Int J Softw Tools Technol Transfer (2004) 6: 257–259 / Digital Object Identifier (DOI) 10.1007/s10009-004-0168-3

Special section on the algorithmics of software model checking

Introductory paper

Matthew Dwyer[1], **Stefan Leue**[2]

[1] Department of Computer Science and Engineering, 256 Avery Hall, University of Nebraska, Lincoln, NE 68588-0115, USA
e-mail: dwyer@cse.unl.edu
[2] Department of Computer and Information Science, University of Konstanz, 78457 Konstanz, Germany
e-mail: Stefan.Leue@uni-konstanz.de

Published online: 17 November 2004 –

1 Introduction

The term "software model checking" has recently been coined to refer to a flourishing area of research in software verification – the formal, automated analysis of program source code. Software model checking is considered an important application of classical model checking, where the model of a software system is analyzed in an automated fashion for compliance with a property specification. While classical model checking assumes the existence of an abstract model of the software system to be analyzed, in software model checking the emphasis is on directly analyzing program code given in a standard programming language, such as Java or C. This introduces a variety of significant obstacles, chief among them the efficient treatment of the complex data, e.g., heap structured data, and control constructs, e.g., procedure calls and exception handling, found in modern programming languages. These obstacles can also be viewed as opportunities for adapting traditional model checking data structures and algorithms to exploit the particular semantics of programming language constructs to gain improved performance. Moreover, while classical model checking emphasizes proving a model correct as the primary objective, an increasingly widely held view is that model checkers can function effectively as anomaly detectors or bug finders, i.e., they locate and explain undesired behavior of the software.

This special section is the second devoted to publishing revised versions of contributions first presented at the International SPIN Workshop Series on Model Checking Software. In recent years this series of workshops has broadened its scope from focusing on the model checker SPIN to covering software model checking technology in general. The editorial introduction by Havelund and Visser to the first STTT special section devoted to SPIN papers [11] provides an excellent overview of the foundational ideas underlying software model checking. That special section was based on papers presented at the 7th International SPIN Workshop held at Stanford University (USA) in August/September 2001. Authors of well-regarded papers from the 8th International SPIN Workshop held in Toronto (Canada), colocated with ICSE 2001 on 10–11 May 2001, and the 9th International SPIN Workshop on Model Checking Software, held 11–13 April 2002 in Grenoble (France) as a satellite event of ETAPS 2002, were invited to submit to this special issue. All three of the papers included here have been extended to include significant new content and have undergone an independent round of reviewing.[1]

2 Software model checking

Model checking is enjoying much attention in academia and industry due to the fact that it can perform deep-semantic reasoning about program behavior in a fully automated fashion, i.e., it does not require interaction from the designer once the model and the property specification have been created. This is particularly valuable for validating concurrent programs where it is difficult to drive traditional testing techniques to exercise unlikely, but still possible, "corner cases" in a program's logic. In addition, when a property violation has been found, most model checkers return an offending trace of the system's behavior, called a counterexample, that helps in locating the cause of the property violation.

There are two prevailing model checking technologies. In symbolic model checking [4, 15], the state space and the state transition function are represented by binary decision diagrams and the property verification corresponds to a symbolic fixed-point computation on the

[1] Reviewing for the paper submitted by Edelkamp, Leue, and Lluch-Lafuente was handled solely by Matthew Dwyer.

set of reachable system states. In explicit-state model checking [5, 12], the system states are explicitly enumerated using a next-state function and property verification corresponds to a systematic search of the state space. Explicit-state model checking has proven to deal very successfully with the irregularly structured models that software verification problems entail. The increasing maturity of model checking technology is documented through the availability of various monographs [3, 5, 16] and model checking tools such as SPIN [12], SMV [15], Bandera [7], JPF [10, 17], and UPPAAL [2]. An earlier STTT special section focused on the pragmatics of model checking [6].

The more direct link in software model checking to the software artifact to be analyzed offers various advantages over model-based classical model checking. First, the manual model-building step is avoided. This relieves the software engineer of the challenge of building a suitable model based on adequate and sound abstractions. Also, when a property violation is found, it is much easier to trace a counterexample that has been produced back to the software code, which enhances error explanation. On the downside, the state spaces of software models are either very large or even infinite. The size of the state space is due to the use of variables over finite, but very large, data domains and due to the concurrent nature of many software systems. Unboundedness of the size of the state space is due, in part, to recursive function and procedure invocations.

Historically, one of the first model checkers to directly analyze software code was the tool Verisoft [9], which offers an incomplete model checking algorithm for verifying safety properties of C programs. It implements the concept of memoryless model checking, which means that only a small finite history of the state space exploration is retained. The tool has been successfully used to analyze telecommunications code for software property violations, in particular deadlock detection. SLAM [1], developed at Microsoft Research, is a software model checking toolset for C programs based on the idea of boolean abstraction. It is capable of checking implementations of real Windows XP device drivers for sequencing properties described as automata; recent experiments with SLAM have analyzed programs of more than 20,000 source lines. While SLAM treats sequential code, the SPIN-based FeaVer system [13] extracts SPIN models from concurrent C code. FeaVer served as a very effective complement to traditional testing in the development of the control software for a voiceover-IP software switch at Lucent Technologies. Much attention has recently also been devoted to the analysis of Java code. The Bandera [7] and Java PathFinder [10, 17] toolsets are the most prominent examples of Java model checkers. The primary characteristic of Bandera is that it combines a variety of program analysis and transformation phases, e.g., slicing and data abstraction, to reduce the model to a form that is significantly more efficient to model check. Java PathFinder is implemented as a customized *stateful* Java interpreter, and as such it can process nearly any Java source code; it has been used by NASA in the verification of mission-critical Java code.

3 Customizing model checking data structures and algorithms for *software*

The successful application of software model checking technology in practical software design processes hinges on the availability of efficient model checking algorithms that are capable of dealing with the tremendous state space sizes that the software systems to be analyzed entail. For many software systems, enormous state spaces remain even after sophisticated abstraction techniques have been applied. This special section focuses on three successful techniques that are capable of significantly improving the performance of existing model checking algorithms in dealing with the complexities of software.

A focus of research has recently been the use of heuristics-guided, informed search algorithms as a replacement of the otherwise uninformed state-space-traversal algorithms. The paper by Groce and Visser produces heuristics based on the structure of the underlying Java code in order to improve the efficiency of finding errors. The idea of this approach is to use the control and concurrency structure of the program in order to achieve a better coverage of the state space when looking for concurrency-related properties, such as deadlock detection. The objective of these heuristics is similar to coverage-increasing heuristics in software testing: a higher or more evenly distributed coverage of the state space increases the chances of finding errors within the time and memory limits available. The most important structural heuristics that these authors suggest include a branch-counting heuristic and a heuristic that attempts to maximize the number of thread switches in order to more easily find concurrency-related faults. The authors implement their heuristics in the Java PathFinder model checker and apply their approach to the DEOS operating system and reengineered Java code of the Deep Space 1 spacecraft as case studies.

The paper by Edelkamp, Leue, and Lluch-Lafuente also addresses heuristics-directed model checking. However, unlike the paper by Groce and Visser, the authors use property-oriented heuristics that help in finding shorter or even optimally short counterexamples when comparing with the standard depth-first search (DFS) strategy commonly used in explicit-state model checkers. Short counterexamples aid in determining the causes of faults in the model since they are easier to comprehend than the typically very long counterexamples obtained through DFS-based model checking. In this paper the authors extend their previous work on directed explicit-state model checking [8] by reconciling it with partial-order concepts, in particular partial-order reduction. This form of automated state space compaction is essential

to the success of explicit-state model checking in analyzing concurrent software models, and hence it needs to be proven that this reduction method is compatible with the directed model checking approach. The authors also introduce heuristics based on Hamming distances between a given error trail and the current system state that help in reducing the length of precomputed counterexamples. The authors apply their approach to various examples of models of real-life concurrent software systems and have implemented their methods in a heuristic extension of SPIN, called HSF-SPIN.

Finally, the paper by Iosif proposes a method to reduce the state space of dynamic concurrent programs. These types of programs are typical for object-oriented systems written in languages like C++ or Java in which object instances are generated and terminated dynamically during execution time of the code. In the paper, the author proposes criteria for determining symmetries between object instances with respect to the threads in which they execute and the heaps on which their data are allocated. The authors also prove that their symmetry reductions are compatible with partial-order reductions. They have implemented their reduction technique in the model checker dSPIN [14], a variant of SPIN that is capable of dealing very efficiently with dynamic systems structures. On the case studies that the authors present significant reductions in the size of the state spaces can be observed.

References

1. Ball T, Rajamani SK (2001) Automatically validating temporal safety properties of interfaces. In: Proceedings of SPIN 2001. Lecture notes in computer science, vol 2057. Springer, Berlin Heidelberg New York
2. Bengtsson J, Larsen KG, Larsson F, Pettersson P, Yi W (1997) UPPAAL – a tool suite for automatic verification of real-time systems. In: Alur R, Henzinger TA, Sonntag ED (eds) Hybrid systems III – Verification and control. Lecture notes in computer science, vol 1066. Springer, Berlin Heidelberg New York, pp 232–243
3. Bérard B, Finkel M, Bidoit A, Laroussine F, Petit A, Petrucci L, Schoenebelen P, McKenzie P (2001) Systems and software verification. Springer, Berlin Heidelberg New York
4. Clarke EM, McMillan KL, Dill DL, Hwang LJ (1992) Symbolic model checking: 10^{20} states and beyond. Inf Comput 98(2):142–170
5. Clarke EM, Grumberg O, Peled DA (1999) Model checking. MIT Press, Cambridge, MA
6. Cleaveland R (1999) Pragmatics of model checking: an STTT special section. Int J Softw Tools Technol Transfer 2(3):208–218
7. Corbett JC, Dwyer MB, Hatcliff J, Laubach S, Pasareanu CS, Robby, Zheng H (2000) Bandera: extracting finite-state models from Java source code. In: 22nd IEEE international conference on software engineering (ICSE)
8. Edelkamp S, Leue S, Lluch-Lafuente A (2004) Directed explicit-state model checking in the validation of communication protocols. Int J Softw Tools Technol Transfer 5(2–3):247–267. DOI: 10.1007/s10009-002-0104-3
9. Godefroid P (2003) Software model checking: the VeriSoft approach. Technical report, Bell Labs Technical Memorandum ITD-03-44189G. Formal Meth Syst Des (in press)
10. Havelund K, Pressburger T (2000) Model checking Java programs using Java PathFinder. Int J Softw Tools Technol Transfer 2(4):366–381
11. Havelund K, Visser W (2002) Program model checking as a new trend. Int J Softw Tools Technol Transfer 4(1):8–20. DOI: 10.1007/s10009-002-0080-7
12. Holzmann GJ (2003) The SPIN model checker, primer and reference manual. Addison-Wesley, Reading, MA
13. Holzmann GJ, Smith MH (2000) Automating software feature verification. Bell Labs Tech J 5(2):72–87
14. Iosif R, Sisto R (1999) dSPIN: A dynamic extension of SPIN. In: Proceedings of the 6th SPIN workshop. Lecture notes in computer science, vol 1680. Springer, Berlin Heidelberg New York, pp 261–276
15. McMillan KL (1993) Symbolic model checking. Kluwer, Dordrecht
16. Peled DA (2001) Software reliability methods. Springer, Berlin Heidelberg New York
17. Visser W, Havelund K, Brat G, Park S (2000) Model checking programs. In: IEEE International conference on automated software engineering. September 2000