MOCHA: Modularity in Model Checking*

R. Alur[1], T.A. Henzinger[2], F.Y.C. Mang[2], S. Qadeer[2], S.K. Rajamani[2], and S. Tasiran[2]

[1] Computer & Information Science Department, University of Pennsylvania, Philadelphia, PA 19104.
Computing Science Research Center, Bell Laboratories, Murray Hill, NJ 07974.
alur@cis.upenn.edu

[2] Electrical Engineering & Computer Sciences Department, University of California, Berkeley, CA 94720.
{tah,fmang,shaz,sriramr,serdar}@eecs.berkeley.edu

1 Introduction

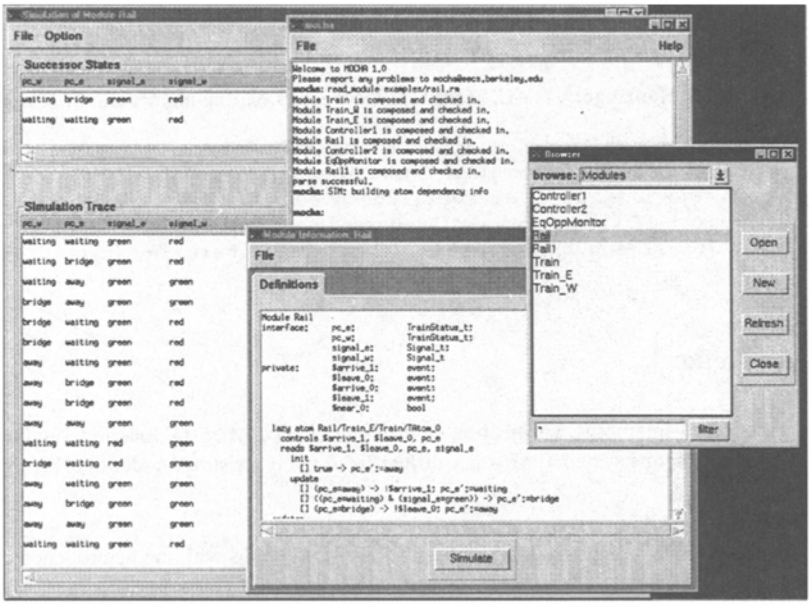
We describe a new interactive verification environment called MOCHA for the modular verification of heterogeneous systems. MOCHA differs from many existing model checkers in three significant ways:

- For modeling, we replace unstructured state-transition graphs with the heterogeneous modeling framework of *reactive modules* [AH96]. The definition of reactive modules is inspired by formalisms such as Unity [CM88], I/O automata [Lyn96], and Esterel [BG88], and allows complex forms of interaction between components within a single transition. Reactive modules provide a semantic glue that allows the formal embedding and interaction of components with different characteristics. Some modules may be synchronous, others asynchronous, some may represent hardware, others software, some may be speed-independent, others time-critical.
- For requirement specification, we replace the system-level specification languages of linear and branching temporal logics [Pnu77,CE81] with the module-level specification language of *Alternating Temporal Logic* (ATL) [AHK97]. In ATL, both cooperative and adversarial relationships between modules can be expressed. For example, it is possible to specify that a module can attain a goal regardless of how the environment of the module behaves.
- For the verification of complex systems, MOCHA supports a range of *compositional and hierarchical verification methodologies*. For this purpose, reactive modules provide assume-guarantee rules [HQR98] and abstraction operators [AHR98]; MOCHA provides algorithms for automatic refinement checking, and will provide a proof editor that manages the decomposition of verification tasks into subtasks.

In this paper, we describe the toolkit MOCHA in which the proposed approach is being implemented. The input language of MOCHA is a machine readable variant of reactive modules. The following functionalities are currently being supported:

- Simulation, including games between the user and the simulator
- Enumerative and symbolic invariant checking and error-trace generation
- Compositional refinement checking
- ATL model checking
- Reachability analysis of real-time systems

* This research was supported in part by the ONR YIP award N00014-95-1-0520, by the NSF CAREER award CCR-9501708, by the NSF grant CCR-9504469, by the ARO MURI grant DAAH-04-96-1-0341, and by the SRC contract 97-DC-324.041.



Successor States

pc_w	pc_e	signal_e	signal_w
waiting	bridge	green	red
waiting	waiting	green	red

Simulation Trace

pc_w	pc_e	signal_e	signal_w
waiting	waiting	green	red
waiting	bridge	green	red
waiting	away	green	green
bridge	away	green	green
bridge	waiting	green	red
bridge	waiting	green	red
away	waiting	green	red
away	bridge	green	red
away	bridge	green	red
away	away	green	green
waiting	waiting	red	green
bridge	waiting	red	green
away	waiting	green	green
away	bridge	green	green
away	away	green	green
waiting	waiting	green	red

```
Welcome to MOCHA 1.0
Please report any problems to mocha@eecs.berkeley.edu
mocha: read_module examples/rail.rm
Module Train is composed and checked in.
Module Train_W is composed and checked in.
Module Train_E is composed and checked in.
Module Controller1 is composed and checked in.
Module Rail is composed and checked in.
Module Controller2 is composed and checked in.
Module EqOppMonitor is composed and checked in.
Module Rail1 is composed and checked in.
parse successful.
mocha: SIM: building atom dependency info

mocha:
```

browse: Modules

```
Controller1
Controller2
EqOppMonitor
Rail
Rail1
Train
Train_E
Train_W
```

```
Module Rail
interface:      pc_e:           TrainStatus_t:
                pc_w:           TrainStatus_t:
                signal_e:       Signal_t:
                signal_w:       Signal_t
private:        $arrive_1:      event:
                $leave_0:       event:
                $arrive_0:      event:
                $leave_1:       event:
                $near_0:        bool

  lazy atom Rail/Train_E/Train/TAtom_0
    controls $arrive_1, $leave_0, pc_e
    reads $arrive_1, $leave_0, pc_e, signal_e
      init
        [] true -> pc_e':=away
      update
        [] (pc_e=away) -> !$arrive_1; pc_e':=waiting
        [] ((pc_e=waiting) & (signal_e=green)) -> pc_e':=bridge
        [] (pc_e=bridge) -> !$leave_0; pc_e':=away
```

MOCHA is intended as a vehicle for the development of new verification algorithms and approaches. It adopts a software architecture similar to VIS [BHS+96], a symbolic model-checking tool from UC Berkeley. Written in C with Tcl/Tk and Tix [Exp97], MOCHA can be easily extended in two ways: designers and application developers can customize their application or design their own graphical user interface by writing Tcl scripts; algorithm developers and researchers can develop new verification algorithms by writing C code, or assembling any verification packages through C interfaces. For instance, MOCHA incorporates the VIS packages for image computation and multi-valued function manipulation, as well as various BDD packages, to provide state-of-the-art verification techniques.

2 Reactive Modules

A formal definition of reactive modules can be found in [AH96]; here we give only a brief introduction. Unlike simple state-transition graphs, reactive modules is a compositional model in which both states and transitions are structured. The state of a reactive module is determined by the values of three kinds of typed variables: the *external variables* are updated by the environment and can be read by the module; the *interface variables* are updated by the module and can be read by the environment; the *private variables* are updated by the module and cannot be read by the environment. The *observable* variables of a module are its external and interface variables.

The state of a reactive module changes in a sequence of rounds. In the first round (the initialization round), the initial values of the interface and private variables are determined. In each subsequent round (an update round), new values of the interface and private variables are determined, possibly dependent on the old values of some variables from the previous round, and possibly dependent on the new values of some variables from the current round. The external variables are initialized and updated nondeterministically.

Value dependencies between variables within an update round are resolved statically. In each update round, some interface and private variables are updated simultaneously, and some sequentially. Variables that are updated simultaneously are grouped together and controlled by an *atom*.

During the execution of an atom (called subround), its variables are updated simultaneously. The new values of the atom variables may depend on the new values of variables that have been updated, by another atom, in an earlier subround. Hence, some atoms can be executed only after the execution of some other atoms. The initialization and update rules for executing an atom are specified via guarded commands. We require that there is a static order of the atoms in which they can be executed in every update round, and that always at least one guarded command is enabled. This ensures that the interaction of an atom with the other atoms (and the environment) is *nonblocking*.

New modules can be built from existing modules using three operations: parallel composition, variable renaming, and variable hiding. The composition of two modules produces a single module whose behavior captures the interaction between the two component modules. Variable renaming changes the name of a variable. Variable hiding changes a variable from interface to private, and therefore renders it unobservable.

3 Simulation

MOCHA provides an interactive simulator with a graphical user-interface for simulating modules. It operates in three different modes: random simulation, manual simulation, and game simulation. In random simulation, all atoms are executed by the simulator, which randomly resolves nondeterminism. In manual simulation, all atoms are executed according to the directions of the user. In game simulation, some of the atoms are executed by the simulator, while the remaining atoms are executed by the user. Each such simulation can be viewed as a game between the user and the simulator, hence the name game simulation.

4 Invariant Checking

MOCHA provides support for checking both state and transition invariants on finite-state modules. For this purpose, we have implemented both symbolic and enumerative state-exploration algorithms:

Symbolic. We represent the transition relation and the set of reached states of a reactive module as binary decision diagrams (BDDs) [Bry86]. We keep the transition relation of a reactive module in a conjunctively partitioned form. Each conjunct is the transition relation of an atom. The image computation routines have been leveraged off VIS, which provides a heuristic [RAP+95] for image computation based on early quantification that has been shown useful in practice.

Enumerative. The current implementation of the enumerative state-exploration routines is rather naive and does not perform any optimizations. It is used primarily by the simulator.

Both the symbolic and enumerative invariant checkers have the capability to produce error traces. The error traces can be displayed graphically with a Tk widget.

5 Compositional Refinement Checking

We briefly describe what it means for one module to refine another. A *trajectory* of a module P is a finite sequence of states obtained by executing P for finitely many rounds. A *trace* of P is obtained by projecting each state of a trajectory of P onto the observable variables. The module P *refines* another module Q, denoted $P \preceq Q$, if every trace of P is also a trace of Q (in addition to some technical side conditions). We have implemented a compositional methodology for refinement checking. The details of the method are explained in an accompanying paper [HQR98].

To illustrate the main aspects of our methodology that deal with the explosion of the implementation state space, consider the refinement check $P_1 \| P_2 \preceq Q$, where $\|$ denotes the parallel composition operation. Suppose that the state space of $P_1 \| P_2$ is too large to be handled by exhaustive state exploration. Typically, Q specifies the behavior of only those variables that are visible at the boundary of $P_1 \| P_2$. Therefore, to obtain suitable constraining environments for P_1 and P_2, we need to construct *abstraction modules* A_1 and A_2 that specify the behavior of the boundary variables and the interface variables between P_1 and P_2. We can then decompose the proof into lemmas using the following assume-guarantee rule:

$$\frac{\begin{array}{c} P_1 \| A_2 \preceq A_1 \| Q \\ A_1 \| P_2 \preceq Q \| A_2 \end{array}}{P_1 \| P_2 \preceq A_1 \| A_2 \| Q \preceq Q}$$

Even if the implementation state space becomes manageable as a result of decomposition, each lemma of the form $P' \preceq Q'$ is still PSPACE-hard in the description of P' and EXPSPACE-hard in the description of Q'. However, for the special case that Q' is *projection refinable* by P' (i.e., all variables of Q' are observable by both P' and Q'), the refinement check reduces to a transition-invariant check on P'—namely, checking whether every move of P' can be mimicked by a move of Q'. The complexity of this procedure is linear in the state spaces of P' and Q'. When Q' is not projection refinable by P', our methodology advocates the use of a *witness module* that, when composed with P', leads to projection refinability.

An assume-guarantee rule very similar to the one described above has been proved sound also for *fair* refinement checking [AH96]. Hence, our methodology applies to fair modules as well.

6 ATL Model Checking

Alternating Temporal Logic (ATL) is a temporal logic designed for specifying requirements of open systems [AHK97]. Consider a set of agents that correspond to different components of a system and its environment. Then, the logic ATL admits formulas of the form $\langle\langle A \rangle\rangle \Diamond p$, where p is a state predicate and A is a subset of the agents. The formula $\langle\langle A \rangle\rangle \Diamond p$ asserts that the agents in A can cooperate to reach a p-state no matter how the remaining agents behave. The semantics of ATL is formalized by defining games such that the satisfaction of an ATL formula corresponds to the existence of a winning strategy.

The model checking problem for ATL is to determine whether a given module satisfies a given ATL formula. The symbolic model-checking procedure for CTL [BCM92] generalizes nicely to yield a symbolic model-checking procedure for ATL. For a set A of agents and a set U of states, let $Pre_A(U)$ be the set of states from which the agents in A can force the system into some state in U in one move. Then, the set of states satisfying the ATL formula $\langle\langle A \rangle\rangle \Diamond p$ is the least set that contains all states satisfying p and is closed under the operator Pre_U. This set can be easily computed by an iterative symbolic procedure. The time complexity of ATL model checking is, like CTL model checking, linear in the size of both the state space and the formula. Thus, the added expressiveness of ATL over CTL comes at no extra cost.

We plan to integrate the game simulator described in Section 3 with the ATL model checker to provide counter-examples. When an ATL specification fails, the ATL model checker synthesizes and outputs a winning strategy as a counter-example, according to which the simulator will play a game with the user. The user tries to win the game by finding an execution sequence that satisfies the specification. We believe that by being forced into playing a losing game, the user can be convinced that the model is incorrect and can be led to the error.

7 Real-Time Modules

MOCHA supports the reachability analysis of real-time systems that are described in the form of *timed modules* as defined in [AH97]. In addition to the discrete-valued variables of reactive modules, a timed module makes use of real-valued *clock variables*. All clock variables increase at the same rate, and keep track of the time elapsed since they have been assigned a value by a guarded command. The guards of later transitions can depend on the values of clocks. The reachability analysis of timed modules is performed by automatically synthesizing a monitor process that restricts the state exploration to only those trajectories that satisfy the timing constraints on the clock variables, as in the analysis of timed automata [AD94].

References

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, vol. 126, pages 183–235, 1994.

[AH96] R. Alur and T.A. Henzinger. Reactive modules. In *Proc. 11th IEEE Symposium on Logic in Computer Science*, pages 207–218, 1996.

[AH97] R. Alur and T.A. Henzinger. Modularity for timed and hybrid systems. In *Proc. 8th International Conference on Concurrency Theory*, LNCS 1243, pages 74–88. Springer-Verlag, 1997.

[AHK97] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. 38th IEEE Symposium on Foundations of Computer Science*, pages 100–109, 1997.

[AHR98] R. Alur, T.A. Henzinger, and S.K. Rajamani. Symbolic exploration of transition hierarchies. In *TACAS 98: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 1384, pages 330–344, 1998.

[BCM92] J.R. Burch and E.M. Clarke and K.L. McMillan and D.L. Dill and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, Vol 98, No 2, pages 142–170, 1992.

[BG88] G. Berry and G. Gonthier. The synchronous programming language ESTEREL: design, semantics, implementation. Technical Report 842, INRIA, 1988.

[BHS$^+$96] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. In *Proc. 8th International Conference on Computer Aided Verification*, LNCS 1102, pages 428–432. Springer-Verlag, 1996.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean-function manipulation. *IEEE Trans. on Computers*, C-35(8), 1986.

[CM88] K.M. Chandy and J. Misra. *Parallel program design: A foundation.* Addison-Wesley, 1988

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.

[Exp97] Expert Interface Technologies. *Tix Home Page.* `http://www.xpi.com/tix/index.html`.

[HQR98] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proc. 10th International Conference on Computer Aided Verification*. Springer-Verlag, 1998.

[Lyn96] N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.

[RAP$^+$95] R.K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R.K. Brayton. Efficient formal design verification: data structures + algorithms. In *Proc. International Workshop on Logic Synthesis*, 1995.