Requirements Eng (2002) 7:221–224
Ownership and Copyright
© 2002 Springer-Verlag London Limited

Requirements
Engineering

Guest Editorial: Special Issue on Model Checking in Requirements Engineering

Steve Easterbrook and Marsha Chechik
Department of Computer Science, University of Toronto, Toronto, Ontario, Canada

In the past decade, practical model-checking techniques have revolutionised research in formal software verification. Until the mid-1980s, when model checking was first introduced [1,2], formal verification research focused primarily on techniques for proving a program is correct against a formal specification. For many reasons, such techniques had had little impact on industrial practice [3]. In contrast, model checking has rapidly transferred into industrial practice, both for hardware and software verification. With its emphasis on partial verification using fully automated techniques, model checking has led to an interest in 'lightweight' formal techniques [4] that can be applied at different levels of abstraction, and during any stage of the development process. Model-checkers have become popular debugging tools and have been used to reason about system requirements [5], software architectures [6], program behaviour [7–9], hardware and circuit designs [10], communication protocols [11] and even user interfaces [12]. Because model checking can be used to analyse abstract behavioural models, it has a number of natural applications in requirements engineering.

A model-checker takes as input a model, M, of a system, expressed as a finite state machine, and a temporal logic formula, φ, and algorithmically determines whether or not the model satisfies the property; i.e., it computes the value of the relation $M \models \varphi$. From an engineering point of view, it is natural to consider the state machine model to be the central artefact, and to talk of checking that various behavioural properties hold of the model. We can summarise the key advantages of model checking over other forms of formal analysis as follows:

1. The procedure is fully automatic and quite fast, often producing an answer in a matter of minutes.
2. If a property is not satisfied, a model-checker will usually produce a counter-example – a sequence of steps leading to the problem, thus showing why the property is not satisfied.
3. Model checking can be applied to partial models, so it is not necessary to fully specify a system nor all its properties before analysing its correctness.

Model checking was first applied to requirements engineering in the work of Atlee and Gannon [5]. In requirements engineering, the state machine typically represents an abstract description of the behaviour of some portion of the system to be specified, or its environment. The properties to be checked typically represent high-level requirements including safety properties (some undesirable situation will never happen) and liveness properties (some desirable situation will eventually occur). Reports of industrial case studies (e.g. [13]) indicate that it is rare to have well-formulated temporal logic properties from the outset. More usually, the model is developed first in an attempt to understand some aspect of a system's behaviour, and the exercise of model checking then involves the interaction of domain experts to discover high-level properties that ought to hold. In this sense, the model-checker becomes an exploration tool, used to discover properties of a model being developed, rather than to verify it against a pre-existing specification. Because model checking does not require completeness of either the model or the properties to be checked, it can be applied at very early stages in requirements modelling.

Correspondence and offprint requests to: Steve Easterbrook, Department of Computer Science, University of Toronto, Toronto, Ontario M5S 2E4, Canada. Email: sme@cs.toronto.edu

It is useful to distinguish between a model-checking engine, which computes the satisfaction relation for a particular temporal logic, and a model-checking framework, which includes modelling languages for expressing the state machine models, abstraction techniques for reducing large models to a size suitable for fast checking, and tools for translating the models into the input language of the checker engine and for visualising the results. Model-checking engines can now be considered mature technology – rather than describe how they work here, we refer the reader to several good tutorial introductions [14,15]. An example of a model-checking framework is the SCRtool from NRL [16]. In general, model-checking frameworks are still relatively immature, and are the subject of much current research. A good model-checking framework is essential for the following reasons:

1. Temporal logics can be hard to work with, and most people have difficulty in finding the correct logical expression for all but the simplest properties.
2. Translations of requirements models to the input languages of model-checking engines often cannot be efficiently checked, because the resulting state spaces are too large.
3. The counter-examples produced by most model-checking engines do not mean anything to the stakeholders, and need to be translated back into the original modelling language.

The first problem has been ameliorated by the development of a set of patterns that allow the analyst to select the appropriate expression from a high-level description of the type of property needed [17]. However, identifying and formalising suitable behavioural properties is still a challenging task.

The second problem is inherently that of the model-checking technology: the size of the state space grows exponentially with the number of variables in the model. This phenomenon is commonly called the state explosion problem. Efficient algorithms, including symbolic model checking, symmetry reduction, etc., allow current state-of-the art model-checkers to handle models with around 400–450 propositional (i.e., Boolean-valued) variables [15]. However, many applications need models with real- or integer-valued variables, and introducing these into a model rapidly expands the state space beyond a size that can be practically checked. Hence, for many applications, good abstraction techniques are essential. For example, a simple abstraction step replaces an integer variable with several Booleans, testing whether the value is in a particular range. In general, finding good abstractions is hard, so model-checking frameworks need to provide automated abstraction techniques tailored to the modelling languages they support, as well as some assurance that the abstractions are sound with respect to the properties being checked. Atlee reports that the performance of some model-checkers is particularly sensitive to certain modelling choices such as the order of introduction of variables, their types, and modularity of the system [18]. Understanding the impact of these choices on the state space of the system and enriching model-checking frameworks to provide support for making these choices correctly are part of ongoing research in this area.

The third problem is a result of the gap between the practical modelling languages used in software engineering and the input languages of model-checking engines. The former are designed to be conceptually easy to use, and to provide modelling primitives appropriate for the application domain. The latter are designed to be efficiently analysed. Abstraction steps applied to the users' models increase this gap. Hence, model-checking frameworks also need to be able to relate the output of the model-checker back to the original model.

The papers in this special issue concern the development of model-checking frameworks suitable for expressing and checking requirements models. They concentrate on validating requirements models expressed in various modelling languages via connections with (extensions of) existing model-checking engines.

The first paper, by Choi, Rayadurgam and Heimdahl, looks at the problem of finding abstractions for requirements models with large state spaces. Requirements for control systems often include variables with large input domains as well as interrelated numeric constraints, e.g., *altitude* < 1.05 * *threshold*, where *altitude* and *threshold* range between 0 ... 40,000 and 2000 ... 35,000, respectively. Model checking such systems can be successful only in the presence of powerful abstractions that reduce the domain of such variables while preserving their relationships. The paper addresses this problem by proposing two types of abstractions: domain reduction and trajectory reduction. Both abstractions are computed statically, and the reduced system can then be verified by any model-checking tool. The first technique can be applied to systems with no data constraints. It is based on a data equivalence relation using the transition conditions on the input variables to partition the infinite input domain into a finite set of partitions from which one representative input is selected. The technique results in abstractions that preserve all CTL* properties of the original system. In systems with data constraints, trajectory reduction maps a possibly infinite set of input variable trajectories through the state space to a single representative trajectory in a finite domain. This technique can handle systems with non-linear constraints, and abstractions generated with this technique preserve ACTL* properties of the original system.

The second paper, by Eshuis, Jansen and Wieringa, is a step towards a framework for model checking hierarchical state machines. The paper considers the problem of how model checking can be applied to analyse object-oriented requirements models, in which a variant of statecharts is used to capture behavioural aspects of objects. Model checking in such a notation is an obvious analysis step, but the case study presented in the paper demonstrates that it is not a straightforward task to couple an existing model-checking engine with the modelling tool. The first part of the paper explores an appropriate semantics for object-oriented requirements models. For statechart-like notations to be used in requirements modelling, they need a requirements-level semantics, that is, they must abstract away from any implementation details, for example by assuming a perfect technology. The proposed semantics describes how the configurations of statecharts evolve in response to external events. For the case study, the authors used existing model-checking engines to analyse their models. The results show both the potential of model checking for debugging and validating requirements models, and also the limitations of current model-checking technology. In particular, they show how a knowledge of the model-checking engine is needed to select appropriate encoding of the models, and suggest that no one engine is sufficient for the range of analyses needed in requirements engineering.

The final paper, by Campbell, Cheng and McUmber, describes a model-checking framework for UML. Analysis of UML models is complicated by the lack of a precise semantics for most of the UML notations. Campbell et al. solve this problem by making their framework adaptable for different choices of semantics, suitable for different application domains. The authors demonstrate the framework using a case study of an adaptive cruise control system. The case study shows how various kinds of analysis can be performed on the formalised UML models to reveal different kinds of problem in the models, and clearly illustrates the role of model checking in requirements analysis.

The papers in this issue together represent a snapshot of the current state of the field. The application of model checking in requirements engineering is still in its infancy, and much current work focuses on the development of model-checking frameworks. In particular, there is clearly a gap between the kinds of models and modelling languages currently used in requirements engineering, and the capabilities of existing model-checking engines. But, as these papers show, that gap is closing.

There are many additional problems that need to be overcome before model checking becomes a routine analysis technique for requirements models. One major problem is the trade-off between expressive power of the modelling languages and their decidability by model-checking techniques. In addition, recent research expanded the potential of the use of model checking in requirements engineering beyond analysing classical state machine models. We now have techniques for reasoning about high-level goals [19], use of the model-checker for test case generation [20,21], formal support for analysing incomplete and inconsistent models [22,23] and the use of the model-checker for general model exploration via query checking [24]. Case studies and further research would be necessary to determine which of these applications of model-checking technology will really scale up to realistic problems, and which will remain confined to toy examples.

We feel that it is also necessary to mention that model checking may not be appropriate for many problems in requirements engineering, because of the level of formality required. As with any formal technique, it is only appropriate when a precise model can be developed, and properties to be checked can be stated precisely.

Acknowledgements. We would like to thank all the authors who submitted papers to this special issue, and especially to the following people who gave their valuable time to provide us with detailed reviews: Myla Archer, Jo Atlee, Ramesh Bharadwaj, Tevfik Bultan, Betty Cheng, Nancy Day, Benet Devereux, Matthew Dwyer, Dimitra Giannakopoulou, Mats Heimdahl, Gerard Holzmann, Michael Huth, Michael Jackson, Somesh Jha, Jeff Magee, Jeff Offutt, Corina Pasareanu, John Penix, Marco Pistore, Alessandra Russo, Mark Ryan, Margaret Smith, Willem Visser, Michal Young.

References

1. Clarke EM, Emerson EA, Sistla AP. Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans Programming Lang Syst 1986;8(2):244–263
2. Vardi MY, Wolper P. An automata-theoretic approach to automatic program verification. In: Proceedings of the 1st symposium on logic in computer science, Cambridge MA, 1986, pp 322–331
3. Craigen D, Gerhart S, Ralston T. Formal methods reality check: industrial usage. IEEE Trans Software Eng 1995;21(2):90–98
4. Jackson D, Wing J. Lightweight formal methods. IEEE Computer 1996;April:21–22
5. Atlee JM, Gannon J. State-based model checking of event-driven system requirements. IEEE Trans Software Eng 1993;19(1):22–40
6. Cheung SC, Kramer J. Checking subsystem safety properties in compositional reachability analysis. In: Proceedings of the 18th IEEE international conference on software engineering (ICSE-18), Berlin. IEEE Computer Society Press, Los Alamitos, CA, 1996
7. Godefroid P. Model checking for programming languages using VeriSoft. In: Proceedings of the symposium on principles of programming languages (POPL'97), 1997, pp 174–186
8. Corbett J, Dwyer M, Hatcliff J, Laubach S, Pasareanu C, Robby, Zheng H. Bandera: extracting finite-state models from Java

source code. In: Proceedings of the 22nd international conference on software engineering. Limerick, Ireland, June 2000, ACM Press

9. Ball T, Rajamani S. Bebop: a symbolic model checker for Boolean programs. In: Proceedings of SPIN 2000 workshop on model checking of software, August–September 2000. Lecture Notes in Computer Science 1885. Springer, Berlin, 2000, pp 113–130
10. Clarke EM, Wing J. Formal methods: state of the art and future directions. ACM Comput Surveys 1996;28(4):626–643
11. Holzmann G. A practical method for verifying event-driven software. In: Proceedings of the 21st international conference on software engineering (ICSE'99), May 1999, pp 597–607
12. Dwyer M, Carr V, Hines L. Model checking graphical user interfaces using abstractions. In: Proceedings of foundations of software engineering, Zurich, Switzerland, September 1997
13. Schneider F, Easterbrook SM, Holzmann GJ. Validating requirements for fault tolerant systems using model checking. In: Proceedings of the 3rd IEEE conference on requirements engineering, Colorado Springs, CO, April 1998
14. Clarke E, Grumberg O, Peled D. Model checking. MIT Press, Cambridge, MA, 1999
15. Clarke E, Schlingloff H. Model checking. In: Robinson J, Voronkov A (eds). Handbook of automated reasoning. Elsevier, Amsterdam, 2000
16. Bharadwaj R, Heitmeyer C. Model checking complete requirements specifications using abstraction. J Automated Software Eng 1999;6(1)37–68
17. Dwyer M, Avrunin G, Corbett J. Patterns in property specifications for finite-state verification. In: Proceedings of the 21st international conference on software engineering. Los Angeles, May 1999, ACM Press
18. Sreemani T, Atlee JM. Feasibility of model checking software requirements: a case study'. In: Proceedings of COMPASS'96, Gaithersburg, MD, June 1996
19. Fuxman A, Pistore M, Mylopoulos J, Traverso P. Model checking early requirements specifications in Tropos. In: Proceedings of the 5th IEEE international symposium on requirements engineering (RE'01), Toronto, Canada, August 2001. IEEE Computer Society Press, Los Alamitos, CA, pp 173–181
20. Gargantini A, Heitmeyer C. Using model checking to generate tests from requirements specifications. In: Proceedings of the joint 7th European software engineering conference and 7th ACM SIGSOFT international symposium on foundations of software engineering (ESEC/FSE99), Toulouse, France, September 1999. ACM Press, New York, pp 146–162
21. Rayadurgam S, Heimdahl MPE. Coverage based test-case generation using model checkers. In: Proceedings of the 8th annual IEEE international conference and workshop on the engineering of computer based systems (ECBS'01), Washington, DC, April 2001. IEEE Computer Society Press, Los Alamitos, 2001, pp 83–93
22. Easterbrook S, Chechik M. A framework for multi-valued reasoning over inconsistent viewpoints. In: Proceedings of the international conference on software engineering (ICSE'01), Toronto, Canada, May 2001. IEEE Computer Society Press, Los Alamitos, CA, pp 411–420
23. Bruns G, Godefroid P. Generalized model checking: reasoning about partial state spaces'. In: Palamidessi C (ed). Proceedings of the 11th international conference on concurrency theory (CONCUR'00), University Park, PA, August 2000. Lecture Notes in Computer Science 1877. Springer, Berlin, 2000, pp 168–182
24. Gurfinkel A, Devereux B, Chechik M. Model exploration with temporal logic query checking. In: Proceedings of SIGSOFT conference on foundations of software engineering (FSE'02), Charleston, SC, November 2002. ACM Press, New York (to appear)