MOCHA: Modularity in Model Checking

R. Alur∗ T. Henzinger† F. Mang† S. Qadeer† S. Rajamani† S. Tasiran†

Abstract. We describe a new interactive verification environment called MOCHA for modular verification of heterogeneous systems. MOCHA differs from existing model checkers in three important ways. First, instead of manipulating unstructured state-transition graphs, it supports the heterogeneous modeling framework of Reactive Modules. Second, instead of traditional temporal logics such as CTL, it uses Alternating Temporal Logic (ATL), a temporal logic that is designed to specify collaborative as well as adversarial interactions between different components. Third, to support hierarchical design and verification, it combines model checking with automated refinement checking. The specific implementation features reported in this paper include game simulation, ATL model checking, compositional refinement checking, and real-time verification. We illustrate the features via a simple railroad controller and a simple public-key encryption protocol.

∗Computer & Information Science, University of Pennsylvania, Philadelphia, PA 19104, and Computing Science Research Center, Bell Laboratories, Murray Hill, NJ 07974. Email: alur@cis.upenn.edu
†Electrical Engineering & Computer Sciences, University of California, Berkeley, CA 94720-1770. Email: \{tah, fmang, shaz, sriramr, serdar\}@eecs.berkeley.edu
1 Introduction

Model checking is a fully automatic technology for finding design errors by exhaustive state-space exploration, and has led in recent years to startling breakthroughs in hardware verification (cf. [CK96]). This is in part because hardware, usually being homogeneous, static, regular, and finite-state by nature, lends itself to the following steps of the traditional model-checking methodology:

1. Flatten a given, complete design into a huge but finite state-transition graph.
2. Specify global graph requirements as formulas of a linear or branching temporal logic.
3. Reduce the graph by detecting redundancies that are irrelevant to the requirement specification, for example, by abstracting data paths, or folding symmetries, or partially ordering independent events.
4. Explore the abstract graph using symbolic representations for boolean states such as BDDs.

We call this four-step methodology, supported by tools such as SMV [McM93], COSPAN [HHK96], MURPH [DDH+92], and VIS [BHS+96], closed model checking, because it views a complete design in isolation, without reference to ongoing design modifications and without interference from other designs.

In comparison, the successful applications of model checking in software and hardware/software codesign have been few and apart (cf. [CW96]). The blame is often assigned to the state-explosion problem: since model checking is based on exhaustive state-space exploration, the size of the state space of a design is the main limiting factor of the technology. However, it is not strictly the size but the structure of a state space that determines model-checking success or failure. The closed approach to model checking explicitly destroys all structure in Step 1, and then rediscovers some of it in Step 3. A significant advance can be made if we replace the closed, flat, approach by a modular approach that exploits rather than destroys design structure. As a possible solution, we propose the following methodology for open model checking:

1. 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 I/O automata [Lyn96], 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.

2. For requirement specification, we replace the system-level specification languages of linear and branching temporal logics [Pnu77, CE81] with Alternating Temporal Logic (ATL) [AHK97]. By taking into account the possible relationships between modules, both cooperative and adversarial module requirements can be specified in ATL. For example, in ATL it is possible to specify that a module can attain a goal regardless of how the environment of the module changes.

3. To support hierarchical design at different levels of abstraction and to aid decomposition of verification tasks, we complement model checking with automated refinement checking for Reactive Modules. The refinement checking problem can be simplified using the compositional and assume-guarantee rules.
The general principle of using modularity and compositionality in verification has a rich history of research. Sample efforts include compositional proof methodologies [AI.93, MP.95], compositional CTL model checking [GL.94], homomorphism checking [Kur.94], property-preserving abstractions [CG.92], minimization [CPS.93], module checking [KV.96], and symbolic refinement checking [McM.97]. However, exploiting modularity in practice still remains a challenge. The novelty of our approach lies primarily in the choice of modeling language and the choice of temporal logic, both of which, in our opinion, are better suited to exploit modularity.

In this paper, we describe the toolkit MOCHA in which the proposed approach is being implemented. MOCHA is an interactive verification environment that reads in Reactive Modules descriptions as input. MOCHA is intended as a vehicle for development of new verification algorithms and approaches. It is designed to allow easy extensions. It follows a software architecture similar to VIS [BHS+96]. Written in C with TcI/Tk and Tix [Exp.97], MOCHA provides two levels of development: 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 algorithms by writing C codes, or assemble any verification packages such as those provided by VIS through the C interfaces.

We report on the following functionalities that are currently being supported in MOCHA:

- Simulation including games between the user and the simulator
- Enumerative and symbolic invariant checking and error-trace generation
- ATL (Alternating-time invariant logic) model-checking
- Compositional refinement checking
- Real-time system verification

We report on two examples. The first example, chosen primarily to illustrate the framework and functionalities, is a simple railroad controller. The second example concerns Needham-Schroeder protocol for public-key encryption, and illustrates a novel use of ATL in design and analysis.

## 2 Reactive Modules

A formal definition of reactive modules can be found in [AH.96]; here we give only a brief introduction. 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 (the update round), new values of interface and private variables are determined, possibly dependent on the old values of some of the variables from the previous round, and possibly dependent on the new values of some variables from the current round. New values are represented by their primed variables.
Within each round, some variables are initialized and updated simultaneously, and some sequentially. Variables that are initialized and updated simultaneously are grouped together and controlled by an atom. During the execution of an atom (called subround), its variables are initialized or updated simultaneously, as defined by the init and the update command respectively. The values of variables that are initialized and updated in a later subround may depend on the new values of the variables that are initialized and updated in an earlier round. Hence, some atoms can only be executed after the execution of some other atoms. This results in a partial ordering of the atoms.

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 in unobservable.

As an example, consider a railway system with two circular railroad tracks, one for the train traveling clockwise, and the other for the train traveling counter-clockwise. At one point of the circle, there is a bridge that is not wide enough to accommodate both tracks. The two tracks merge on the bridge, and for controlling the access to the bridge, there is a signal at either entrance. If the signal at the western entrance is green, then a train coming from the west may enter the bridge; if the signal is red, the train must wait. The signal at the eastern entrance to the bridge controls trains coming from the east in the same fashion.

Figure 1 shows the Reactive Modules description for the a generic train Train. The module has three interface variables: pc of enumerative type ({away, wait, bridge}) and arrive, leave of type event (E); and one external variable signal. An event variable $x$ is a boolean variable with restricted operations: $x!$ stands for the assignment $x' := \neg x$ (emission of the event) and $x?$ stands for the condition $x' \neq x$ (checking for presence). The module has only one atom which controls all the interface variables. The keyword lazy indicates that the atom may choose not to update the variables, in which case the variables retain their old values. This is useful to model the assumption regarding independence of the speeds of different modules. The keyword reads indicates the old values of the read variables ($pc$, arrive, leave, signal) are used for updating the controlled variables.

The module does the following: when the train approaches the bridge, it sends the event arrive to the railroad controller and checks the signal at the entrance to the bridge ($pc = wait$). When the signal is red, the train stops and keeps checking the signal. When the signal is green, the train proceeds onto the bridge ($pc = bridge$). When the train exits from the bridge, it sends the event leave to the controller and travels around the circular track ($pc = away$). Multiple copies of the Train are created by variable renaming. $Train_W$, which represents the train traveling clockwise, is constructed by renaming variables $pc$ to $pc_W$, arrive to $arrive_W$, signal to $signal_W$ and leave to $leave_W$. $Train_E$, which represents the train traveling counter-clockwise, is constructed in a similar fashion.

Figure 1 also shows a controller controlling the signals to prevent collisions of the two trains. Here, $\mathbb{B}$ represents the boolean type; the keyword awaits indicates that the controlled variables of the atom can be updated (initialized) only after the awaited variables have been updated (initialized). The complete railway system is represented by the module Rail, which is the composition of the trains with the controller, with variables $arrive_W$, $arrive_E$, $leave_W$ and $leave_E$ hidden.
module Train
  interface pc: {away, wait, bridge} ; arrive, leave : E
  external signal: {green, red}
  lazy atom controls pc, arrive, leave reads pc, arrive, leave, signal
  init
    [] true → pc' := away
  update
    [] pc = away → arrive!; pc' := wait
    [] pc = wait ∧ signal = green → pc' := bridge
    [] pc = bridge → leave!; pc' := away
endatom
endmodule

Train_w = Train[pc, arrive, signal, leave := pc_w, arrive_w, signal_w, leave_w]
Train_e = Train[pc, arrive, signal, leave := pc_e, arrive_e, signal_e, leave_e]

module Controller
  private near_w, near_e: E
  interface signal_w, signal_e: {green, red}
  external arrive_w, arrive_e, leave_w, leave_e: E
  atom controls near_w reads near_w, arrive_w, leave_w awaits arrive_w, leave_w
  init
    [] true → near_w' := false
  update
    [] arrive_w? → near_w' := true
    [] leave_w? → near_w' := false
endatom
atom controls near_e reads near_e, arrive_e, leave_e awaits arrive_e, leave_e
  init
    [] true → near_e' := false
  update
    [] arrive_e? → near_e' := true
    [] leave_e? → near_e' := false
endatom
lazy atom controls signal_w, signal_e reads near_w, near_e, signal_w, signal_e
  init
    [] true → signal_w' := red; signal_e' := red
  update
    [] near_w ∧ signal_e = red → signal_w' := green
    [] near_e ∧ signal_w = red → signal_e' := green
    [] ¬near_w → signal_w' := red
    [] ¬near_e → signal_e' := red
endatom
endmodule

Rail = hide arrive_w, arrive_e, leave_w, leave_e in Train_w || Train_e || Controller

Figure 1: Railroad controller
3 Simulation

MOCHA provides a simulator with an interactive graphical user interface to simulate modules. It operates in three different modes: random simulation, manual simulation, and game simulation. In random simulation, all the atoms are executed by the simulator, which randomly chooses a possible update sequence of the atoms. In manual simulation, all the atoms are executed according to the choice of the user. In game simulation, some of the atoms are executed by the simulator, while the remaining are executed by the user. Each such simulation can be conceived as a game between the user and the simulator, hence the name game simulation.

To start the simulation, the user specifies the module to be simulated, as well as the atoms that are played by the user. The remaining atoms will be played by the simulator. Each state is updated by the user and the simulator with each player choosing a possible execution of the atoms it controls. The simulator resolves the nondeterminism in the actions of its atoms randomly. For example, the user can control the execution of the Controller, whereas the simulator will control the two trains Train\textsubscript{E} and Train\textsubscript{W} to simulate the rail system in Figure 1.

In random simulation and manual simulation, the order of execution of the atoms is immaterial; all linearizations of the underlying partial ordering of the atoms are equivalent. In game simulation, however, the order is more important. For example, when choosing the execution of the user atoms, the user is not able to make a decision based on the execution of the simulator atoms which are executed later. In our implementation, we always pick the “worst” linearization: atoms that are not executed by the user come as late as possible in the linearization so that the user has the least information about the choices of the simulator.

4 Invariant and Refinement Check

4.1 Invariant Checking

MOCHA provides support for checking two types of invariants on finite state reactive modules by performing reachability analysis. Let \( P \) be a reactive module and let \( T_P(s, s') \) be its transition relation. Let \( \text{reach}_P(s) \) be the set of reachable states of \( P \).

1. State invariant defines a subset of the set of states of the module and is specified as a boolean predicate \( \sigma(s) \) on the set of states in terms of only unprimed variables. It is checked by computing \( \text{reach}_P \) and verifying that \( \text{reach}_P \) implies \( \sigma \).

2. Transition invariant defines a subset of the set of transitions of the module and is specified as a boolean predicate \( \tau(s, s') \) in terms of both primed and unprimed variables. It is checked by computing \( \text{reach}_P(s) \) and verifying that the conjunction \( \text{reach}_P \land T_P(s, s') \) implies \( \tau(s, s') \).

We have implemented both symbolic and enumerative reachability analysis.

1. 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 \textit{conjectively partitioned} form. Each partition is the transition relation of an atom. The image computation routines have been leveraged off VIS [BHS+96], a symbolic model checking tool from UC Berkeley. VIS provides a heuristic [RAP+95] for image computation based on \textit{early quantification} that has been shown to be quite efficient in practice.
2. **Enumerative.** The current implementation of the enumerative reachability analysis is rather naive and does not perform any optimization. Currently, it is used by the simulation engine only and is not suitable for serious verification tasks.

During the reachability analysis for checking invariants, *history-free* variables, i.e., variables that are not read by any atom, and event variables are not stored as part of the explored state space. It can be shown that these optimizations do not affect the soundness of the invariant check.

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.

### 4.2 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 observable variables. Given two reactive modules $P$ and $Q$, $P$ is a refinement of $Q$, denoted $P \preceq Q$, if (1) every interface variable of $Q$ is an interface variable of $P$, (2) every external variable of $Q$ is an observable variable of $P$, and (3) the trace language of $P$ is contained in the trace language of $Q$. Using symbolic reachability analysis, we have implemented a compositional refinement check for reactive modules in MOCHA. The details of our approach are explained in an accompanying paper [HQR98].

To illustrate the main aspects of our methodology that deal with explosion of the implementation state space, consider the refinement check $P_1 \parallel P_2 \preceq Q$, where $\parallel$ denotes parallel composition operation and $\preceq$ denotes the refinement relation on modules. The state space of $P_1 \parallel P_2$ is too large to be handled by exhaustive state search algorithms. A naive compositional approach would try to prove (1) $P_1 \preceq Q$ and (2) $P_2 \preceq Q$, and conclude $P_1 \parallel P_2 \preceq Q$. Though this rule is sound, it is not useful in practice — $P_1$ usually behaves like $Q$ only in a suitable constraining environment, and so does $P_2$. Typically, $Q$ specifies the behavior of only those variables that are visible at the boundary of $P_1 \parallel P_2$. Therefore, to get abstract constraining environments for $P_1$ and $P_2$, we need to construct *abstraction modules* $A_1$ and $A_2$, that specify the behavior of not only the boundary variables but also the interface variables between $P_1$ and $P_2$. Now, constraining environments for $P_1$ and $P_2$ are provided by $A_2$ and $A_1$ respectively. We can then decompose the proof using the following assume-guarantee rule:

\[
\begin{align*}
P_1 \parallel A_2 & \preceq A_1 \\
A_1 \parallel P_2 & \preceq A_2 \\
A_1 \parallel A_2 & \preceq Q
\end{align*}
\]

\[
\begin{align*}
P_1 \parallel P_2 & \preceq A_1 \parallel A_2 \preceq Q
\end{align*}
\]

Even if the implementation state space becomes manageable as a result of decomposition, the refinement check $P \preceq Q$ is PSPACE-hard in the description of $P$ and EXPSPACE-hard in the description of $Q$. For the special case that all variables of $Q$ are also present in $P$ (we say that $P$ is *projection comparable* with $Q$ in such cases), the refinement check reduces to a transition invariant check on $P$ — checking if every move of $P$ can be mimicked by $Q$. The complexity of this procedure is linear in the state spaces of $P$ and $Q$. Suppose $P$ is not projection comparable with $Q$. Our methodology advocates the use of a *witness module* that can be composed with $P$ to make it projection comparable with $Q$. The construction of witness modules also requires manual effort.
An implementation of a similar assume-guarantee rule for hardware designs was described in [McM97]. That work did not deal with fairness. On the other hand, an assume-guarantee rule very similar to the one described above is sound for fair refinement check [AH96]. Hence, our methodology allows us to deal with fair modules also.

5 ATL Model-Checking

Alternating-time Temporal Logic

Alternating Temporal Logic (ATL) is a temporal logic designed to write requirements of open systems [AHK97]. An open system is a system that interacts with its environment and whose behavior depends on the state of the system as well as the behavior of the environment. Models for open systems (e.g., Reactive Modules) distinguish between internal nondeterminism, choices made by the system, and external nondeterminism, choices made by the environment. Consequently, besides universal (do all computations satisfy a property?) and existential (does some computation satisfy a property?) questions, a third question arises naturally: can the system resolve its internal choices so that the satisfaction of a property is guaranteed no matter how the environment resolves the external choices? Such an alternating satisfaction can be viewed as a winning condition in a two-player game between the system and the environment.

More generally, let $\Sigma$ be a set of agents corresponding to different components of the system, one of which may correspond to the external environment. Then, the logic ATL admits formulas of the form $\langle A \rangle \diamond p$, where $p$ is a state predicate and $A$ is a subset of agents. The formula $\langle A \rangle \diamond p$ means that the agents in the set $A$ can cooperate to reach a $p$-state no matter how the remaining agents resolve their choices. This is formalized by defining games, and satisfaction of ATL formulas corresponds to existence of winning strategies in such games. The alternating path quantifier $\langle A \rangle$ is a generalization of the path quantifiers of branching-time logics: the existential path quantifier corresponds to $\langle \Sigma \rangle$, and the universal quantifier corresponds to $\langle \emptyset \rangle$.

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 [McM93] 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 step. Then, the set of states satisfying the ATL formula $\langle A \rangle \diamond p$ is the least set that (i) contains all states satisfying $p$ and (ii) is a fixpoint of the operator $Pre_U$. This set can easily be computed by an iterative symbolic procedure. Thus, the added expressiveness of ATL over CTL comes at no extra cost.

Example

Consider again the railway system described in section 2. Using the ATL model-checker, we are able to prove the following properties:

1. The most important property is the safety requirement which states that the two trains should not be allowed to move on to the bridge at the same time. More specifically, the invariant

$$safe : \forall a \neg (pc_E = bridge \land pc_W = bridge)$$

should hold for all the reachable states of the system.
2. A train cannot enter the bridge without the help of the Controller. In ATL, this property is stated as

$$\neg\langle\langle\text{Train}_E\rangle\rangle(\text{pc}_E = \text{bridge}).$$

However, with the help of the Controller, it can move on to the bridge:

$$\langle\langle\text{Train}_E, \text{Controller}\rangle\rangle(\text{pc}_E = \text{bridge}).$$

3. It is obvious that $\text{Train}_E$ can choose to move on to the bridge infinitely often, provided $\text{Train}_W$ does not stay on the bridge forever. Hence, the following assertion is true:

$$\langle\langle\langle\text{Train}_E, \text{Controller}, \text{Train}_W\rangle\rangle(\text{pc}_E = \text{bridge}).$$

Note this is equivalent to $\forall \square \exists(\text{pc}_E = \text{bridge})$ in CTL. However, this can only be true with the cooperation of $\text{Train}_W$. In particular, the following property fails:

$$\langle\langle\langle\text{Train}_E, \text{Controller}\rangle\rangle(\text{pc}_E = \text{bridge}).$$

Implementation

We have implemented a symbolic ATL model-checker in MoCHA . The model-checking algorithm (as presented in [AHK97]) is very similar to that of CTL model-checking, except in the pre-image computation. In Reactive Modules, each agent corresponds to an atom. For each external variable, there is an extra agent which controls it. The exact nesting of the existential and universal quantifiers depends on the linearization of the atoms. In general, the variables controlled by the agents specified in the path quantifier will be quantified out existentially, while the rest universally. Since in general universal and existential quantifiers do not commute, and the linearization of the atoms is not unique, we always pick the worst linearization: we assume that all the agents that are not specified in the path quantifier will come as late as possible in the linearization. The details will be presented in the full paper.

Counter-examples and Witnesses

Visualization of counter-examples/witnesses is a major feature in a verification environment like MoCHA . Our goal is to integrate the game simulator described in section 3 with the ATL model-checker to provide counter-examples/witnesses: the ATL model-checker synthesizes and outputs a winning strategy as counter-example/witness, 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 playing a losing game, the user can be convinced that their model is incorrect and subsequently discover the bug in their model (in case of counter-example).

ATL and Refinement: the Needham-Schröeder protocol

In this section, we put forward an informal relationship between ATL and refinement, and give an example to illustrate a possible use of ATL in authentication protocol design.

Let $M$ be a model. Let $\Sigma_0$ and $\Sigma_1$ be sets of agents of $M$, and let $\varphi$ be a path formula. Clearly $\langle\langle\Sigma_0\rangle\rangle\varphi$ implies $\langle\langle\Sigma_0 \cup \Sigma_1\rangle\rangle\varphi$. The converse, however is generally not true. Suppose $M$ satisfies
\[\langle \Sigma_0 \cup \Sigma_1 \rangle \varphi, \text{ but not } \langle \Sigma_0 \rangle \varphi. \] Then, there exists a refinement \( M' \) of \( M \) that satisfies \( \langle \Sigma_0 \rangle \varphi \). Such a refinement is obtained from \( M \) by restricting the behavior of the agents in \( \Sigma_1 \) so that they always collaborate with the agents in \( \Sigma_0 \) to achieve \( \varphi \).

Consider the simplified Needham-Schroeder’s public-key encryption protocol, as discussed in [Low96]. The faulty protocol is given as below:

1. \( A \to B : A.B.\{N_a, A\} K_b \)
2. \( B \to A : B.A.\{N_a, N_b\} K_a \)
3. \( A \to B : A.B.\{N_b\} K_b \)

Here the agent \( A \) is the initiator and the agent \( B \) is the responder. \( A \) randomly selects a nonce \( N_a \) and sends it along with his own ID to \( B \), encrypted with \( B \)'s public-key \( K_b \). \( B \) decrypts the message, obtains the nonce \( N_a \) and sends it back to \( A \) along with a randomly selected nonce \( N_b \), encrypted with \( A \)'s public-key \( K_a \). \( A \) then decrypts this message, obtains the nonce \( N_b \) and sends it back to \( B \). \( B \) then checks if this nonce is the same as the one it sent earlier.

The protocol is modeled as three interacting modules, the initiator \( A \), the responder \( B \), and an intruder \( I \). We assume that the module \( A \) nondeterministically initiates a request to talk to \( B \) or \( I \). The module \( I \) nondeterministically chooses to (1) overhear and remember every message passing through the network, and decrypt any message when it has the key, (2) intercept and/or delete any message, or (3) generate messages using any combination of knowledge that it knows, such as replaying any overheard messages. We want to check that \( A \) and \( B \) are correctly authenticated. The correctness criterion is given as an invariant in ATL:

\[
\text{auth} : \langle \rangle \Box ((B.\text{state} = \text{COMMIT} \land B.\text{talk} = A) \rightarrow A.\text{request} = B))
\]

It states that \( B \) will not commit to talk to \( A \) unless \( A \) has initiated a request to talk to \( B \). When this property is checked against our model, Mocha is able to uncover the same execution sequence that leads to false authentication as described in [Low96]. Instead of fixing the protocol as suggested in [Low96] directly, we try the following approach. The idea is that we successively weaken the property to be verified, and try to discover a possible refinement of the incorrect protocol to implement the correct one. So we check a weaker and more natural property: \( A \) and \( B \) should have a strategy to avoid false authentication. In ATL, it is stated as:

\[
\text{auth}' : \langle A, B \rangle \Box ((B.\text{state} = \text{COMMIT} \land B.\text{talk} = A) \rightarrow A.\text{request} = B))
\]

Mocha reports success when checking \( \text{auth}' \). On close examination of the original attack and the model, it easy to see that \( A \) can avoid the attack simply by never initiating a request (which is not interesting), or by never talking to \( I \). However, this is not a realistic assumption as \( I \) could be an ordinary network user and it is not realistic to assume that \( A \) never talks to \( I \). In order to correct the protocol, we change our model to make \( A \) always talk to \( I \). Then we check the same property against the modified model. This time Mocha reports fail, implying that no refinement of \( A \) or \( B \) implements the correct protocol.

Suppose we suspect that by including an encrypted ID in the reply message (2) from \( B \) to \( A \), this protocol can be corrected. Yet, we do not know whose ID to include. So we modify \( B \) to
module Delay
interface out: E
external in: E
private state: \{stable, unstable\} : x : C
atom controls state, out, x awaits in'
init
\[ \text{true} \rightarrow state' := \text{stable} ; out' := \text{in'} \]
update
\[ state = \text{stable} \land \text{in'} \neq \text{in} \rightarrow state' := \text{unstable} ; x' := 0 \]
\[ state = \text{unstable} \land x \geq 2 \rightarrow state' := \text{stable} ; out' := \text{in'} \]
delay
\[ state = \text{stable} \rightarrow \text{true} \]
\[ state = \text{unstable} \land x \leq 3 \rightarrow x' \leq 3 \]

Figure 2: Delay element

nondeterministically include either A’s or B’s ID in the reply message. The modified protocol looks like this:

1. \( A \rightarrow B : A.B.\{N_a, A\}_{K_b} \)
2. \( B \rightarrow A : B.A.\{N_a, N_b, X\}_{K_a} \)
3. \( A \rightarrow B : A.B.\{N_b\}_{K_b} \)

where \( X \) is either \( A \) or \( B \). We also modify \( A \) to check the included ID. The Reactive Modules description of the modified agents \( A \) and \( B \) is given in Appendix B for reference.

This is still an incorrect protocol, and the property \( \text{auth} \) fails. However, the weaker requirement \( \text{auth'} \) is satisfied. This implies that some refinement of the above incorrect protocol implements the correct one. There are two possible refinements: setting \( X \) to either \( A \) or \( B \). Verification with \textsc{Mocha} reveals that setting \( X \) to \( A \) is not the right refinement. Setting \( X \) to \( B \) is the correct one, and it is exactly the same correction suggested in [Low96].

6 Support for Timed Modules

\textsc{Mocha} supports real-time systems that are described in the form of \textit{timed reactive modules} as defined in [AH97]. In addition to the discrete-valued variables of reactive modules, a timed module makes use real-valued \textit{clock variables}. All clock variables increase at the same rate, and keep track of time elapsed after they have been assigned a value by a guarded command. The guards of later transitions can depend on the values of clocks. Each location specifies an invariant on clock values that must hold for the module to be able to remain in that location. Figure 2 is a simple timed module representing a delay element, the output of which follows its input with a delay in the range \([2, 3]\). The clock \( x \) is reset to 0 when the module becomes unstable. The guard \( x \geq 2 \) at \( \text{state} = \text{unstable} \) makes sure that at least two time units elapse before the module can become stable again. The invariant \( x \leq 3 \) expressed by the delay statement limits the time spent in \( \text{state} = \text{unstable} \) to 3.
MOCHA restricts guards on clocks and clock invariants to be positive Boolean combinations of inequalities of the form $x \leq c$ and $x \geq c$, where $c \in \mathbb{N}$. This is adequate for modeling the bounds on delays of any physical system. In [HMP92], it is proven that, with this restriction, for each trace $\gamma$ of a timed module, there exists a trace $[\gamma]$ such that (i) the sequence values that discrete variables take on is the same for $\gamma$ and $[\gamma]$ and (ii) all updates of discrete variables take place at integer-valued points in time. This enables clocks to be modeled as integer-valued variables that increase at the same rate. Timed modules are converted by MOCHA into (untimed) modules, equivalent to the original ones in the sense described. With this, all algorithms presented in previous sections are applicable to timed modules as well. Compositional and assume-guarantee style proof rules are valid also for timed modules (see [TAK+96, AH97]).

References


Appendix A: A snapshot of a session with mocha

File: Option

- Successor States

<table>
<thead>
<tr>
<th>state_w</th>
<th>state_e</th>
<th>signal_w</th>
<th>signal_e</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait</td>
<td>wait</td>
<td>red</td>
<td>red</td>
</tr>
<tr>
<td>wait</td>
<td>away</td>
<td>red</td>
<td>red</td>
</tr>
<tr>
<td>wait</td>
<td>wait</td>
<td>red</td>
<td>green</td>
</tr>
<tr>
<td>wait</td>
<td>away</td>
<td>red</td>
<td>green</td>
</tr>
</tbody>
</table>

- Simulation Trace

<table>
<thead>
<tr>
<th>state_w</th>
<th>state_e</th>
<th>signal_w</th>
<th>signal_e</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait</td>
<td>bridge</td>
<td>red</td>
<td>green</td>
</tr>
<tr>
<td>wait</td>
<td>away</td>
<td>red</td>
<td>green</td>
</tr>
<tr>
<td>wait</td>
<td>wait</td>
<td>red</td>
<td>red</td>
</tr>
<tr>
<td>wait</td>
<td>bridge</td>
<td>red</td>
<td>green</td>
</tr>
<tr>
<td>wait</td>
<td>bridge</td>
<td>red</td>
<td>green</td>
</tr>
<tr>
<td>wait</td>
<td>away</td>
<td>red</td>
<td>green</td>
</tr>
<tr>
<td>wait</td>
<td>bridge</td>
<td>red</td>
<td>green</td>
</tr>
<tr>
<td>wait</td>
<td>away</td>
<td>red</td>
<td>green</td>
</tr>
</tbody>
</table>

File: Miscellaneous

Definitions

- Interface

```plaintext
interface:
  signal_w: Signal_t;
  signal_e: Signal_t;
  pc_w: TrainStatus_t;
```

- private

```plaintext
private: $arrive_0: event;
  $leave_0: event;
  $near_0: bool;
  $arrive_1: event;
  $leave_1: event;
  $near_1: bool;
```

File: Help

Welcome to MOCHA 1.0
Please report any problems to mocha@eecs.berkeley.edu
mocha: parse examples/fall2cm
Module Train is composed and checked in.
Module Train_W is composed and checked in.
Module Train_E is composed and checked in.
Module Controller is composed and checked in.
Module Rail is composed and checked in.
parse successful.

mocha: mocha: Module Rail initialized.
mocha:
Appendix B: Needham-Schroeder’s Public-key protocol

/ *
This is a modeling of NS public-key authentication protocol
Alice is the initiator, Bob is the responder, and Ivan is the intruder
*/

type KEY : {KEY_A, KEY_B, KEY_I, KEY_W}
type NONCE: {NONCE_A, NONCE_B, NONCE_I, NONCE_W}
type AGENT: {ALICE, BOB, IVAN, NONE}
type STATE: {IDLE, SENT, RECV, COMMIT}

/* the message is modeled as a structure */
#struct message: {key:KEY; id:AGENT; n1:NONCE; n2:NONCE } #endstruct

/* Alice is the initiator. */
module Alice
interface Astate: STATE; request:AGENT; a2i:message
external i2a:message
private nonceRecv: NONCE; ok:event

/* this atom changes the state of Alice */
atom controls Astate reads ok, Astate, request
awaits ok
init
□ true -> Astate’:=IDLE
update
□ Astate = IDLE -> Astate’ := SENT; request’ := IVAN
□ Astate = SENT & ok? -> Astate’ := COMMIT
endatom

/* this atom changes the output */
atom controls a2i reads a2i awaits request, nonceRecv, Astate
init
□ true -> a2i.n1’ := NONCE_W; a2i.n2’ := NONCE_N
update
□ Astate’ = SENT -> a2i.key’ := KEY_I; a2i.id’ := ALICE; a2i.n1’ := NONCE_A
□ Astate’ = COMMIT-> a2i.key’ := KEY_I; a2i.n1’ := nonceRecv’
endatom

/* this atom checks the reply message 2. */
/* the encrypted id is also checked */
atom controls ok, nonceRecv reads i2a, request, Astate, nonceRecv
init
□ true -> nonceRecv’ := NONCE_N
update
□ i2a.key = KEY_A & i2a.n1 = NONCE_A & "(i2a.n2 = NONCE_W) & "(Astate = IDLE)
  & (i2a.id = request | i2a.id = ALICE) -> nonceRecv’ := i2a.n2; ok!
□ Astate = IDLE -> nonceRecv’ := NONCE_N
endatom

endmodule
/* Bob is the responder */
module Bob
interface Bstate:STATE; Btalk:AGENT; b2i:$message
external i2b:$message
private ok:event; nonceRecv:NONSE

/* this atom changes the state of Bob */
atom controls Bstate reads ok, Bstate awaits ok
  init
    [] true -> Bstate' := IDLE
  update
    [] Bstate = IDLE & ok? -> Bstate' := RECV
    [] Bstate = RECV & ok? -> Bstate' := COMMIT
endatom

/* this atom checks messages 1 and 3 */
atom controls ok, nonceRecv, Btalk
reads i2b.key, i2b.id, i2b.n1, i2b.n2, Bstate, Btalk, nonceRecv, ok
  init
    [] true -> Btalk' := NONE; nonceRecv' := NONCE_N
  update
    [] Bstate = IDLE & i2b.key = KEY_B & 
      "(i2b.id = NONE) & 
      "(i2b.id = IVAN) -> ok!!; Btalk' := i2b.id; nonceRecv' := i2b.n1
    [] Bstate = RECV & i2b.key = KEY_B & i2b.n1 = NONCE_B -> ok!!
endatom

/* this atom outputs the reply message 2.*/
/* it nondeterministically choose what id to be encrypted */
atom controls b2i awaits Bstate, nonceRecv, Btalk
  init
    [] true -> b2i.n1' := NONCE_N; b2i.n2' := NONCE_N
  update
    [] Bstate' = RECV -> b2i.key' := if (Btalk'=IVAN) then KEY_I else KEY_A fi;
      b2i.id' := nondet(ALICE, BOB);
      b2i.n1' := nonceRecv'; b2i.n2' := NONCE_B
endatom
endmodule

/* the protocol is modeled as the composition of Alice, Bob and Ivan */
SYS := Alice || Bob || Ivan