JMOCHA: A Model Checking Tool that Exploits Design Structure

R. Alur[†] **L. de Alfaro**[*] **R. Grosu**[‡] **T.A. Henzinger**[*] **M. Kang**[†] **C.M. Kirsch**[*]
R. Majumdar[*] **F. Mang**[*] **B.Y. Wang**[†]

[*] Department of Electrical Engineering and Computer Science, University of California, Berkeley

[†]Department of Computer and Information Science, University of Pennsylvania

[‡]Department of Computer Science, State University of New York, Stony Brook

1 Introduction

Model checking is emerging as a practical tool for automated debugging of embedded software [6, 10, 7]. In model checking, a high-level description of a system is compared against a logical correctness requirement to discover inconsistencies. Since model checking is based on exhaustive state-space exploration, and the size of the state space of a design grows exponentially with the size of the description, scalability remains a challenge. The goal of our research is to develop techniques for exploiting modular design structure during model checking, and the model checker JMOCHA is based on this theme. Instead of manipulating unstructured state-transition graphs, it supports the hierarchical modeling framework of *Reactive Modules* [3]. JMOCHA is a growing interactive software environment for speci£cation, simulation, and veri£cation, and is intended as a vehicle for the development of new veri£cation algorithms and approaches. It is written in Java and uses native C-code BDD libraries from VIS [5]. JMOCHA offers: (1) A *graphical user interface* that looks familiar to Windows/Java users. (2) A *simulator* that displays traces in a message sequence chart fashion. (3) *Requirement veri£cation* both by *symbolic* and *enumerative* model checking. (4) *Implementation veri£cation* by checking trace containment. (5) A *proof manager* that aids *compositional* and *assume-guarantee* reasoning [9]. (6) A *scripting language* called SLANG for the rapid and structured development of new veri£cation algorithms. jMocha is available publicly at http://www.eecs.berleley.edu/∼mocha. It is a successor and extension of the original Mocha tool [2], that was entirely written in C.

2 The Modeling Language

The language REACTIVE MODULES [3] is a *modeling* and *analysis* language for *heterogeneous concurrent* systems with synchronous and asynchronous components. As a modeling language it supports high-level, partial system descriptions, rapid prototyping, and simulation. As a language for concurrent systems, it allows a modular description of the interactions among the components of a system. As an analysis language it allows the speci£cation of requirements either in temporal logic [4] or as abstract modules.

3 The Graphical User Interface

As in modern Windows or Java tools, the interaction between the user and JMOCHA is controlled by a *graphical user interface*. One may use JMOCHA as a syntax-directed editor for the REACTIVE MODULES language. Once one has edited and saved a tree of £les, JMOCHA may generate a *proof context (or state)* in a separate `Project` window provided there are no syntactic errors. The context is shown in a convenient tree notation. A selected node in the tree (module or judgment) may be then simulated and veri£ed, respectively.

4 The Simulator

The behavior (executions) of a reactive system can be visualized in a *message sequence charts (MSC)* like fashion by using the *simulator*. To run the simulator, the user selects a module and the submodules/variables to be traced. For each selected variable, a vertical line shows its evolution in time. The value of a variable is displayed only when it changes. Clicking on a box, which displays a change, shows which other variables (and values) contributed to the change. The same format is used to display the counter-examples generated by the model checkers during failed veri£cation attempts. The simulator can be used either in *random-simulation* or in *manual-simulation* mode.

5 The Invariant Checkers

JMOCHA allows the speci£cation of requirements in a rich temporal logic called *alternating temporal logic (ATL)* [4]. By far the most common requirements are *invariants*, and thus it is of utmost importance to implement invariant checking ef£ciently. JMOCHA provides both £ne-tuned enumerative and symbolic state search routines for invariant checking. The enumerative, state-based algorithms are often preferable for asynchronous systems; the symbolic, decision-diagram based algorithms, for synchronous systems. More general ATL formulas can be checked by de£ning algorithms using the scripting SLANG, as shown in Section 7. These algorithms can call on both enumerative and symbolic search as subroutines.

Enumerative Invariant Checking

The enumerative checker uses the standard on-the-¤y algorithm for detecting violations of invariants starting from the initial states. We have implemented various features and optimizations in the JMOCHA enumerative search engine. For example, as in SPIN [10], each state (excluding combinational variables) is stored as bit string to save space using

compression. For asynchronous modules JMOCHA provides a search heuristic called hierarchical reduction [1] that merges several internal steps into one, and this in a hierarchical manner. For well-structured architectures such as rings and trees, this leads to signi£cant savings.

Symbolic Invariant Checking
While the enumerative checker works directly on the internal representation generated by the parser, the symbolic checker works on a *multi-valued decision diagram (MDD)* encoding provided by the VIS C-package from Berkeley [5]. The checker consists of two components: a *model generator* and an *invariant checker*. The model generator produces an MDD representation of the transition relation and of the set of initial states. The transition relation is partitioned in a conjunctive form. The invariant checker uses an image computation routine from VIS that has a very ef£cient early quanti£cation heuristic. While most of the symbolic model checker is written in Java, it calls the VIS MDD routines, written in C, to construct and manipulate MDDs ef£ciently. A main objective of the symbolic model checker is to support bit vectors and arrays ef£ciently.

6 The Re£nement Checkers

Re£nement checking gives users the possibility to verify if a module P (the implementation) *re£nes* another more abstract module P' (the speci£cation). Formally, P re£nes P' if the traces of P are contained in the set of traces of P'. Due to the high computational complexity of checking trace containment, the re£nement checkers in JMOCHA check if the speci£cation module *simulates* the implementation module assuming that (1) the speci£cation contains no private variables, and (2) all variables of the speci£cation appear in the implementation as well. In this case, simulation checking reduces to checking a transition invariant: £rst, each initial state of the implementation must be an initial state of the speci£cation, and second, each reachable transition of the implementation must satisfy the transition relation of the speci£cation [9]. This can be done ef£ciently using either enumerative or symbolic search.

Compositional and Assume-Guarantee Reasoning
JMOCHA supports *compositional rules* that allow to decompose the proof of re£nement between composite modules to the proof of re£nement between their submodules (in the most general context). For the more complex cases in that a module re£nes another module only in a particular context (e.g. the speci£cation context) JMOCHA also provides *assume/guarantee rules* [3, 9] . Given a re£nement judgment, the *proof manager (or prover)* of JMOCHA suggests all decompositions that are possible according to a built-in database of proof rules, which includes the compositional and assume-guarantee rules. Once a rule is selected, the premises are added to the proof manager as new proof goals, and they are displayed in the judgment browser. The user can then apply either further decomposition rules or discharge each proof obligation by invoking the re£nement checker.

7 The Scripting Language SLANG

SLANG is a Scripting LANGuage for the veri£cation of REACTIVE MODULES, designed with the goals of rapid prototyping of veri£cation algorithms and automation of veri£cation tasks. In addition to the usual datatypes, SLANG provides access to the datatypes speci£c to JMOCHA, including module expressions, logical expressions, MDDs, and module variables. It also provides several prede£ned functions that implement various model-checking tasks (e.g. pre, post, init). This functionality of SLANG is suf£cient to model check all ATL and μ-calculus requirements and to compute state equivalences such as bisimilarity, over £nite-state as well as in£nite-state systems (in the latter case, a SLANG script may not terminate) [8].

Acknowledgements
We thank Himyanshu Anand, Ben Horowitz, Franjo Ivancic, Michael McDougall, Marius Minea, Oliver Moeller, Shaz Qadeer, Sriram Rajamani, and Jean-Francois Raskin for their assistance in the development of JMOCHA. The MOCHA project is funded in part by the DARPA grant NAG2-1214, the NSF CAREER awards CCR95-01708 and CCR97-34115, the NSF grant CCR99-70925, the NSF ITR grant CCR0085949, the MARCO grant 98-DT-660, and the SRC contracts 99-TJ-683.003 and 99-TJ-688.

REFERENCES

[1] R. Alur, R. Grosu, and B.-Y. Wang. Automated re£nement checking for asynchronous processes. In *Proc. 3rd FMCAD*, LNCS. Springer-Verlag, 2000.

[2] R. Alur, T.A. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proc. 10th CAV*, LNCS 1427, pages 516–520, 1998.

[3] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[4] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. 38th FOCS*, pages 100–109, 1997.

[5] R. Brayton et al. VIS: A system for veri£cation and synthesis. In *Proc. 8th CAV*, LNCS 1102, pages 428–432, 1996.

[6] E.M. Clarke and O. Grumberg and D.A. Peled. Model Checking. *The MIT Press*, 1999.

[7] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting £nite-state models from Java source code. In *Proc. 22nd ICSE*, pages 439–448, 2000.

[8] T.A. Henzinger and R. Majumdar. A classi£cation of symbolic transition systems. In *Proc. 17th TACS*, LNCS 1770, pages 13–34, 2000.

[9] T.A. Henzinger, S. Qadeer, and S. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proc. 10th CAV*, LNCS 1427, pages 521–525, 1998.

[10] G.J. Holzmann. The model checker SPIN. *IEEE Trans. Software Engineering*, 23(5):279–295, 1997.