Special section on tools and algorithms for the construction and analysis of systems

Guest editors’ introduction: Advancements and extensions of verification techniques

Joost-Pieter Katoen¹, Perdita Stevens²

¹University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science, Formal Methods and Tools Group, P.O. Box 217, 7500 AE Enschede, The Netherlands
E-mail: katoen@cs.utwente.nl
²University of Edinburgh, School of Informatics, JCMB, King’s Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK
E-mail: Perdita.Stevens@ed.ac.uk

Published online: 29 July 2004 – © Springer-Verlag 2004

Abstract. This special section is devoted to a selection of journal versions of papers that appeared originally in the Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), which took place in Grenoble, France in April 2002 as a constituent event of the European joint conferences on Theory and Practice of Software (ETAPS). All papers are relevant to the field of systems validation. The first three papers advance and extend model-checking techniques, the fourth presents algorithms for run-time verification, and the last paper is about animation and test generation for formal system specifications.

Keywords: Model checking – Probabilistic systems – Run-time verification – Testing – Tools

The five selected high-quality papers are representative of the recent advancements and extensions to verification techniques, and systems validation more generally. The first three papers present improvements and extensions to model checking, while the last two papers deal with test generation from formal system specifications and the automated verification of tests, respectively. Model checking is a successful verification technique in industry that is based on a systematic investigation of the state space. Due to its ability to generate diagnostic feedback in the form of counterexamples, model checking is often used as an intelligent debugging tool. However, the diagnostic information provided is sometimes limited and hard to understand. In particular for systems with a larger state space, either it may be problematic to model check them at all (no feedback) or the counterexamples are typically incomprehensible. Other important restrictions are that model checking typically focuses on checking qualitative properties (and does not allow for quantitative properties such as quality-of-service aspects) and checks models rather than real implementations in hardware and software (or both). In this special issue contributions are collected that report on advancements of validation techniques in overcoming these problems.

1 Extending and advancing model checking

The first two papers in this group are concerned with improving existing model-checking algorithms. The first paper proposes an improvement to counterexample generation, while the second paper is concerned with the use of genetic algorithms to enable the systematic exploration of state spaces that are some orders of magnitude larger than those that currently can be handled. The
last paper in this group presents a tool for model checking probabilistic models, models in which (some) state transitions encode the probability of making a transition between states rather than just the existence of such a transition.

*Fate and free will in error traces* [8] by HoonSang Jin, Kavita Ravi and Fabio Somenzi.

One of the main strengths of model checking is the possibility it presents for the automated generation of counterexamples when a property is refuted. Notwithstanding the argument that counterexamples are crucial to understanding system bugs, counterexample traces may sometimes become too large and too complex for manual inspection and analysis. This paper describes an approach to segmenting a counterexample into sub-parts that describe fate (i.e. inevitable system development toward the error) and sub-parts corresponding to free will (i.e. choices that, if avoided, might have prevented the error). It is assumed that properties are specified in a branching-time temporal logic like CTL* [4]. An algorithm is presented to generate counterexamples in the form of fate and free segments. The paper reports on some case studies using a revalidation of this algorithm in the VIS [2] model checker. Additionally, an alternative, computationally more involved, algorithm is presented to obtain a fate-and-free counterexample with a minimal number of free segments. This algorithm serves as a basis to generate the shortest counterexamples.


The major practical restriction of model checking is the often excessive size of the state space of the model at hand. Although various effective techniques have been developed in the past, there is still an urgent need to investigate whether state spaces that are several orders of magnitude larger than those currently being handled can still be treated. This paper proposes to prune the size of the state space by exploiting heuristic search strategies that are based on genetic algorithms and presents the necessary amendments to model checking to enable this approach. The technique is implemented in the (stateless) model-checking engine VeriSoft [5], a verifier for concurrent C software. The paper reports extensively on experiments conducted and discusses heuristics for detecting deadlocks and assertion violations using genetic algorithms. Case studies include a public-key authentication algorithm and the well-known dining philosophers problem and show significant progress over random search algorithms.

*Probabilistic symbolic model checking with PRISM: a hybrid approach* [9] by Marta Kwiatkowska, Gethin Norman and Dave Parker.

While model checking traditionally focuses on the verification of functional (i.e. qualitative) properties of the system, in practice quantitative aspects are also relevant. In particular, systems are subject to various phenomena, such as message loss or processor failure, that have a stochastic nature. In the last decade, this has led to a growing interest in the verification of probabilistic systems. This paper reports on a software tool called PRISM that allows for model checking of probabilistic systems. Properties are expressed in probabilistic extensions to CTL, whereas models of various kinds are supported, ranging from Markov chains (both discrete-time and continuous-time) to Markov decision processes, a model in which probabilistic and non-deterministic branching co-exist. The paper reports on the tool and details the use of a combination of multi-terminal binary decision diagrams (for storing the state space) and explicit techniques (to store the solution vector) to diminish the state space explosion problem. Several case studies show the effectiveness of this combined approach.

2 Test generation and verification

One of the prerequisites of model-based verification techniques such as model checking is the availability of a system model that describes the possible system behaviour at a sufficiently detailed level. These models are typically described in modelling formalisms tailored to specific application domains. Prior to carrying out an extensive (and costly) verification, it is useful to perform a first sanity check by simulating or animating the model (i.e. its description). Furthermore, once the verification phase has finished, it is important to have the means to validate whether the software (or hardware, or even combination of both) fulfills the original requirements. At this stage, testing techniques are prominent.

The first paper in this group is concerned with the animation of, and the automated generation of tests from, formal specifications, while the second paper is concerned with the verification of executions of real systems.

*CLPS-B – A constraint solver to animate a B specification* [3] by Fabrice Bouquet, Bruno Legeard and Fabien Peureux.

This paper focuses on the B method [1], a modelling formalism based on first-order logic extended with set constructors and relations. In particular, the paper presents a method for evaluating a B formal specification using constraint programming. It describes techniques for animating finite B specifications (i.e. simulating the execution of operations) as well as methods for automated test generation from B specifications. Animation is done by keeping a representation of a set of