Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Alternative Approaches to Symbolic Verification: Preface by the Section Editor ⋆

Rance Cleaveland

Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794-4400, USA

Abstract. The papers in this special section present a sampling of new symbolic approaches for determining whether or not a system satisfies its specification. Abstracts of these articles appeared originally in the Proceedings of the 1999 Symposium on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99).

Key words: model checking – symbolic verification – deductive reasoning – system verification – formal methods

1 Introduction

System verification and validation refer to a host of activities that system developers perform in order to certify that a system implementation conforms to its designers' expectations. *Validation* typically involves the use of incomplete, informal, but tractable techniques: examples range from code/design reviews to testing at various levels. In situations in which a specification is missing or ambiguous, validation often represents the only possibility for certifying system behavior. *Verification* has a more formal, mathematical connotation; verifying a system typically requires proving a theorem that a system satisfies properties desired of it. For verification to be possible one typically requires that the implementation and specification be defined in a mathematically rigorous manner. The term *formal methods* is traditionally synonomous with verification, although the phrase's meaning has broadened in recent years to include other mathematically-driven system construction and analysis techniques.

One may distinguish between *symbolic* and *concrete* approaches to system validation and verification. Symbolic approaches rely on the analysis of a syntactic system artifact, which may be a program or specification. Concrete approaches rely on an analysis of explicit system behavior: test cases together with expected results exemplify such concrete artifacts.

While system validation has traditionally used a mixture of symbolic and concrete techniques (code reviews feature an informal analysis of a symbolic artifact, namely source code, while testing focuses on the concrete activity of running the program on tests and assessing the results), until the mid-1980s system verification focused exclusively on the use of symbolic techniques to prove system correctness [4–6]. On the one hand, the use of symbolic reasoning represented a natural consequence of system verification as involving the proof of theorems about systems. On the other hand, the undecidability of general verification problems suggested that the only way systems could be verified was via human-constructed proofs undertaken possibly with machine assistance in the form of theorem provers.

Model checking [3,8] represented a departure from the symbolic point of view on verification. The original model-checking approaches were concrete; they explicitly represented the entire operational behavior of systems and explored the resulting data structures to determine whether or not properties held. (These approaches are sometimes called *explicit-state* because they manipulate descriptors for system states directly.) Model checking's reliance on state enumeration limited its applicability to finite-state systems, but it did not require users to conduct proofs of correctness by hand; instead, the model checker in essence attempted to prove systems correct by trying "all possible" proofs. Model-checking tools' simplicity of use spurred much research into model checking, although critics of the technology noted its inability to "scale" (i.e. handle large system descriptions) because of the large memory requirements needed to store system state spaces.

In a fascinating twist, one of the major advances in addressing the "scale problem" in model checking involved the re-introduction of symbolic reasoning into model-checking routines. In its original form [2] *symbolic model checking*

⋆ Research supported in part by NSF grants CCR-9996806 and CCR-9988489, and ARO grants P-38682-MA, DAAD190110003 and DAAD190110019.

used propositional formulas to encode sets of system states; these formulas were in turn encoded as (ordered) binary decision diagrams. This representation afforded tremendous improvements in the space usage of model checkers. Moreover, the use of a symbolic representation (propositional formulas) for sets of system states has spurred interest in the use of other symbolic, yet automated, approaches to system verification.

This special section provides a glimpse into some of the novel approaches to symbolic system verification that are currently being studied. The topics presented in the special section are by no means a complete account of the numerous ways in which symbolic reasoning is being re-incorporated into system verification, but they represent an interesting sampling of techniques oriented around new uses of symbolic analysis.

2 The Papers

The papers in this special section were all initially presented at the 1999 Conference on Tools and Algorithms for the Construction and Analysis of Systems, which was held in Amsterdam. A summary of each paper follows.

The first paper, "Constraint-Based Deductive Model Checking," shows how model-checking problems can be cast as constraint-satisfaction problems that, in many cases, can be solved symbolically by constraint-logic programming engines even when the system being checked has an infinite number of states. These insights usefully augment the traditional symbolic approach to model checking [2] by allowing formulas in different fragments of first-order logic, and not just those in propositional logic, to be used to describe sets of system states. The authors show how model checking can be achieved using these representations for sets of system states, and they demonstrate that an existing constraint-logic programming engine can be used to solve model-checking questions efficiently. The net result of this work is, on the one hand, a uniform account of symbolic model checking that supports infinite- as well as finite-state systems, and on the other, an indication as to how this declarative account of model checking can be implemented efficiently.

The second paper, "Bisimulation Verification Using the Up-to Techniques," presents strategies and an associated tool for proving that the behavior of one system is equivalent to that of another. The notions of equivalence used are based on *bisimulation* [7], which has been studied extensively in the literature. The technique developed in this paper for proving bisimulation equivalence uses syntactic manipulations of system descriptions to reduce the proof obligations a decision procedure must discharge in order for systems to be found to be equivalent. This basis in syntactic manipulation is what makes the method symbolic.

The third paper, "A Lightweight Framework for Hardware Verification," proposes a novel framework that supports mixtures of formal and informal reasoning about the correctness of hardware designs. The paper notes that while designers often find it difficult to prove mathematically that their designs work, they are adept at giving informal justifications of correctness, which in turn may or may not be correct. To provide machine-assisted precision in support of these traditional informal arguments of correctness, the authors develop a simple theorem prover for first-order logic that they then augment with domain-specific (informally validated) decision procedures for reasoning about circuit properties such as timing. The inclusion of these decision procedures is intended to allow users to express their informal arguments of correctness systematically, at the price of possibly unsound reasoning, since the decision procedures are encoded outside the theorem-prover formalism. (Of course, reasoning conducted within the framework of the theorem prover is guaranteed to be valid.) While not guaranteeing correctness in the mathematically rigorous sense, the tool does force users to be precise about their intuitions, and this can lead to the discovery of mistakes and oversights in designs.

The fourth paper, "An Industrial Experiment on a CC-NUMA Architecture: From Formal Specification, Verification and Code Generation to Tests Generation and Execution," describes an extended case study involving the use of formal verification and validation tools to analyze the behavior of a multiprocessor computer architecture. The tools in question permit system descriptions given in the LOTOS [1] specification language to be compared for behavioral conformance, checked for different properties, compiled into C, and used for generating test cases to be applied to system implementations. One of the main novelties of this piece of work resides in the authors' demonstration that LOTOS, which has traditionally been used to describe network protocols, also provides a useful framework for describing high-level aspects of computer architectures. In addition, while the validation tools used in the case study are not "symbolic" in the traditional sense (they rely on state-space exploration), the authors exploit the structure of their LOTOS specifications to improve the performance of these routines, through the use of semantic minimization and abstraction to reduce state-space sizes.

References

1. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.
2. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
3. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *Tenth Annual ACM Symposium on Principles of Programming Languages (POPL '83)*, pages 117–126, Austin, Texas, January 1983. ACM Press.
4. E.W. Djikstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.
5. D. Gries. *The Science of Programming*. Springer-Verlag, Heidelberg, 1981.
6. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, October 1969.

7. R. Milner. *Communication and Concurrency*. Prentice-Hall, London, 1989.
8. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, Turin, April 1982. Springer-Verlag.