Int J Softw Tools Technol Transfer (2004) 6: 1–3 / Digital Object Identifier (DOI) 10.1007/s10009-004-0144-y

Special section on verification, model checking, and abstract interpretation

Preface by the section editors

Lenore Zuck, Paul Attie, Agostino Cortesi

Published online: 1 July 2004 – © Springer-Verlag 2004

Abstract. The papers in this special section present a sample of recent approaches to modeling and verification of software-based systems. This research was initially presented at the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '03). The choice of papers for the issue was based on their merit as well as on the fact that, as a group, they represent the main current research areas in the field of software-based systems.

1 Introduction

The crucial role of rigorous and formal methods in specification, verification, and analysis of software systems has long been recognized. The past decade has witnessed increasing applications of formal methods in the development of safety-critical and hardware systems. We expect this trend to continue as more consumer goods are "software embedded", and expected to conform to high reliability standards.

Formal methods for software can be compared and evaluated according to the following criteria:

– Degree of automation, with deductive (manual) methods at one end and fully automatic methods at the other;
– Methodology used, where currently the leading methods are model checking and abstract interpretation;
– Degree of abstraction applied, with no abstraction at one end and finite-state abstraction at the other;
– Ambitiousness of goals, with verification at one end and synthesis at the other;
– Development stage of applied methods, with static analysis methods at one end and run-time methods at the other.

Degree of automation: from deductive to automatic

Deductive methods, where one builds a proof from the ground up using rules of deductive reasoning, are extremely powerful; yet they constitute an art requiring considerable expertise and creativity. While new automatic theorem provers allow one to check proofs, construction of the proofs is often an elusive and laborious task.

In contrast, fully automatic methods, as their name implies, require no manual intervention. However, they can only be directly applied to problems that are decidable, which rules out infinite-state software systems as well as many systems that are just too large to handle by automatic methods. Hardware systems, which are usually simpler than software systems, are good candidates for automatic methods.

Some methods are at neither extreme of this dichotomy. For example, many of the new theorem provers allow the user to "suggest" to the system the proof strategy. While such provers are not strictly automatic, they often do not require the same level of expertise expected from a user of a closely guided theorem prover.

Formal methods: from model checking to abstract interpretation

The main formal methods that have been applied to the verification, analysis, and synthesis of software systems are model checking and abstract interpretation. Abstract interpretation is mainly applied to the verification of systems with infinite-data domains. Roughly speaking, a system is *abstracted* into a smaller system such that properties of the abstract system imply corresponding properties of the concrete system. The abstraction/refinement is ideally achieved by means of a Galois connection. Abstract interpretation methods have been successfully applied to sequential programs over infinite-data domains.

Model checking constructs a finite-state model of the system and algorithmically checks whether it is a model of some specification. When the result is negative, a counterexample is produced that shows where the model violates the specification. When the result is positive, one may safely conclude that the verified system is correct. Model checking has been successfully applied to hardware systems and to systems with a small number of states and intricate control structure (e.g., forms of concurrency) that abstract interpretation techniques cannot easily accommodate.

Of course, a formal method can combine the two. For example, there is an increasing body of work on systems that are too large to be handled by model checking and whose control structure is too complex to be handled by abstract interpretation. Using abstract interpretation techniques, one can abstract such a system into a (possibly intricate-control) finite-data system and analyze the new systems using model checking.

Degree of abstraction: from none to finite-state abstraction

The high complexity, or undecidability, of run-time properties of programs yields the need for every analysis technique to face the tradeoff between accuracy and efficiency. Accuracy of the analysis can be tuned according to different degrees of abstraction. Very rarely an exhaustive approach (leading to exactness) can be applied, namely, when the set of possible executions is somehow bounded. In general, both in abstract interpretation and in model checking and related techniques, a suitable abstract representation of concrete execution states has to be designed that preserves correctness (in terms of overapproximation) and termination of the analysis.

Goals: from testing to synthesis

The most popular use of formal mehods is *testing*: given a software module, identify a set of tests such that, if the system passes all tests in the set, we can be reasonably sure the system is correct. Ideally, one would like to obtain *verification*: given a software module and a property, formally show that *all* executions of the module satisfy the property. Model checking and abstract interpretation are both analytic *debugging* methods. If they succeed in verifying a property, one can conclude that an abstract model of the system satisfies (possibly an abstract model of) the property. If they fail, one must check whether it is the abstraction that creates a ‘false negative’ or the concrete program. *Synthetic* methods, on the other hand, derive a program from its specifications. There are several ways to obtain the derivation, e.g., *stepwise-refinement* starts with a high-level operational specification and gradually refines it into an implementation.

Refinement is usually accomplished using two main paradigms, *preorders* and *equivalences.* In preorder-based refinement, an *inclusion preorder* is defined based on behaviors, and an implementation is considered correct if its behaviors are included in those of the specification. The behavioral inclusion is usually established by using a simulation relation. The overall methodology, then, is to start with a high-level operational specification and construct a sequence of intermediate system descriptions such that the behavior of each description is included in the behavior of the previous one. Gradually, implementation details are added in, so that the last description constitutes a suitable implementation. Such implementation details can deal with nonfunctional properties, e.g., distribution, fault-tolerance, performance, etc.

In *equivalence-based* refinement, algebraic reasoning is used to replace a component of the current system with another one that has equivalent behavior (w.r.t. a defined notion of equivalent behavior) but that is more detailed, i.e., closer to an implementation.

When applied: from static to run time

Traditionally, verification is *static* – a software system is given in full detail and possibly abstracted, and some properties are analyzed/verified. When systems are sufficiently complex, static methods may often fail to deal with them efficiently. For systems that are not safety critical, where detection of errors shortly before or even when they occur still allows for some actions that will avoid catastrophic consequences, run-time verification offers an attractive alternative: rather than proving the system is correct, monitor it when it is running and notify of any detected errors. The main weakness of run-time methods is that only a restricted class of properties can be monitored.

There are, of course, hybrid approaches, e.g., translation validation, where instead of verifying an optimizing compiler each of its executions is verified against its specifications.

The papers

The papers chosen for this special issue all present new research in the application of formal methods to specific problem domains. The five papers here represent verification, synthesis, and a combination of model checking, abstract interpretation, and static and run-time verification. The application domains are just as broad and include compilers, telecommunication protocols, security protocols, distributed systems, and timed systems. Preliminary versions of the papers were presented at the

Fourth Conference on Verification, Model Checking, and Abstract Interpretation held at New York University in January 2003 [1].

Based on the criteria described above, the papers can be described by the following table.

#	degree	AI/MC	Abstraction	Goals	Stage
1	Full	MC	Full	T/Syn	S and RT
2	Some	AI	Tunable	V	Between S and RT
3	Full	MC	Some	V	S
4	Semi	TP	(Simulation)	V	S
5	Full	MC	Full	Perfor.	S

In the table, *MC* stands for model checking, *AI* for abstract interpretation, *TP* for theorem proving, *T* for testing, *V* for verification, *Syn* for synthesis, *S* for static analysis, and *RT* for run time. For example, the first paper describes a fully automatic method, using model checking and full abstraction, for testing at both the static analysis and run-time stages. The fourth paper uses abstraction by simulation, and the goal of the fifth paper is performance enhancement.

Below is a short summary of the papers.

The first paper, "Behavior-based Model Construction" by Hardi Hungar and Bernhard Steffen, presents a technique for constructing a model of a system by observing behaviors of its implementation. The framework combines in a novel way machine learning, abstract interpretation, verification, testing, and model checking. The work is motivated by large-scale telecommunication applications that are not amenable to existing verification techniques. The paper focuses on the machine-learning aspect of the proposed process that provides an abstraction of the system and outlines the use of other techniques such as testing and model checking to enhance the learning process. The resulting combination is very promising from both practical and theoretical standpoints.

The second paper, "Certification of Compiler Assembly Code by Invariant Translation" by Xavier Rival, presents an abstract-interpretation-based method for automatically verifying that assembly code generated by a compiler preserves invariant (safety) properties of the source code. The approach is based on translation validation, where, rather than verifying a translator, which is often an attainable task, one verifies the correctness of each translation. The paper proposes to use static analysis tools to generate invariants for the source code and translate them into *candidate invariants* using debugging information generated during compilation. The candidate invariants are then checked against the compiled code, and when they match, one may conclude that the source code and the compiled code display similar (abstract) behaviors. The approach was tested on a prototype implementation, and the description of the methodology is accompanied by a rich yet easy-to-follow running example.

The third paper, "A Logical Encoding of the π-calculus: Model Checking Mobile Processes Using Tabled Resolution" by Ping Yang, C.R. Ramakrishnan, and Scott Smolka, describes MMC – a π-calculus-based model checker for mobile systems that uses the XSB tabled logic programming engine to directly embed several formal systems (e.g., the π- and spi-calculi and the modal μ-calculus) as logic programs. The paper shows that these logic programs correctly implement the existing formal systems and that some attacks on security protocols can be found. The performance of the proposed system on some synthetic benchmarks improves upon that of the Mobility Workbench, the first automated analysis tool for the π-calculus.

The fourth paper, "Using Simulated Execution in Verifying Distributed Algorithms" by Toh Ne Win, Michael Ernst, Stephen Garland, Dilsun Kirli, and Nancy Lynch, explores the use of simulation as a precursor and aid to formal verification. Distributed algorithms are expressed as I/O automata (IOA) and can be simulated using the IOA environment together with the Daikon invariant detection tool. When a distributed algorithm is simulated, Daikon outputs a set of guessed candidate invariants. These can then be verified to be actual invariants, for example using the connection between IOA and the Larch Prover. Finally, given a candidate forward simulation relation from an implementation to a specification, the IOA toolset can help in checking the validity of the forward simulation by means of "paired execution": it generates an execution of the implementation together with a corresponding execution of the specification. The user can then inspect these executions and detect problems with the forward simulation.

The fifth paper, "Efficient verification of timed automata with BDD-like data structures" by Farn Wang, investigates the design of data structures based on BDDs for the efficient model checking of timed automata. It proposes a new structure, the clock-restriction diagram, which has been implemented in the author's real-time model checker. The paper presents extensive experimental data on several benchmarks that compare clock-restriction diagrams with other popular data structures for timed automata. The performance figures show that clock-restriction diagrams provide performance benefits in many cases.

References

1. Zuck LD, Attie PC, Cortesi PD, Mukhopadhyay S (eds) (2003) Proceedings of the 4th international conference on verification, model checking, and abstract interpretation (VMCAI 2003), New York. Lecture notes in computer science, vol 2575. Springer, Berlin Heidelberg New York