MOCHA: A Model Checking Tool that Exploits Design Structure

R. Alur[†] **L. de Alfaro**[*] **R. Grosu**[‡] **T.A. Henzinger**[*] **M. Kang**[†] **R. Majumdar**[*] **F. Mang**[*] **C.M. Kirsch**[*] **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 (see [7] for a survey, and [12, 11] for sample model checkers, and [8] for applications to software analysis). 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 MOCHA is based on this theme of exploiting modularity. Instead of manipulating unstructured state-transition graphs, it supports the hierarchical modeling framework of *Reactive Modules* [3]. The hierarchy is exploited by the tool in three ways. First, verification tasks such as refinement checking can be decomposed into subgoals using assume-guarantee rules [10]. Second, instead of traditional temporal logics such as CTL, it uses *Alternating Temporal Logic* (ATL), a game-based temporal logic that is designed to specify collaborative as well as adversarial interactions between different components [4]. Third, the MOCHA algorithms incorporate optimizations based on the hierarchical reduction of sequences of internal transitions [5].

MOCHA is a growing interactive software environment for specification, simulation, and verification, and is intended as a vehicle for the development of new verification algorithms and approaches. MOCHA is available in two versions, cMOCHA (Version 1.0.1) and JMOCHA (Version 2.0). This paper describes JMOCHA (for an introduction to cMOCHA, see [2]). Like its predecessor, JMOCHA offers the following:

- Support for *modular* specification and reasoning about *heterogeneous* systems with both synchronous and asynchronous components.
- System *execution* by randomized or manual trace generation.
- Requirement verification by model checking. MOCHA supports both *symbolic* and *enumerative* search.
- Implementation verification by checking trace containment between implementation and specification modules. For decomposing proofs, MOCHA supports an *assume-guarantee* principle.

JMOCHA is written in Java and uses native C-code BDD libraries from VIS [6]. It provides the following improvements over cMOCHA:

- A new *graphical user interface* written in Java that looks familiar to Windows/Java users.
- A new *simulator* with a graphical user interface that displays traces in a message sequence chart fashion.
- A *proof manager* for managing verification proofs such as assume-guarantee proofs.
- An new *enumerative checker* for invariant and refinement checking with optimizations such as hierarchical reduction of unobservable steps.
- A new *scripting language* called SLANG for the rapid and structured development of new verification algorithms.

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 an analysis language it allows the specification of requirements either in temporal logic or as abstract modules. Finally, as a language for concurrent systems, it facilitates a modular description of the interactions among the components of a system.

The basic structuring units, or the molecules of a system, are *reactive modules*. The modules have a well-defined interface given by a set of *external (or input)* variables and a set of *interface (or output)* variables. A module may also have a set of *private* variables. All variables are typed, and MOCHA supports a standard set of finite and infinite types, such as booleans and integers. A module is built from *atoms*, each grouping together a set of *controlled* (interface or private) variables with exclusive updating rights. *Updating* is defined by two nondeterministic guarded commands: an *initialization* command and an *update* command. In these commands unprimed variables, such as x, refer to the old value of the corresponding variable, and primed variables, such as x', refer to the new value of the corresponding variable. An atom is said to *await* another atom if its initialization or update commands refer to primed variables that are controlled by the other atom. The variables change their values over time in a sequence of *rounds*. The first round consists of the execution of the initialization command of each atom, and the subsequent rounds consist of the execution of the update command of each atom, in an order consistent with the await dependencies. A round of an atom is therefore a *subround* of the module. If no guard of the update command is enabled, then the atom idles, i.e., the values of the variables do not change. If the update command of an atom has a branch with a true guard and no updating action, then it may at any time either take a transition or idle. Such an atom is called *lazy*, and is useful for modeling asynchronous interaction.

For example, consider the specification of a village telephone system that contains four telephones. The specification consists of two modules: the first one models the environment, i.e., the users, and the second one models the system. A phone is either on-hook or off-hook, and the module `UserSpec` nondeterministically toggles at most one telephone between on-hook and off-hook.

```
type hookType is {on, off}
module UserSpec is
  interface h1,h2,h3,h4: hookType;
lazy atom ToggleHook
  controls h1,h2,h3,h4
  reads h1,h2,h3,h4
  init
    [] true -> h1’ := on; h2’ := on; ...
  update
    [] h1 = on -> h1’ := off;
    [] h1 = off -> h1’ := on;
    ...
```

Modules can be *composed* if they have disjoint sets of interface variables, and their union of atom sets does not contain a circular await dependency. Given a specification `SystemSpec` of the telephone system, specification module `Spec` is defined as:

```
module Spec is UserSpec || SystemSpec
```

For encapsulation Reactive Modules allows the *hiding* of interface variables, and for instantiation it allows the *renaming* of interface and external variables. Hiding and parallel composition permit hierarchical descriptions of complex systems.

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*. The GUI consists of five menus, three tool bars, a desktop, and a status text panel. The menus are `File`, `Edit`, `Simulate`, `Check`, and `Options`. The tool bars are associated with `File Edit`, `Simulate`, and `Check`. The menu items and the tool bar buttons are activated/deactivated in a way consistent with the state of the proof manager.

One may use jMocha as a syntax-directed editor window for the Reactive Modules language. One may open more than one file and the labels associated to their windows allow to conveniently switch from one window to another. One may edit the files by using the menu items in the `Edit` menu or the associated toolbar. One can cut and paste from one editor window into another editor window. The editor windows highlight the Reactive Modules keywords and comments. One can enable/disable *parsing on the fly* and a *pop-up window* prompting the user with the allowed next tokens.

Once one has edited and saved a tree of Reactive Modules files one may simulate and model check them in the *project mode*. In this mode the proof manager expands all import declarations that include modules from other files, and calls the parser and the type checker on the expanded code. If there are no syntactic errors, it generates a *proof context (or state)* that is displayed in a separate `Project` window that appears on the left-hand side of the desktop, as shown in Figure **??**. The project window displays the Mocha proof context in a convenient tree notation. Each node in the tree may be expanded or collapsed by clicking on it. The proof context consists of several subcontexts: `types`, `modules`, `formulas`, and `judgments`. A selected module and judgment in the project window may be simulated and verified, 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, the display parameters, and the submodules/variables to be traced. For each selected variable, a vertical line shows its evolution in time. The vertical lines are split into segments, each corresponding to a discrete time unit or equivalently, to a round of the associated module. 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 verification attempts.

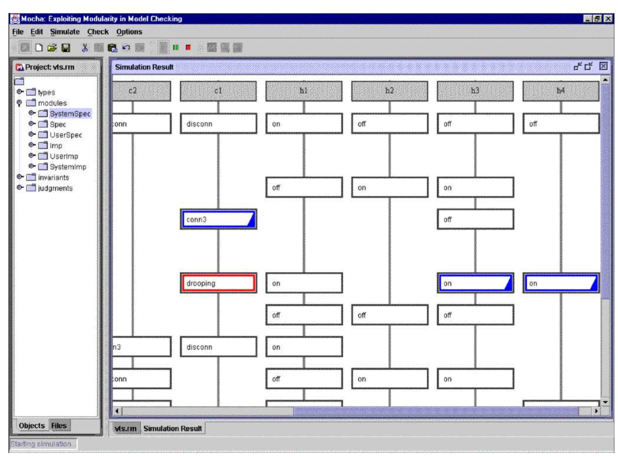

Figure 1: The simulator

The simulator can be used in *automatic* or *manual* mode. In automatic simulation, in each round, Mocha chooses one state randomly out of all the possible next states. One can stop the simulation temporarily by clicking the pause button, or permanently by clicking the stop button. In manual simulation, at each step, the user is requested to choose one state from the set of possible next states, both for the module and for its environment.

5 The Invariant Checkers

jMOCHA allows the specification 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 efficiently. Whith this in mind, jMOCHA provides both fine-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 defining 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-fly algorithm for detecting violations of invariants starting from the initial states. We have implemented various features and optimizations in the jMOCHA enumerative search engine. Each state is stored as bit string to save space using compression, as in SPIN [11]. Variables that are only awaited, but not read by any atoms (e.g., the unlatched variables of combinational circuits) are not stored. For modules that consist of lazy atoms only, jMOCHA provides a heuristic called hierarchical reduction to reduce search space [5]. The basic idea is to merge several internal steps into one, and this is applied in a hierarchical manner. For well-structured architectures such as rings and trees, this leads to significant 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 of state sets provided by the VIS C-package from Berkeley [6]. MDDs are a generalization of binary decision diagrams (BDDs) to enumerated datatypes. 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 naturally partitioned by the atoms in a conjunctive form. The invariant checker uses an image computation routine from VIS which has a very efficient early quantification 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 efficiently. A main objective of this release of the symbolic model checker was to support bit vectors and arrays efficiently.

6 The Refinement Checkers

Refinement checking gives users the possibility to verify if a module (the implementation) *refines* another module (the specification). Typically, the specification is a more abstract, nondeterministic version of the implementation. Formally, a module P refines module P', denoted by $P \preceq P'$, 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 refinement checkers in jMOCHA check if the specification module *simulates* the implementation module assuming that (1) the specification contains no private variables, and (2) all variables of the specification appear in the implementation as well. In this case, simulation checking reduces to checking a transition invariant: first, each initial state of the implementation must be an initial state of the specification, and second, each reachable transition of of the implementation must satisfy the transition relation of the specification [10]. This can be done efficiently using either enumerative or symbolic search.

For example, for the telephone system, one can write a more refined module `UserImp` modeling the users, and the intended refinement relation can be stated as

```
judgment  J1 is UserImp < UserSpec
```

There are several ways to circumvent the restrictions (1) and (2) about the specification variables. For example, one can make all private specification variables become interface variables. If a specification variable is not included in the implementation, a witness module can be constructed to assign values to the variable. The witness is in turn composed with the implementation and checked against the specification [10, 1].

Assume-Guarantee Reasoning

```
module Spec is UserSpec || SystemSpec
module Imp is UserImp || SystemImp
judgment J0 is SystemImp < SystemSpec
judgment J1 is UserImp < UserSpec
judgment J2 is Imp < Spec
```

The lines above define the specification module `Spec` and the implementation module `Imp` as the parallel composition of `UserSpec` and `SystemSpec` and respectively of `UserImp` and `SystemImp`. We wish to verify the judgment `J2`. While this can be proved directly, it can also be reduced to simpler proof obligations. Typical *compositional rules* allow this goal to be reduced to the subgoals `J0` and `J1` asserting component-wise refinements. It turns out that the implementation module `SystemImp` is not a refinement of `SystemSpec` in an unconstrained environment (so `J0` fails). However, `SystemImp` refines `System-Spec` in the more restrictive context given by the abstract module `UserSpec`. Therefore one can use the assume-guarantee rule [3, 10] which states that `J2` holds provided (1) `UserImp || SystemSpec` refines `UserSpec` and (2) `UserSpec || SystemImp` refines `SystemSpec`.

Given a refinement 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 above assume-guarantee rule. 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 refinement checker.

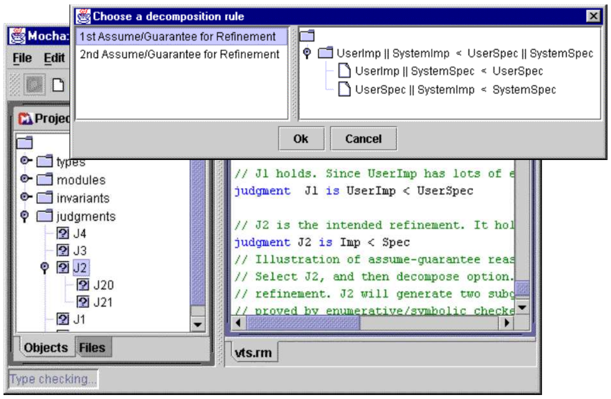

Figure 2: Proof manager and assume/guarantee reasoning

7 The Scripting Language SLANG

SLANG is a Scripting LANGuage for the verification of REACTIVE MODULES, designed with the goals of rapid prototyping of verification algorithms and automation of verification tasks. SLANG is a structured imperative language with run-time type checking. Upon request, JMOCHA provides a window for the interactive input and execution of SLANG commands. In addition to the usual datatypes, such as integers, strings, and arrays, SLANG provides access to the datatypes specific to JMOCHA, including module expressions, logical expressions (such as invariants), MDDs, and module variables. The set of predefined operators of SLANG includes the usual arithmetic, logical, and string operators. In addition, SLANG provides several predefined functions that implement various model-checking tasks. For example, if P is a module expression and ϕ is a predicate on module variables, then the function `create_mdd(`P`,`ϕ`)` returns the MDD that defines the states satisfying ϕ in the state space of P. For MDDs Φ, Φ_1, Φ_2, and for a module P, the available functions include `and(`Φ_1`,`Φ_2`)`, `or(`Φ_1`,`Φ_2`)`, `not(`Φ`)`, `equal(`Φ_1`,`Φ_2`)`, `init_reg(`P`)` (which returns the MDD representing the initial states of P), `pre(`P`,`Φ`)` and `post(`P`,`Φ`)` (which compute the MDDs representing the successor and predecessor states of the set of states represented by Φ). Other functions include functions for checking invariants and refinement relations. The usual control constructs are available in SLANG, such as if-then-else and while loops.

As an example of the capabilities of SLANG, the following function `backforth_invcheck (M, phi)` checks whether the module `M` implements the invariant `phi`, by using a mix of forward reachability from the initial condition and backward reachability from the complement of the invariant. The functions provided by SLANG are sufficient to model check all ATL and μ-calculus requirements and to compute state equivalences such as bisimilarity, over finite-state as well as infinite-state systems (in the latter case, a SLANG script may not terminate) [9].

```
def backforth_invcheck (M, phi) {
  R_back  := zeroMdd;  R_forw  := zeroMdd;
  NR_back := not(phi); NR_forw := init_reg(M);
  while ( !equal (R_back, NR_back)
          && !equal (R_forw, NR_forw)
          && empty (and (NR_forw, NR_back))) {
    R_forw  := NR_forw;
    NR_forw := or (NR_forw, post (M, NR_forw));
    R_back  := NR_back;
    NR_back := or (NR_back, pre  (M, NR_back));
  }
  return (empty (and (NR_forw, NR_back))); }
```


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.

REFERENCES

[1] R. Alur, R. Grosu, and B.-Y. Wang. Automated refinement 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. Alur and B.-Y. Wang. "Next" heuristic for on-the-fly model checking. In *Proc. 10th CONCUR*, LNCS 1664, pages 98–113, 1999.

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

[7] E.M. Clarke and R.P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

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

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

[10] 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.

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

[12] K. McMillan. *Symbolic model checking: An approach to the state explosion problem.* Kluwer Academic Publishers, 1993.