# MOCHA: Exploiting Modularity in Model Checking

L. de Alfaro* R. Alur† R. Grosu† T. Henzinger* M. Kang†
R. Majumdar* F. Mang* C. Meyer-Kirsch* B.Y. Wang†

August 2, 2000

## 1 Introduction

MOCHA is a growing interactive software environment for specification, simulation and verification of *concurrent systems*. The main objective of MOCHA is to exploit the modularity in the design structure during model checking. It is intended as a vehicle for 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 capabilities:

- System specification in the language of REACTIVE MODULES. Reactive modules allow the formal specification of heterogeneous systems with synchronous and asynchronous components. Reactive Modules support modular and hierarchical structuring and reasoning
- System *execution* by randomized or manual trace generation. In the manual mode, the user may choose at each step one of the possible next state of the system.
- Requirement verification by *invariant checking*. MOCHA supports both *symbolic* and *enumerative* search. The symbolic model checker is based on BDD engines developed by the UC Berkeley *VIS project*.
- Implementation verification by checking trace containment between implementation and specification modules. The check can be performed automatically if the specification module has no private variables, and otherwise, the user has to supply a witness module defining the refinement mapping. For decomposing proofs, MOCHA supports an *assume-guarantee* principle.

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

- An updated *graphical user interface* written in Java that looks familiar to Windows/Java users: it has a project window and a desktop, has a syntax directed editor, allows concurrent threads, can be easily extended and debugged.
- A *new simulator* with a graphical user interface that displays traces in a message sequence chart (MSC) fashion and shows the dependencies among variable updates.
- A *proof manager* for managing verification proofs such as assume-guarantee proofs.
- An *enhanced enumerative checker* for invariant checking as well as refinement checking with many new optimizations like hierarchic reduction.

*Department of Electrical Engineering and Computer Science, University of California, Berkeley
†Department of Computer and Information Science, University of Pennsylvania

- A new *scripting language* called SLANG for rapid and structured algorithm development. Slang provides primitive functions for symbolic manipulation of transitions systems and states, and new symbolic algorithms can be programmed by writing SLANG scripts.

The architecture of JMOCHA is shown in Figure 1 where the free bidirectional arrows denote user interaction via the graphical user interface.

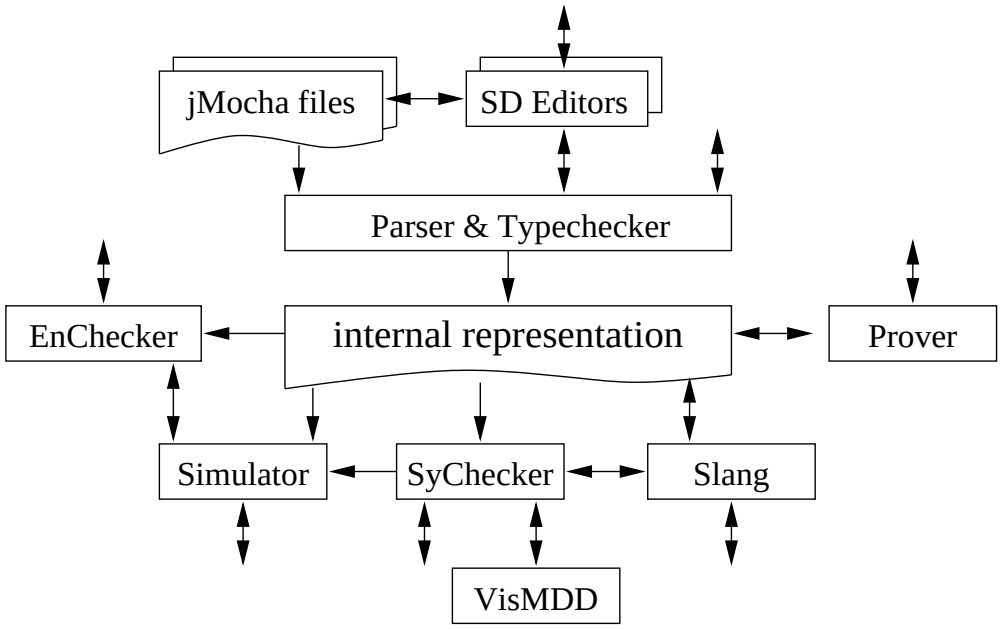


Figure 1: JMOCHA tool architecture

The rest of the paper describes each of the above components. In Section 2 we introduce the specification language *reactive modules*. In Section 3 we describe the graphical user interface. In Section 4 we describe the simulator. In Section 5 we describe the checkers. In Section 7 we describe the scripting language.

## 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 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. These variables are also called *observable* variables. A module may also have a set of *private* variables. Variables are *typed*. The types supported are enumerated types, events, arrays and bitvectors. New enumerated or array types may be introduced for convenience.

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 in an order consistent with the await dependencies.

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*. By using the keyword lazy, the idling transition is implicitly added to an atom.

Reactive modules can be *composed* to build hierarchic reactive modules, if they have disjoint sets of interface variables and their union of atom sets does not contain a circular await dependency. To facilitate composition and enhance modularity, interface variables may be *hidden* and observable variables may be *renamed*. For example, if $M, M_1$ and $M_2$ are appropriate modules, $x$ is an interface, $y$ is an external variable of $M$ and $u, v$ are fresh variable names for $M$ then $M_2 \| M_2$ is the composition of $M_1$ with $M_2$, `hide` $x$ `in` $M$ is the module $M$ with $x$ hidden and $M[x, y := u, v]$ is the module $M$ with $x$ and $y$ renamed by $u$ and $v$.

For example, consider the specification of a village telephone system that, for simplicity, contains only 4 telephones. The specification consists of two modules: the first one models the environment, i.e., the users, the second one models the system. The types below, define the states of the phones and the lines in the telephone system. A line is either disconnected, drooping, or connected to one of the phones. A phone is either on-hook or off-hook.

```
type connType is { disconn, conn1, conn2, conn3, conn4, drooping }
type hookType is { on, off }
```

The module `UserSpec` is a very abstract model of the users. It nondeterministically toggles at most one telephone between on-hook and off-hook.

```
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; h3' := on; h4' := on;
  update
    [] h1 = on -> h1' := off;
    [] h1 = off -> h1' := on;
    [] h2 = on -> h2' := off;
    [] h2 = off -> h2' := on;
    [] h3 = on -> h3' := off;
    [] h3 = off -> h3' := on;
    [] h4 = on -> h4' := off;
    [] h4 = off -> h4' := on;
```

The specification module `SystemSpec` below, defines a telephone system that establishes and destroys connections between communication partners. The extra variable `p` is used to select the partner pairs: `p=0` means 1-4/2-3, `p=1` means 1-3/2-4 and `p=2` means 1-2/3-4. The atom `Conn1` is defined as follows. If the user hangs up, it sets `c1` to `disconnected`. If the partner hangs up, it sets `c1` to `drooping`. If the phone is off-hook and disconnected, it checks the partner (selected by `p`) and tries to connect.

```
module SystemSpec is
 interface c1,c2,c3,c4 : connType; p : (0..2);
 external h1,h2,h3,h4 : hookType;

atom selectPartner
 controls p
 init
  [] true -> p' := nondet;
 update
  [] true -> p' := nondet;

atom Conn1
 controls c1
```

```
reads c1,c2,c3,c4,p
awaits h1,h2,h3,h4,p
init
 [] true -> c1' := disconn;
update
 [] (h1' = on) -> c1' := disconn;
 [] (c1 = conn2) & (h2' = on) -> c1' := drooping;
 [] (c1 = conn3) & (h3' = on) -> c1' := drooping;
 [] (c1 = conn4) & (h4' = on) -> c1' := drooping;
 [] (c1=disconn) & (h1'=off) & (p=0) & (c4=disconn) & (h4'=off) -> c1' := conn4;
 [] (c1=disconn) & (h1'=off) & (p=1) & (c3=disconn) & (h3'=off) -> c1' := conn3;
 [] (c1=disconn) & (h1'=off) & (p=2) & (c2=disconn) & (h2'=off) -> c1' := conn2;
```

The atoms `Conn2` to `Conn4` are not shown in this specification. They are specified in a similar way to `Conn1`.The line below defines the specification module `Spec` as the parallel composition of `UserSpec` and `SystemSpec`.

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

# 3 The Graphical User Interface

Similarly to modern Windows or Java tools, the interaction between the user and JMocha is controlled by a *graphical user interface (GUI)*. 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`. They contain buttons with intuitive icons that may be used as shortcuts for the most frequently used menu items. One can drag the tool bars at any convenient place outside the tool bars standard location.

The menu items and the tool bar buttons are activated/deactivated in a way consistent with the state of the proof manager. This avoids undesired input and guides the user by telling him what are the available options. At the beginning only three buttons and correspondingly four menu items are active: `Open Project, New File, Open File` and `Exit`. Clicking these buttons (menu items) one can use Mocha in an *editor* and a *project mode*.

## 3.1 Mocha in Editor Mode

If one clicks `NewFile` or `Open File` then one may use Mocha as a syntax directed editor window for the Reactive Modules language. Both options also activate the other `File` menu and `Edit` menu items. One may open more than one file and the labels associated to their windows allow to conveniently switch from one window to another even if they maximized, as shown in Figure 2. One may edit the files by using the menu items in the `Edit` menu or the associated toolbar. The edit action always takes place in the currently selected editor frame (topmost frame). One can cut and paste from one editor window in another editor window.
The editor windows highlight the Reactive Modules keywords and comments. One can enable/disable *parsing on the fly* by clicking the check box item `Enable Parsing` inside the sub menu `Editor Options` of the top menu `Options`. In case of an error while typing, the first erroneous token is highlighted in red. One can further enable a pop up window prompting the user with the allowed next tokens. Clicking on the pop up options, the associated text is automatically inserted at the current cursor position. This allows not only to correct almost all syntactic errors at typing but also to learn the Reactive Modules language, as shown in Figure 3. One may enable/disable the pop up mode by clicking the check box item `Enable Pop up` inside the sub menu `Editor Options` of the top menu `Options`. The specifications of modules can be imported from other files using the `import` command.

## 3.2 Mocha in Project Mode

To make it reasonably fast, the *on-the-fly parser* neither does expand any import declarations, nor does any type-checking, nor does generate any code. Once one has edited and saved a tree of reactive modules files

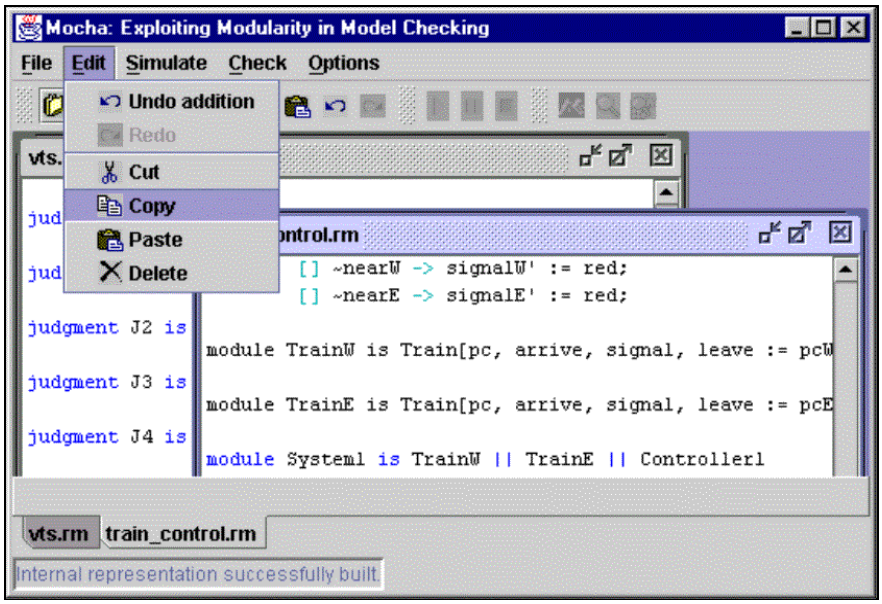


Figure 2: Using MOCHA in editor mode

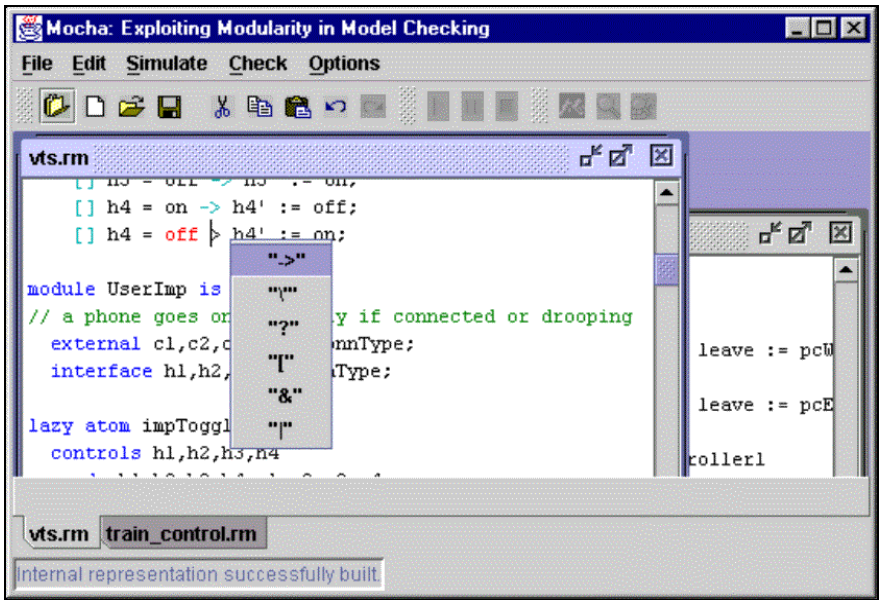


Figure 3: Using parsing on the fly

one may want to simulate and check them. For this purpose, one has to press the `Open Project` button or the associated menu item inside the `File` menu and select the root reactive modules file.

In this case the proof manager expands all the import declarations 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 4.

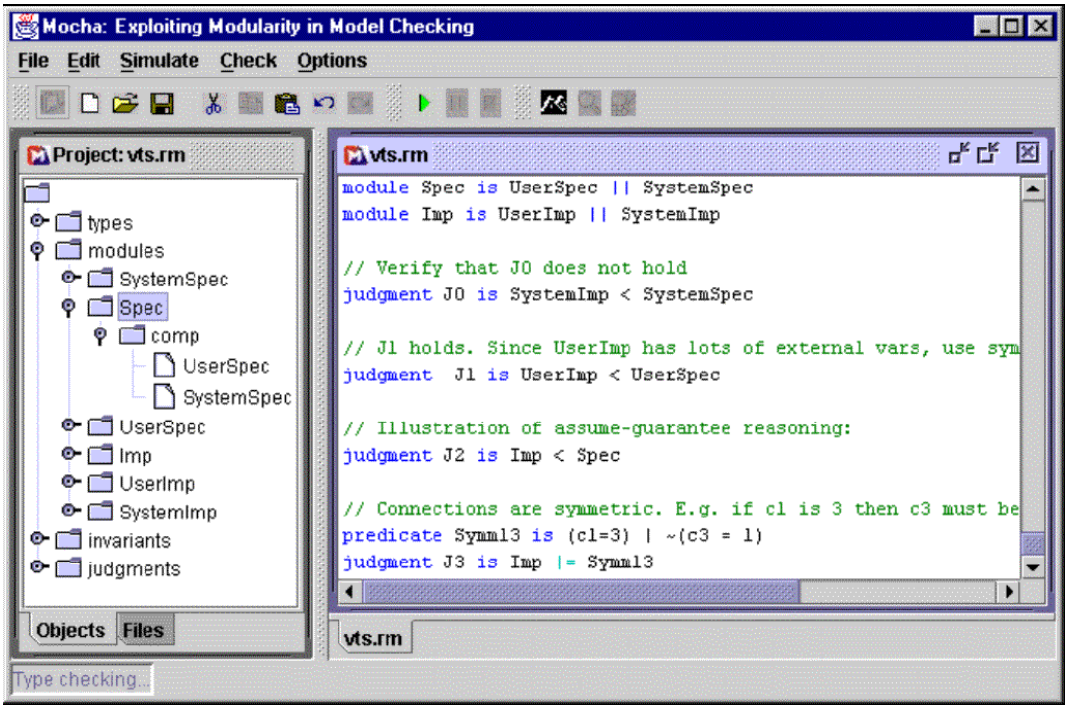


Figure 4: Using MOCHA in project mode

In the project mode one may open and edit reactive modules files in the same way one does it in the edit mode. Moreover, it enables the menu items `Parse` and `Type Check` inside the top menu `Check`. Clicking one of these items, JMOCHA parses and type checks the root file (and its associated imports) again and updates, if there is no error, the project window. In case of error, the project window displays the last consistent state. Note that before parsing or type checking all files opened in the desktop are automatically saved. Note also that the `Type Check` option first invokes the parser to make sure that the code to be type checked has no parsing errors

The project window displays the MOCHA proof context in a very convenient, tree notation. Each node in the tree may be expanded or collapsed by clicking it. The proof context consists of several sub contexts: `types`, `modules`, `formulas` and `judgments`. They are initially collected from the associated reactive modules files. When a module is selected, two buttons get highlighted: the reach and the run buttons. When selecting a judgment two other buttons get selected: check and in case of refinement statements, decompose. They are discussed later.

# 4 The Simulator

The behaviour of a reactive system may be visualized in a *message sequence charts (MSC)* like fashion by using the *simulator*. Alternatively, one may view the graphical display as an intuitive visualization of the *executions* of a reactive module. In these executions, the values of the variables are displayed only when they change. Clicking on the box displaying a value of a variable, shows what other variables (and their

values) contributed to the change. This information can be used for debugging or simply for understanding in detail the behavior of the given reactive module description.

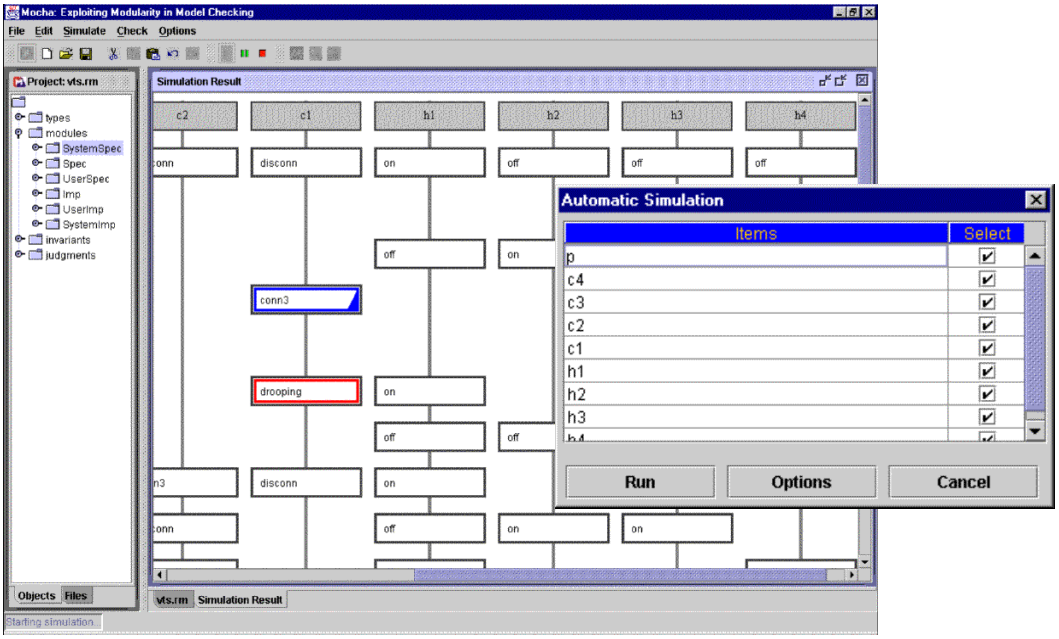


Figure 5: The simulator

Starting the simulator, the user gets a *simulation dialog* that allows him to choose the level of simulation (variable, atom or module) and what variables are to be displayed, as shown in Figure 5. For each, 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 simulator can be used in a stand-alone fashion. One has two options to run the simulator in this mode. One is to hand over the control to MOCHA(*automatic simulation*), the other is to control every step by oneself (*manual simulation*). One can choose this option from the 'Simulator' menu. In automatic simulation, when one clicks the simulation button and then the start button, MOCHA will take control of the simulation. It will make the simulation proceed by choosing 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.

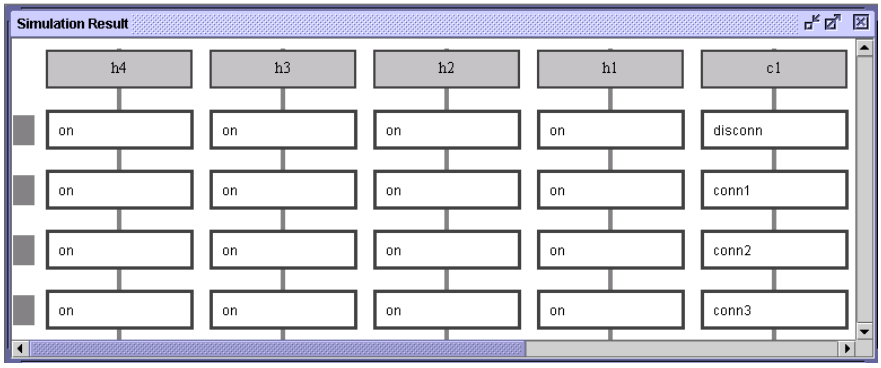


Figure 6: Manual simulation

If one wants to control the step-by-step execution of a module, one needs to select manual simulation before starting the simulation. After clicking a module, click the simulation start 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. Figure 6 shows an example of manual simulation. In the output window, one can see many states beginning with grayed boxes. Clicking one of these grayed boxes selects the associated state and the simulation proceeds one step. The simulation then proceeds in a similar way.

In the simulation output window, if one clicks any value box in a certain state, that box is inlined with a red color and all the boxes it depended on to get this value are inlined in blue. Clicking the box again, hides the above information. For example, in Figure 7, clicking the box labeled `l1 = 1` of the last state, results in inlining this box in red an the `off` and `l1 = 0` boxes in blue.

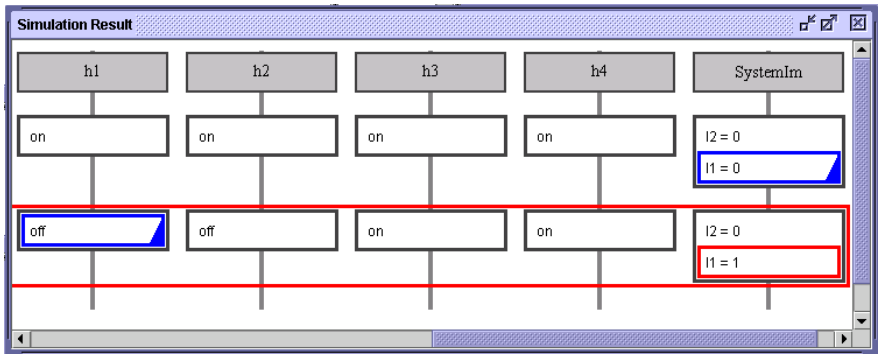


Figure 7: Counterexample display

Using MOCHA's checker, one can verify whether a given specification (judgement) is true. With an exhaustive search, MOCHA might find a state where the judgement fails. In this case, the simulator, which is integrated with the checker, will automatically provide a graphical sequence of states where the final state fails the given judgement. Figure 7 shows an example. The last state is surrounded by a red border, meaning that it violates the specified invariant.

# 5 The Invariant Checkers

JMOCHA allows specification of requirements as *alternating temporal logic (ATL)* formulas. Currently however, it has a built in checker only for a restricted (but most common) class of ATL formulas, namely *invariants*. A state (or transition) invariant is a predicate that is required to hold on all reachable states (or transitions) of a reactive module. This is not a limitation because the user may define itself more powerful checkers by using SLANG, as shown in Section 7.

For example, an invariant of the village telephone system specification, is the property that connections are symmetric. The following formula specifies that if phone 1 is connected to 3, then phone 3 should be connected to 1:

```
predicate Symm13 is (c1=3) | ~(c3 = 1)
judgment J3 is Spec |= Symm13
```

One may check this invariant either enumeratively or symbolically.

## 5.1 Enumerative Invariant Checking

The core of the enumerative search engine is a routine to compute the successors of a given state. It first generates all possible values of the external variables. It then goes through each atom in the order consistent with await-dependencies. For an atom, each guard in the guarded command is evaluated to check whether it is enabled, and then actions corresponding to all enabled guards are executed. After all atoms have been processed, all controlled variables are assigned to their new values, and the invariant predicate is checked

if it is satisfied by successor states. The invariant judgment is valid if the search engine has traversed all reachable states.

We have implemented various features and optimizations in the JMOCHA enumerative search engine. Some of them are listed below:

- For every transition, information about how the updated value of a variable depends upon the old/new values of other variables is generated. This information can be visualized with the help of the simulator.
- Each state is stored as bit string to save space using compression.
- Unlatched (not read by any of the atoms) variables are not stored in the table.
- Event variables are not stored in the table (they are updated on the fly during computation of successor states).
- Independent atoms are grouped together. Each group generates partial successor states. Successor states are cross products of these partial states.

A user can check enumeratively whether an invariant holds in a module as follows. First he selects `Enumerative check` in the `Check` options. Then he selects the judgment and clicks either the `Check` button (the magnifying glass) in the toolbar or the `Check` menu item inside the `Check` menu, as shown in Figure 8. If the invariant is not satisfied, JMOCHA produces a counterexample execution along with its variable dependency information in a simulator window.

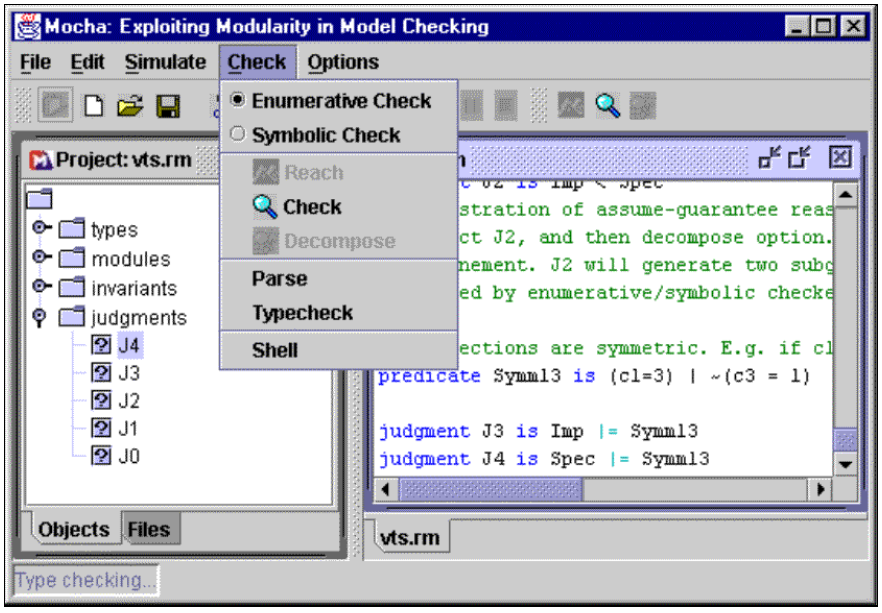


Figure 8: Enumerative invariant checking in MOCHA

For modules which consist of only lazy atoms, JMOCHA provides a heuristic called hierarchical reduction to reduce search space [4]. The basic idea is to merge several internal steps into one. If the module is composed of independent submodules, the search space may be reduced by the heuristic.

## 5.2 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]. 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 [8] that has a very efficient early quantification heuristic. Note that most of the symbolic model checker is written in Java. However, it calls the VIS MDD routines that are written in C, to construct and manipulate MDDs efficiently.

For example, if the default checker is the symbolic checker and the selected judgment for the village telephone system example is J4, then clicking the check button starts the symbolic ckecker that produces the result shown in Figure 9.

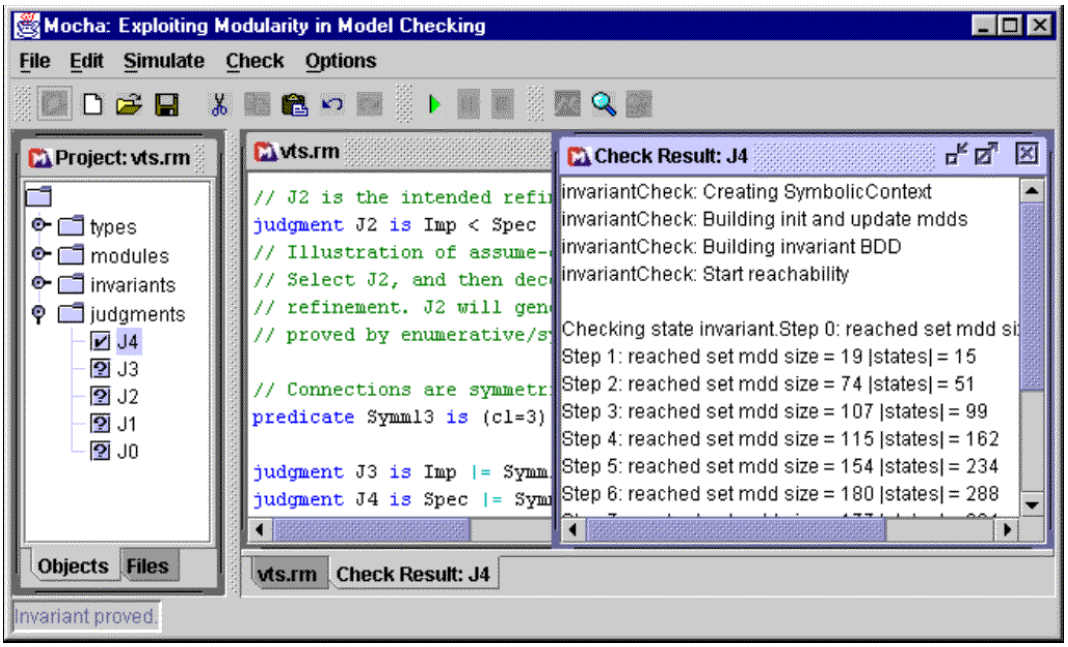


Figure 9: Symbolic invariant checking in MOCHA

A main objective of this release of the symbolic model checker was to support bit vectors and arrays efficiently. In particular, the efficient representation of non-constant references to components of compound data types like bit vectors and arrays. For example, the occurrence of a variable $i$ in $a[i]$ is a non-constant reference to an array $a$. We use enumeration (like the enumerative checker) to compute the values of non-constant references and, additionally, of any closed expression (a closed expression is an expression with all variables bound by a quantifier). Thus the model generator computes the actual values of each reference or closed expression with respect to its quantifier pattern before constructing the MDD representation. Unquantified variables in reference expressions are assumed to be universally quantified. Note that the model generator keeps track of the instantiations of the variables to constrain the MDD representation.

# 6 The Refinement Checkers

Refinement checking gives users the possibility to verify if a module (the implementation) *refines* another module (the specification). 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 implementation module simulates the specifiction module assuming that (1) the specification contains no private (or hidden) variable and (2) that all variables of specification module appear in the implementation module as well. In this case, checking simulation relation is reduced to checking transition invariant: first, the refinement checker checks that all the initial states of the implementation module are contained in that of the specification module, and each reachable transition of of the implementation satisfies the transition relation of the specification module. This can be done efficiently

either symbolically or enumeratively. If there is an implementation execution which is not permitted by specification, JMOCHA will reproduce the execution (along with the variable dependency information) in a simulation window. The user can then change the design by examining the execution.

For example, an intended refinement of the module `UserSpec` is the module `UserImp` below. It makes sure that a phone goes on-hook only if it is connected or drooping.

```
module UserImp is
  external c1,c2,c3,c4 : connType;
  interface h1,h2,h3,h4 : hookType;
lazy atom impToggleHook
  controls h1,h2,h3,h4
  reads h1,h2,h3,h4,c1,c2,c3,c4
  init
    [] true -> h1' := on; h2' := on; h3' := on; h4' := on;
  update
    [] h1 = on -> h1' := off;
    [] ~(c1 = disconn) -> h1' := on;
    [] h2 = on -> h2' := off;
    [] ~(c2 = disconn) -> h2' := on;
    [] h3 = on -> h3' := off;
    [] ~(c3 = disconn) -> h3' := on;
    [] h4 = on -> h4' := off;
    [] ~(c4 = disconn) -> h4':= on;
```

The intended refinement relation can be stated in JMOCHA as below. It can be subsequently checked either enumeratively or symbolicaly.

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

There are several ways to circumvent the simulation restrictions 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 module, a witness module can be built to assign values to the variable. The witness is in turn composed with the implementation module and checked against the specification [7, 6, 1]

## 6.1 Enumerative Refinement Checking

When an implementation module is checked against a specification, the enumerative search engine generates all possible successor states of implementation, as described in the invariant checking algorithm. Similarly, all successors of the specification are generated in the same way. It then projects all implementation states to specification states and checks if the projections are included in the specification. The judgment is valid if the search engine has traversed all reachable states of the implementation.

To perform the refinement checking, the user first selects the refinement judgment in the project window, then chooses the `Check` menu. The enumerative search engine will report how many states have been visited. The optimizations of enumerative invariant checking are also applied. In particular, JMOCHA uses different algorithm to check lazy modules.

## 6.2 Symbolic Refinement Checking

In this case, checking simulation relation is reduced to checking transition invariant: first, the refinement checker checks that all the initial states of the implementation module are contained in that of the specification module, and each reachable transition of of the implementation satisfies the transition relation of the specification module.
For example, the result of checking the refinement judgment `J1` symbolically is shown in Figure 10.

## 6.3 Assume/Guarantee Reasoning

Consider the implementation module `SystemImp` (shown below) of the the village telephone system, where the connections are hot-lines (1–2 and 3–4). The variable `p` has no role in this case. It is there just because

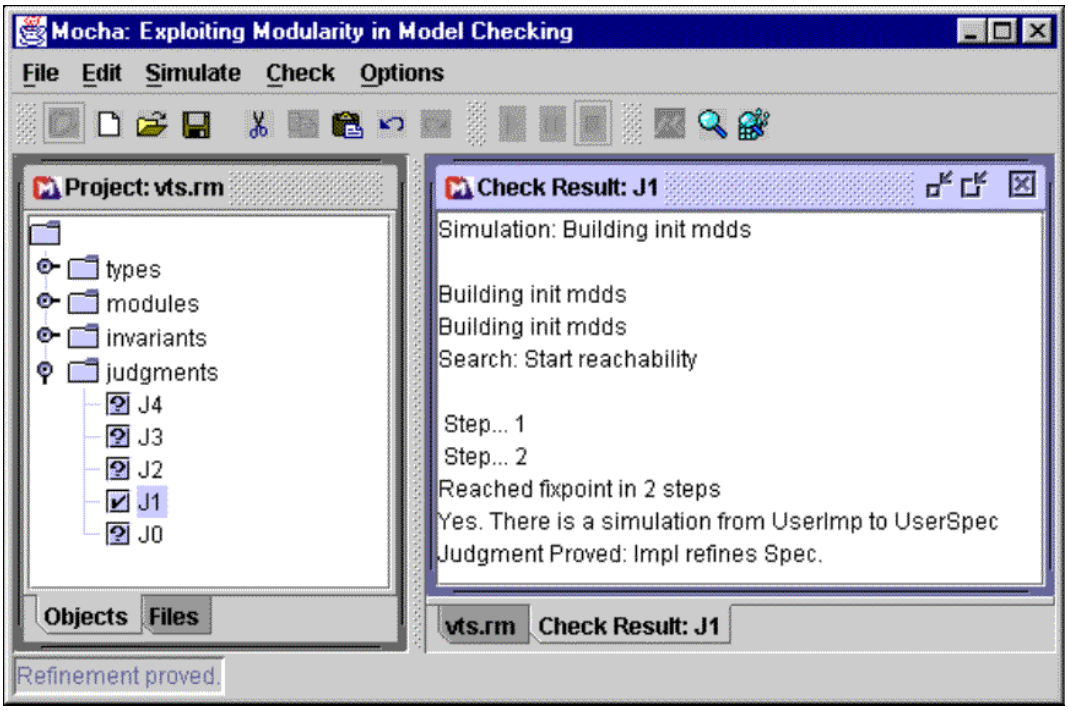


Figure 10: Symbolic refinement checking in JMocha

the specification module `SystemSpec` cannot have private variables. The variables `l1` and `l2` are used for the finite-state control of the hot-line 1 and respectively hot-line 2.

```
module SystemImp is
  interface c1,c2,c3,c4 : connType; p : (0..2);
  external h1,h2,h3,h4 : hookType;
  private l1,l2 : (0..7);

atom selectHotPartner
  controls p
  initupdate
    [] true -> p' := 2;

atom Line1
   controls c1,c2,l1
   reads l1
   awaits h1,h2
   init
     [] true -> c1' := disconn; c2':= disconn; l1':=0;
   update
     [] (l1=0) & (h1'=off) -> l1':=1;
     [] (l1=0) & (h2'=off) -> l1':=2;
     [] (l1=1) & (h1'=on) -> l1':=0;
     [] (l1=2) & (h2'=on) -> l1':=0;
     [] (l1=1) & (h2'=off) -> c1':=conn2; c2':=conn1; l1':=3;
     [] (l1=2) & (h1'=off) -> c1':=conn2; c2':=conn1; l1':=3;
     [] (l1=3) & (h1'=on) -> c1':=disconn; c2':=drooping; l1':=4;
     [] (l1=3) & (h2'=on) -> c2':=disconn; c1':=drooping; l1':=5;
     [] (l1=4) & (h2'=on) -> c2':=disconn; l1':=0;
     [] (l1=5) & (h1'=on) -> c1':=disconn; l1':=0;
     [] (l1=4) & (h1'=off) -> l1':= 6;
     [] (l1=5) & (h2'=off) -> l1':=7;
     [] (l1=6) & (h1'=on) -> l1':=4;
     [] (l1=7) & (h2'=on) -> l1':=5;
     [] (l1=6) & (h2'=on) -> l1':=1; c2':=disconn;
     [] (l1=7) & (h1'=on) -> l1':=2; c1':=disconn;
```

...

For example, for the first line, `l1=0` models the idle situation, `l1=1` models the situation where the phone 1 is off-hook and waiting for phone 2, `l1=2` models the situation where the phone 2 is off-hook and waiting for phone 1, `l1=3` models the situation where the phones 1 and 2 are connected, `l1=4` models the situation where the phone 1 is on-hook and phone 2 is drooping, `l1=5` models the situation where the phone 2 is on-hook and phone 1 is drooping, `l1=6` models the situation where the phone 1 goes off-hook while phone 2 is drooping, `l1=7` models the situation where the phone 2 goes off-hook while phone 1 is drooping. The line `l2` is used in a similar way for the second hot-line.

The lines below define the specification module `Spec` and the implementation module `Imp` as the parallel composition of `UserSpec` and `SystemSpec` and respectively of `UserImp` and `SystemImp`. The implementation module `SystemImp` is not a refinement of `SystemSpec` for any environment. It is easy to verify that `J0` does not hold. As a consequence, one may not use compositional reasoning, to prove that `Imp` refines `Spec` as stated in judgment `J2`. But this judgment is indeed true, because `SystemImpl` refines `SystemSpec` in the more restrictive contexts given by the user modules.

```
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
```

In this case, one may either try to prove the judgment `J2` directly, but this will involve a quite large state space, or use the following assume-guarantee rule [3, 7]: if $P_1 \| P_2' \preceq P_1'$ and $P_1' \| P_2 \preceq P_2'$ then it follows that $P_1 \| P_2 \preceq P_1' \| P_2'$, where $P_1, P_2, P_1'$ and $P_2'$ are reactive modules.

Given a refinement judgment, the *proof manager (or prover)* of jMocha can suggest as many decompositions as possible according to a built in database of proof rules that includes the above assume-guarantee rule. Once a decomposition is selected, the additional proof obligations will be added to the proof manager as new proof judgments, and they will be displayed in the judgment browser. The user can then discharge each of these proof obligations by invoking the refinement checker as usual.

For example, as shown in Figure 11, one may select `J2` in the project window, and then click decompose button (magnifying glass over a cube) in the tool bar or the check menu. This will pop up a window with two decomposition rules. The first rule is intended for the symbolic checker. In has unconstrained external variables. The second rules is intended for the enumerative checker. In this case it is better to constrain the external variables. Choosing the first assume-guarantee rule, the two subgoals of the rule `J20` and `J21` are automatically inserted in the project window. Both are easily proved by the symbolic checker.

# 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 (among which 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 $\phi$ is a region expression, then the function `create_mdd(`$P$`,`$\phi$`)` returns the mdd that defines the states satisfying $\phi$ on the state-space of $P$. For mdds $\Phi$, $\Phi_1$, $\Phi_2$, and for a module $P$, the available functions include the following ones:

- `and(`$\Phi_1, \Phi_2$`)`, `or(`$\Phi_1, \Phi_2$`)`, `not(`$\Phi$`)` compute the corresponding boolean functions on the mdds;
- `equal(`$\Phi_1, \Phi_2$`)` returns 1 (for *true*) if the mdds $\Phi_1$ and $\Phi_2$ are equal, and 0 (*false*) otherwise;

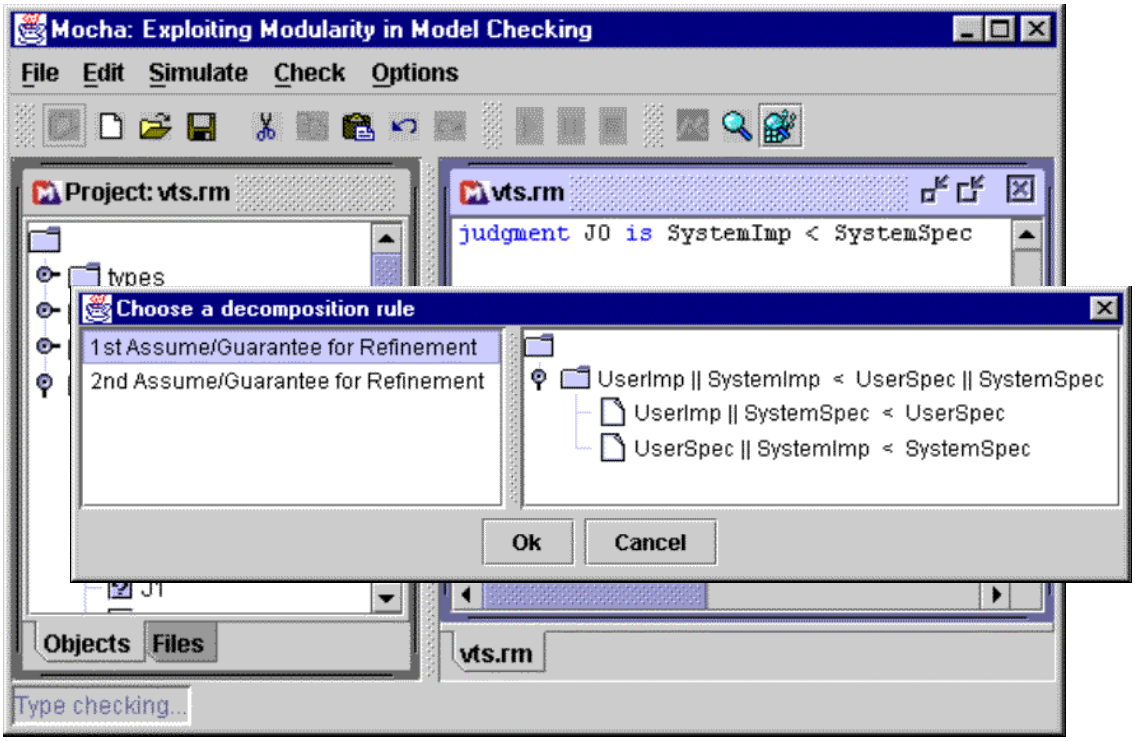


Figure 11: The proof manager and Assume/guarantee reasoning in JMocha

- `init_reg(`$P$`)` returns the MDD representing the initial states of $P$;
- `pre(`$P$`, Φ)` and `post(`$P$`, Φ)` compute the MDDs representing the successor and predecessor states of the set of states represented by $\Phi$.

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. New functions can be defined using the `def` construct: for example, `def f(x) { returns (x+1); }` defines the increment-by-1 function `f`. In Slang, functions are first-order objects, and they can be passed as arguments, assigned, and returned as results: higher-order functions can be straightforwardly written in Slang. Complex definitions can be read from a file using the `source(`*filename*`)` command, which parses and iterprets the commands present in the specified file. 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.

```
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))); }
```

The constant `zeroMdd` represents an MDD with empty truth-set. The function returns 1 (*true*) if module `M` satisfies invariant `phi`, and 0 (*false*) otherwise.

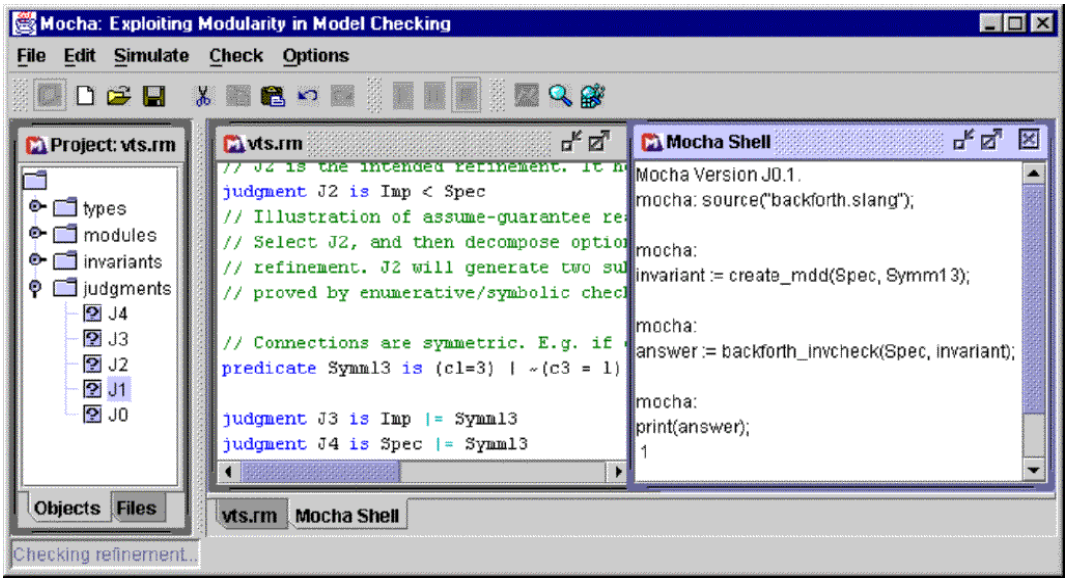


Figure 12: Using SLANG for backwards-and-forwards reachability analysis

In the screenshot of Figure 12, this function is applied to the verification of the invariant `Symm13` of module `Spec`. The definition of the function is in the file `/tmp/local/backforth.slang`, that is sourced into SLANG. Next, the MDD `invariant` is created from the expression `Symm13`, and the invariant is checked using the function `backforth_invcheck`.


### Acknowledgements

We thank Himyanshu Anand, Ben Horowitz, Franjo Ivancic, Michael McDougall, Marius Minea, Oliver Moeller, Shaz Qadeer, Sriram Rajamani, and Jean-Fraccois Raskin for their assistance in development of JMOCHA. The MOCHA project is funded in part by the Defense Advanced Research Projects Agency (DARPA (NASA) grant NAG2-1214), the National Science Foundation (NSF CAREER award CCR95-01708 and CCR97-34115, and award CCR99-70925), the Microelectronics Advanced Research Corporation (MARCO grant 98-DT-660), and the Semiconductor Research Corporation (SRC contract 99-TJ-683.003 and 99-TJ-688).


## References


[1] R. Alur, R. Grosu, and B.-Y. Wang. Automated refinement checking for asynchronous processes. In *Proceedings of the Third International Workshop on Formal Methods in Computer-Aided Design*. 2000.

[2] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proceedings of the 10th International Conference on Computer Aided Verification*, LNCS 1427, pages 516–520. Springer-Verlag, 1998.

[3] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999. Invited submission to FLoC'96 special isuue. A preliminary version appears in *Proc. 11th LICS, 1996.*

[4] R. Alur and B.-Y. Wang. "Next" heuristic for on-the-fly model checking. In *CONCUR'99: Concurrency Theory, Tenth International Conference*, LNCS 1664, pages 98–113. Springer-Verlag, 1999.

[5] The VIS Group. Vis: A system for verification and synthesis. In *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432, 7 1996.

[6] T.A. Henzinger, X. Liu, S. Qadeer, and S.K. Rajamani. Formal specification and verification of a dataflow processor array. In *Proceedings of the International Conference on Computer-aided Design*, pages 494–499, 1999.

[7] T.A. Henzinger, S. Qadeer, and S. Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV 98: Computer-aided Verification*, LNCS 1427, pages 521–525, 1998.

[8] R.K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R.K. Brayton. Efficient formal design verification: Data structures + algorithms. In *Proceedings of the International Workshop on Logic Synthesis*, 1995.