ring (Martin, 1985). This implicit ring design represents an entire family of specific ring design members in which each family member associates with a concrete number of processes. Automatically verifying the correctness of such parameterized systems cannot be realized by model checking, however. This is because the sequence of (even finite-state) components is unknown or infinite, and exhaustively searching the resulted unbounded-state space is out of the reach of model checking. This problem is called *Generalization Verification* (Demri et al., 2006) and is proved in general undecidable (Apt and Kozen, 1986).

3.5 Human Intervention

Another disadvantage is human intervention is needed in all four stages of model checking. The first stage of model checking is modelling systems to obtain a checkable formal model. As discussed in Section 3.1, this process is laborious and error-prone. Methods that attempt to address the issues are largely inadequate as their applicability and scalability are questionable.

3.6 Decidability Issues

As Turing (1936) pointed out, computability of a sound and complete algorithmic solution for any sufficiently powerful programming model, even under restrictions such as finite-state spaces, is completely intractable. The problem of model checking is a concrete instance to obviously illustrate this undecidability problem. In Section 3.4, we discussed this in the context of verifying parameterized systems and here we focus on software model checking.

In contrast to the pure model checking technique, software model checking does the verification at the source code level without requiring manually constructing a formal model for the system under consideration. Two basic principles of this technique are (1) reasoning about a system at the source code level and (2) finding a right abstraction level for the system to carry out the verification.

Recent tools often leverage predicate abstraction as well as decision procedures to verify the correctness of practical systems. The former allows the abstraction to be parameterized by and specific to a program. The obtained abstract program is represented by a Boolean program and can be relegated to the later.

4 STATE-SPACE EXPLOSION

In practice, the number of states needed to model a system accurately may be extremely large and easily exceed the amount of available computer memory. This is known as the state space explosion problem. In sequential programs, verification models are generated by means of unfolding a program graph over program locations and variables. Let *L* and *V* represent the sets of locations and variables of a program graph, respectively. The number of states of the unfolded verification model is

$$|L| \cdot \prod_{x \in V} |\text{domain}(x)|$$

The number of states thus grows exponentially with the number of variables in the program graph. Even simple program graphs with just a small number of variables, this bound can be excessive. For example, consider a program graph with 10 locations and a bit-type array variable of 100 bits, the bound grows up to $10 \cdot 2^{100}$. Furthermore, in case the set of program locations or the data domain of any program variable is infinite, the underlying verification model yields an unbounded state space. The model checking problem for such program graphs is undecidable. This observation clearly explains why model checking is mainly appropriate to control-intensive applications but tremendously hard to deal with data-intensive applications.

In concurrent programs, the state space of the whole system is the Cartesian product of the local state spaces of components. For example, consider a parallel system P consisting of n components P_i ($1 \leq i \leq n$), the number of states of this system:

$$P = P_1 \,|||\, P_2 \,|||\, P_3 \ldots |||\, P_n$$

is indicated as follows:

$$S = |S_1| \cdot |S_2| \cdot |S_3| \cdots |S_n|$$

Here S_i represents the state space of component P_i. The number of states of the verification model for the complete system therefore grows exponentially with the number of components. In addition, the exponential increase in the local state space of each component as discussed in sequential programs also makes the model checking problem extremely hard to exhaustively cover the combinatorial growth of state space for the system. If the number of components of the system is infinite, the model checking problem becomes undecidable as discussed in Section 3.4 – Generalization Verification.

In fact, using model checking to verify the correctness of realistic systems is too complex (and even impossible). For decades, the problem of state space explosion has been the driving force behind much of the research in model checking and the development of new model checkers. We survey some of these techniques.

4.1 Symbolic Model Checking

The first algorithms for CTL model checking represent transition relations explicitly by adjacency lists and hence just handle concurrent systems with the fairly number of states (Clarke and Emerson, 1981), (Queille and Sifakis, 1982). In contrast, symbolic model checking (McMillan, 1993) uses Binary Decision Diagrams (Bryant, 1986) to represent the sets of states and transition relations, and then computes a fixed point of an operator for the CTL formula relying on mu-calculus (Emerson, 1996). It can thus obtain a compact model for proving correctness and handling the booming of state space due to program variables and data types. Symbolic model checking has been applied to successfully verify many practical systems (Burch et al., 1991), (Clarke et al., 1993). It however does not work well when BDDs grow too large.

4.2 Partial Order Reduction

The aim of partial order reduction is to prune the state space exploration of concurrent programs by exploiting the independence of concurrently executed events and also the redundancies in the state space with respect to a given property being checked (Valmari, 1990), (Godefroid, 1990), (Peled, 1994). Two events are independent of each other if regardless of the ordering of their executions, the result will be the same. The interleaving of transitions of such independent events can be therefore restricted to one representative when constructing the state space for proving the property. This effect becomes even more drastic on increasing the number of concurrent processes – the state space of the full transition system grows exponentially in the number of processes whereas the reduced state space consists of a single path that grows just linear. Partial order reduction however has little effect when systems consist of processes that are tightly connected or few independent events exist.

4.3 Abstraction

Abstraction is one of the most successful techniques reported so far. One approach is the cone of influence reduction. It attempts to reduce the state space of the state transition graph by focusing on portions of the system description that preserve all relevant information for the behaviours of interest as identified by the specification (Balarin and Sangiovanni-Vincentelli, 1993), (Kurshan, 1994). Irrelevant portions for verifying the desired property are then removed and the size of the corresponding transition system model is reduced significantly. Another approach is data abstraction. It involves finding a mapping between the actual data values or data structures in the system and a small set of abstract data values (Clarke et al., 1992), (Bensalem et al., 1992). For example, a stack class can be mapped to an integer which holds the size information of the stack. The size of the obtained abstract model therefore becomes tractable. The most predominant abstraction technique now is predicate abstraction as discussed in Section 3.1.

5 CONCLUSIONS

Model checking has been demonstrated an effective technique for proving correctness and ensuring reliability of systems. Applicability of the technique in industry is still restricted, nonetheless. This uncloses a number of research directions for the future. First, devise sufficient data structures and algorithms to handle large search spaces. Second, improve and integrate abstraction and compositional reasoning techniques together with others to deal with high complexity and large systems, especially software systems. Third, develop mechanisms to better reason systems in the presence of expressive heap abstractions and concurrent interactions. Last but not least, support reasoning modern programming language features such as object-orientation, dynamic dispatch, abstract data types, higher-order control flow and continuations.

In this paper, we have provided a comprehensive picture of the capability and the applicability of model checking in practice. We approached the technique in terms of its disadvantages and highlighted obstacles of its practical application from the point of view of verification users. We clearly delineated the context for each disadvantage and pointed out its difficulties when applied to specific systems. We also provided a perspective of research works in this context and emphasized outcomes and shortcomings of each work. The paper would be therefore useful for verification users in practical usage and others doing research in model checking.

REFERENCES

Myers, G. J., 1979. *The Art of Software Testing*. Wiley.

Bledsoe, W. W., Loveland, D. W. (eds.), 1984. *Automated Theorem Proving: After 25 Years*. Contemporary Mathematics, V29. American Mathematical Society.

Clarke, E. M., Grumberg, O., and Peled, A., 1999. *Model Checking*. MIT Press.

Holzmann, G. J., Smith, M. H., 2000. Automating Software Feature Verification. *Bell Labs Technical Journal*.

Clarke, E. M., Grumberg, O., Hiraishi, H., Jha, S., Long, D. E., McMillan, K. L., Ness, L. A., 1993. Verification of the Futurebus+ Cache Coherence Protocol. *CHDL*.

Holzmann, G. J., 1990. *Design and Validation of Computer Protocols*. Prentice-Hall,Inc., Upper Saddle River, NJ.

Ball, T., Cook, B., Levin, V., Rajamani, S. K., 2004. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *IFM '04: Integrated Formal Methods*.

Fix, L., 2008. Fifteen Years of Formal Property Verification in Intel. *25 Years of Model Checking*.

Clarke, E. M., 2011. *Introduction to Model Checking*. Carnegie Mellon University. Retrieved from: http://www.cs.cmu.edu/~emc/15817-s11/reading.html.

Holzmann, G. J., 2011. *Logic Model Checking*. California Institute of Technology. Retrieved from: http://spinroot.com/spin/Doc/course/.

Eisner, C., Fisman, D., 2006. *A Practical Introduction to PSL (Series on Integrated Circuits and Systems)*. Springer, New York.

Havelund, K., Lowry, M., Park, S., Pecheur, C., Penix, J., Visser, W., White, J., 2000. Formal Analysis of the Remote Agent Before and After Flight. *Proc. 5th NASA Langley Formal Methods Workshop*, Williamsburg, VA.

Havelund, K., Lowry, M., Penix J., 2001. Formal Analysis of a Space-Craft Controller Using SPIN. *IEEE Transactions on Software Engineering*, v.27 n.8.

Clarke, E. M., Emerson, E. A., 1981. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. *Logic of Programs*. Springer-Verlag.

Queille, J. P., Sifakis, J., 1982. Specification and Verification of Concurrent Systems in CESAR. *Proceedings of the 5th Colloquium on International Symposium on Programming*.

Holzmann, G. J., 2004. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley.

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

Pnueli, A., 1977. The Temporal Logic of Programs. In *18th IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society Press.

Vardi, M. Y., Wolper, P., 1986. An Automata-Theoretic Approach to Automatic Program Verification. In *1st Annual Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press.

Havelund, K., Pressburger, T., 2000. Model Checking JAVA Programs Using JAVA PathFinder. *Int'l J. Software Tools for Technology Transfer*.

Hatcliff, J., Dwyer, M. B., Zheng, H., 2000. Slicing Software for Model Construction. *Higher-Order and Symbolic Computation*.

Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby, Zheng, H., 2000. Bandera: Extracting Finite-State Models from Java Source Code. *Proceedings of the 22nd International Conference on Software Engineering*.

Holzmann, G. J., 2001. From Code to Models. *In Proceedings of the 2nd International Conference on Application of Concurrency to System Design*.

Godefroid, P., 1997. VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software. *Proceedings of the 9th International Conference on Computer Aided Verification*.

McMillan, K. L., 2003. Interpolation and Sat-Based Model Checking. In *CAV*.

Henzinger, T. A., Jhala, R., Majumdar, R., Qadeer, S., 2003. Thread-Modular Abstraction Refinement. *CAV*.

Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., Zhu, Y., 2003. Bounded Model Checking. *Advances in Computers*, vol. 58. Academic Press.

Rajan, S. P., Tkachuk, O., Prasad, M. R., Ghosh, I., Goel, N., Uehara, T., 2009. WEAVE: WEb Applications Validation Environment. In *ICSE*.

Martin, A., 1985. The Design of a Self-Timed Circuit for Distributed Mutual Exclusion. In *Proceedings of the 1985 Chapel Hill Conference on VLSI, Computer Science Press, Rockville, MD*.

Demri, S., Laroussinie, F., Schnoebelen, Ph., 2006. A Parametric Analysis of the State-Explosion Problem in Model Checking. *Journal of Computer and System Sciences*, v.72 n.4, p.547-575.

Apt, K. R., Kozen, D., 1986. Limits for the Automatic Verification of Finite-State Concurrent Systems. *Information Processing Letters*.

Kurshan, R. P., McMillan, K. L., 1995. A Structural Induction Theorem for Processes. *Information and Computation*.

Wolper, P., Lovinfosse, V., 1989. Verifying Properties of Large Sets of Processes with Network Invariants. In *Automatic Verification Methods for Finite State Systems*.

Clarke, E. M., Grumberg, O., Jha, S., 1995. Verifying Parametrized Networks Using Abstraction and Regular Languages. In *CONCUR'95*.

Kesten, Y., Pnueli, A., 2000. Control and Data Abstractions: The Cornerstones of Practical Formal Verification. *Software Tools for Technology Transfer*.

Turing, A. M., 1936. On Computable Numbers, with an Application to the Eintscheidungsproblem. In *Proceedings of the London Mathematical Society*.

Qadeer, S., Rehof, J., 2005. Context-Bounded Model Checking of Concurrent Software. *TACAS*.

Bryant, R., 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*.

Emerson, E. A., 1996. Model Checking and the Mu-calculus. *Descriptive Complexity and Finite Models*.

Burch, J. R., Clarke, E. M., Long, D. E., 1991. Representing Circuits More Efficiently in Symbolic Model Checking. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*.

Valmari, A., 1990. A Stubborn Attack On State Explosion. *CAV*.

Godefroid, P., 1990. Using Partial Orders to Improve Automatic Verification Methods. *CAV*.

Peled, D., 1994. Combining Partial Order Reductions with On-the-fly Model-Checking. *CAV*.

Balarin, F., Sangiovanni-Vincentelli, A. L., 1993. An Iterative Approach to Language Containment. *CAV*.

Kurshan, R. P., 1994. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press.

Clarke, E. M., Grumberg, O., Long, D. E., 1992. Model Checking and Abstraction. *POPL*.

Bensalem, S., Bouajjani, A., Loiseaux, C., Sifakis, J., 1992. Property Preserving Simulations. *CAV*.