Verification of an Elevator System with MOCHA

Lin Mei, Yuan Gan
Department of Computer Science, University of Toronto
January 28, 2004

Abstract

Unlike many other existing model checkers, MOCHA is designed for the modular verification of heterogeneous systems. Instead of manipulating unstructured state-transition graphs, it supports the hierarchical modeling framework of Reactive Modules. It overcomes the state-explosion problem by exploiting the modular structure naturally present in many system designs. Instead of traditional temporal logics such as CTL and LTL, it uses Alternating Temporal Logic, a module-level specification language. For the verification of complex systems, its algorithms incorporate optimizations based on the hierarchical reduction of sequences of internal transitions. In this paper, we study the language Reactive Modules, the Alternating Temporal Logic and use a case study of an elevator system to try out MOCHA. Finally, we compare it with other tools SMV and SPIN and draw conclusions.

1 Introduction

MOCHA (Modularity in Model Checking) is a growing interactive software environment for the modular and hierarchical verification of heterogeneous systems. MOCHA is developed in a joint venture of R. Alur at University of Pennsylvania and T.A. Henzinger at University of California at Berkeley in 1998[1]. It is available in two versions, cMocha (Version 1.0.1) which is written in C and jMocha (Version 2.0) which is written in Java[2].

MOCHA distinguishes from many other model checkers in modeling, specification, simulation and verification:

♦ Different modeling language: instead of using unstructured state-transition graphs, MOCHA uses the heterogeneous modeling framework of reactive modules [3]. The reactive modules provide semantic glue that allows the formal embedding and interaction of components with different characteristics. The modules can be synchronous or asynchronous, represent hardware or software, and be speed-independent or time-critical.

♦ Nontraditional requirement specification: instead of supporting the system-level specification languages of linear and branching temporal logics, MOCHA supports the module-level specification language of Alternating Temporal Logic (ATL) which can express both cooperative and adversarial relationships between different modules.

♦ Improved algorithms: MOCHA implements enumerative, as well as symbolic, state-exploration algorithms and both checkers have the capability to produce error traces. MOCHA supports a range of compositional and hierarchical verification methodologies so that it integrates assume-guarantee rules, abstraction operators and automatic refinement checking.

♦ Supporting three kinds of simulation, namely, 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.

In this paper, we use a case study of a three-level elevator to explore many of these features under the verification environment of cMocha.

2 Elevator System Specification

Before we get into MOCHA, it is necessary to introduce the system we are going to model. The case study we are using to demostrate MOCHA is an elevator system for an apartment building.

The system consists of an elevator that services three floors of the building and a controller that communicates with the elevator and schedules its moves. Each floor has a request button that a user presses to get the elevator to come to that floor and open its doors. Inside the elevator, there is one request button for each of the three floors; passengers press these buttons to get the elevator to go to a particular floor and open its doors. To go from floor *i* to floor *k*, the elevator must visit floors *i+1* through *k-1*, although it does not have to open doors there. If there are no requests to service, an elevator stays at a floor with its doors open. As passengers press buttons, the controller schedules the elevator to service the requests, trying to minimize the waiting time. The elevator has a "passenger present" detector and a "door open" button. When someone steps into the elevator, the doors should close and remain closed unless the "door open" button is pressed. Since we do not want the passenger to keep doors open, the elevator can react to the "door open" button at most twice. As passengers leave the elevator, the "passenger present" detector is reset.

To make the specification complete, we have made some assumptions about the environment. First, assume that the user should be able to press request buttons at any point, so we leave the button pressing action nondeterministic. Second, to avoid leaving some request unfulfilled, the elevator runs in a pattern that it moves in one direction to satisfy all outstanding requests in route, until it reaches the last floor in that direction or until there are no more requests in that direction.

3 Model checker MOCHA

3.1 Reactive Modules

The input language that MOCHA uses for model description is that of Reactive Modules Language. Unlike simple state-transition graphs, reactive modules form a compositional model in which both states and transitions are structured. Reactive modules are built from atoms, and atoms are built from variables which are the elementary particles of systems.

In MOCHA, a system is considered to be discrete, deadlock-free, nondeterministic. The state of the system is described by a set of state variables: each system state corresponds to an assignment of values to the variables. So, in the elevator system, the state of the system is essentially the movement status and position of the elevator car and the status of the door. The behavior of the system consists in an initial round, which initializes the variables to their initial values, followed by an infinite sequence of update rounds, which assign new values to the variables, thus describing the evolution of the system’s state. Atoms and modules are used to specify the initial and update rounds for all the variables.

3.1.1 Variables

3.1.1.1 Variable Type

MOCHA provides the simplest data types such as Boolean, integer and natural; the keywords for these types are **`bool`**, **`int`**, and **`nat`**, respectively. Other common data types of MOCHA are ranges, enumerations, arrays, and bitfields.

For example, we declare the moving status of the elevator as enumerated type and the counter for the times of door opening as range type:

```
type movingType : {UP, DOWN, IDLE}
type counterType : (0..2)
```

Another more interesting data type is **`event`**. A variable that is used to represent an event can be declared of type **`event`**. Events are represented by toggling the value of Boolean variables. If `x` is an event variable, then we toggle it with the command `x!` to generate an event and we can test whether it has been toggled with the Boolean expression `x?` to count that event. These are only operations of `event` type. The only thing that matters is whether the value of an event variable has been toggled or not, we do not really care about what value is stored in `x`. This methodology makes it easier to toggle the variable and detect the toggling and also make the value of the variable irrelevant.

Since MOCHA does not support shared memory, the communication between different modules is accomplished by using `event` variables. The events that may happen in our elevator system are `opendoor` event, `closedoor` event, elevator `stop` event and elevator `go` event. They become communication channel between the controller and the elevator.

3.1.1.2 *Latched* and *updated* values

Reactive modules are to specify how variables change their values over the evolution of the system. In each evolution round, every variable has two values. The value of a variable `x` at the beginning of the round is called the *latched* value, represented by unprimed symbol `x`. The value of a variable at the end of the round is called *updated* value, represented by primed symbol `x'`. In an update round, the updated value of a variable may depend on the latched value of this variable, and on the latched value of other variables. Such a dependency between the values of variables within a single round is called an *await* dependency.

3.1.1.3 Variable Classification

Variables are declared in the beginning of a module and specified their usage in the beginning of the atom that needs to access them. In the module level, variables are partitioned into three sets: *private variables*, *interface variables* and *external variables*, according whether the variable is controlled by the module or by the environment. In the atom level, variables are partitioned into three types: *controlled variables*, *read variables*, and *awaited variables*, according whether the atom can update the variable's value or can only read its value. We will discuss in details in the next section.

3.1.2 Atoms

An atom is the basic unit of a module. It groups all related variables together and describes the initial condition and transition relation between them. An atom deals with three types of variables:

- ♦ *Controlled variables*. The variables for which the atom can specify the values in each round.

- *Read variables*. The variables whose *latched* value can be read by the atom to decide the next values of the controlled variables. The *updated* value of a *read* variable is not accessible to its atom.
- *Awaited variables*. The variables whose *updated* value can be read by an atom in order to decide the *updated* value of the controlled variables. The *latched* value of an *awaited* variable is not accessible to its atom.

In order to avoid inconsistent specifications, the *awai*t dependencies must be acyclic. To avoid circularity problems, MOCHA requires that no variable is controlled by more than one atom and no variable is both awaited and controlled.

The state of a reactive module changes in a sequence of rounds. The first round is called the *initial round*, and determines initial values for all variables. Each subsequent round is called and *update round*, and determines new values for all variables. Figure 3.1 shows the atoms which specify the requests of a floor caused by non-deterministic pressing of external button and internal button of our elevator system, as well as the updating of floor request on the change of button pressing. Specifically, the **`init`** action of `Buttons` is followed by the **`init`** action of `FloorReq;` in each **`update`** round, after the **`external`** variables have been updated, the **`update`** action of `Buttons` is followed by the **`update`** action of `FloorReq`. In this manner, all **`await`**ed values are available when they are needed during the execution and all nondeterminism in the completion of a round is caused by the nondeterminism of the environment.

```
atom Buttons
     controls inButton, exButton
     reads    arrive, inButton, exButton
     awaits   arrive
     init
         []true -> exButton' := false; inButton' := false
     update
         []arrive'  & ( exButton | inButton )-> exButton' := false;
         []~arrive' &  ~exButton -> exButton' := nondet
         ...
     endatom

atom FloorReqq
     controls floorRequest
     awaits exButton, inButton
     init
         []true -> floorRequest' := false
     update
         []true -> floorRequest' := (exButton' | inButton')
     endatm
```

Figure 3.1 Atom `Buttons` and `FloorReq`

The Figure 3.2 is the definition of atoms showing the communication in the elevator system using event methodology. The variables `stop,` `go` are declared as **`event`** type. The construct `stop?` quires the toggling of an event variable while the `stop!` toggles the value of an event. The atom `MoveControl`, which is part of the controller, is responsible to determine the next action of the elevator. It detects requests from reading the variable `floor` and the current `position` of the elevator and schedules a stop event to the elevator car in case that a request in the current floor should be satisfied. If there is no request in the current floor, the controller sends a `go` event to the elevator. The atom `elevcar`, which is part of the elevator, waits for the command from the controller by toggling the value of `stop` and `go`.

```
   atom MoveControl
        controls keepGoing, stop, go
        reads    stop, go, position, floor0, floor1, floor2,
        init
            []true -> keepGoing' := false;
        update
            [] position=f0 & floor0 -> stop! ; keepGoing' := false
            ...
            [] position=f0 & ~floor0 & ( floor1 | floor2 )
                                    -> go! ; keepGoing' := true
            ...
       endatom

    atom elevcar
        controls moving, door
        reads    stop, go
        awaits   stop, go
        init
            []true -> moving' := false ; door' := OPEN;
        update
            []stop? -> moving' := false; door' := OPEN
            ...
            []go?   -> moving' := true; door' := CLOSED
            ...
        endatom
```

Figure 3.2 Communications in MOCHA

You cannot feed just above atom as input to MOCHA since the smallest unit of input to the MOCHA parser is a module, not an atom. Next, we look at how to define a module.

3.1.3 Modules

A reactive module is a collection of atoms. It is a system, or a system component that interacts with other systems or other components. The variables appearing in a module may be controlled either by the module or by the environment and they are partitioned into atoms. In the initialization round and in every update round, the module and the environment take turns in the form of subrounds to initializes or update an atom of variables that they control.

A module description involves the following three classes of variables according whether or not their values can be observed by the environment. And the state of a reactive module is determined by the values of these three kinds of variables:

- *Private variable*. The variables that are only read and modified by some atom of the module and neither read nor modified by the environment.
- *Interface variable*. The variables that are read by both the module and the environment, but only modified by the module.
- *External variable*. The variables that are read by both the module and the environment, but only modified by the environment.

Therefore, from the above definition, a module is in charge of its *private* variables and *interface* variables, while the environment can observe the *interface* variable and *external* variables of one module. Given a module *P* consisting of a set $U = \{U_1, U_2, \dots U_i, \dots U_n\}$ of atoms, we refer *privP*, *extP*, *intfP* as the *private*, *external*, *interface* variables of module P respectively and *readU*$_i$, *ctrU*$_i$, *awaitU*$_i$ as the *read*, *controlled*, *awaited* variables of atom U_i respectively, the following relationships among different variables immediately follow from their definition:

$$privP \cap extP \cap intfP = \phi$$

$$\bigcup_{i=1}^{i=n} ctrU_i = intfP$$

$$ctrU_i \cap awaitU_i = \phi$$

We give a concrete example after we develop the `elevator` module to show the relationship between module and environment in term of various classes of variables.

Now, we are ready to develop modules for the elevator system. The Figure 3.3 is the `elevator` module which represents the elevator car of our system. It communicates with the controller to serve requests.

```
module elevator
    external  stop, go, opendoor, closedoor : event;
              elevnextdire : movingType
    interface elevdire :  movingType;
              moving : bool;
              position : posiType;
              door : doorStatusType;
              passengerPresent : bool;
              openDoorTimes : counterType
    atom elevcar
        controls moving, position, elevdire, door,
                 passengerPresent, openDoorTimes
        reads    moving, position, elevdire, door,
                 elevnextdire,  openDoorTimes,
                 go, stop, opendoor, closedoor
        awaits   stop,  go, opendoor, closedoor
        init
            []true -> moving' := false ; position' := f0; door' := OPEN;
            ...
        update
            []stop? -> moving' := false; door' := OPEN;
                       passengerPresent' := nondet
            []go? & (elevnextdire=UP)  -> elevdire' := elevnextdire;door' := CLOSED;
                                          moving' := true;
                                          position' := if( position = f0) then f1
                                                  else if (position = f1) then f2
                                                  else position fi fi            ...
            ...
        endatom
    endmodule
```

Figure 3.3 Module `elevator`

The various classes of module variables and their relationships in the `elevator` module are shown in Table 1:

Private variable	**Interface variable**	**External variable**
	`elevdire, moving, position, door, passengerPresent, openDoorTimes`	`stop, go, opendoor, closedoor, elevnextdire`
Controlled variable by module `elevdire, moving, position, door, passengerPresent, openDoorTimes`		
	Visible variable by environment `elevdire, moving, position, door, passengerPresent, openDoorTimes, stop, go, opendoor, closedoor, elevnextdire`	

Table 1. The relationship of variables in the module `Elevator`

As we have mentioned before, the state changes of a reactive module can be described as a sequence of rounds. While in each round, there are several subrounds: one for the external variables, and one per atom. When executing the above module `elevator`, in each round, first the **`external`** variables `stop, go, opendoor, closedoor,` and `elevnextdire` are assigned arbitrary value of the correct types and domain, and then the atoms are executed and the controlled variables `elevdire, moving, position, door` , etc get updated.

To take a closer look to the execution of a set of atoms *U*, we analyze the initialized trajectories of *U*. An initialized trajectory of *U* is the outcome of the execution. Figure 3.4 is a trajectory of module `floor` with atoms `Buttons` and `FloorReq`. `arrive` is the external variable of module `Floor`, while `exButton`, `inButton`, `floorRequest` are controlled variables in the atoms. There are three subrounds in the module `floor`: one for external variable `arrive`, one for the atom `Buttons`, and another for the atom `FloorReq`. The subround of updating the variables `exButton`, `inButton` awaits the subround of updating external variable `arrive`. And the subound of atom `FloorReq` awaits the subround of atom `Buttons`. And in the execution of `FloorReq`, the value of variable `floorRequest` depends on the value of primed `exButton`, `inButton`, which means that the value of `floorRequest` will be updated in the same round as the atom `Buttons`.

arrive	0		0		1		0		0		1		1		0		0		0		1	
exButton		0		1		0		0		0		0		0		1		1		1		0
inButton		0		0		0		1		1		0		0		1		1		1		0
floorRequest		0		1		0		1		1		0		0		1		1		1		0

Figure 3.4 Initialized trajectory of the atoms in module `Floor`

The trajectory is depicted graphically in the form of a *timing diagram* in Figure 3.5. The vertical dotted lines of the timing diagram represent boundaries between rounds. The diagram is interpreting the *await* dependency: the variables `exButton` and `inButton` changes their values, in each round, only after `arrive` have changed its value; the variable `floorRequest` changes its value only after either `exButton` or `inButton` changes its value.

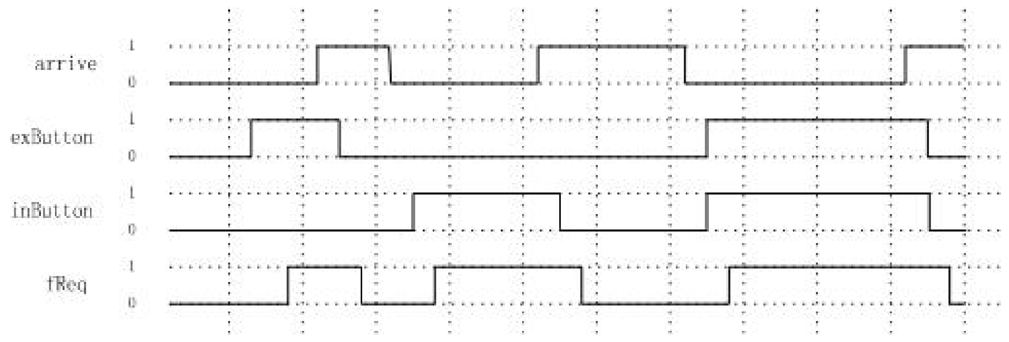


Figure 3.5 The timing diagram for the trajectory from above

MOCHA provides three mechanisms to create new module from pre-defined module by *variable hiding*, *variable renaming* and *parallel composition*. The renaming is useful for creating different instances of a module, and for avoiding name conflicts. In our elevator system, we define the module `floor and` create the first floor `floor_0 by renaming it:`

```
      floor_0 := floor[arrive,floorRequest,exButton,inButton :=
                                              arrive0, floor0, exBtn0, inBtn0]
```

The parallel composition `||` operation combines two modules into a single module whose behavior captures the interaction between the two original component modules. Two modules can be composed only if their variable declarations are mutually consistent, and if the combined await dependencies of two modules are not circular. The composition `P || Q` is asynchronous iff both `P` and `Q` are asynchronous, and `P || Q` is round-insensitive iff both `P` and `Q` are round-insensitive [5].

After defining all component modules for our elevator system, namely, `floor0, floor1, floor2, elevator, openDoorBtn` and `controller,` we have all components of the system. We create the entire elevator system by parallel composition. To construct module abstractions of degrees of detail, we hide some interface variables, so that only some status data that the user can view, such as `elevnextdire`, `arrive0`, etc...

```
      ElevSystem := hide elevnextdire, arrive0, arrive1, arrive2
                         in (floor_0 || floor_1 || floor_2 || elevator
                             || openDoorBtn || controller) endhide
```

The structure of modules can be graphically depicted using *block diagram* Figure 3.6. From the diagram, we can see that the relationship of *await* dependencies among all component modules is not circular. The complete elevator system *Reactive Modules* specification can be found in Appendix A. After building the system, we specify all the properties that the system should hold. Once we have both system specification and property specification, we verify them in MOCHA.

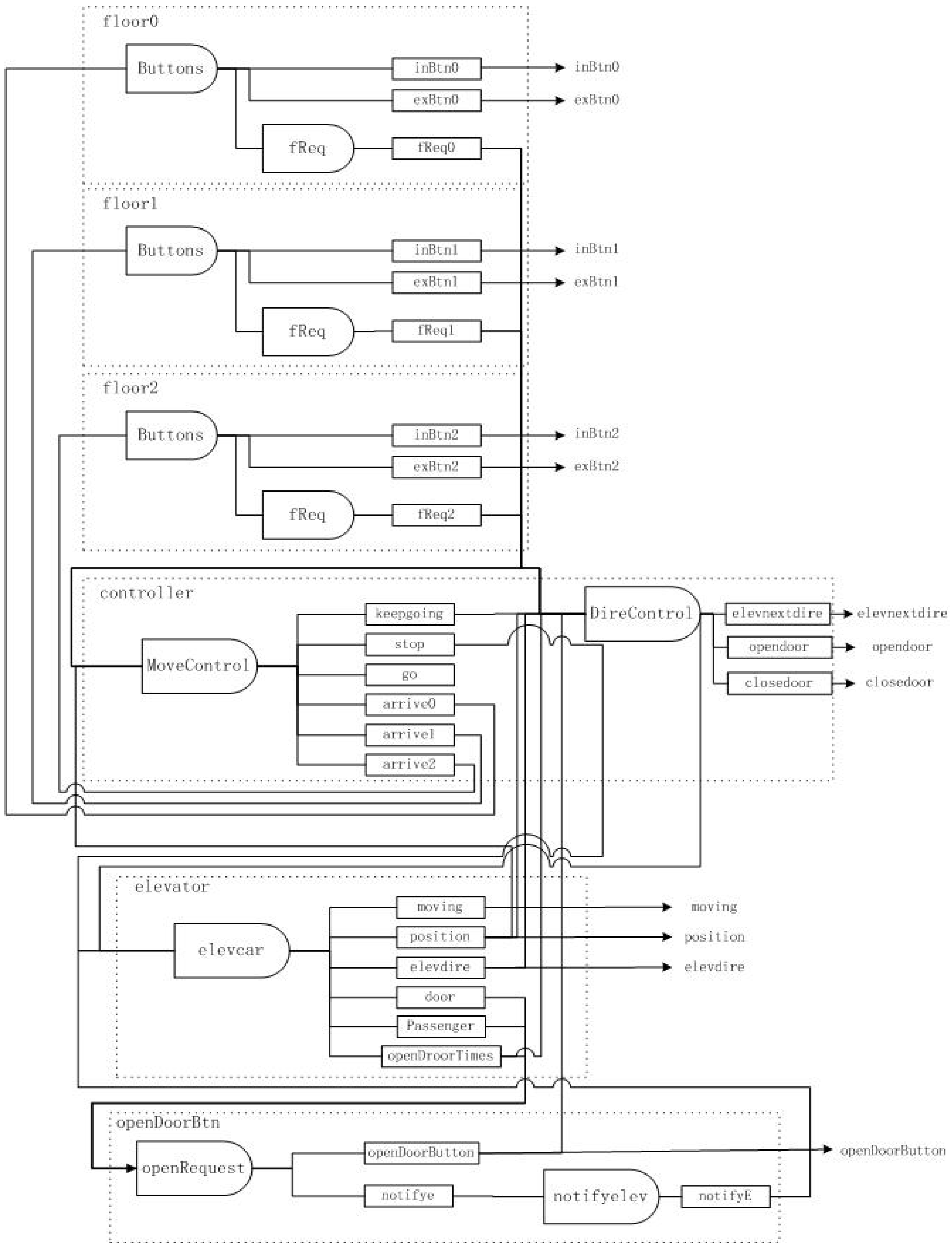


Figure 3.6 Block Diagrams for the elevator system modules

3.2 Property Specifications

MOCHA can check specification of a module in three different ways: invariants, alternating-time temporal logic, and refinement. Refinement checking is to check whether the given module is a refinement of the specification which is also given as a reactive module. Since in this case study we do not check when a module refines another module, we only introduce the first two specifications.

3.2.1 Invariants

An invariant is a Boolean predicate which is intended to hold in all reachable states of a module. The syntax for describing an invariant is the same as that for a Boolean expression.

Consider the elevator modules, an important property that should hold true throughout the execution is that "The elevator never moves with its door open". This is a very suitable property that we can describe as invariant specification:

```
inv "inv1" ~( moving & (door = OPEN))
```

3.2.2 Alternating Temporal Logic

As a generalization of the temporal logic CTL [4], Alternating Temporal Logic is interpreted over concurrent game structure and is designed for specifying requirements of open systems [5]. It captures compositions of open systems by interpreting them over game structure with multi-players, instead of just two-player games between the system and the environment. The players represent different components of the system and the environment. To give a general picture about ATL, we consider a set A of players and an ATL formula <<A>>φ. Let us consider dividing all players into a protagonist and antagonist. The game proceeds in an infinite sequence of rounds, i.e. one **`init`** round and infinite **`update`** rounds. To execute an update round, the protagonist chooses for every player in A an action. Then the antagonist chooses for every player not in A an action, and the system (game) is updated to a new state. In this way, the game produces a computation. The protagonist wins the game if the resulting computation satisfies φ. In the following sections, we will use the term "player" or "agent" to refer a module in the system.

In addition using all the same *path quantifiers* and *temporal operators* as CTL, ATL has two additional *path quantifiers*:

$$<< names>>, \qquad [[names]]$$

$<< names >> \varphi$ where φ is a path formula and $names \subseteq \sum \; of \;\; players$, means that the listed players *names* have a strategy to produce a computation that satisfying φ, no matter how the other unlisted players play the game, i.e. irrespective of how the players in $\sum \backslash name$ behave. Therefore, we can derive that CTL path quantifier **`A`** corresponds to ATL path quantifier $<< \phi >>$ and CTL path quantifier **`E`** corresponds to ATL path quantifier $<< \sum >>$.

$[[names]]\varphi$ where φ is a path formula and $names \subseteq \sum \; of \;\; players$, means that the players that are unlisted in the quantifier have a strategy to produce a computation satisfying φ, no matter how the other listed players play the game. We see that [[ ]] is the dual of $<< \;\; >>$. Therefore, we can derive that CTL path quantifier **`A`** corresponds to ATL path quantifier [[$\sum$]] and CTL path quantifier **`E`** corresponds to ATL path quantifier [[ϕ]].

3.2.2.1 CTL properties

We should note that ATL is more expressive than CTL, which means that all CTL formula can be verified by MOCHA without any modification.

Consider the elevator model, the property we described as an invariant in Section 3.2.1 also can be described as the following ATL formula:

```
            AG ( moving -> AF(door = CLOSED))
```

To make MOCHA recognize it, the syntax of specifying an ATL property is:

```
            atl "atl1"  A G ( moving => A F(door = CLOSED)),
```

where "`atl1`" is the name of this property, which is given by the user.

The property that "Requests to use the elevator are eventually serviced" can be specified as:

```
            atl  A G ( exBtn0 => A F ((door = OPEN) & (position = f0))).
```

The property that "Requests to be delivered to a particular floor are eventually serviced" can be specified as:

```
            atl A G ( inBtn0 => A F ((door = OPEN) & (position = f0)));
```

The above properties will hold vacuously if the external button and internal button are never pressed. So in order to make sure such case never happen and the system is doing some service, we should specify and verify another property that "Eventually, the external/internal buttons will be pressed" for vacuity checking.

```
                                  atl E F ( exBtn0 )

                                  atl E F ( inBtn0 )
```

The property that "The elevator should keep its door open until there is a request to use it" can be expressed as follows:

```
     A G ( (door = OPEN) => A ( (door = OPEN) W (floor0 | floor1 | floor2)));
```

The property that "When someone steps into the elevator, the door should close and remain closed unless the door open button is pressed" can be expressed as follows:

```
     A G ( (~moving & passengerPresent & (door = OPEN)) =>

                               A ((door = CLOSED) U openDoorButton ));
```

The above property will hold vacuously if there is always no passenger present. Therefore, we should specify and verify another property that "Eventually, the passenger will present" as follows:

```
            E F ( (~moving & passengerPresent & (door = OPEN))
```

The complete list of properties of the elevator system can be found in Appendix B.

3.2.2.2 Properties in ATL but not in CTL

Since ATL is a generalization of the temporal logic CTL, it is more expressive than CTL. We can take the advantage of two additional *path quantifiers* of ATL to specify some properties that cannot be expressed in CTL.

There are six agents (or modules) in our elevator system: `floor_0`, `floor_1`, `floor_2`, `elevator`, `openDoorBtn`, and `controller`. If we consider the evolution of an elevator system as a game, in each step of the game, these agents choose a move, and the combination of choices determines a transition from the current state to a successor state. By cooperation, some agents may have a strategy to ensure a certain property hold no matter the rest of agents in the system do. In some other cases, some agents do not have any strategy to force a certain property hold no matter how they behave. We are interested in verifying such CTL-unable properties.

The vacuity check for the property "Eventually, the buttons will be pressed" which was verified using CTL formula can be verified more precisely using ATL formulas:

```
atl <<floor_0>> F (exBtn0);
atl <<floor_0>> F (inBtn0);
```

This ATL formula means that the module `floor_0` has a strategy to make the external/internal button be pressed eventually no matter how the other modules behaves. This formula is stronger than CTL formula and it is not expressible in CTL.

The controllability of a system whose safe states are given by the state predicate φ is impossible expressed in CTL. In ATL, it can be specified by $<< control\_agent >> \mathrm{G}\ \varphi$. This asserts that the *control_agent* has a strategy to ensure the satisfaction of φ. In our elevator system, the property that "The `floor_1` has the discretion to make the external button not be pressed, no other module can force it to do otherwise" states the controllability of the `floor_1` which represents the second floor over the external button, and it can be specified as the following ATL formula:

```
atl A G ( ~exBtn1 => <<floor_1>> G ~exBtn1).
```

It is often useful to express an ATL formula in a dual form. As we mentioned in the beginning of this section, the path quantifier [[ ]] is the dual of << >>. While ATL formula <<A>> φ where A is a set of agents intuitively means that the agents in A can cooperate to make φ true, the dual formula [[A]] φ states that the agents in A cannot cooperate to make φ false [5]. Let us consider the second floor in our elevator system again. The external button `exBtn1` is controlled by the `floor_1` module; only the `floor_1` module has strategies to produce a trajectory to enforce the button `exBtn1` turn on. Other agents in the system such as `controller` or `elevator` cannot make the button turn on. The dual form of this property can be stated as "whenever the request button is off, all agents together except floor_1 does not have a strategy to produce a trajectory to force the external button be pressed" which can be specified as follow:

```
atl A G ( ~exBtn1 =>
     [[controller, elevator, floor_0, floor_2, openDoorBtn ]] G ~exBtn1);
```

The ATL is intended for compositional reasoning. It makes the property which holds in the a component of the system consistently hold in the composite system. More specifically, if A satisfies the ATL formula $<< A >><> \varphi$, then the composite system which contains A also satisfies this formula. Consider the property that "The requests on the third floor are eventually serviced" and its CTL formula is:

```
A G ( floor2 => A F ( (door = OPEN) & (position = f2)));
```

This CTL formula asserts that it is for *all* modules in the elevator system to cooperate so that the request on the third floor is eventually serviced, i.e. the elevator stops at the third floor and the door is open. The reason that the composite system `ElevSystem` satisfies this property is because that the two components, namely `controller` and `elevator`, satisfy this property. So, we can give the genuine ATL formula for this property:

```
<< >> G ((position=f0) & floor2 =>
     <<controller, elevator>> F ((position = f2)&(door=OPEN)));
```

This ATL formula states that whenever the elevator is currently in the first floor and the second floor is required, the controller and the elevator can cooperate so that the elevator will eventually be at the second floor.

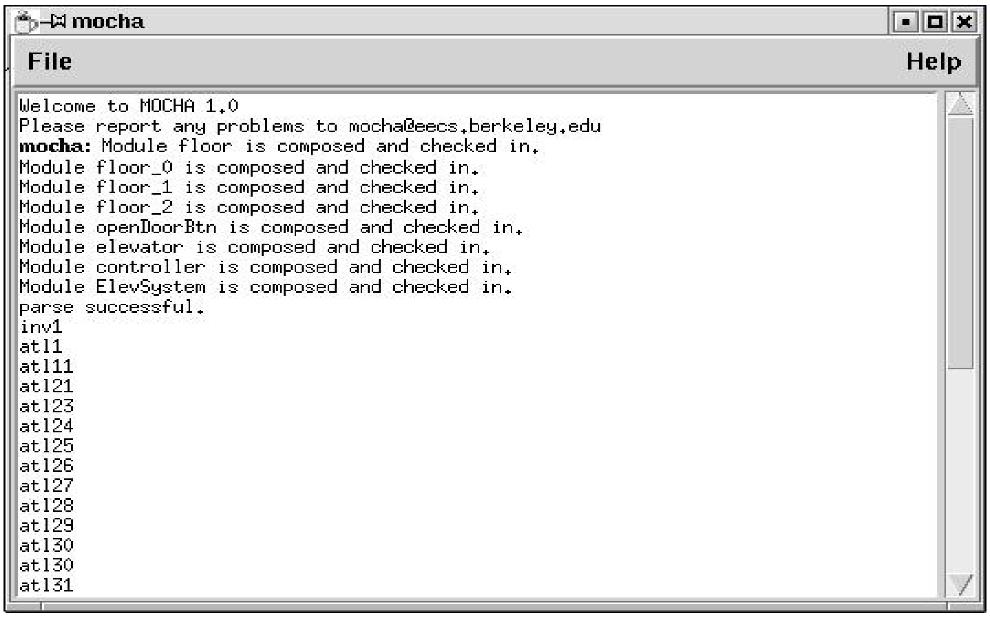
The complete list of ATL properties can be found in Appendix B.

3. 3 Simulation

MOCHA allows the user to execute any module in three modes: manual, random, and game, via a Tk-based GUI for interacting with the tool and viewing the execution trace.

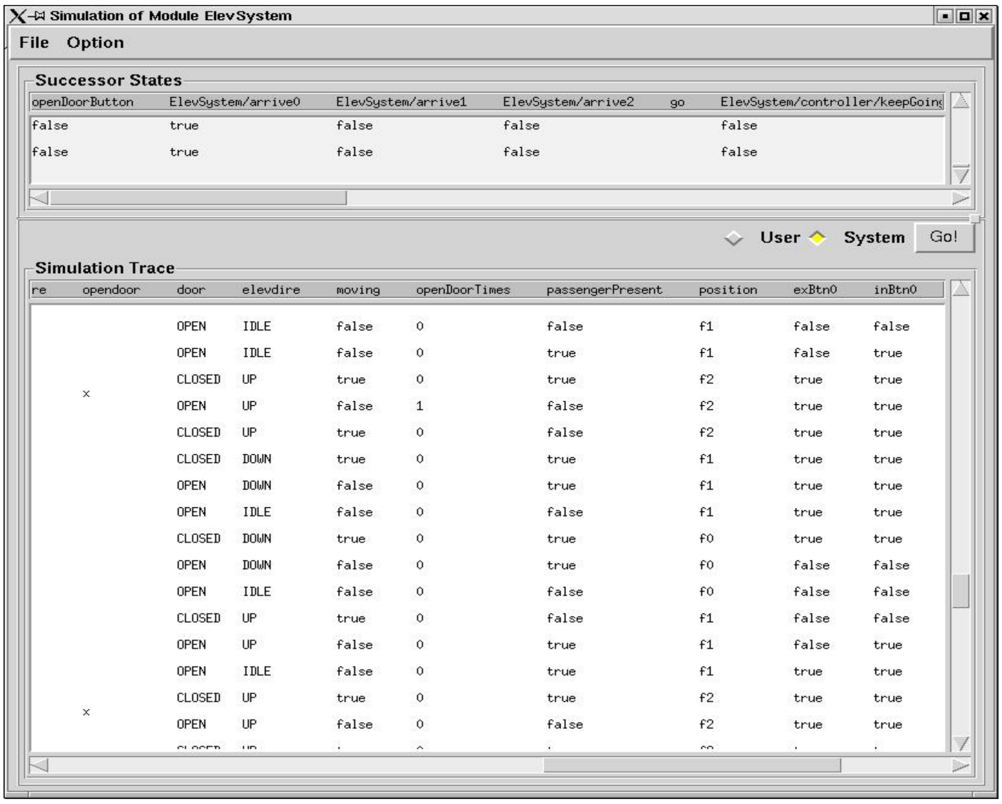
- Manual execution. The GUI displays all possible initial states and the user can select any one of them to execute.
- Random execution. User can specify the number of rounds that it wants to execute the module for.
- Game execution. The user plays a game against the computer. The user controls the update of a subset of the set of atoms of the module being simulated. In every round, the user chooses to update the variables of the atoms he controls and the system updates the rest of the atoms randomly. This is a way of performing guided execution.

The module definitions should be entered into a file named with the suffix `.rm`, in our case, `elev.rm`. The properties specified in invariants and ATL formulas can be stored in a file named with suffix `.spec`. Before simulation and verification, one should load these files into MOCHA. The load of the elevator system `elev.rm` module and the specification is shown in Figure 3.7. We can see that MOCHA treats a single module as a system. The composite system `ElevSystem` is also a module. MOCHA allowed user to simulate any module as user needs. For example, we can just choose to simulate the module `elevator`. In this case, during the simulation, the external variables such as `opendoor`, `stop`, etc, which are controlled by the module `controller`, are arbitrarily assigned a value by the environment. Since we are interested in our entire elevator system, a random simulation execution of `ElevSystem` module is shown in Figure 3.8.



```
Welcome to MOCHA 1.0
Please report any problems to mocha@eecs.berkeley.edu
mocha: Module floor is composed and checked in.
Module floor_0 is composed and checked in.
Module floor_1 is composed and checked in.
Module floor_2 is composed and checked in.
Module openDoorBtn is composed and checked in.
Module elevator is composed and checked in.
Module controller is composed and checked in.
Module ElevSystem is composed and checked in.
parse successful.
inv1
atl1
atl11
atl21
atl23
atl24
atl25
atl26
atl27
atl28
atl29
atl30
atl30
atl31
```

Figure 3.7 Read module and specification



Successor States

openDoorButton	ElevSystem/arrive0	ElevSystem/arrive1	ElevSystem/arrive2	go	ElevSystem/controller/keepGoing
false	true	false	false		false
false	true	false	false		false

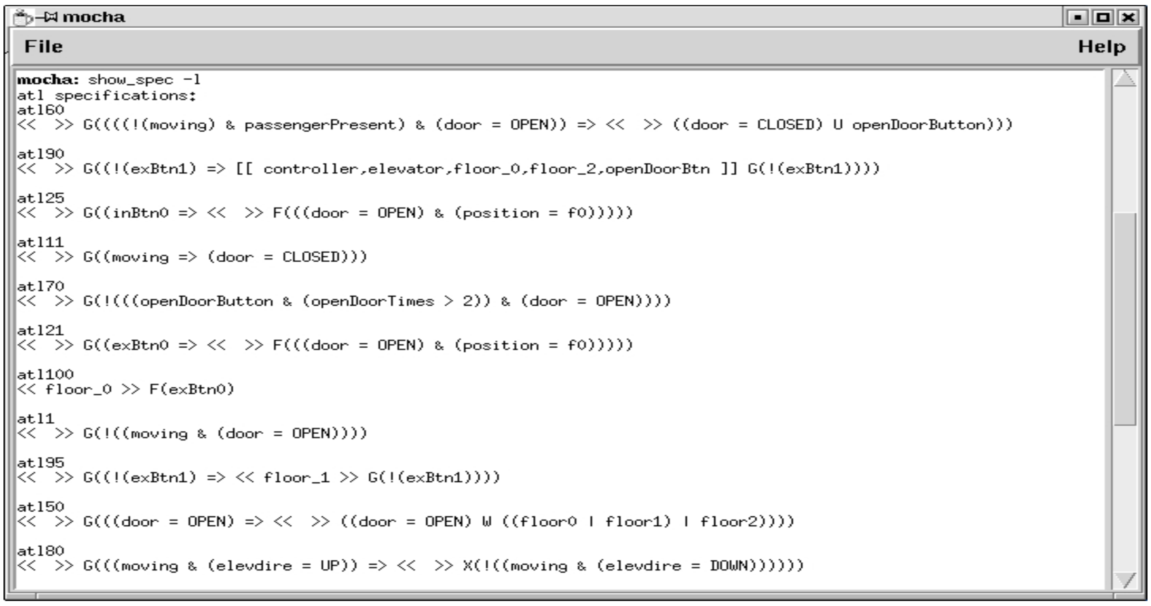
Simulation Trace

re	opendoor	door	elevdire	moving	openDoorTimes	passengerPresent	position	exBtn0	inBtn0
		OPEN	IDLE	false	0	false	f1	false	false
		OPEN	IDLE	false	0	true	f1	false	true
		CLOSED	UP	true	0	true	f2	true	true
	x	OPEN	UP	false	1	false	f2	true	true
		CLOSED	UP	true	0	false	f2	true	true
		CLOSED	DOWN	true	0	true	f1	true	true
		OPEN	DOWN	false	0	true	f1	true	true
		OPEN	IDLE	false	0	false	f1	true	true
		CLOSED	DOWN	true	0	true	f0	true	true
		OPEN	DOWN	false	0	true	f0	false	false
		OPEN	IDLE	false	0	false	f0	false	false
		CLOSED	UP	true	0	false	f1	false	false
		OPEN	UP	false	0	true	f1	false	true
		OPEN	IDLE	false	0	true	f1	true	true
		CLOSED	UP	true	0	true	f2	true	true
	x	OPEN	UP	false	0	false	f2	true	true

Figure 3.8 Simulation of Module `ElevSystem`

3.4 Verification

Mocha translates CTL-formed formulas into ATL form. Figure 3.9 shows the specifications that have been loaded and translated.

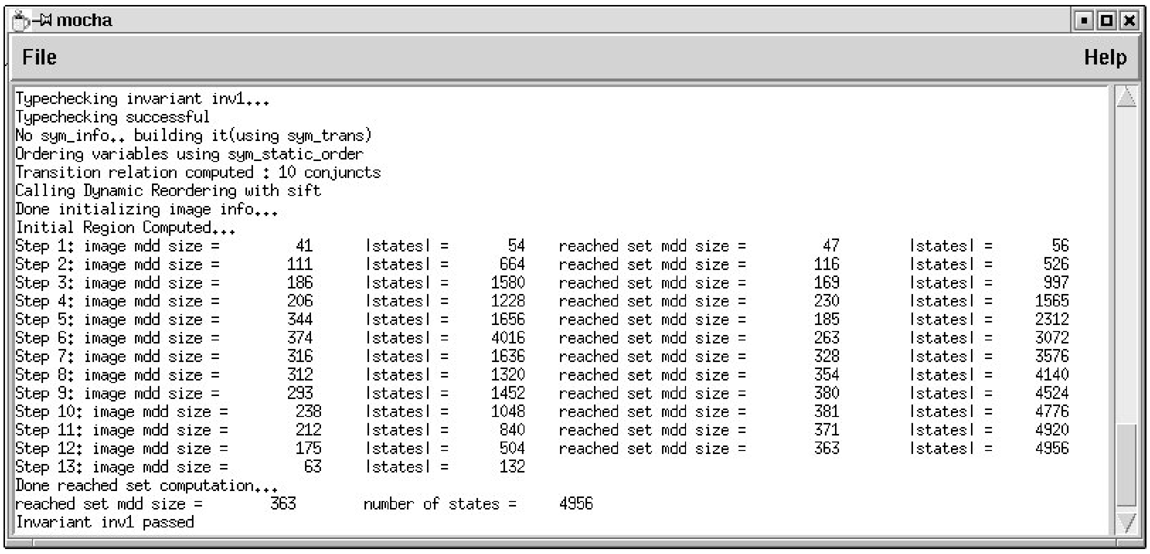


```
mocha: show_spec -l
atl specifications:
atl60
<<  >> G((((!(moving) & passengerPresent) & (door = OPEN)) => <<  >> ((door = CLOSED) U openDoorButton)))

atl90
<<  >> G((!(exBtn1) => [[ controller,elevator,floor_0,floor_2,openDoorBtn ]] G(!(exBtn1))))

atl25
<<  >> G((inBtn0 => <<  >> F(((door = OPEN) & (position = f0)))))

atl11
<<  >> G((moving => (door = CLOSED)))

atl70
<<  >> G(!(((openDoorButton & (openDoorTimes > 2)) & (door = OPEN))))

atl21
<<  >> G((exBtn0 => <<  >> F(((door = OPEN) & (position = f0)))))

atl100
<< floor_0 >> F(exBtn0)

atl1
<<  >> G(!((moving & (door = OPEN))))

atl95
<<  >> G((!(exBtn1) => << floor_1 >> G(!(exBtn1))))

atl50
<<  >> G(((door = OPEN) => <<  >> ((door = OPEN) W ((floor0 | floor1) | floor2))))

atl80
<<  >> G(((moving & (elevdire = UP)) => <<  >> X(!((moving & (elevdire = DOWN))))))
```

Figure 3.9 specifications loaded and translated

3.4.1 Invariant Checking

We check the property that “The elevator never moves with its door open” using invariant checking, which is shown in Figure 3.10. We can see Mocha invariant checker is using symbolic checking. It produces a *Binary decision diagrams* (BDDs) [15] representation of the transition relation and of the set of initial states. The checking result states that the invariant is passed. The time that takes to verify this property is one seconds.



```
Typechecking invariant inv1...
Typechecking successful
No sym_info.. building it(using sym_trans)
Ordering variables using sym_static_order
Transition relation computed : 10 conjuncts
Calling Dynamic Reordering with sift
Done initializing image info...
Initial Region Computed...
Step 1: image mdd size =        41      |states| =      54    reached set mdd size =        47        |states| =      56
Step 2: image mdd size =       111      |states| =     664    reached set mdd size =       116        |states| =     526
Step 3: image mdd size =       186      |states| =    1580    reached set mdd size =       169        |states| =     997
Step 4: image mdd size =       206      |states| =    1228    reached set mdd size =       230        |states| =    1565
Step 5: image mdd size =       344      |states| =    1656    reached set mdd size =       185        |states| =    2312
Step 6: image mdd size =       374      |states| =    4016    reached set mdd size =       263        |states| =    3072
Step 7: image mdd size =       316      |states| =    1636    reached set mdd size =       328        |states| =    3576
Step 8: image mdd size =       312      |states| =    1320    reached set mdd size =       354        |states| =    4140
Step 9: image mdd size =       293      |states| =    1452    reached set mdd size =       380        |states| =    4524
Step 10: image mdd size =       238     |states| =    1048    reached set mdd size =       381        |states| =    4776
Step 11: image mdd size =       212     |states| =     840    reached set mdd size =       371        |states| =    4920
Step 12: image mdd size =       175     |states| =     504    reached set mdd size =       363        |states| =    4956
Step 13: image mdd size =        63     |states| =     132
Done reached set computation...
reached set mdd size =       363       number of states =     4956
Invariant inv1 passed
```

Figure 3.10 invariant checking and the results

3.4.2 CTL and ATL checking

The algorithm of Mocha ATL checking is generalized from the symbolic model-checking procedure for CTL. For a set A of agents, to check an ATL formula $<< A >> <> p$ is to compute the least fixed point of the set of Agents that contains all states satisfying p. Mocha computes this by an iterative symbolic procedure so that the time complexity of ATL model checking is still linear in the size of state space and the formula, although ATL is more expressive than CTL.

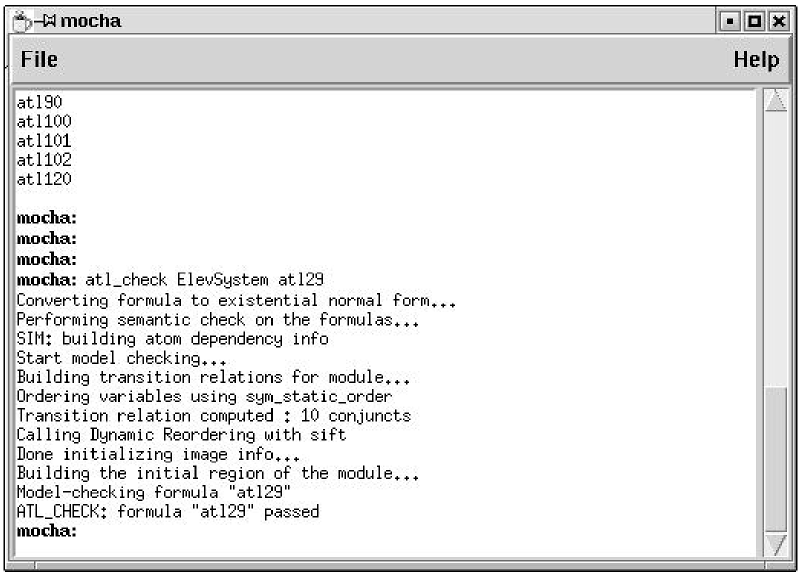
To illustrate the above MOCHA ATL checker property, we compare the checking results of both CTL formula and ATL formula for the property “Requests to use the elevator are eventually serviced”. The result of checking CTL formula

```
        “A G ( floor2 => A F ( (door = OPEN) & (position = f2)))”
```

is shown in Figure 3.11. The result of checking ATL formula

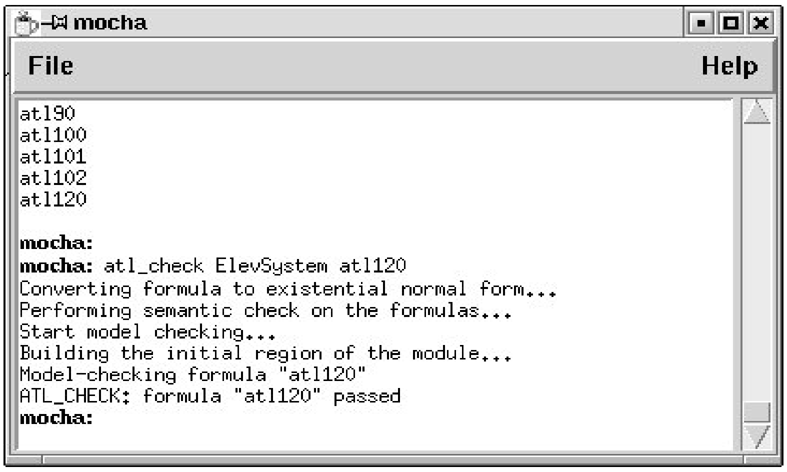
```
        “<<>>G (((position=f0) & floor2) =>
           <<controller,elevator>> F ((position = f2)&(door=OPEN)))”
```

is shown in Figure 3.12.



```
atl90
atl100
atl101
atl102
atl120

mocha:
mocha:
mocha:
mocha: atl_check ElevSystem atl29
Converting formula to existential normal form...
Performing semantic check on the formulas...
SIM: building atom dependency info
Start model checking...
Building transition relations for module...
Ordering variables using sym_static_order
Transition relation computed : 10 conjuncts
Calling Dynamic Reordering with sift
Done initializing image info...
Building the initial region of the module...
Model-checking formula "atl29"
ATL_CHECK: formula "atl29" passed
mocha:
```

Figure 3.11 CTL formula checking and the results



```
atl90
atl100
atl101
atl102
atl120

mocha:
mocha: atl_check ElevSystem atl120
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl120"
ATL_CHECK: formula "atl120" passed
mocha:
```

Figure 3.12 ATL formula checking and the results

Both formulas are verified in one seconds. Therefore, the added expressiveness of ATL over CTL does not add extra cost to verification.

The verification results of all the properties can be found in Appendix C.

3.5 Problems

There are some problems we encountered while verifying the properties using MOCHA. Some problems might be caused by the limitation of the model checker, while other problems might be caused by the imperfection of the elevator system specification. We list these problems below:

♦ We originally intended to use jMOCHA as the environment to explore Reactive Modules and ATL model checking. It provides many improvements over cMOCHA and is written in Java. However, its execution is noticeably slower than cMOCHA, especially when the system state space gets larger. Another problem we found is that when the array type data is used in the module, jMOCHA behaves unexpectedly.

- When verifying some ATL properties, cMOCHA is crashed with segmentation fault. We observed that those ATL formulas have a common feature that they all refer some variables of range type. For instance, the ATL formula:

  ```
  A G ~(openDoorButton & (openDoorTimes > 2) & (door = OPEN))
  ```

  states that the elevator react "open door" at most twice. The variable `openDoorTimes` is ranging in (0..2).

- Some ATL properties are failed when verified by MOCHA. However, MOCHA does not provide counterexample and back trace methodologies, it is difficult to debug and optimize our model.

4 Related works and comparisons

Hardware and software systems will inevitably grow in scale and functionality. With the increase in complexity, the likelihood of subtle errors is much greater. Model checking is one way to enable developers to construct systems that operate reliably despite this complexity. Currently, model checkers are widely used in industry and in academia. SMV and SPIN are among the most popular ones in model checking. SMV (Symbolic model verifier) [6] is a model-checker for checking that finite-state systems satisfy specifications given in CTL. It uses the OBDD-based symbolic model checking algorithm. It was intended for hardware systems. Its modeling language is SMV, which hierarchically describes finite-state machines at any level of detail, both synchronous and asynchronous. SPIN [7] is Bell Labs' tool implemented in C and intended for checking correctness of process interactions. The underlying language is PROMELA (Protocol or Process Meta Language), which is a C-like non-deterministic language based on Hoare's CSP (communicating sequential processes). SPIN contains a symbolic simulator, model checker, and verification with LTL (linear temporal logic) and invariants. SPIN uses partial order reduction to reduce state space. Based on the experience of modeling and verifying a similar elevator system in both SMV and SPIN, we draw a summarization map of these two types of model checkers.

i. The modeling language

- SMV

The language in SMV for describing the model is a simple parallel assignment. The description of a complex finite-state system can be decomposed into *modules* with one module called *main*.

Modules can have parameters, which may be state components, expressions, or other *modules*.

Modules can be composed either synchronously or using interleaving. If the keyword *process* precedes an instance of a module, interleaving is used; other wise synchronous composition is assumed. The interleaving model allows for a particularly efficient representation of the transition relation: the set of states reachable by one step of the program is the union of the sets reachable by each individual process. By this means, the reachability graph is not needed, while sometimes complexity in representing the set of states reachable in n steps is increased up to s^n possibilities. Fairness constraint is used for specifying that each process has to execute infinitely often.

In modules, each variable is a state machine described by *init* and *next*. And there's no loop construct in SMV.

The state transitions in a model may be either deterministic or nondeterministic. Nondeterminism can reflect actual environment of the system being modeled, or it can be used to describe a more abstract model where certain details are hidden. When modeling an elevator system, the request from inside the elevator and at the floors can be modeled as nondeterministic environment.

♦ SPIN

Promela(Process Meta Language) allows for the dynamic creation of concurrent processes. Promela programs consist of *processes*, message *channels*, and *variables*. The state of a variable or channel can only be changed or inspected by processes. The behavior of a process is defined in a *proctype* declaration.

Process can receive parameters of all basic data types and message channels. But data arrays and process types are not allowed.

A *proctype* definition only declares process behavior, it doesn't execute it. Initially, just one process will be executed: a process of type *init*. Independent processes are composed asynchronously by the operation of *run* processes defined by *proctype*. Except for some basic constructs, there are also do loops, assertions and atomic steps in Promela.

Based on Dijkstra's guarded command language, every statement in Promela program is guarded by a condition and blocks until condition becomes true. The executability of statement is the basic means of synchronization.

Processes can communicate using channels and global variables.

Channels and global variable define the environment in which the processes run. Promela doesn't provide nondeterministic assignment of variables. So, when modeling an elevator system, we need to model the nondeterministic environment which can be executed by nondeterministic choice of statements in processes.

ii. Simulator

♦ SMV doesn't provide a separate simulator. If the result of model-checking is false, there is a counterexample to prove it. If the result is true, no extra information is given.

♦ SPIN provides random simulations of the system's execution. It can be used as a simulator not only when a counterexample is generated but also for random and manually guided simulation.

iii. Verifier

♦ SMV can verify properties expressed in CTL formula. These CTL properties are given as *SPEC* statements in the program.

♦ SPIN can generate an exhaustive state space searching program for a model to check deadlock, unreachable code, and violated assertions. Assertion is a way provided to check validity of system invariant.

SPIN can also check properties specified as an LTL formula by LTL claim. The claim is translated into NEVER claim and stored either in *.ltl* file or at the end of model file. All variables in LTL claims have to be global, and only one LTL property can be verified at a time.

Due to the implicit use of universal quantification over the set of computations, LTL cannot express existential, or possibility, properties.

iv. Algorithms

- ♦ SMV uses the OBDD-based symbolic model checking algorithm. OBDDs represent sets of states and transitions in Kripke structures. And the algorithm is based on the manipulation of Boolean formulas including fixpoints calculation and iterative techniques.
- ♦ Spin represents the system as a finite state machine and visits each reachable state explicitly (using Nested DFS). performs on-the-fly computation; uses partial order reduction; efficient memory usage(state compression, Bit-state hashing).

  Spin performs on-the-fly computation: System is the asynchronous composition of processes and the global transition is never build. For each state, the successor states are enumerated using the transition relation of each process.

  Spin also tries to realize efficient memory usage. Techniques applied includes: Hash table, partial order reduction, minimized automata reduction, as well as working on non-binary variables (MDD).

Given that none of formal method is likely to be suitable for describing and analyzing every aspect of a complex system, a practical approach is to use different methods in combination. Comparing with those tools mentioned, MOCHA is interesting to us because it integrates diverse methods and tools for modeling heterogeneous, reactive systems, particularly embedded systems. It can model heterogeneous systems such as those including hardware and software, analog and digital, ad electrical and mechanical devices. It can also model systems that are complex in the sense that they mix widely different operations, such as signal processing, feedback control, sequential decision making, and user interfaces.

Comparing with SMV and SPIN, the following proprieties are remarkable in MOCHA:

- ♦ The system description language is based on reactive modules. The language REACTIVE MODULES is a modeling and analysis language for heterogeneous concurrent systems with synchronous and asynchronous components. The efficiency of most verification tools often depends on the specific synchrony assumption supported by the underlying model [3]. SMV, as a language intended for hardware description, assumes synchronous progress and BDD-based model checking is successful in this domain. Many protocol description languages, like Promela, assume asynchronous interleaving, and the most effective verification strategy is explicit on-the-fly search with reduction techniques based on partial orders and symmetries. While both synchrony and asynchrony can be forced, in one way or another, into most currency models, this often comes at the cost of inefficiencies in verification. For example, the use of interleaving model in SMV through the keyword *process* increases the number of transitions exponentially. And the introduction of synchronization points into asynchronous models restricts the applicability of efficient search methods in verification [12]. By contrast, the reactive modules provide a uniform framework which allows separating complexity of verification methods from model dependence.
- ♦ MOCHA provides support for checking invariants on finite-state modules by implementing both symbolic and enumerative state-exploration algorithms. In cMocha, the enumerative state-exploration which is used primarily by the simulator doesn't perform any optimizations, while the symbolic state-exploration use BDDs to represent the transition relation and the set of reached

states of a reactive module. In jMocha, various features and optimizations of enumerative search engine is implemented [13]. And the symbolic checker in jMOCHA works on a multi-valued decision diagram (MDD).

- ♦ Reactive modules, which permit the modular and hierarchical description of heterogeneous systems, provides support for modular proof principles, such as assume-guarantee reasoning and hierarchical verification, based on built-in abstraction operators such as **next** [3]. The technique of dividing the verification task at hand into simpler tasks is one way for combating state-explosion problem. In mocha a compositional methodology for refinement checking is implements based on assume-guarantee rules and made use of the infrastructure of reactive modules [14].

- ♦ MOCHA can verify properties specified in ATL formula. Temporal logic comes in two varieties: linear-time temporal logic assumes implicit universal quantification over all paths that are generated by the execution of a system; branching-time temporal logic allows explicit existential and universal quantification over all paths. While alternating-time temporal logic (ATL) provides a more general variety of temporal logic, which offers selective quantification over those paths that are possible out comes of games, such as the game in which the system and the environment alternate moves[5].

  The logics LTL and CTL are interpreted over Kripke structure, which offer a natural model for the computations of a closed system, whose behavior is completely determined by the state of the system. While reactive modules, as a concurrent game structure, require to be viewed as open systems, which interact with environment and whose behavior depend on the state of the system as well as the behavior of the environment. ATL is better suitable for compositional reasoning and expressing properties of open systems. For instance, if a component A satisfies the CTL formula $EG\varphi$, we can't conclude that a composite system that contains A as a component also satisfies $EG\varphi$. On the other hand, if A satisfies the ATL formula $<< A >> G\varphi$, then so does the composite system..

Given the features above, it would be worth further research to investigate how MOCHA could provides us with a wealth of experience in tool integration.

5 Conclusions

MOCHA is an interactive verification environment for the modular verification of heterogeneous. It models system with heterogeneous modeling framework of reactive modules, specifies properties with the module-level specification language of ATL, in which both cooperative and adversarial relationships between modules can be expressed. MOCHA also provides automatic refinement checking. The MOCHA toolkit includes the following functionalities: simulation, enumerative and symbolic invariant checking and error-trace generation, compositional refinement checking, ATL model checking, reachability analysis of real-time systems. As an integrated model checking environment, MOCHA is a good choice to do some case study on.

From the experience of building an elevator system with MOCHA as well as SPIN and SMV, we found that as a model checker based on reactive modules, MOCHA has some advantages in scalability which is realized by parallel composition of modules, providing both enumerative and symbolic invariant checking, supporting game execution, and verifying properties specified in ATL formula. As a tool integrating so many interesting features in modeling checking, MOCHA is worthy of learning and analyzing.

From the case study of an elevator system, we also find some limitations in MOCHA. First, RML is not as convenient of building large complex software or hardware systems for programmers as programming language which is more similar to natural language. Second, the way in which variables are updated in a single round in MOCHA is *nonatomic synchrony* [5]. To make sure that the possibility of nonterminating computation within a single is avoided, a restrictive strategy is applied by statically linearizing the partial order on the atoms. It is useful, while at the same time requires user to spend more time on figuring out relationship between modules as well as atoms in detail. Finally, the ATL property checker still cannot provide the counter example execution, which makes debug extremely hard.

References

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

[2] L. de Alfora R. alur R. Grosu T. Henzinger M. Kang. *MOCHA: exploiting Modularity in Model Checking.*

[3] R. Alur and T.A. Henzinger. *Reactive Modules.* In Proceeding of the 11th Annual Symposium on Logic in Computer Science, pages 207-218. IEEE Computer Society Press, 1996.

[4] E. M. Clarke and E. A. Emerson. *Design and synthesis of synchronization skeletons using branching-time temporal logic*. In Workshop one Logic of Programs, Lecture Notes in Computer Science 131, pages 2-71. Springer-Verlag, 1981.

[5] R. Alur, T.A. Henzinger, and O. Kupferman. *Alternating-time temporal logic.* In Proceedings of the 38th Annual Symposium on Foundations of Computer Science, pags 100-109. IEEE Computer Society Press, 1997.

[6] K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Technical Report CMU--CS--92--131, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1992. PhD Thesis...

[7] Gerard J. Holzmann. *The Model Checker Spin*,IEEE Trans. on Software Engineering,Vol. 23, No. 5, May 1997, pp. 279-295.

[8] http://www.dcs.ed.ac.uk/home/cwb

[9] Rajeev Alur, Thomas A. Henzinger, Pei-Hsin Ho. *Automatic symbolic verification of embedded systems.* IEEE Real-Time Systems Symposium, 1996

[10] C. Daws, S. Yovine .*Two examples of verification of multirate timed automata with Kronos*, Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95), Pisa, Italy (1995)

[11] Edmund M.Clarke, Jr., Orna Grumberg, and Doron A.Peled. *Model Checking*

[12] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1985

[13] L.De Alfaro, R.Alur, *MOCHA: Exploiting Modularity in Model Checking*, 2000

[14] Thomas A. Henzinger Shaz Qadeer, Sriram K. Rajamani, *You Assume, We Guarantee: Methodology and Case Studies*, Proceedings of the 10th International conference on Computer-aided Verification, Lecture Notes in Computer Science 1427, Springer-Verlag, 1998, pp.440-451

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

Appendix A: Elevator System Modules

```
/*********************************************************************

        A REACTIVE MODULES of AN THREE-LEVEL ELEVATOR CONTROL SYSTEM

                            Yuan Gan, Lin Mei
                                09-01-2004

*********************************************************************/

-- the moving direction of the elevator car
type movingType : {UP, DOWN, IDLE}

-- the status of the elevator car
type elevStatusType : {STOP, GO}

-- the states of door
type doorStatusType : {OPEN, CLOSED}

-- floor number
type posiType : {f0, f1, f2}

-- the counter counting the times of door opening
type counterType : (0..2)

module floor
-- represents one floor of the building which the elevator serves. There are an
-- external and an internal buttons in each floor which issue the requests to
-- the elevator. When a request is satisfied, the controller issues an arrive
-- event.

    external  arrive : bool
    interface floorRequest : bool;
              exButton : bool;
              inButton : bool

    atom Buttons
        controls inButton, exButton
        reads    arrive, inButton, exButton
        awaits   arrive

        init
            []true -> exButton' := false; inButton' := false
        update
            -- if the request is served, reset the request to be false
            []arrive'  & ( exButton | inButton )-> exButton' := false;
                                                   inButton' := false
            -- otherwise, the request occurs nondeterministcally
            []~arrive' &  ~exButton -> exButton' := nondet
            []~arrive' &  ~inButton -> inButton' := nondet
            []default ->  exButton' := exButton; inButton' := inButton
        endatom

    atom FloorReqq
        controls floorRequest
        awaits exButton, inButton
        init
            []true -> floorRequest' := false
        update
            []true -> floorRequest' := (exButton' | inButton')
        endatom
endmodule


-- There are three floors in the building, we easily create the three floor
-- modules by variable renaming.

floor_0 := floor[arrive,  floorRequest, exButton, inButton :=
                                            arrive0, floor0, exBtn0, inBtn0]
floor_1 := floor[arrive,  floorRequest, exButton, inButton :=
                                            arrive1, floor1, exBtn1, inBtn1]
floor_2 := floor[arrive,  floorRequest, exButton, inButton :=
                                            arrive2, floor2, exBtn2, inBtn2]

module openDoorBtn
--represents the “open door” button inside the elevator.
    external   door : doorStatusType; openDoorTimes : counterType;
               passengerPresent : bool
    interface  openDoorButton : bool

    atom Openrequest
        controls openDoorButton
        reads    door, openDoorTimes, passengerPresent

        init
            []true -> openDoorButton' := false
        update
            --if there is passenger inside the elevator, opendoor button
            --can be pressed nondeterminstically.
            []openDoorTimes < 2 & passengerPresent -> openDoorButton' := nondet
            []default -> openDoorButton' := false
        endatom
endmodule


module elevator
--represents the elevator car. It communicates with the controller to get
--the next movement data for the elevator moving direction and the statue of the
--door.

    external  stop, go, opendoor, closedoor : event;
              elevnextdire : movingType
    interface elevdire :  movingType;
              moving : bool;
              position : posiType;
              door : doorStatusType;
                     passengerPresent : bool;
                     openDoorTimes : counterType

    atom elevcar
        controls moving, position, elevdire, door,
                 passengerPresent, openDoorTimes
        reads    moving, position, elevdire, door,
                 elevnextdire,  openDoorTimes,
                 go, stop, opendoor, closedoor
        awaits   stop,  go, opendoor, closedoor

        init
            []true -> moving' := false ; position' := f0;
                      elevdire' := IDLE; door' := OPEN;
                      passengerPresent' := false
        update
            -- stop command from the controller! Stop moving and open the door
            -- and passenger can either leave the elevator or stay in it.
            []stop? -> moving' := false;
                       door' := OPEN;
                       openDoorTimes' := 0;
                       passengerPresent' := nondet

            -- go command from the controller! moves according the next direction
            -- issued by the controller.
            []go? & (elevnextdire=UP)  -> elevdire' := elevnextdire;
                                          door' := CLOSED;
                                          moving' := true;
                                          openDoorTimes' := 0;
                                          position' := if( position = f0) then f1
                                                  else if (position = f1) then f2
                                                  else position fi fi

            []go? & (elevnextdire=DOWN)-> elevdire' := elevnextdire;
                                          door' := CLOSED;
                                          moving' := true;
                                          openDoorTimes' := 0;
                                          position' := if( position = f2) then f1
                                                  else if (position = f1) then f0
                                                  else position fi fi

            --No movement command and next direction is IDLE, so should open the
            --door and wait
            [](elevnextdire=IDLE)->       elevdire' := elevnextdire;
                                          door' := OPEN;
                                          moving' := false;
                                          openDoorTimes' := 0;
                                          position' := position;
                                          passengerPresent' := nondet

            --opendoor is pressed!
            []opendoor? -> moving' := false;
                           position' := position;
                           door' := OPEN;
                                  elevdire' := elevdire;
                           openDoorTimes' := openDoorTimes + 1

            --passenger is present!
            []closedoor? -> moving' := false;
                            position' := position;
                            door' := CLOSED;
                                   elevdire' := elevdire

            -- if there is no any events, the elevator stops there.
            []default -> moving' := false; door' := OPEN; position' := position
```

```
        endatom

endmodule


module controller
-- controls the elevator car to serve the requests from each floor.
-- It detects requests from reading the variable floor[0,1,2] and determines the
-- next direction according the current direction of elevator and which floor is
-- required.

    external position : posiType;
             floor0, floor1, floor2 : bool;
             elevdire : movingType;
             openDoorButton : bool;
             openDoorTimes : counterType;
             passengerPresent : bool;
             door : doorStatusType
    interface elevnextdire : movingType;
              stop, go, opendoor, closedoor : event;
              arrive0, arrive1, arrive2 : bool
    private   keepGoing : bool

    atom MoveControl
        controls keepGoing, stop, go, arrive0, arrive1, arrive2
        reads    stop, go, position, floor0, floor1, floor2, arrive0,
                 arrive1, arrive2

        init
            []true -> keepGoing' := false; arrive0' := false;
                           arrive1' := false; arrive2' := false
        update
            -- satisfy the first floor request
            [] position=f0 & floor0 -> stop! ; arrive0':=true; arrive1' := false;
                                    arrive2' := false; keepGoing' := false

            -- satisfy the second floor request
            [] position=f1 & floor1 -> stop! ; arrive0':=false; arrive1' := true;
                                    arrive2' := false; keepGoing' := false

            -- satisfy the third floor request
            [] position=f2 & floor2 -> stop! ; arrive0':=false; arrive1' := false;
                                    arrive2' := true; keepGoing' := false

            -- no need to stop, so keep going
            [] position=f0 & ~floor0 & ( floor1 | floor2 ) ->
                                                   go! ; keepGoing' := true

            [] position=f1 & ~floor1 & ( floor0 | floor2 ) ->
                                                   go! ; keepGoing' := true

            [] position=f2 & ~floor2 & ( floor0 | floor1 ) ->
                                                   go! ; keepGoing' := true


            [] default -> keepGoing' := false
       endatom
```

```
atom DireControl
    controls elevnextdire, opendoor, closedoor
    reads    elevnextdire, elevdire, position,
                  floor0, floor1, floor2, door,
                       openDoorButton, openDoorTimes, passengerPresent,
             opendoor, closedoor
    awaits   keepGoing

    init
        []true -> elevnextdire' := IDLE
    update

        ---** The elevator is currently in UP direction**---

        -- if it is at the upper most floor, and any of lower floor is
        -- required, go DOWN.
        [] keepGoing' & elevdire=UP & position=f2 & (floor0| floor1)
                                   -> elevnextdire' := DOWN

        -- if it is at the low most floor, and any of upper floors is
        -- request, still go UP.
        [] keepGoing' & elevdire=UP & position=f0 & (floor1 | floor2)
                                   -> elevnextdire' := UP

        -- if it is at the middle floor, the priority is given to the
        -- upper floor requests.
        [] keepGoing' & elevdire=UP & position=f1 &  floor2
                                   -> elevnextdire' := UP
        [] keepGoing' & elevdire=UP & position=f1 & ~floor2 & floor0
                                   -> elevnextdire' := DOWN


        ---**The elevator is currently in DOWN direction**---

        -- if it is at the lower most floor, and any of upper floors is
        -- required, go UP.
        [] keepGoing' & elevdire=DOWN & position=f0 & (floor1 | floor2)
                                   -> elevnextdire' := UP

        -- if it is at the upper most floor, and any of lower floors is
        -- required, go DOWN.
        [] keepGoing' & elevdire=DOWN & position=f2 & (floor0 | floor1)
                                   -> elevnextdire' := DOWN

        -- if it is at the middle floor, the priority is given to the
        -- lower floor requests.
        [] keepGoing' & elevdire=DOWN & position=f1 & floor0
                                   -> elevnextdire' := DOWN
        [] keepGoing' & elevdire=DOWN & position=f1 & ~floor0 & floor2
                                   -> elevnextdire' := UP


        ---**The elevator currently is IDLE **---

        -- if it is at the upper most floor, and any of lower floors is
        -- required, go DOWN.
        [] keepGoing' & elevdire=IDLE & position=f2 & (floor0 | floor1)
                                   -> elevnextdire' := DOWN
```

```
            -- if it is at the upper most floor, and any of lower floors is
            -- required, go DOWN.
            [] keepGoing' & elevdire=IDLE & position=f0 & (floor1 | floor2)
                                       -> elevnextdire' := UP

            -- if it is at the middle floor, the priority is given to the
            -- upper floor requests.
            [] keepGoing' & elevdire=IDLE & position=f1 & floor2
                                       -> elevnextdire' := UP
            [] keepGoing' & elevdire=IDLE & position=f1 & ~floor2 & floor0
                                       -> elevnextdire' := DOWN

            -- if the elevator is not moving and passenger is present
            -- door should be closed
            [] ~keepGoing' & passengerPresent & (door = OPEN) -> closedoor!

            -- if the elevator is not moving and open door button is pressed,
            -- open the door if open door times is less than twice.
            [] ~keepGoing' & openDoorButton & (openDoorTimes < 2) & (door = CLOSED)
                                              -> opendoor!

            -- if none of three floors is required, it should be IDLE.
            [](~floor0) &  (~floor1) & (~floor2) -> elevnextdire' := IDLE

            []default -> elevnextdire' := elevnextdire
        endatom
endmodule


-- We create the entire elevator system by parallel composition. To construct
-- module abstractions of degrees of detail, we hide some interface variables
-- so that only some status data the user can view, such as elevdire, door,
-- moving, etc.

ElevSystem := hide elevnextdire,  arrive0, arrive1, arrive2,
              keepGoing in (floor_0 || floor_1 || floor_2 || elevator
                           || openDoorBtn || controller) endhide
```

Appendix B: Elevator System Properties

```
/*************************************************************************


    The invariant and ATL properties of a three-level elevator system


*************************************************************************/

--The elvator never moves with its door open.
--This property can be expressed either in invariant form or ATL form
inv "inv1"
 ~( moving & (door = OPEN) );

atl "atl1"
 A G ~( moving & (door = OPEN) );

atl "atl11"
 A G ( moving => (door = CLOSED) );


--Requests to use the elevator are eventually serviced.

atl "atl21"
 A G ( exBtn0 => A F ( (door = OPEN) & (position = f0)));

atl "atl22"
 A G ( exBtn1 => A F ( (door = OPEN) & (position = f1)));

atl "atl23"
 A G ( exBtn2 => A F ( (door = OPEN) & (position = f2)));


--Requests to be delivered to a particular floor are eventually serviced.

atl "atl24"
 A G ( inBtn0 => A F ( (door = OPEN) & (position = f0)));

atl "atl25"
 A G ( inBtn1 => A F ( (door = OPEN) & (position = f1)));

atl "atl26"
 A G ( inBtn2 => A F ( (door = OPEN) & (position = f2)));


--The above two types properties can be tested together since
--floor0 = (exBtn0 | inBtn0).

atl "atl27"
 A G ( floor0 => A F ( (door = OPEN) & (position = f0)));

atl "atl28"
 A G ( floor1 => A F ( (door = OPEN) & (position = f1)));

atl "atl29"
 A G ( floor2 => A F ( (door = OPEN) & (position = f2)));
```

```
--Vacuity check for the above property formulas.
atl "atl30"
 E F (exBtn0);

atl "atl31"
 E F (exBtn1);

atl "atl32"
 E F (exBtn2);

atl "atl33"
 E F (inBtn0);

atl "atl34"
 E F (inBtn1);

atl "atl35"
 E F (inBtn2);


--The elevator should keep its door open until there is a request to use it.
atl "atl50"
 A G ( (door = OPEN) => A ( (door = OPEN) W (floor0 | floor1 | floor2)));


--When someone steps into the elevator, the door should close and remain closed unless
--the door open button is pressed.
atl "atl60"
 A G ( (~moving & passengerPresent & (door = OPEN)) => A ((door = CLOSED) U
openDoorButton ));

--Vacuity check for the above property,
--i.e. Eventually, the passenger will present" as follows:
atl "atl61"
 E F (~moving & passengerPresent & (door = OPEN));


--The elevator react "open door" at most twice
atl "atl70"
 A G ~(openDoorButton & (openDoorTimes > 2) & (door = OPEN));


--The elevator cannot change its direction when it is moving.
atl "atl80"
 A G ( (moving & (elevdire = UP)) => A X ~( moving & (elevdire = DOWN)));


--The floor has the discretion to make the external button not be pressed,
--No other module can force it to do otherwise.
atl "atl95"
 A G ( ~exBtn0 => <<floor_0>> G ~exBtn0);

atl "atl96"
 A G ( ~exBtn1 => <<floor_1>> G ~exBtn1);

atl "atl97"
```

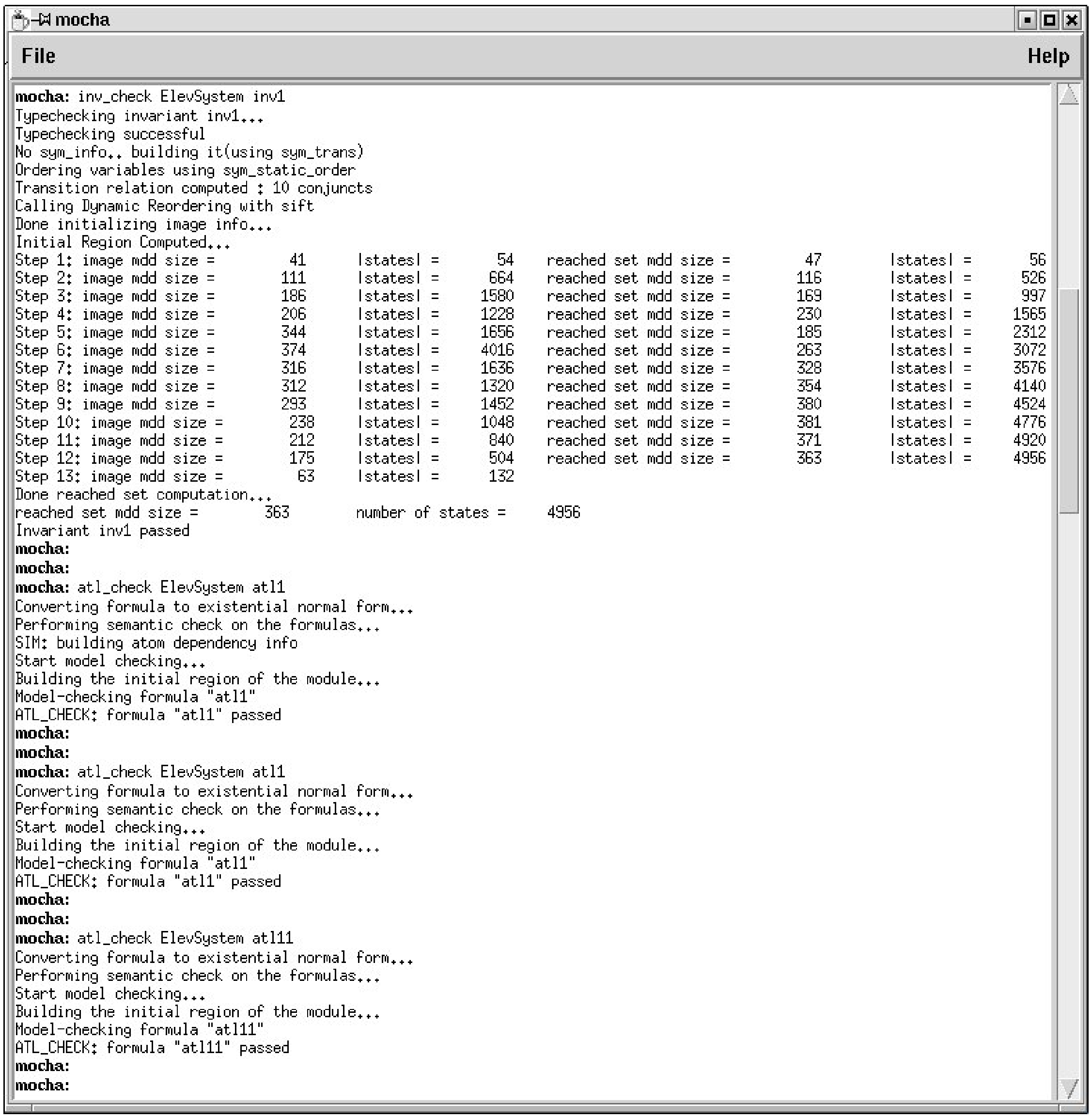
```
 A G ( ~exBtn2 => <<floor_2>> G ~exBtn2);


--Whenever the request button is off, the controller does not have a strategy
--to produce a trajectory to force the external button not be pressed .
atl "atl90"
 A G ( ~exBtn1 => [[controller, elevator, floor_0, floor_2, openDoorBtn ]] G ~exBtn1);


--The floor has a strategy to make the external button be pressed,
--no matter how the other module behaves.
atl "atl100"
 <<floor_0>> F (exBtn0);

atl "atl101"
 <<floor_1>> F (exBtn1);

atl "atl102"
 <<floor_2>> F (exBtn2);

--Whenever the elevator is currently in the first floor and the second floor
--is required, the controller and the elevator can cooperate so that the
--elevator will be at the second floor.
atl "atl120"
 << >> G ( ((position=f0) & floor2) =>
                  <<controller, elevator>> F ((position = f2)&(door=OPEN)));

--atl "atl130"
-- <<elevator>> F ((positon=f1) & floor2) => <<elevator>> F (elevdire=DOWN));
--
```

Appendix C: Result of invariant checking and ATL checking

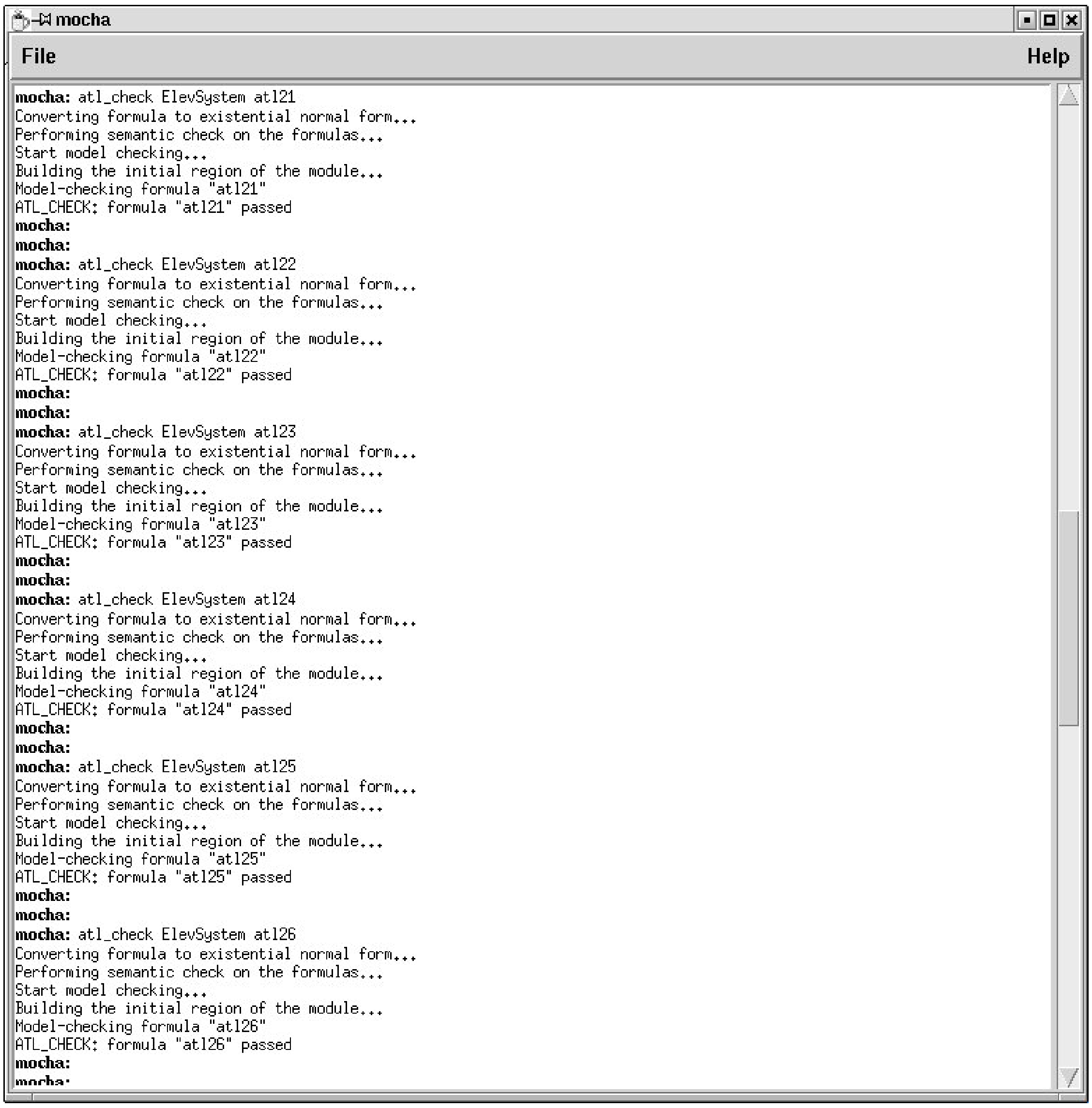
The number of properties that checked is consistent with the properties specified in Appendix B. MOCHA does not report the time that takes to verify properties. We manually recorded the time. It took about 2 seconds to check the properties:

```
atl "atl95"  A G ( ~exBtn0 => <<floor_0>> G ~exBtn0);
atl "atl96"  A G ( ~exBtn1 => <<floor_1>> G ~exBtn1);
atl "atl97"  A G ( ~exBtn2 => <<floor_2>> G ~exBtn2);
```

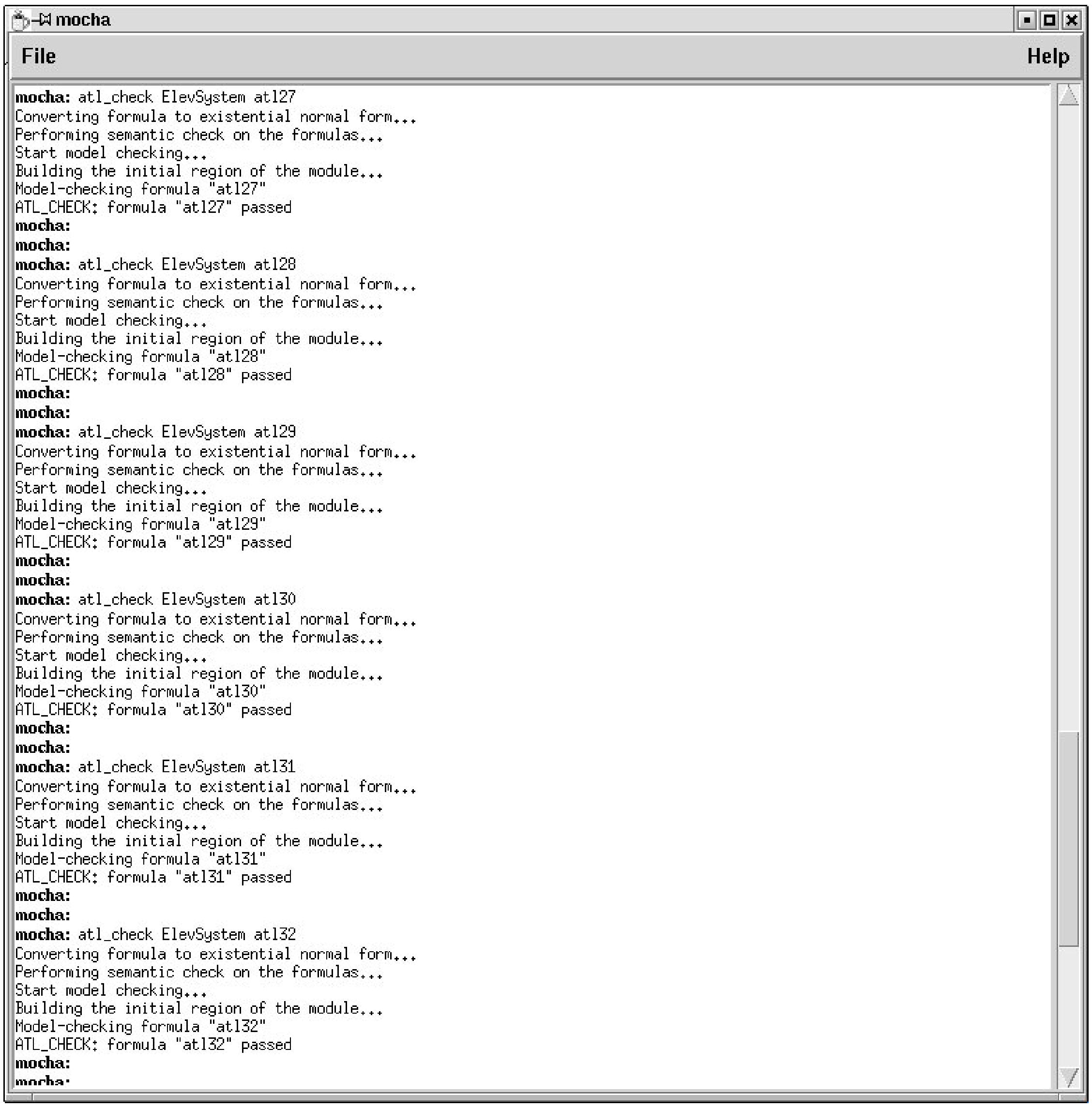
All the other properties are verified in one second.



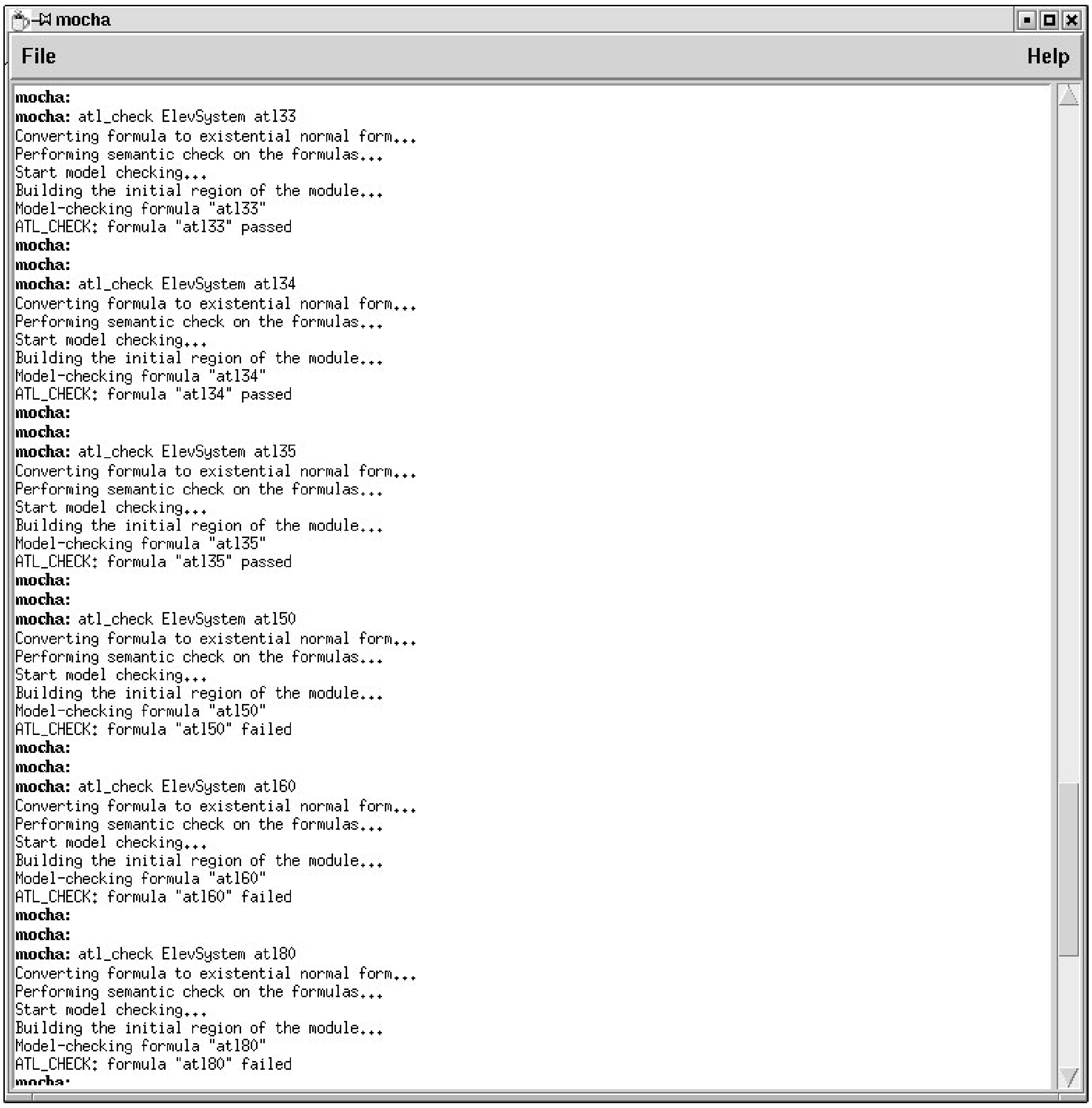
```
mocha: inv_check ElevSystem inv1
Typechecking invariant inv1...
Typechecking successful
No sym_info.. building it(using sym_trans)
Ordering variables using sym_static_order
Transition relation computed : 10 conjuncts
Calling Dynamic Reordering with sift
Done initializing image info...
Initial Region Computed...
Step 1: image mdd size =        41      |states| =      54    reached set mdd size =        47        |states| =      56
Step 2: image mdd size =       111      |states| =     664    reached set mdd size =       116        |states| =     526
Step 3: image mdd size =       186      |states| =    1580    reached set mdd size =       169        |states| =     997
Step 4: image mdd size =       206      |states| =    1228    reached set mdd size =       230        |states| =    1565
Step 5: image mdd size =       344      |states| =    1656    reached set mdd size =       185        |states| =    2312
Step 6: image mdd size =       374      |states| =    4016    reached set mdd size =       263        |states| =    3072
Step 7: image mdd size =       316      |states| =    1636    reached set mdd size =       328        |states| =    3576
Step 8: image mdd size =       312      |states| =    1320    reached set mdd size =       354        |states| =    4140
Step 9: image mdd size =       293      |states| =    1452    reached set mdd size =       380        |states| =    4524
Step 10: image mdd size =       238     |states| =    1048    reached set mdd size =       381        |states| =    4776
Step 11: image mdd size =       212     |states| =     840    reached set mdd size =       371        |states| =    4920
Step 12: image mdd size =       175     |states| =     504    reached set mdd size =       363        |states| =    4956
Step 13: image mdd size =        63     |states| =     132
Done reached set computation...
reached set mdd size =       363       number of states =     4956
Invariant inv1 passed
mocha:
mocha:
mocha: atl_check ElevSystem atl1
Converting formula to existential normal form...
Performing semantic check on the formulas...
SIM: building atom dependency info
Start model checking...
Building the initial region of the module...
Model-checking formula "atl1"
ATL_CHECK: formula "atl1" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl1
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl1"
ATL_CHECK: formula "atl1" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl11
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl11"
ATL_CHECK: formula "atl11" passed
mocha:
mocha:
```



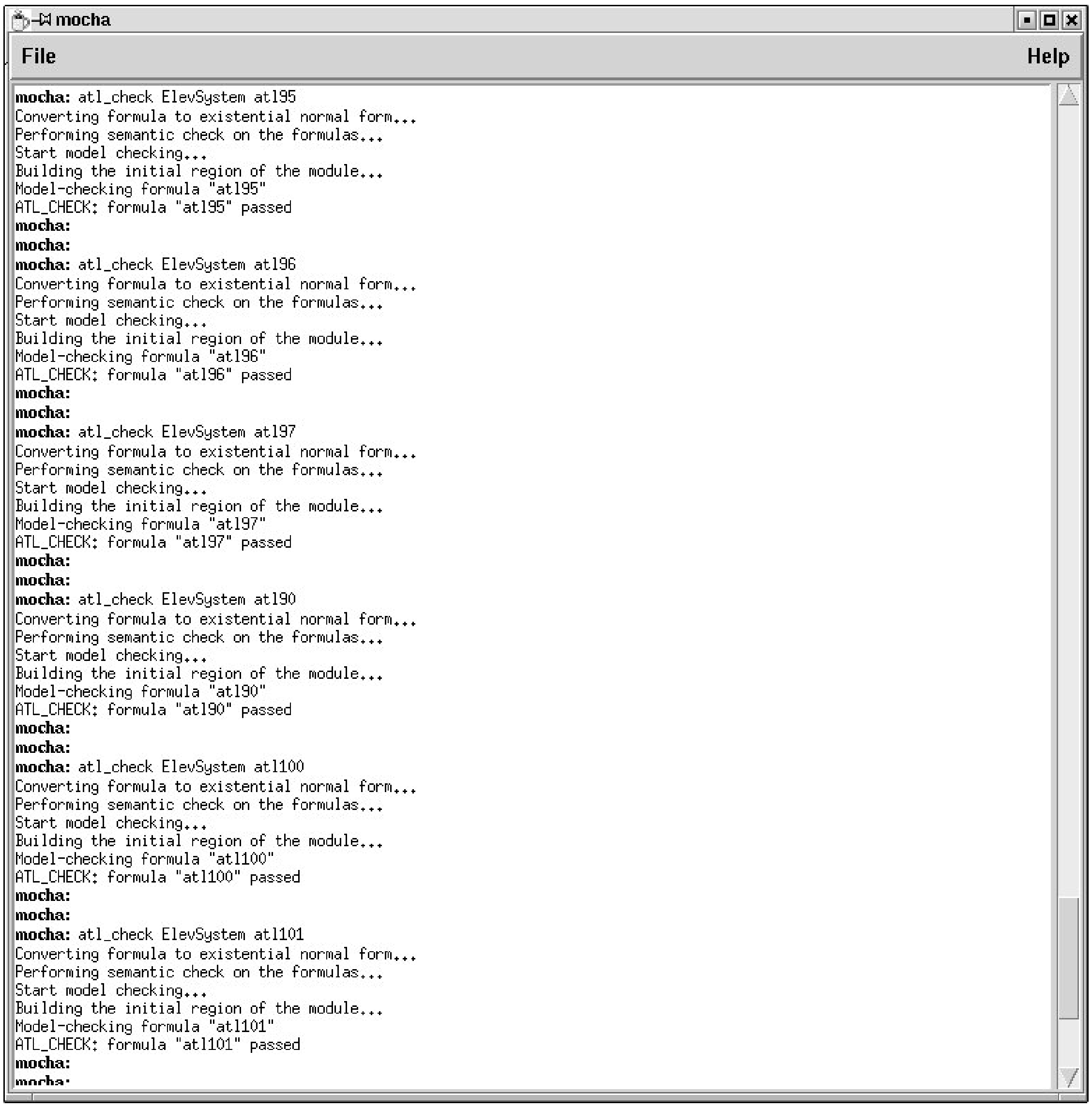
```
mocha: atl_check ElevSystem atl21
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl21"
ATL_CHECK: formula "atl21" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl22
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl22"
ATL_CHECK: formula "atl22" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl23
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl23"
ATL_CHECK: formula "atl23" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl24
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl24"
ATL_CHECK: formula "atl24" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl25
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl25"
ATL_CHECK: formula "atl25" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl26
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl26"
ATL_CHECK: formula "atl26" passed
mocha:
mocha:
```



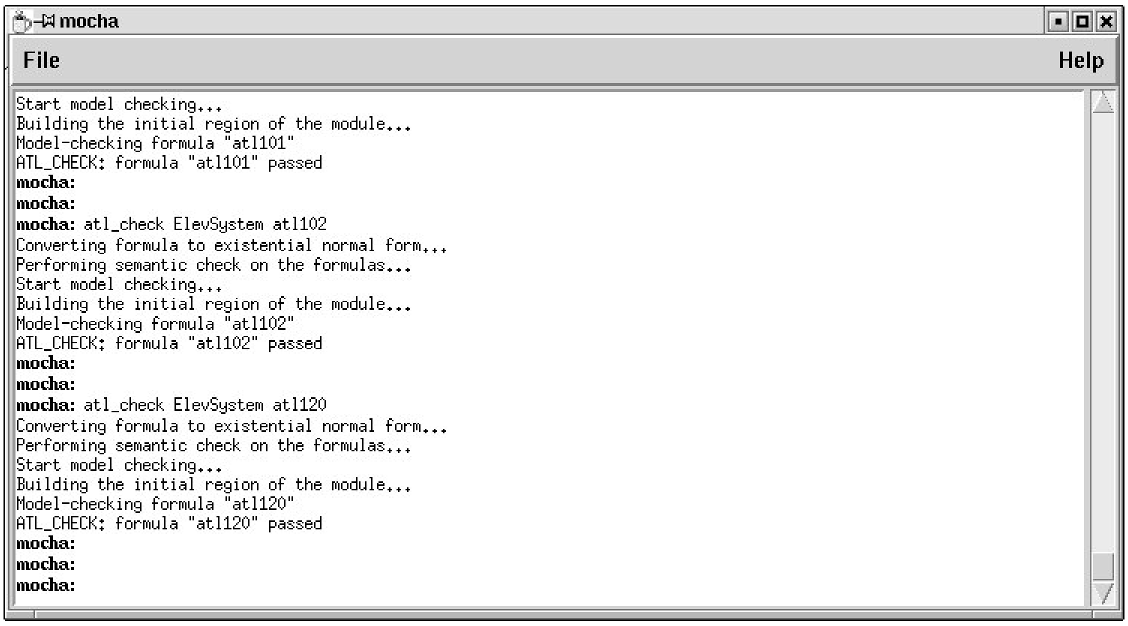
```
mocha: atl_check ElevSystem atl27
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl27"
ATL_CHECK: formula "atl27" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl28
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl28"
ATL_CHECK: formula "atl28" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl29
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl29"
ATL_CHECK: formula "atl29" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl30
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl30"
ATL_CHECK: formula "atl30" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl31
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl31"
ATL_CHECK: formula "atl31" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl32
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl32"
ATL_CHECK: formula "atl32" passed
mocha:
mocha:
```



```
mocha:
mocha: atl_check ElevSystem atl33
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl33"
ATL_CHECK: formula "atl33" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl34
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl34"
ATL_CHECK: formula "atl34" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl35
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl35"
ATL_CHECK: formula "atl35" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl50
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl50"
ATL_CHECK: formula "atl50" failed
mocha:
mocha:
mocha: atl_check ElevSystem atl60
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl60"
ATL_CHECK: formula "atl60" failed
mocha:
mocha:
mocha: atl_check ElevSystem atl80
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl80"
ATL_CHECK: formula "atl80" failed
mocha:
```



```
mocha: atl_check ElevSystem atl95
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl95"
ATL_CHECK: formula "atl95" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl96
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl96"
ATL_CHECK: formula "atl96" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl97
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl97"
ATL_CHECK: formula "atl97" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl90
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl90"
ATL_CHECK: formula "atl90" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl100
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl100"
ATL_CHECK: formula "atl100" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl101
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl101"
ATL_CHECK: formula "atl101" passed
mocha:
mocha:
```



```
Start model checking...
Building the initial region of the module...
Model-checking formula "atl101"
ATL_CHECK: formula "atl101" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl102
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl102"
ATL_CHECK: formula "atl102" passed
mocha:
mocha:
mocha: atl_check ElevSystem atl120
Converting formula to existential normal form...
Performing semantic check on the formulas...
Start model checking...
Building the initial region of the module...
Model-checking formula "atl120"
ATL_CHECK: formula "atl120" passed
mocha:
mocha:
mocha:
```