Mocha User Manual

R. Alur
Computer and Information Science Department
University of Pennsylvania, Philadelphia
alur@cis.upenn.edu

L. de Alfaro T.A. Henzinger S.C. Krishnan
F.Y.C. Mang S. Qadeer S.K. Rajamani S. Taşıran
Electrical Engineering and Computer Sciences Department
University of California, Berkeley
{dealfaro,tah,krishnan,fmang,shaz,sriramr,serdar}@eecs.berkeley.edu

www.eecs.berkeley.edu/mocha
Original date: circa 1998
Current Revision: September 29, 2004

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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

- For modeling, we replace unstructured state-transition graphs with the heterogeneous modeling framework of *reactive modules* [AH96]. The definition of reactive modules is inspired by formalisms such as Unity [CM88], I/O automata [Lyn96], and Esterel [BG88], and allows complex forms of interaction between components within a single transition. Reactive modules provide a semantic glue that allows the formal embedding and interaction of components with different characteristics. Some modules may be synchronous, others asynchronous, some may represent hardware, others software, some may be speed-independent, others time-critical.

- For requirement specification, we replace the system-level specification languages of linear and branching temporal logics [Pnu77, CE81] with the module-level specification language of *Alternating Temporal Logic* (ATL) [AHK97]. In ATL, both cooperative and adversarial relationships between modules can be expressed. For example, it is possible to specify that a module can attain a goal regardless of how the environment of the module behaves.

- For the verification of complex systems, MOCHA supports a range of *compositional and hierarchical verification methodologies*. For this purpose, reactive modules provide assume-guarantee rules [HQR98] and abstraction operators [AHR98]; MOCHA provides algorithms for automatic refinement checking, and will provide a proof editor that manages the decomposition of verification tasks into subtasks.

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

- Execution, including games between the user and MOCHA

- Enumerative and symbolic invariant checking and error-trace generation
- Compositional refinement checking
- ATL model checking
- Reachability analysis of real-time systems

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

1.1 Organization of this manual

Chapter 2 is a tutorial introduction to MOCHA. Chapter 3 goes into the syntax and semantics of REACTIVEMODULES in great depth, and is replete with examples, both simple and moderately complex. Chapter 4 discusses the specification formalism for specifying requirements of modules accepted by MOCHA. The commands available at the MOCHA shell prompt are covered in Chapter 5. The support in MOCHA for modular verification by assume-guarantee reasoning, and other means to circumvent the intractability of monolithic verification is dealt with in Chapter 6. Real-time modules and verification is the subject of Chapter 7.

1.2 How to read this manual

The best way to learn to use MOCHA is to taste it quickly and let it stimulate you, and then let the high take care of the rest. Start with the tutorial in Chapter 2 in front of a terminal. Then perhaps read Chapter 5 (again in front of a terminal).

Thereafter, continue having fun, and read the other parts of the manual as and when needed. The tables in Chapter 3, a listing of which can be found on Page 4, should suffice for the grammar of REACTIVEMODULES. An index can be found at the end of this manual. If you have any problems, please do not hesitate to send email to `mocha@eecs.berkeley.edu`.

Chapter 2

Tutorial introduction to MOCHA

2.1 Introduction

REACTIVEMODULES is the modeling formalism and input language to MOCHA. REACTIVEMODULES is rich enough to model systems with heterogeneous components: synchronous, asynchronous, speed-independent or real-time, finite or infinite state, etc. In this tutorial chapter we illustrate the facilities of MOCHA by considering two simple examples, one from hardware and the other from software.

The hardware example is a simple counter adapted from a similar example in the Symbolic Model Verifier (SMV) example suite []; the tricks developed here should enable the reader to translate any design in the commonly used subsets of hardware description languages (HDLs) such as VERILOG [] or VHDL [] into REACTIVEMODULES. The software example we consider is Peterson's mutual-exclusion protocol. In addition to modeling these examples in REACTIVEMODULES, specifying and verifying correctness requirements, we walk the reader through an interactive session with the MOCHA tool.

2.2 3-bit counter

Consider a counter that counts the number of 1's in a binary input stream, modulo 8. We construct this out of three 1-bit counter cells.

The Mealy Finite State Machine (FSM) shown in Figure 2.1, `counterCell` implements the 1-bit counter-cell. `counterCell` receives a `carryIn` bit as input, maintains a `sumBit`, and outputs a `carryOut` bit. `sumBit` is stored in a register. The register assumes an initial value 0 (the initialization circuitry is not shown in Figure 2.1). During each clock cycle, the combinational logic computes the values of the next-state of the register and the carry-out bit as

$$(\texttt{carryOut}, \texttt{sumBit}') = (\texttt{sumBit} + \texttt{carryIn}) \tag{2.1}$$

where `carryOut` is the high order bit of the sum and `sumBit` is the low order bit of the sum. Notice, the next-state of `sumBit`, being a latch, is denoted by `sumBit'`,

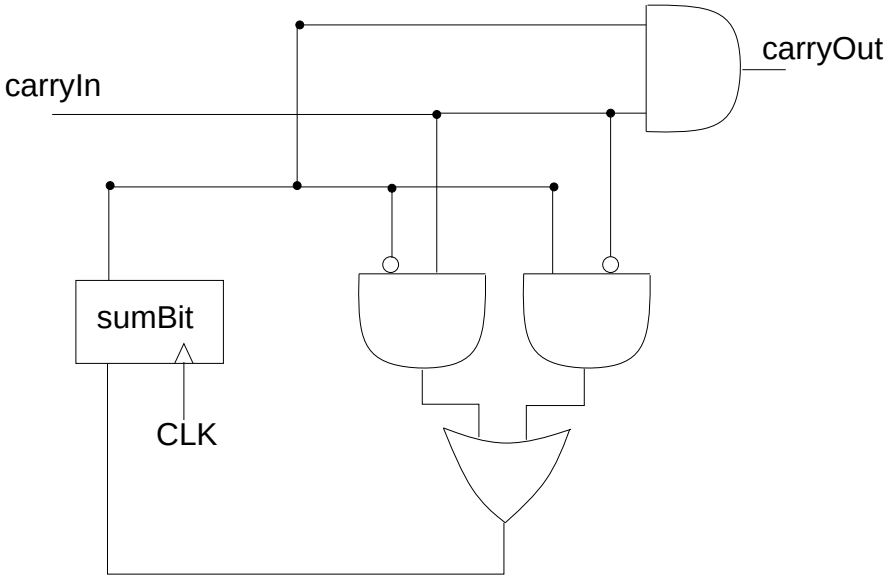

Figure 2.1: `counterCell`: Mealy FSM

Figure 2.2: 3-bit counter

whereas the other signals are outputs of combinational gates. It is easy to see that the FSM of Figure 2.1 implements (2.1).

The 3-bit counter is produced by connecting up three of the counter-cells as shown in Figure 2.2. The `carryOut` bit of the first (second) cell is the `carryIn` bit of the second cell (third). This is achieved by physically connecting a wire between the appropriate terminals. The `sumBit`'s of the three counter-cells store the three bit sum of the input stream.

2.3 Modeling the 3-bit counter in ReactiveModules

A ReactiveModules specification consists of one or more *modules*. A module roughly corresponds to a module in a HDL or a function in a programming language. A module consists of set of variables and a set of rules to define the evolution in time of the subset of variables it controls. The input variables are called the *external* variables and the output variables are called the *interface* variables. The other variables of the module that are not exchanged with its environment are called *private* variables. The private and interface variables of a module are controlled by it.

For the `counterCell`, `sumBit` is a private variable, `carryIn` is an external variable, and `carryOut` is an interface variable, as in Figure 2.3.

In synchronous digital hardware, in each cycle, the primary inputs (PIs) and the present state values of latches propagate to determine the next state values of latches and the outputs (including the primary outputs). Then, all the latches simultaneously assume their next state values, and the process repeats in this manner every clock cycle.

Similarly, with ReactiveModules, in every *round* (like cycle) the variables are updated. No apriori distinction is made between variables for latches and variables for combinational gates. Instead, the semantic distinction between these is captured in their update rules specified by *atoms*.

If the value of a variable x—the present value—at the beginning of a round is denoted by x, the update round computes the next or primed value x'. The initial value computation that happens in the init round is specified by the init section of the atom, and the subsequent updates in the update rounds are specified by the update sections of the atoms.

An atom specifies the next state function as a set of guarded commands: when the guard is satisfied the action is taken. The next-state function of the variable an atom controls may be a function of the present value of variables (i.e., unprimed version) it *reads* and primed value of variables it *awaits*. In particular, both the guards and the commands may only involve expressions over the unprimed variables that are read and the primed variables that are awaited. For instance, the next value of `carryOut` (i.e. `carryOut'`), is a function of the present value of `sumBit` (being a latch) and the next value of `carryIn` (being a primary input): its atom, in Figure 2.3, reads `sumBit` and awaits `carryIn`.

Also, this means that the next value of a variable will be computed after the next values of the variables it awaits are computed. Note that cyclic await dependencies are disallowed. In fact, the partial order induced by the await dependencies among all atoms must be completable to a linear order.

Figure 2.3 shows the ReactiveModules description of the counter-cell. In each round, the next value of carryIn is first computed, and then sumBit and carryOut may be computed in each order. A sample execution of the module is given in the table below:

Round:	0 (initial)	1	2	3	4	···
`carryIn`:	T	T	T	F	T	···
`sumBit`:	F	T	F	F	T	···
`carryOut`:	F	F	T	F	F	···

From the exercise of modeling the counter-cell in ReactiveModules, we can derive the rules in Table 2.1 to translate a Mealy FSM module to ReactiveModules.

2.3.1 Module instantiation and composition

New modules can be created from previously defined modules by instantiating them with renaming as well as by composing together modules. The desired interconnec-

```
module counterCell
  private sumBit : bool
  external carryIn : bool
  interface carryOut : bool

-- this is a comment; a comment line starts with --

atom controls sumBit reads sumBit awaits carryIn
init
  [] true -> sumBit' := false
update
  [] ~sumBit -> sumBit' := carryIn'
  [] sumBit  -> sumBit' := if (~carryIn') then true else false fi
endatom

atom controls carryOut reads sumBit awaits carryIn
init
  [] true -> carryOut' := false
update
  [] true -> carryOut' := sumBit & carryIn'
endatom

endmodule  -- end counterCell
```

Figure 2.3: 1-bit counter-cell

If a signal x is a function of signal y, then in the atom controlling variable x:

1. if y is a latch-output, the variable y should be read, i.e., its unprimed version should be used.
2. if y is a gate output, the variable y should be awaited, i.e., its primed version should be used.
3. if y is an input signal, the variable y should be awaited, i.e., its primed version should be used.

Table 2.1: Recipe for variable assignment in translating a Mealy FSM module to ReactiveModules

tion of the counter-cells in Figure 2.2 is done as follows:

```
cell0 := counterCell[carryIn, carryOut := input, out0]
cell1 := counterCell[carryIn, carryOut := out0, out1]
cell2 := counterCell[carryIn, carryOut := out1, out2]
threebitcounter := hide out0, out1 in (cell0 || cell1 || cell2) endhide
```

Three instances of `counterCell` are created by renaming the external and interface variables using common names for the desired interconnections. For instance, `out0` is the `carryOut` of `cell0` and the `carryIn` of `cell1`.

The actual interconnection is achieved by the parallel composition of the three instances: `cell0`, `cell1`, and `cell2`, and hiding the `carryOut` variables of `cell0` and `cell1`, making these private variables of `threebitcounter`. The module `threebitcounter` has one external variable `input` and one interface variable `out2`.

The order of `cell0`, `cell1`, and `cell2` in the parallel composition statement is not important. For example, the previous parallel composition statement is equivalent to the following line:

```
threebitcounter := hide out0, out1 in (cell1 || cell0 || cell2) endhide
```

2.4 Other issues in modeling hardware

2.4.1 Connecting a latch output to a primary input

At times we might need to compose modules and connect a latch from one module to the input of another module. This would lead to a violation of item 1 of Table 2.1 since we decreed that an input variable is to be awaited (by item 3 of Table 2.1).

To work around this we have to insert a unit-delay non-inverting buffer with the appropriate initial value between the latch-output and the input of the module it connects to. A unit-delay non-inverting buffer with initial value 0 is modeled in ReactiveModules as follows:

```
module noninvertingbuffer0
  external latchoutput
  interface onedelayed

--unit delay non inverting buffer with initial value 0
atom controls onedelayed reads latchoutput
init
  [] true -> onedelayed' := false
update
  [] true -> onedelayed' := latchoutput
endatom
endmodule
```

We have one additional rule to invoke in translating a network of Mealy FSMs to ReactiveModules, and it is stated in Table 2.2.

If y is a latch-output and the output of a module A and is being connected to x an external variable of module B, introduce a unit-delay non-inverting buffer with the appropriate initial value between y and x.

Table 2.2: Rule for connecting the latch-output of one module to the input of another module

2.4.2 Non-determinism

Non-determinism is useful in modeling systems at an abstract level. There are two ways to model non-determinism in ReactiveModules:

1. With a non-deterministic assignment.
2. By having multiple guarded commands with the same guard.

We illustrate both ways of having non-determinism with a simple example. We design a module that will serve as the input to the three-bit-counter `threebitcounter` to result in a closed system, i.e., one with no inputs (no external variables), and will non-deterministically output a 0 or 1. The first way of modeling this is shown below:

```
module nondetinput
  interface output : bool

atom controls output
init update
  [] true -> output' := nondet
endatom
endmodule
```

The variable `output` is assigned a value non-deterministically from the range of values it can assume. The keyword `nondet` is used to assign to a variable a nondeterministic element of the variable domain.

The second equivalent way of modeling the module `nondetinput` is by using multiple guarded commands with the same guard is:

```
module nondetinput
  interface output : bool

atom controls output
init update
  [] true -> output' := false
  [] true -> output' := true
endatom
endmodule
```

This second method is useful (and the only way) currently to model a nondeterministic assignment to a variable from a *subset* of the possible values it may assume.

2.4.3 Simple exercises

1. Modify the module `threebitcounter` to output a signal whenever the three bit sum has value 0.

2. Create a new module that is the composition of threebitcounter and nondetinput.

2.5 Running MOCHA

All the module definitions have to be entered into a single file named typically with the suffix .rm; in our case, (say) this file is counter.rm (Figure 2.4). MOCHA is invoked by typing `mocha` at the shell prompt. If you do not want to bring up MOCHA with its GUI, start MOCHA in the text mode typing `mocha -t`.

The module is read and parsed with the `read_module` command. MOCHA displays the names of the modules that were successfully parsed. In the case of a parse error, an appropriate message is displayed.

```
mocha:   read_module counter.rm
Module counterCell is composed and checked in.
Module cell0 is composed and checked in.
Module cell1 is composed and checked in.
Module cell2 is composed and checked in.
Module threebitcounter is composed and checked in.
Module nondetinput is composed and checked in.
Module InputModule is composed and checked in.
Module closedthreebitcounter is composed and checked in.
parse successful.
```

The command `show_mdls` lists the modules that have been read in.

```
mocha: show_mdls
closedthreebitcounter
InputModule
cell0
cell1
cell2
counterCell
threebitcounter
```

MOCHA provides many methods of verifying the correctness of a design: execution (i.e., simulation), invariant checking, refinement checking, and ATL model

```
-- this is a comment
-- 3 bit counter found in the SMV example suite
module counterCell
  private sumBit : bool
  external carryIn : bool
  interface carryOut : bool

atom controls sumBit reads sumBit awaits carryIn
init
  [] true -> sumBit' := false
update
  [] ~sumBit -> sumBit' := carryIn'
  [] sumBit -> sumBit' := if (~carryIn') then true else false fi
endatom

atom controls carryOut reads sumBit awaits carryIn
init
  [] true -> carryOut' := false
update
  [] true -> carryOut' := sumBit & carryIn'
endatom
endmodule  -- end counterCell

cell0 := counterCell[ carryIn, carryOut := input, out0 ]
cell1 := counterCell[ carryIn, carryOut := out0, out1 ]
cell2 := counterCell[ carryIn, carryOut := out1, out2 ]

threebitcounter := hide out0, out1 in
   (cell0 || cell1 || cell2) endhide

module nondetinput
  interface output : bool

atom controls output
init update
  [] true -> output' := nondet
endatom
endmodule

InputModule := nondetinput[ output := input ]
closedthreebitcounter := hide input in
   (InputModule || threebitcounter) endhide
```

Figure 2.4: Input file: counter.rm

checking. We highlight execution and ATL model checking in this tutorial, being the distinguishing features of MOCHA, and treat refinement checking and assume-guarantee reasoning in subsequent chapters.

2.5.1 Executing modules

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. To execute a module, first read it in using the `read_module` command. Then choose it by selecting it in the browser, that is in turn brought up by choosing under the browse option under the "File" pull down menu.

After selecting the module, press the open button. For instance, choose the module `closedthreebitcounter`. A new window will pop up containing the REACTIVEMODULES definition of the module selected. Now click the "Execute" button. A new window pops up giving the three options for execution as radio buttons. For the Game execution option, the user gets to choose the atoms for which he can specify the next state (i.e., resolve the non-determinism); the other atoms' next states will be chosen by MOCHA; use the browse button to choose the atoms the user wants to control.

The upper window presents (by default) a table of the external and interface variables for the chosen module and possible values at the current state. The user gets to choose the particular tuple he wants to proceed with in the case of manual or game execution. Additional variables to view can be selected by choosing the "Select Variables" option under the options pull down menu. For instance, if you chose to execute `closedthreebitcounter` under the game mode, and chose to play the module `InputModule`, you will have to choose the input variable to be made visible. When the user's button is lit, the user should make a choice from the upper window and then press the "Go!" button. When it's the system's turn pressing the "Go!" button, executes the system's move. Try it!

2.6 ATL model checking

ATL is a new temporal logic that subsumes CTL and is appropriate for reasoning about open systems. The difference between ATL and CTL is that the path quantifiers in ATL are parametrized by a set of atoms, and a formula is true along all paths that the parametrized atoms can take the system into, no matter how the other atoms behave.

For instance, the CTL formula $AF(out2)$ is not true with respect to the module `closedthreebitcounter`. We obtain a counterexample to the formula if the input to the counter is set to 0. In this case the sum is always 0 and the carry-out from the third (`out2`) counter-cell never becomes 1.

The formula $AF(out2)$ stated in ATL is:

$$<<>> F(out2)$$

where the set of atoms parametrizing the path quantifier (within angle brackets) is empty. The formula is to be read as follows: no matter how the agents (i.e, atoms) behave it is the case that the system reaches a state where `out2` is true.

Similarly, the exists (E) path quantifier of CTL is the ATL path quantifier parametrized by all the atoms. For instance, the module `closedthreebitcounter` satisfies the CTL formula $EF(out2)$. The equivalent ATL formula is

$$<< U >> F(out2)$$

where U stands for a list of all the atoms in the module `closedthreebitcounter`.

ATL thus let us specify games where we can divide up the set of atoms into two teams; with some atoms in one team and the remainder in the other team, we can pose adversarial questions as to whether a team can enforce a condition no matter how the other team behaves.

For instance, we can pose the following ATL formula that is stronger than the CTL formula $EF(out2)$:

$$<< InputModule >> F(out2) \tag{2.2}$$

This formula asks: what are the states from which the module `InputModule`, i.e. the team comprised of atoms in the module `InputModule` has a strategy to make sure that `closedthreebitcounter` reaches a state where `out2` is true no matter how the other modules (atoms) behave. Notice that in CTL the only two teams possible are where one is the empty set and the other all the atoms, whereas ATL let us partition the atoms into two teams any way we want.

The ATL formula of (2.2) is also true of `closedthreebitcounter`, because again (for instance) a winning strategy for `InputModule` is to set the input to 1 sometime. The reader should be convinced that it is not difficult to come up with instances of modules that can be distinguished by ATL formulae but not by CTL formulae. Note that in ATL formulae we may use the short hand A and E to stand for the parametrized path quantifier with the empty set of atoms and the set of all atoms, respectively.

The ATL formulae can be entered into a file whose name is suffixed by `.spec`, named say `counter.spec`:

```
atl "atl1" A F (out2) ;
atl "atl2" E F (out2) ;
atl "atl3" << InputModule >>  F (out2);
```

Lines containing ATL formulae start with word `atl`, followed by the name of the formula, and then the formula, with a semi-colon terminating the line. The command `show_spec` prints out the names of ATL formulae read in (as well as invariants read in). Try it.

The command for ATL model checking is `atl_check`. The following command:

```
atl_check closedthreebitcounter atl1
```

which says check if the module `closedthreebitcounter` satisfies the ATL formula `atl1`. When you run this, you will get a message reporting a failure. On the otherhand:

```
atl_check closedthreebitcounter atl2
```

should pass as should `atl_check closedthreebitcounter atl3`.

2.7 Peterson's mutual exclusion protocol

As an example of a concurrent program consisting of processes that communicate through read-shared variables, we consider a mutual-exclusion protocol, which ensures that no two processes simultaneously access a common resource. The modules P_1 and P_2 of Figure 2.5 model the two processes of Peterson's solution to the mutual-exclusion problem for shared variables. Each process has a program counter (`pc1`, `pc2`) and a flag ($x1$, $x2$), both of which can be observed by the other process. The program counter indicates whether a process is outside its critical section (`outCS`), requesting the critical section (`reqCS`), or occupying the critical section (`inCS`). In each update round, a process looks at the latched values of all variables (reads them) and, nondeterministically, either updates its controlled variables or sleeps (i.e., leaves the controlled variables unchanged: achieved by the last guarded command with a true guard and no actions), without waiting to see what the other process does. Note that each process may sleep for arbitrarily many rounds: nondeterminism is used to ensure that there is no relationship between the execution speeds of the two processes.

Interleaving. Unlike in interleaving models, both processes may modify their variables in the same round. While Peterson's protocol ensures mutual exclusion even under these weaker conditions, if one were to insist on the interleaving assumption, one would add a third module that, in each update round, nondeterministically schedules either or none of the two processes. Alternatively, one could describe the complete protocol as a single module containing a single atom whose update action is the union of the update actions of the atoms of Figure 2.5. The guarded command that specifies a union of actions consists simply of the union of all guarded assignments of the individual actions. This style of describing asynchronous programs as an unstructured collection of guarded assignments is pursued in formalisms such as UNITY [CM88] and MURφ [Dil96].

Write-shared variables. The original formulation of Peterson's protocol uses a single write-shared boolean variable x, whose value always corresponds to the value of the predicate $x1 = x2$ in our formulation. If one were to insist on modeling x as a write-shared variable, one would add a third module with the interface variable x and awaited external variables such as $P_i\_sets\_x\_to\_0$, which is a boolean interface variable of the i-th process that indicates when the process wants to set x to 0. This style of describing write-shared memory makes explicit what happens when several processes write simultaneously to the same location.

2.8 Verification

The two modules `P1` and `P2` are composed to give module `Pete`. We verify that the version of Peterson's mutual exclusion protocol in Figure 2.5 does implement mutual exclusion. The following invariant says that both processes are not simultaneously in their critical sections:

```
inv "mutex" ~ (pc1 = inCS & pc2 = inCS);
```

The invariant can be entered into a file (say) `pete.spec`. The name of the invariant is `mutex`. Start up MOCHA, and first read in the module definitions of Figure 2.5 by typing:

```
mocha:  read_module pete.rm
```

The specification is read in with the `read_spec` command:

```
mocha:  read_spec pete.spec
```

The mutual-exclusion is tested with the command:

```
mocha:  inv_check Pete mutex
```

which says check if the module `Pete` satisfies the invariant `mutex`. `Pete` should satisfy the invariant.

2.9 Where to find the files for the examples in this chapter

The REACTIVEMODULES files as well as the ATL specs., etc., can be found on the WWW at `http://www-cad.eecs.berkeley.edu/mocha/demo.html`

```
module P1
  interface pc1:  {outCS,reqCS,inCS} ; x1 :  bool
  external pc2:  {outCS,reqCS,inCS} ; x2 :  bool
  atom controls pc1, x1 reads pc1, pc2, x1, x2
  init
    [] true -> pc1' := outCS
  update
    [] pc1=outCS                          -> pc1' := reqCS; x1' := x2
    [] pc1=reqCS & (pc2=outCS | ∼(x1=x2)) -> pc1' := inCS
    [] pc1=inCS                           -> pc1' := outCS
    [] true                               ->
  endatom
endmodule


module P2
  interface pc2:  {outCS,reqCS,inCS} ; x2 :  bool
  external pc1:  {outCS,reqCS,inCS} ; x1 :  bool
  atom controls pc2, x2 reads pc1, pc2, x1, x2
  init
    [] true -> pc2' := outCS
  update
    [] pc2=outCS                        -> pc2' := reqCS; x2' := ∼x1
    [] pc2=reqCS & (pc1=outCS | x1=x2) -> pc2' := inCS
    [] pc2=inCS                         -> pc2' := outCS
    [] true                             ->
  endatom
endmodule
Pete:= hide x1, x2 in (P1 || P2) endhide
```

Figure 2.5: Asynchronous mutual-exclusion protocol

Chapter 3

Reactive Modules

The input language that MOCHA uses for language description is that of *reactive modules.* The language of reactive modules is vaguely similar to a programming language. However, REACTIVEMODULES have a different emphasis than most programming languages. Programs written in ordinary programming languages are usually meant to describe procedures or processes in full detail, to enable their efficient execution. On the other hand, it may not be desirable (or indeed possible) to encode all the details of a system into a REACTIVEMODULES description. Hence non-determinism plays in REACTIVEMODULES a more important role than in ordinary programming languages, as it enables to abstract from such details. REACTIVEMODULES also provides extensive facilities for the modular description of a system, and for modeling both synchronous and asynchronous types of behavior.

The structure of a REACTIVEMODULES description resembles that of a conventional (imperative) programming language: to the statements of the language correspond *atoms,* and to the procedures correspond *reactive modules* (modules, for short). A complete description consists of one or more modules.

3.1 The input file

Structure of the file. A file contanining a REACTIVEMODULES description is usually given a name ending with the `.rm` extension, which is what MOCHA assumes by default. Keywords in REACTIVEMODULES can be separated by any type of *whitespace,* where whitespace can be a space, a new-line, or a tab. Comments in REACTIVEMODULES are introduced with the character sequence `--`: anything from `--` to the end of the line is considered as comment, and disregarded by MOCHA.

Splitting a description in multiple files. A long (or short) REACTIVEMODULES description can be split into multiple files for ease of handling. Each module must be contained in a single file. The complete description can be loaded into MOCHA simply by loading the separate files one after the other, as if reading a single file in successive steps.

Typographical conventions. In this manual, we write in typewriter font the keywords, such as `atom`, that have to be entered as specified, and we write in italics the items, such as *var, x*, for which you have to substitute the appropriate values. We use also the following special symbols:

- ... denotes an arbitrary number of items in a list, as in x_1 , ... ,x_n, where we assume that $n \geq 1$. The separator between list elements, when present (“,” in this example) indicates the separator that has to be used between list items.
- | denotes alternative: in `true` | `false`, you can write either `true` or `false`.
- [·] denotes an option: in [*atom_name*], the input *atom_name* can be provided or omitted.
- round parentheses in roman typeface, as in (·), are used for grouping.

3.2 Atoms

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. 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. You can also think of the initial round and the update rounds as the two elements that describe a transition system: the possible outcomes of the initial round correspond to the initial states of the transition system, and the update rounds define the transition relation. Atoms and modules are used to specify the initial and update update rounds for all the variables.

3.2.1 Atoms

An *atom* is the basic unit used to describe the initial condition and transition relation of a group of related variables. The syntax of an *atom* is given in Table 3.1. The keyword `lazy` is optional. Its meaning will be described in Section 3.2.3. The atom has an (optional) name *atom_name* which is used only as a reminder of the atom's purpose. The *identifier* is an alphanumerical string which begins with a letter. The *identifier* may also contain “_” (underscores) and “.” (dots).

An atom has three types of variables: the *controlled* variables $u_1, \ldots, u_i$, the *read* variables $v_1, \ldots, v_j$, and the *awaited* variables $w_1, \ldots, w_k$. An admissible identifier for a variable is an *identifier* followed by the optional expression *array_declare* which restricts the declaration to particular elements of an array structure. The exact meaning will be described in Section 3.4.6.

Controlled variables. The controlled variables of the atom represent the variables for which the atom can establish the value at the next round. Each variable is controlled by at most one atom: this insures that no conflict can arise between different atoms trying to update the same variable to different values.

atom ::= [**lazy**] **atom** [*atom_name*]
controls u_1 , ... ,u_i
[**reads** v_1 , ... ,v_j]
[**awaits** w_1 , ... ,w_k]
[**init**
[*guarded_command*$_1$
...
guarded_command$_m$]]
update
guarded_command$_1$
...
guarded_command$_n$
endatom

u ::= *identifier*[*array_declare*] v ::= u w ::= u

atom_name ::= *identifier*

Table 3.1: Atom syntax

Read variables. The read variables of an atom are the variables whose current value can be read by an atom in order to decide the next values of the controlled variables. Precisely, if a variable x is not read, then the new values of the controlled variables do not depend on the current value of x.

Awaited variables. The awaited variables of an atom are the variables whose *next value* can be read by an atom in order to decide the next value of the controlled variables. A variable cannot be both awaited and controlled: otherwise, in order to determine the next value of the variable, you would have to already know this next value — such a circularity can be often problematic. In order to avoid all such circularity problems, MOCHA keeps track of the *awaits relation* $\succ$: if an atom controls a variable x and awaits a variable y, then $x \succ y$ holds. MOCHA then computes a global relation $\succ_g$ by taking the union of the $\succ$ relations for all the atoms. To avoid await circularities, MOCHA checks that the relation $\succ_g$ is *acyclical.*

Guarded commands. The *guarded_command* statements following the **init** keyword specify the values of the controlled variables at the end of the initial round; the *guarded_command* statements following the **update** keyword specify the values of the controlled variables at the end of an update round. The syntax of guarded commands is presented in Table 3.2. A guarded command statement consists of two parts: a *guard,* that is a boolean expression specifying when the guarded command

guarded_command ::= `[]` *guard* `->` $[command_1; \ldots; command_m]$
guard ::= *boolean_expr* | `default`

Table 3.2: Guarded command syntax

```
atom incrdecr
  controls x
  reads x
init
  [] true -> x' := 0
update
  [] true -> x' := x + 1
  [] true -> x' := x - 1
endatom
```

Figure 3.1: Atom `incrdecr`

can be executed, and a list of *commands,* used to specify the next value of the controlled variables. The special guard `default` will be discussed in Section 3.2.3.

3.2.2 Simple examples of atoms

The atom of Figure 3.1 controls a variable `x`, which we assume to be of type integer (we will explain later how to specify types for variables). The atom specifies that the variable `x` has initially value 0, and that the value of `x` is either decremented or incremented by 1 at each round. Thus, variable `x` performs an (unbounded) random walk. Note that `x` must also be listed among the read variables: otherwise, the value of `x` at the end of the round would not be able to depend on the current value of `x`!

In the atom, the guards of all the guarded commands are always true — in fact, they are equal to the boolean constant `true`. Given a variable x, you can access the current value of x (i.e. the value of x at the beginning of the round) by writing x; you can access the *next* value of x (i.e. the value of x at the end of the round) by writing x'. The guarded commands for the initial round cannot access the current (unprimed) value of the variables, since such value has not been defined yet — the purpose of the initial round is to define such initial value!

The guards of guarded commands need not be mutually exclusive: the guarded command that is executed is selected non-deterministically among the ones whose guards are true. Hence, according to the above atom each state has two successor states: one in which the value of `x` is increased by 1, and one in which the value of `x` is decreased by 1.

You cannot feed the above atom as input to MOCHA: the smallest unit of input

```
module randomwalk
  interface x : int

  atom incrdecr
    controls x
    reads x
  init
    [] true -> x' := 0
  update
    [] true -> x' := x + 1
    [] true -> x' := x - 1
  endatom
endmodule
```

Figure 3.2: Module `randomwalk`

to the MOCHA parser is a module, not an atom. To give the atom as input to MOCHA, you can embed it into a minimal module as shown in Figure 3.2. In this module declaration, variable `x` is declared of type *int* (the MOCHA shorthand for integer), and is listed as an *interface* variable of the module, indicating that the module can modify it, and it is visible from outside of the module as well (so other modules can look at its value).

Unit-delay vs. zero-delay

The module `DelayedAnd` of Figure 3.3 has three interface variables `x1`, `x2`, and `y`, of type `bool`, which is the MOCHA keyword for boolean. Atom `randx` updates `x1` and `x2` non-deterministically at each round: the keyword `nondet` is used to assign to a variable a non-deterministic element of the variable domain (in this case, the domain of `x1` and `x2` is {`true`, `false`}). This non-deterministic update simulates random inputs to our AND gate. Atom `delayedAnd` assigns to the next value of `y` the logical AND of `x1` and `x2`. Abbreviating `true` and `false` with T and F, we can represent one of the possible behaviors of module `DelayedAnd` as follows:

Round:	0 (initial)	1	2	3	5	6	7	8	9	10	···
`x1`:		T	T	F	T	T	F	T	F	T	T	···
`x2`:		T	F	T	T	T	F	F	T	T	F	···
`y`:		F	T	F	F	T	T	F	F	F	T	···

Under index i, for $i > 0$, we list the current values of the variables at round i. Thus, at round 6, the current value of `x1` and `x2` is F, and the current value of `y` is F. As you can see from this example, variable `y` contains the logical AND of `x1` and `x2` with *one round of delay:* in fact, according to module `DelayedAnd` the *next* value of `y` is the AND of the *current* values of `x1` and `x2`. To eliminate this delay, you can await the next values of `x1` and `x2`, and use their next (rather than the current) values

```
module DelayedAnd
  interface x1, x2, y : bool

  atom randx
    controls x1, x2
  init
    [] true -> x1' := nondet; x2' := nondet
  update
    [] true -> x1' := nondet; x2' := nondet
  endatom

  atom delayedAnd
    controls y
    reads x1, x2
  init
    [] true -> y' := nondet
  update
    [] true -> y' := x1 & x2
  endatom
endmodule
```

Figure 3.3: Module `DelayedAnd`

to compute the next value of `y`. This is what module `SynchAnd` of Figure 3.4 does. One of the many behaviors of module module `SynchAnd` is as follows:

Round:	0 (initial)	1	2	3	5	6	7	8	9	10	···
x1:	T	T	F	T	T	F	T	F	T	T	···
x2:	T	F	T	T	T	F	F	T	T	F	···
y:	T	F	F	T	T	F	F	F	T	F	···

The syntax of boolean expressions is presented in Table 3.3. An admissible identifier for a boolean variable is an *identifier* with an optional `'` followed by the optional expression *index_refer*. The presence of an *index_refer* expression indicates the dereferencing of an array structure to a particular element. The exact meaning will be described in Section 3.4.6. The syntax of *int_range_expr* and *nat_expr* expressions will be explained in Section 3.4.1. The construct *event_var*`?` will be explained in Section 3.4.2. The equivalence test between enumeration types will be explained in Section 3.4.4.

The meaning of the boolean operators, together with their precedence, are listed in Table 3.4. Finally, the syntax of the commands is given in Table 3.5. The `forall` construct as well as the syntax of *index_assign* expressions will be described in Section 3.4.6. The syntax of *bitvector_expr* expressions will be described in Section 3.4.7. The *event_var*`!` construct will be described in Section 3.4.2.

boolean_expr ::= *boolean_var* | *event_var*`?` | `(`*boolean_expr*`)` | ∼*boolean_expr*
| *boolean_expr*$_1$ *binary_boolean_op* *boolean_expr*$_2$
| `if` *boolean_expr* `then` *boolean_expr*$_1$
`else` *boolean_expr*$_2$
| *comparison* | `true` | `false`

boolean_var ::= *identifier*[`'`][*index_refer*]

event_var ::= *identifier*

numerical_expr ::= *int_range_expr* | *nat_expr*

comparison ::= *numerical_expr*$_1$ *comparison_op* *numerical_expr*$_2$
| (*element*$_1$ | *enum_var*$_1$) `=` (*element*$_2$ | *enum_var*$_2$)

enum_var ::= *identifier*[`'`][*index_refer*]

binary_boolean_op ::= `&` | `|` | `<=>` | `=>`

comparison_op ::= `>` | `<=` | `=` | `>=` | `<`

Table 3.3: Boolean expression syntax

Operator:	Meaning:
∼	negation
`&`	conjunction
`|`	disjunction
`=>`	implication
`<=>`	equivalence
Precedence:	
∼	Highest
`&`, `|`	Medium
`=>`, `<=>`	Lowest

Table 3.4: Meaning and precedence of boolean operators

```
module SynchAnd
  interface x1, x2, y : bool

  atom randx
    controls x1, x2
  init
    [] true -> x1' := nondet; x2' := nondet
  update
    [] true -> x1' := nondet; x2' := nondet
  endatom

  atom delayedAnd
    controls y
    awaits x1, x2
  init
  update
    [] true -> y' := x1' & x2'
  endatom
endmodule
```

Figure 3.4: Module `SynchAnd`

command	::=	`forall` *i* *x*`'[`*i*`]` `:=` *expr*
		| *x*`'[`*index_assign*`]` `:=` *expr* | `nondet`
		| *event_var*`!`
expr	::=	*boolean_expr* | *numerical_expr* | *bitvector_expr*
x	::=	*identifier*
i	::=	*identifier*

Table 3.5: Command syntax

- *The keyword* `init` *is missing.* Then, the controlled variables of the atom are initialized non-deterministically.
- *The keyword* `init` *is present, but no initial guarded command is present.* Then, the initial round is assumed to coincide with the update round.

Table 3.6: Rules for omitting the `init` keyword, or the initial guarded commands

Rounds and sub-rounds

To understand the semantics of module `SynchAnd` of Figure 3.4, you can imagine each round as consisting of one or more *sub-rounds*. During each subround, one of the atoms updates the variables it controls. The order in which the atoms execute their subrounds is arbitrary, except that if an atom A awaits some variable that is controlled by another atom B, then the sub-round of atom A must precede that of atom B. The order of execution of the sub-rounds does not change the set of possible successor states, provided the above constraint is respected (can you see why?).

Omitting the `init` keyword

Atom delayedAnd of Figure 3.3 shows that you can omit the guarded commands following the `init` keyword: in this case, the initial round is equivalent to the update round, so that atom `delayedAnd` can be equivalently written as follows:

```
atom delayedAnd
  controls y
  awaits x1, x2
init
  [] true -> y' := x1' & x2'
update
  [] true -> y' := x1' & x2'
endatom
```

You can also omit the `init` keyword entirely. In this case, the controlled variables of the atom are initialized non-deterministically, as if they were idle. Table 3.6 summarizes these conventions.

3.2.3 Guards

Using guards, you can constrain the random walk of variable `x` of module `randomwalk` (Figure 3.2) to the interval $[0, 10]$ as shown in Figure 3.5.

```
module randomwalk010
  interface x : (0..10)

  atom incrdecr010
    controls x
    reads x
  init
    [] true -> x' := 0
  update
    [] x < 10  -> x' := x + 1
    [] x > 0 -> x' := x - 1
  endatom
endmodule
```

Figure 3.5: Module `randomwalk010`

The variable `x` has a type that ranges over the integers from 0 to 10, included (type declarations will be discussed in Section3.4). The guards are used to specify when the corresponding command can be executed. In the above example, when $\texttt{x} = 0$, only the first command

```
[] x < 10 -> x' := x + 1
```

can be executed, so that `x` can be incremented but not decremented. Similarly, for $\texttt{x} = 10$ only the second guarded command can be executed. For $0 < \texttt{x} < 10$, both guarded commands can be executed, and `x` can be both incremented and decremented.

Guards can also depend on the *next* value of a variable. The module `CountUp` of Figure 3.6 contains two atoms: an atom `toggle`, whose boolean output `x` changes at arbitrary points in time, and the atom `counter`, which counts with its output `count` the number of "positive fronts" (changes from `false` to `true`) of `x`.

Omitting guards

If no guard is true, then the atom *idles* for one round: this means that all the variables controlled by the atom are idle, and their value is updated according to Table 3.7. The reason for the seemingly strange Rule 2 is that if a variable is not read, its current value is not available to the atom. Therefore, the atom is unable to insure that the updated value for the variable is equal to its current value. To help prevent errors, MOCHA issues a warning whenever a variable is controlled but not read. Using this feature, you can rewrite the previous module `CountUp` as shown in Figure 3.7. Note that atom `counterPrime` is idle whenever variable `x` does not change its value.

```
module CountUp
  interface x : bool; count : int

  atom toggle
    controls x
    reads x
  update
    [] true -> x' := ~x
    [] true -> x' := x
  endatom

  atom counter
    controls count
    reads x, count
    awaits x
  init
    [] true -> count' := 0
  update
    [] ~(x <=> x') -> count' := count + 1
    [] x <=> x' -> count' := count
  endatom
endmodule
```

Figure 3.6: Module `CountUp`

A variable x that is idle in the initial round is updated to a non-deterministic value in its domain. A variable x that is idle in an update round is updated as follows:

1. if x is both controlled and read, then the value of x is left unchanged;
2. if x is controlled but not read, then the value of x at the next round is chosen non-deterministically from the domain of x.

Table 3.7: Updating idle variables

```
module CountUpPrime
  interface x : bool; count : int

  atom toggle
    controls x
    reads x
  update
    [] true -> x' := ~x
    [] true -> x' := x
  endatom

  atom counterPrime
    controls count
    reads x, count
    awaits x
  init
    [] true -> count' := 0
  update
    [] ~(x <=> x') -> count' := count + 1
  endatom
endmodule
```

Figure 3.7: Module `CountUpPrime`

```
module randomwalkabove
  interface x : (0..10); count : int

  atom incrdecr010
    controls x
    reads x
  init
    [] true -> x' := 0
  update
    [] x < 10  -> x' := x + 1
    [] x > 0 -> x' := x - 1
  endatom

  atom countconsec
    controls count
    reads count
    awaits x
  init
    [] true -> count' := 0
  update
    [] x' > 5  -> count' := count + 1
    [] default -> count' := 0
  endatom
endmodule
```

Figure 3.8: Module `randomwalkabove`

The `default` guard

If you want a guarded command to be executed when the guards of all other commands are false, you can use the keyword `default` as guard. For example, suppose that you want to count the number of consecutive times (including the current one) in which the random walk of module RANDOMWALK010 (Figure 3.5) is greater than 5. You can do this as shown in Figure 3.8.

Omitting variable updates

If an atom controls more than one variable, it is possible to omit some of the variable updates from the guarded commands. If the guarded command is selected, the variables whose updates are omitted are idle, and their value is updated according to Table 3.7. This convention often enables a considerable space saving. Module `GrayCode` of Figure 3.9 generates outputs `x` and `y`, that change cyclically following the Gray code sequence 00, 01, 11, 10. The module uses a variable `pc` that is *private* to the module: it is visible only from within the module itself.

```
module GrayCode
  interface x, y : (0..1)
  private pc : (0..3)

  atom gray
    controls x, y, pc
    reads    x, y, pc
  init
    [] true -> x' := 0; y' := 0; pc' := 0
  update
    [] pc = 0 -> pc' := 1; x' := 1
    [] pc = 1 -> pc' := 2; y' := 1
    [] pc = 2 -> pc' := 3; x' := 0
    [] pc = 3 -> pc' := 0; y' := 0
  endatom
endmodule
```

Figure 3.9: Module `GrayCode`

Lazy atoms

A *lazy atom* is an atom whose controlled variables can remain unchanged during any update round. Lazy atoms can be declared with the help of the keyword `lazy` (see Table 3.1). Precisely, using the keywork `lazy` is equivalent to adding the update guarded command with empty body:

```
[] true ->
```

to the atom. When the `lazy` keyword is used, the atom must read all the variables it controls. Using keyword `lazy`, you can encode module `CountUpPrime` (Fig 3.7) in the alternative way of Figure 3.10. Since atom `toggleSecond` can sleep at any time, the guarded command `[] true -> x' := x` is no longer necessary. However, to insure that variable `x` retains its value when the atom sleeps, it is now necessary to declare it as read (see Table 3.7).

3.3 Modules

A *module* is a collection of atoms, together with a declaration of the variables that occur in the module. There are two types of modules: *simple modules,* obtained by specifying directly the atoms composing the module, and *composite modules,* obtained by combining or modifying existing modules.

3.3.1 Simple Modules

The syntax of simple modules is given in Table 3.8. After the keyword `module`, you must provide the module name *module_name*. The module name is followed by a

```
module CountUpSecond
  interface x : bool; count : int

  lazy atom toggleSecond
    controls x
    reads x
  update
    [] true -> x' := ~x
  endatom

  atom counterPrime
    controls count
    reads x, count
    awaits x
  init
    [] true -> count' := 0
  update
    [] ~(x <=> x') -> count' := count + 1
  endatom
endmodule
```

Figure 3.10: Module `CountUpSecond`

simple_module ::= `module` *module_name*
in_out_decl$_1$
...
in_out_decl$_m$
atom$_1$
...
atom$_n$
`endmodule`

in_out_decl ::= (`private` | `interface` | `external`)
var_decl$_1$; ... ; *var_decl*$_k$

var_decl ::= x_1 , ... , x_i : *type*

module_name ::= *identifier*

Table 3.8: Simple module syntax

list of variable declarations, which is followed in turn by the list of atoms composing the module. The keyword `endmodule` concludes the module specification.

Associated with each module are three sets of variables: the *private variables,* the *interface variables,* and the *external variables.* These sets of variables have the following meaning:

- The private variables are the variables that are controlled by some atom of the module, and that cannot be read or awaited by other modules. The value of a private variable is thus local to the module.
- The interface variables are the variables that are controlled by some atom in the module, and that can be read or awaited by atoms in other modules. These variables cannot however be controlled by atoms of other modules, according to the general rule stating that a variable can be controlled by at most one atom.
- The external variables are the variables whose value can be read or awaited by the atoms in the module. These variables cannot be controlled by any atom in the module.

Each of the private and interface variables of the module must be controlled by some atom. External variables, on the other hand, do not need to be read nor awaited by any atom. Again, the awaits relation $\succ_g$ computed for the module must be acyclical.

The three sets of private, interface and external variables, together with the type of each variable, are specified using the syntax given in Table 3.8. The symbol *type* in the table indicates a *type declaration:* two possible declarations are `int` (for integer type) and `bool` (for boolean type). Type declarations will be covered in Section 3.4.

You have seen examples of module declarations in the previous section as well as the tutorial chapter; additional examples will be presented in Section 3.7.

3.3.2 Composite Modules

There are three operations defined on modules in MOCHA: *variable hiding,*, *variable renaming,* and *parallel composition.* These operations create new *composite* modules, which can in turn be combined into other composite modules.

Hiding Variables

The hiding of interface variables allows you to construct module abstractions of varying degrees of detail. For instance, after composing two modules, it may be appropriate to convert some interface variables to private variables, so that they are used only for the interaction of the component modules, and are no longer visible to the environment of the compound module.

Given a module (simple or composite) with name P, denote with $External(P)$, $Private(P)$, $Interface(P)$ the set of external, private, and interface variables of P, respectively. Given any list of interface variables $x_1, \ldots, x_n \in Interface(P)$, you can *hide* $x_1, \ldots, x_n$ using the construct:

Q := **hide** x_1 , ... ,x_n **in** P **endhide**

The resulting module Q is identical to P, except that:

$$\begin{aligned} \mathit{Private}(Q) &= \mathit{Private}(P) \cup \{x_1, \ldots, x_n\} \\ \mathit{Interface}(Q) &= \mathit{Interface}(P) \setminus \{x_1, \ldots, x_n\}. \end{aligned}$$

In other words, the effect of hiding the interface variables $x_1, \ldots, x_n$ in P is to make them private, so that other modules cannot access their content.

Variable Renaming

The renaming operation is useful for creating different instances of a module, and for avoiding name conflicts. Let P be the name of a module, and let x be a variable (external, private, or interface) of a module. You can rename variable x to y with the construct:

Q := P [x := y]

The resulting module Q is identical to P, except that variable x has been renamed to y. The new variable y is of the same type of x, and it does not need to be explicitly declared: MOCHA infers its type and class (external, private, or interface) from the renaming operation. You can also rename several variables at once: the construct

Q := P [x_1 , ... ,x_m := y_1 , ... ,y_m]

simultaneously renames the variables $x_1, \ldots, x_m$ to $y_1, \ldots, y_m$. This simultaneous renaming is especially useful to exchange variables. Note that the variables $y_1, \ldots, y_m$ must be all distinct — even if they are external variables of the module of the same type, in which case renaming them to the same name would make sense (it would correspond to drawing the various inputs from the same source). For instance, if a module P has two variables x_1 and x_2 and we want to exchange them, the following

Q := P [x_1 := x_2] [x_2 := x_1]

leaves Q and P being identical except that in Q the variable x_2 is mapped to x_1. The correct way to exchange them is

Q := P [x_1, x_2 := x_2, x_1]

MOCHA propagates the variable renames to the awaits relation $\succ$ in the obvious way.

Parallel composition

You can use parallel composition to combine two modules into a single module whose behavior captures the interaction between the two component modules. Two modules with names P and Q are *compatible* if the following conditions hold:

program	::=	*module_decl* | *type_def* | (*module_decl* | *type_def*) *program*
module_decl	::=	*simple_module* | *module_name* `:=` *composite_module*
composite_module	::=	*module_name*
		| `(`*composite_module*`)`
		| `hide` x_1`,` ... `,`x_n `in` *composite_module* `endhide`
		| *composite_module* `[` x_1`,` ... `,`x_m `:=` y_1`,` ... `,`y_m `]`
		| *composite_module*$_1$ `||` *composite_module*$_2$

Table 3.9: Syntax of module declarations

- the sets of interface variables of P and Q are disjoint, i.e. $Interface(P) \cap Interface(Q) = \emptyset$;
- the global awaits relation $\succ_g$ for the two modules is acyclic.

If P and Q are compatible, you can form their parallel composition (and give it name R) with the construct:

R `:=` P `||` Q

The external, private, and interface variables of R are given by:

$$\begin{aligned} Private(R) &= Private(P) \cup Private(Q) \\ Interface(R) &= Interface(P) \cup Interface(Q) \\ External(R) &= [External(P) \cup External(Q)] \setminus Interface(R) \end{aligned}$$

Note that $External(R)$ is equivalent to $[External(P) \setminus Interface(Q)] \cup [External(Q) \setminus Interface(P)]$ since the sets of interface variables of P and Q are disjoint.

Summary of module declarations

Table 3.9 summarizes the possible module declarations. The non-terminal *program* is the start symbol of the grammar for ReactiveModules. The expression *type_def* will be described in Section 3.4. Note that while the name of an atom is used only for mnemonic purposes, the name of a module is used by Mocha to construct more complex modules. In fact, the names of modules will also be mentioned by verification commands, as you will see later.

As an example, we give two different approaches to constructing an OR gate. The first approach, sometimes called the *behavioral* approach, specifies the behavior of the OR gate directly:

int_range_expr	::=	*int_nat_constant* | *int_nat_var*
		| `(`*int_range_expr*`)` | `-`*int_range_expr*
		| *int_range_expr*$_1$ `+` *int_range_expr*$_2$
		| *int_range_expr*$_1$ `-` *int_range_expr*$_2$
		| `if` *boolean_expr* `then` *int_range_expr*$_1$ `else` *int_range_expr*$_2$
int_nat_constant	::=	*constant*
int_nat_var	::=	*identifier*[`'`][*index_refer*]

Table 3.10: Syntax of integer and range expressions

```
module BehavOr
  external in1, in2: bool
  interface out: bool

  atom controls out awaits in1, in2
  update
      [] true -> out' := in1' | in2'
  endatom
endmodule
```

In the second approach, shown in Figure 3.11, we assume that all we have at our disposal is two-input NAND gates, and we create an OR gate by connecting these NAND gates in the appropriate way. This approach is sometimes called the *structural* approach, because it describes a system by specifying its structure, rather than directly its behavior. Note that we cannot obtain a NOT gate simply by renaming the two inputs `in1` and `in2` of the NAND gate to the same name NORIN: we must introduce a splitter module SPLIT, which makes two signals out of the same signal.

3.4 Types and Expressions

The simplest types of MOCHA are boolean, integer and natural. Other types include events, ranges, enumerations, arrays, and bitfields. Boolean variables are declared with the help of the keyword `bool`, and the syntax and semantics of boolean expressions has already been described in Tables 3.3 and 3.4.

```
module nand
  external in1, in2 : bool
  interface out : bool

  atom controls out awaits in1, in2
  init update
    [] true -> out' := ~ (in1' & in2')
  endatom
endmodule

module split
  external in : bool
  interface out1, out2 : bool

  atom controls out1, out2 awaits in
  init update
    [] true -> out1' := in'; out2' := in'
  endatom
endmodule

StructOr := hide in1a, in1b, in2a, in2b, cin1, cin2 in
     split [in, out1, out2 := in1, in1a, in1b]
  || split [in, out1, out2 := in2, in2a, in2b]
  || nand  [in1, in2, out  := in1a, in1b, cin1]
  || nand  [in1, in2, out  := in2a, in2b, cin2]
  || nand  [in1, in2, out  := cin1, cin2, out]
  endhide
```

Figure 3.11: Modules for structural OR

nat_expr	::=	*int_nat_constant* | *int_nat_var*
		| (*nat_expr*) | nat_expr_1 + nat_expr_2
		| if *boolean_expr* then nat_expr_1
		else nat_expr_2

Table 3.11: Syntax of natural expressions

3.4.1 Naturals and integers

Integer variables are declared using the keyword `int`; their domain ranges over the integers. Natural variables are declared using the keyword `nat`; their domain ranges over the non-negative integers. The syntax of integer (and range) expressions is given in Table 3.10; the syntax of natural expressions is given in Table 3.11. A *constant* is a numerical string. Natural variables can be assigned to integer ones, and they can be used in integer expressions; the converse is not permitted.

3.4.2 Events

In MOCHA, events are represented by toggling the value of boolean variables. A variable that is used to represent events can be using the keyword `event`. This has two advantages. First, it makes it easier to toggle the variable, and detect the toggling. Second, and more importantly, it notifies MOCHA that the value of the variable is irrelevant: the only thing that matters is whether the value has been toggled or not. Therefore, the presence of an additional event variable does not increase the size of the state space. This will be explained more in detail in Section 3.6.

If `x` is an event variable, then we can toggle it with the command `x!`, and we can test whether it has been toggled with the boolean expression `x?` (see Table 3.3). Aside from these two operations, there are no expressions of event type, and events cannot be compared using the operators `>`, `<`, etc. The simple module `EventCount` of Figure 3.12 consists of two atoms: an atom `generate` that generates events, and an atom `counts` that counts them. Note that the atoms that refer to event variables must also read them, to be able to distinguish when they are toggled.

3.4.3 Range types

A range declaration in MOCHA has the form `(0..`*max*`)`, where *max* is a non-negative integer constant. The expressions using range types share the same syntax of integer expressions, given in Table 3.10. In these expressions, you must follow the restrictions:

1. You cannot mix different range types (or range types and `int` or `nat`) in the same expression.

2. You cannot assign an expression of a range type to a different range type, or to an `int` or `nat`.

Arithmetic operations are performed modulo *max*+1 on the range type, thus insuring that their value belongs to the same range type of the operands. There is no type-casting operator in MOCHA, making the translation between different range types cumbersome. A module that translates a variable of type `(0..3)` to integers is given in Figure 3.13: you see that you don't want to be doing this all the time.

```
module EventCount
  external e : event
  external c : nat

  lazy atom generate
    controls e
    reads e
  update
    [] true -> e!
  endatom

  atom counts
    controls c
    reads c, e
    awaits e
  init
    [] true -> c' := 0
  update
    [] e? -> c' := c + 1
  endatom
endmodule
```

Figure 3.12: Module `EventCount`

```
module RangeTranslation
  external x : (0..3)
  interface y : int

  atom translate
    controls y
    awaits x
  init update
    [] x' = 0 -> y' := 0
    [] x' = 1 -> y' := 1
    [] x' = 2 -> y' := 2
    [] x' = 3 -> y' := 3
  endatom
endmodule
```

Figure 3.13: Module `RangeTranslation`

```
module UpDownWalk
  private ud : { up, down }
  interface c : int

  atom generate
    controls ud
  update
    [] true -> ud' := nondet
  endatom

  atom count
    controls c
    reads c
    awaits ud
  init
    [] true -> c' := 0
  update
    [] ud' = up   -> c' := c + 1
    [] ud' = down -> c' := c - 1
  endatom
endmodule
```

Figure 3.14: Module `UpDownWalk`

3.4.4 Enumeration type

To declare a variable of enumeration type, you simply list between curly braces the possible values of the variable, separated by commas. The only operations you can perform on an enumeration type are checking for equality (using operator `=`), and assignments. The minimal module `UpDownWalk` of Figure 3.14 contains two atoms: one that generates an infinite sequence of ups and downs, and another that counts up or down, as dictated by the former atom.

A value in an enumeration cannot be used in more than one distinct range type. For example, in Figure 3.15 it was necessary to capitalize the values `Water` and `Pasta` for the range of `z` in order to avoid using twice the values for the range of `x`. On the other hand, note that it is possible to use the same enumeration value in *the same* range type multiple times, as illustrated in Figure 3.16.

3.4.5 Naming types

Equality of types in MOCHA is defined structurally (as opposed to being based on inheritance). This means that two variables have the same type iff they are declared of structurally equivalent types, even though the type declarations may be distinct. For example, the module `EnumAssign` of Figure 3.16 is correct: `y'` can be assigned `x'`. You can also associate a name with enumeration and range types (but not with

```
module RecycleNames
  external x : { water, bread, pasta }
  interface z : { Water, Pasta, Bananas }

  atom copy
    controls z
    awaits x
  init update
    [] x' = water -> z' := Water
    [] x' = pasta -> z' := Pasta
  endatom
endmodule
```

Figure 3.15: Module `RecycleNames`

```
module EnumAssign
  external x : { water, bread, pasta }
  interface y : { water, bread, pasta }

  atom copy
    controls y
    awaits x
  init update
    [] true -> y' := x'
  endatom
endmodule
```

Figure 3.16: Module `EnumAssign`

```
type menu : { water, bread, pasta }

module EnumAssignBis
  external x : menu
  interface y : menu

  atom copy
    controls y
    awaits x
  init update
    [] true -> y' := x'
  endatom
endmodule
```

Figure 3.17: Module `EnumAssignBis`

integer, natural, boolean, and event types). To do so, you include the declaration

`type` *type_name* `:` *complex_type*

in the input file, before the modules that use type *type_name*. The type *complex_type* must be an enumeration, range, array, or bitvector type. Using this declaration makes it easier to write structurally equal types, and also makes the code more readable. Using this type declaration, you can rewrite the example of Figure 3.16 as shown in Figure 3.17.

3.4.6 Arrays

MOCHA also has array and bitvector types. Arrays are essentially as in ordinary programming languages, except that multidimensional arrays are not provided, and other minor limitations are present. Each array variable has to be controlled by the same module, but different array elements can be controlled by different atoms: hence, you can think of arrays as a bus of wires or signals, not necessarily controlled jointly. The syntax for declaring an array type is:

`array` *index_type* `of` *element_type*

Type *index_type* is the type of the array index, and can be an enumeration type or a range type (perhaps the most common case). Type *element_type* is the type of the array elements, and can be a boolean, integer, natural, enumeration, range, or bitvector type. An example of an array declaration is

```
interface x :  array (0..10) of bool
```

Table 3.12 describes the constructs to express dereferencing of arrays in the context of variable declarations in atoms as well as in expressions and assignments.

You can declare that an atom controls the whole array `x` by writing `controls x`. If the atom controls only some elements of `x`, you can specify these simply by listing them, as in `controls x[0], x[2], x[4]`. You can specify which elements are read or awaited in a similar way. Note that the array elements that are controlled, read, or awaited must be specified using constants or elements of enumeration types only: you cannot write `controls x[1 + 2]` or `awaits x[a]` if `a` does not denote an element of an enumeration type. If an array is controlled by the module (i.e. if it is declared as `interface` or `private` in the module), then all array elements must be controlled by some atom of the module.

You can refer to array elements by writing the variable name followed by a range expression in square brackets, as in `x[4]` or `x[z + 1]`. In an expression, you can refer to an element of an array of type `array` *index_type* `of` *element_type* whenever you can refer to a variable of type *element_type*.

You can assign values to individual array elements by using constants to specify them or use the forall assignment to all elements of the array together. Moreover, you must specify the array element being updated using constants only: for example, you can write `x'[3] := true`, but not `x'[y] := true` or even `x'[3+1] := true`. Thus, there is no direct way to update entry `a` of array `x`. But, indirectly the same effect can be achieved by using the forall assignment statement as described below.

The forall assignment is used to assign values simultaneously to all array entries. An example of the forall assignment is

```
forall i x'[i] := if (i = a) then y else x[i] fi
```

`i` is the index variable that is bound by the forall statement. The effect is to assign to entry `i` the value of the expression on the right side with the value of `i` subsituted for `i`. Notice also that the above example is just another way of saying

```
x'[a] := y
```

Note that forall assignments may only occur in *command* expressions as given in Table 3.5.

Modulo arithmetic is used for indexing whenever the index type of an array is a range type, e.g., if array `x` has index type `(0..3)`, `x[3+3]` refers to `x[2]`. An array `x` can be indexed only by a variable of identically the same type as its index type. For example, an array with index type `(0..7)` cannot be indexed with a var of type int, or type `(0..5)`.

An atom can read or await parts of an array, but then no expression in the atom can index the array by a variable. Similarly, if an atom controls only part of the array it can only update array elements specified by constants.

3.4.7 Bitvectors

Bitvectors are essentially arrays of booleans, except that you can perform arithmetic and logical operations on a bitvector as a whole, and as a consequence each bitvector variable has to be controlled by a single atom. The index type of a bitvector is always a range type. An example of a declaration is

array_declare ::= `[`*int_nat_constant* | *element*`]`

index_refer ::= `[`*numerical_expr* | *enum_var* | *element*`]`

index_assign ::= *array_declare*

Table 3.12: Array access syntax

bitvector_expr ::= *int_nat_constant* | *bitvector_var*
| `(`*bitvector_expr*`)` | `-`*bitvector_expr*
| *bitvector_expr*$_1$ `+` *bitvector_expr*$_2$
| *bitvector_expr*$_1$ `-` *bitvector_expr*$_2$
| $\sim$*bitvector_expr*
| *bitvector_expr*$_1$ *binary_boolean_op bitvector_expr*$_2$
| `if` *boolean_expr* `then` *bitvector_expr*$_1$
`else` *bitvector_expr*$_2$

bitvector_var ::= *identifier*`['][`*index_refer*`]`

Table 3.13: Bitvector expression syntax

```
interface x :  bitvector 8
```

Here, `x` has eight elements indexed by the range type `(0..7)`. The rules for reading or awaiting parts of bitvector variables are identical to those for arrays. It is not possible for different atoms to control parts of the same array; the whole bitvector has to be controlled by the same atom. The forall assignment can be used with bitvectors also.

The operations available for bitvectors are arithmetic operations of addition and subtraction (modulo 2^k for length k bitvectors), and bitwise logical operations `&`, `|`, $\sim$, `=>`, and `<=>`. Table 3.13 provides the syntax of *bitvector_expr* expressions. Two bitvectors can be added (similarly for other operations) only if they have the same lengths. Constants can also be used as an argument to these operations. In such expressions, the constant is interpreted as a constant bitvector of the length of the other argument. Let `x`, `y`, `z` be bitvectors of length three.

```
z' := (x & y) + 5
```

The numeral 5 in the above expression will be interpreted as the constant bitvector `101`.

3.4.8 Summary of type declarations

Table 3.14 provides a summary of the syntax of the types available in MOCHA. The *element* symbol indicates an alphanumerical string (beginning with a letter) denoting an element of an enumeration type. The following restrictions apply:

- If the type *index_type* of an array index is specified by a previously-defined type name *type_name*, then *type_name* cannot refer to another array type.
- If the type *element_type* of an array element is specified through a previously defined type name *type_name*, then *type_name* cannot refer to another array or bitvector type. In particular, this rules out arrays of arrays (and thus multidimensional arrays) and arrays of bitvectors.

For arithmetic expressions on *size_constant*: $*$, /, and % have equal precedence, that is higher than $+$ and $-$ which have equal precedence. Equal precedence operations are left-associative.

3.4.9 Finite and infinite types in verification

A type is *finite* if it consists of finitely many possible values, and it is *infinite* otherwise. Examples of finite types are boolean, enumeration, range, and event types. The infinite types are integer and natural, as well as the composite types (arrays) built from integers or naturals. MOCHA can deal much better with finite types than with infinite ones. In particular, if a REACTIVEMODULES description consists of only finite types, then MOCHA can use both enumerative and symbolic model-checking for the verification. If also infinite types are used, then only enumerative model-checking can be used. This can prevent the use of some of the most efficient verification methods implemented in MOCHA.

type ::= **bool** | **int** | **nat** | **event** | *type_name*
| *enum_type* | *range_type*
| **array** *index_type* **of** *element_type*
| **bitvector** *size_constant*

type_name ::= *identifier*

enum_type ::= {$element_1$, ... , $element_n$}

element ::= *identifier*

range_type ::= **(0 ..** *size_constant***)**

type_def ::= **type** *type_name* **:** *enum_type*
| **type** *type_name* **:** *range_type*
| **type** *type_name* **:** **array** *index_type* **of** *element_type*
| **type** *type_name* **:** **bitvector** *size_constant*

index_type ::= *enum_type* | *range_type*

element_type ::= **bool** | **int** | **nat** | *type_name*
| *enum_type* | *range_type*
| **bitvector** *size_constant*

size_constant ::= *int_nat_constant*
| **(***size_constant* **)**
| $size\_constant_1$ **-** $size\_constant_2$
| $size\_constant_1$ **+** $size\_constant_2$
| $size\_constant_1$ ***** $size\_constant_2$
| $size\_constant_1$ **/** $size\_constant_2$
| $size\_constant_1$ **%** $size\_constant_2$

Table 3.14: Syntax of types and type definitions

3.5 Macro expansion

The parser for ReactiveModules provides a limited macro-expansion facility.

The declaration:

```
#define name string
```

defines a macro `name` that refers to the string `string`. Any expression that involves `$name` will be evaluated after first substituting `string` for it.

For instance the code fragment:

```
#define BITWIDTH 4
type valType      : bitvector  $BITWIDTH
```

defines valType to be a bitvector of 4 bits.

The other macro allowed is the `foreach` macro.

```
#foreach i = (1 .. $BITWIDTH + 1)
type valType_$i : bitvector $i
#endforeach
```

The above code fragment defines four types: `valType_1`, `valType_2`, `valType_3`, and `valType_4`. The index string may be of index_type (see Table 3.14), but cannot be a type_name; it has to be the type itself. That is,

```
type food : {pasta, food, water}
#foreach i = food
#endforeach
```

is not allowed, but `foreach i = {pasta, food, water}` is fine.

3.6 Efficiency Considerations

The efficiency of the symbolic model-checking methods of Mocha depends crucially on the size of the state space, i.e. on the total number of system states. A *system state* is simply an assignment of values to the variables (external, interface, and private) of the ReactiveModules description. Several factors affect the size of the state space. First, if the description includes variables of an infinite type, then the state space is infinite, and the symbolic model-checking cannot be performed.

If all the variables are finite, the size of the state space is proportional to the (product of) size constants used to dimension arrays and range types, and to the number of values for enumeration types. While this, and similar facts, are well-known in model-checking, the distinction between read and awaited variables enables Mocha to limit the size of the state space. In particular, variables that are awaited but not read, and event variables, do not contribute to the size of the state space.

3.6.1 Awaited, but not read, variables

If a variable is never read, but only awaited, then the variable does not contribute to the size of the state space. For example, the variables `in1a`, `in1b`, `in2a`, `in2b`, `cin1`, `cin2` of Figure 3.11 do not contribute to the size of the state space: they are not read by any of the modules of the figure, and since they are hidden, we are sure that no other module can read them. Such variables are also called *history-free*. The variables `in1`, `in2` and `out` may or may not contribute to the size of the state space, depending on whether or not they are read by some other module.

Variables that are not read, but only awaited, are often used to model wires of hardware components. The reason why these variables are not read is that these variables, like wires, have no memory, and thus their state needs not be remembered. In fact, when computing the transition relation of the system, the values of these variables in the current state are not used to determine the successor state.

Variables that are read are also called *history-dependent* variables.

3.6.2 Event variables

Event variables also do not contribute to the size of the state space, even though they must be both read and awaited by all atoms using them. The reason is that they current value (which would contribute to the size of the state space) is not relevant: all that matters is whether they retain their value, or whether they are toggled, when going from one state to the next. Hence, as in the previous case, the values of event variables at the current state are not used to determine the successor state, and hence they need not be remembered.

3.7 More examples

3.7.1 Synchronous message-passing protocols

The modules `Sender` and `Receiver` of Figure 3.18 communicate via events in order to transmit a stream of messages.

The private variable `pc` of the sender indicates if it is producing a message (`pc` = `produce`), or attempting to send a message (`pc` = `send`). The private variable `pc` of the receiver indicates if it is waiting to receive a message (`pc` = `receive`), or consuming a message (`pc` = `consume`). Messages are produced by the atom `AProd`, which requires an unknown number of rounds to produce a message. Once a message is produced, the event `doneP` is issued, and the message is shown as `msgP` (the actual value of message is chosen non-deterministically from the finite type `msgType`). Once a message has been produced, the sender is ready to send the message, and `pc` is updated. When ready to send a message, the sender sleeps until the receiver becomes ready to receive, and when ready to receive a message, the receiver sleeps until the sender transmits a message.

The synchronization of both agents is achieved by two-way handshaking in three subrounds within a single update round. The first subround belongs to the receiver.

```
type msgType :  bool
type sendCtrlType :  {produce, send}
type recCtrlType :  {receive, consume}
module Sender
  external ready :  event
  interface transmit :  event; msgS, msgP :  msgType
  private pc :  sendCtrlType; doneP :  event
  atom controls pc, transmit, msgS
    reads pc, transmit, msgS, doneP, msgP, ready awaits doneP, ready
  init
    [] true -> pc' := produce
  update
    [] pc=produce&doneP? -> pc' := send
    [] pc=send&ready?    -> transmit!; msgS' := msgP; pc' := produce
  endatom
  lazy atom AProd controls doneP, msgP reads pc, doneP, msgP
  update
    [] pc=produce -> doneP!; msgP' := nondet
  endatom
endmodule

module Receiver
  external transmit :  event; msgS :  msgType
  interface ready :  event; msgC :  msgType
  private pc :  recCtrlType; doneC :  event; msgR :  msgType
  atom controls pc, msgR
    reads pc, transmit, doneC awaits transmit, msgS, doneC
  init
    [] true -> pc' := receive
  update
    [] pc=receive & transmit? -> msgR' := msgS'; pc' := consume
    [] pc=consume & doneC?    -> pc' := receive
  endatom
  lazy atom controls ready reads pc, ready
  update
    [] pc=receive -> ready!
  endatom
  lazy atom ACons controls doneC, msgC reads pc, doneC, msgR
  update
    [] pc=consume -> doneC!; msgC' := msgR
  endatom
endmodule
```

Figure 3.18: Synchronous message-passing protocol

If the receiver is ready to receive a message, it issues the interface event `ready` to signal its readiness to the sender. The second subround belongs to the sender. If the sender sees the external event `ready` and is ready to send a message, it issues the interface event `transmit` to signal a transmission. The third subround belongs to the receiver. If the receiver sees the external event `transmit`, it copies the message from the external variable `msgS` to the private variable `msgR`. The sender goes on to wait for the production of another message, and the receiver goes on to consume `msgR`. Messages are consumed by the atom `ACons`, which requires an unknown number of rounds to consume a message. Once a message is consumed, the event `doneC` is issued, the consumed message is shown as `msgC`, and the receiver waits to receive another message.

3.7.2 A train controller

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

Figure 3.19 shows the Reactive Modules description for the a generic train `Train`. The module has three interface variables: `arrive` and `leave` of type event and `pc` of enumerative type $\{\texttt{away}, \texttt{wait}, \texttt{bridge}\}$; and one external variable `signal`. The module has only one atom which controls all the interface variables. By declaring the atom to be lazy, we are able to model the assumption regarding independence of the speeds of different modules.

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

Figure 3.19 also shows a controller controlling the signals to prevent collisions of the two trains. The complete railway system is represented by the module `RailroadSystem`, which is the composition of the trains with the controller, with variables `arriveW`, `arriveE`, `leaveW` and `leaveE` hidden.

```
module Train
  interface pc :  {away, wait, bridge}; arrive, leave :  event
  external signal :  {green, red}
  lazy atom controls pc, arrive, leave reads pc, arrive, leave, signal
  init
    [] true -> pc' := away
  update
    [] pc=away                -> arrive!; pc' := wait
    [] pc=wait & signal=green -> pc' := bridge
    [] pc=bridge              -> leave!; pc' := away
  endatom
endmodule
TrainW := Train[pc,arrive,signal,leave := pcW,arriveW,signalW,leaveW]
TrainE := Train[pc,arrive,signal,leave := pcE,arriveE,signalE,leaveE]

module Controller
  private nearW, nearE :  bool
  interface signalW, signalE :  {green, red}
  external arriveW, arriveE, leaveW, leaveE :  event
  atom controls nearW reads nearW, arriveW, leaveW awaits arriveW, leaveW
  init
    [] true -> nearW' := false
  update
    [] arriveW? -> nearW' := true
    [] leaveW?  -> nearW' := false
  endatom
  atom controls nearE reads nearE, arriveE, leaveE awaits arriveE, leaveE
  init
    [] true -> nearE' := false
  update
    [] arriveE? -> nearE' := true
    [] leaveE?  -> nearE' := false
  endatom
  lazy atom controls signalW, signalE reads nearW, nearE, signalW, signalE
  init
    [] true -> signalW' := red; signalE' := red
  update
    [] nearW&signalE=red -> signalW' := green
    [] nearE&signalW=red -> signalE' := green
    [] ~nearW            -> signalW' := red
    [] ~nearE            -> signalE' := red
  endatom
endmodule

RailroadSystem := hide arriveW, arriveE, leaveW, leaveE in TrainW || TrainE
|| Controller endhide
```

Figure 3.19: Railroad controller

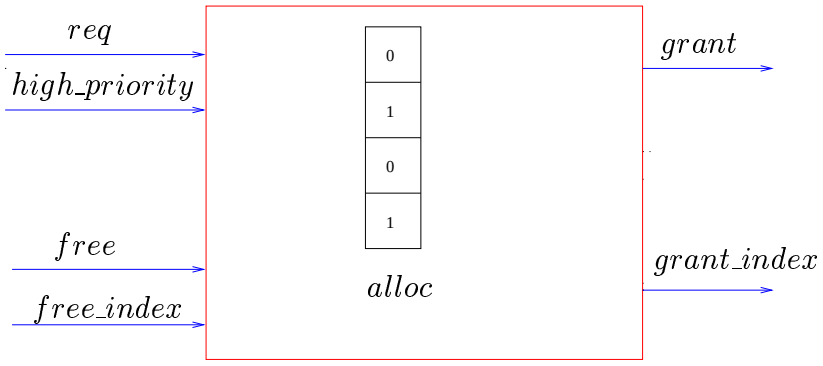

Figure 3.20: Resource Manager

3.7.3 A resource manager

Consider a resource manager used to resolve accesses to instances of a shared resource. A block diagram of a resource manager is given in Figure 3.7.3. There are four instances of the resource available. The environment interacts with the resource manager to allocate and free resources. If a resource is available, the resource manager responds by granting an instance of the resource.

Figure 3.21 shows a reactive modules description of the resource manager. The module `RManager` receives allocate and free requests through the boolean external variables `req` and `free` respectively. The environment also sets the external variable `highPriority` to `true` in the case of a high-priority request. For a normal-prioroty request, `highPriority` is set to `false`. The manager may grant a high-priority request, even if only one instance of the resource is available. A normal-priority request, on the other hand may be granted only if at least two instances of the resource are available. The module `RManager` signals grants to the environment through the interface variable `grant` of type `bool`. The interface variable `grantIndex` is set to the index of the resource that is granted. While freeing the resource, the environment is expected to set the external variable `freeIndex` to the appropriate index of the resouce that is being freed.

Private variable `alloc`(a boolean array) is used to maintain a bit for each resource to indicate if it is free or allocated. The history-free private variable `halfEmpty` of type `bool` is set to true if at least two instances of the resource are free. It is used in handling normal-priority requests.

Module `RManagerImpl` in Figure 3.22 is a "less-abstract" lower-level description of the resource allocator, Note that the modules `RManager` and `RManagerImpl` have same set of external and interface variables. However, some design decisions have been taken to implement the non-deterministic choices allowed in `RManager`. The

```
type indexType : (0..3)
type sindexType : (0..4)
type regType : array indexType of bool

module Rmanager
  external req : bool; free : bool; free_index : indexType;
           high_priority : bool
  interface grant : bool; grant_index : indexType; alloc : regType;
            half_empty : bool
  atom ALLOC controls alloc
       reads alloc awaits req, grant, grant_index, free, free_index
  init
    [] true -> forall i alloc'[i] := false
  update
    [] true -> forall i alloc'[i] :=
       if(grant' & grant_index'=i) then true else
       if (free' & free_index'=i) then false else alloc[i] fi fi
  endatom
  atom HALF_EMPTY controls half_empty reads alloc
  init
    [] true -> half_empty' := true
  update
    [] ~alloc[0] -> half_empty' :=
       if (~alloc[1] | ~alloc[2] | ~alloc[3]) then true else false fi
    [] ~alloc[1] -> half_empty' :=
       if (~alloc[0] | ~alloc[2] | ~alloc[3]) then true else false fi
    [] ~alloc[2] -> half_empty' :=
       if (~alloc[0] | ~alloc[1] | ~alloc[3]) then true else false fi
    [] ~alloc[3] -> half_empty' :=
       if (~alloc[0] | ~alloc[1] | ~alloc[2]) then true else false fi
    [] default -> half_empty' := false
  endatom
  atom GRANT_INDEX controls grant_index
  init update
    [] true -> grant_index' := nondet
  endatom
  atom GRANT controls grant
       reads alloc awaits req, high_priority, half_empty, grant_index
  init
    [] true -> grant' := false
  update
    [] req' & high_priority' & ~alloc[grant_index'] -> grant' := true
    [] req' & half_empty' & ~alloc[grant_index'] -> grant' := true
    [] default -> grant' := false
  endatom
endmodule
```

Figure 3.21: Resource Manager (Specification)

```
module RManagerImpl
  external req : bool; free : bool; free_index : indexType;
           high_priority : bool
  interface grant : bool; grant_index : indexType; alloc : regType;
           half_empty : bool
  private sum : sindexType

  atom ALLOC controls alloc reads alloc
             awaits grant, grant_index, free, free_index
  init
    [] true -> forall i alloc'[i] := false
  update
    [] true -> forall i alloc'[i] :=
       if (grant' & grant_index'=i) then true else
       if (free' & free_index'=i) then false else alloc[i] fi fi
  endatom
  atom SUM controls sum reads sum, alloc awaits grant, free, free_index
  init
    [] true -> sum' := 0
  update
    [] grant' & ~(free' & alloc[free_index']) -> sum' := sum + 1
    [] ~grant' & (free' & alloc[free_index']) -> sum' := sum - 1
  endatom
  atom HALF_EMPTY controls half_empty reads sum
  init
    [] true -> half_empty' := true
  update
    [] sum > 2 -> half_empty' := false
    [] default -> half_empty' := true
  endatom
  atom GRANT_INDEX controls grant_index reads alloc
  init
    [] true -> grant_index' := 0
  update
    [] ~alloc[0] -> grant_index' := 0
    [] alloc[0] & ~alloc[1] -> grant_index' := 1
    [] alloc[0] & alloc[1] & ~alloc[2] -> grant_index' := 2
    [] alloc[0] & alloc[1] & alloc[2] -> grant_index' := 3
  endatom
  atom GRANT controls grant reads alloc, sum awaits req, high_priority
  init
    [] true -> grant' := false
  update
    [] req' & high_priority' & sum <= 3 -> grant' := true
    [] req' & sum <= 2 -> grant' := true
    [] default -> grant' := false
  endatom
endmodule
```

Figure 3.22: Resource Manager (Implementation)

`grantIndex` variable always points to the available resource with the least index, if one is available (as opposed to the non-deterministic choice in `RManager`) and there is a new private variable `sum` that keeps track of the number of resources that have been allocated. We will later use MOCHA to show that `RManagerImpl` is a valid implementation of the module `RManager`.

Chapter 4

Specifications

So far, we described the modeling language used to model systems in MOCHA. In this section we discuss how to describe specifications. Later, we will illustrate how to use MOCHA to check if a specification holds true on a module. There are three different ways to state specifications in MOCHA: (1) invariants, (2) alternating-time temporal logic, and (3) refinement.

4.1 Invariants

An invariant of a module is a predicate that is intended to hold true in all reachable states of a module. The syntax for describing an invariant is similar to that for a boolean expression inside a module. The formal specification is given in Table 4.1. The *full_identifier* construct is a generalized form of an *identifier* which will be explained in Section 4.6. A *full_identifier* may distinguish different variables with the same identifier but which belong to different modules.

An invariant I is a predicate on the states of a module. A state either satisfies or does not satisfy I. A module M satisfies I if all the reachable states of M satisfy I. For example, consider the module `Pete` from Figure 2.5. We would like to state that $P1$ and $P2$ can never be in the critical section at the same time. This is specified by the invariant

```
~((pc1 = inCS) & (pc2 = inCS))
```

4.2 Alternating-time temporal logic

We briefly discuss how to specify Alternating Temporal Logic (ATL) formulas in MOCHA. The reader is referred to [AHK97] for an introduction to this logic.

ATL is a generalization of the temporal logic CTL [CE81]. If p is a predicate on the states of a module, then the CTL formula `EF`p means that a state satisfying p is reached along *some* execution of the module, while the CTL formula `AF`p means that a state satisfying p is reached along *every* execution of the module. The temporal logic ATL is designed to write requirements of *open* systems [AHK97], and is defined

inv_formula ::= *atomic_proposition* | ∼*inv_formula*
| *inv_formula*$_1$ *binary_boolean_op* *inv_formula*$_2$

atomic_proposition ::= *atomic_expr*$_1$ *comparison_op* *atomic_expr*$_2$
| (*element*$_1$ | *full_var*$_1$) `=` (*element*$_2$ | *full_var*$_2$)

atomic_expr ::= *int_nat_constant* | *full_var*
| `(`*atomic_expr*`)` | `-`*atomic_expr*
| *atomic_expr*$_1$ `+` *atomic_expr*$_2$
| *atomic_expr*$_1$ `-` *atomic_expr*$_2$

full_var ::= *full_identifier*`[[`*atomic_expr* | *element*`]]`

Table 4.1: Invariant formula syntax

path_formula ::= `N` *state_formula* | `G` *state_formula* | `F` *state_formula*
| `(` *state_formula*$_1$ `U` *state_formula*$_2$ `)`
| `(` *state_formula*$_1$ `W` *state_formula*$_2$ `)`

state_formula ::= *path_quantifier* *path_formula* | *atomic_proposition*
| ∼*state_formula*
| *state_formula*$_1$ *binary_boolean_op* *state_formula*$_2$

path_quantifier ::= `A` | `E` | `<<` *names* `>>` | `[[` *names* `]]`

names ::= [*module_name*$_1$`,` ... `,`*module_name*$_n$]
| [*full_atom_name*$_1$`,` ... `,`*full_atom_name*$_n$]

full_atom_name ::= *full_identifier*

Table 4.2: ATL formula syntax

by generalizing the existential and universal path quantifiers of CTL. For instance, let Σ be a set of agents corresponding to different components of the system and the external environment. Then, the logic ATL admits formulas of the form `<<` A `>>` `F` p, where p is a state predicate and A is a subset of agents. The formula `<<` A `>>` `F` p means that the agents in the set A can cooperate to reach a p-state no matter how the remaining agents resolve their choices. This is formalized by defining games, and satisfaction of ATL formulas corresponds to existence of winning strategies in such games. The syntax for ATL formulas is defined inductively by the grammar in Table 4.2 and is explained subsequently.

Atomic formulas. *atomic_proposition* as given in Table 4.1 represents the simplest form of state formulas, and is made up of comparisons of expressions of different types, including integers, natural numbers, ranges and enumerative types.

Path formulas. Path formulas are obtained from state formulas using the temporal operators `N` (next), `F` (eventually), `G` (always), `U` (until), and `W` (while). The formal syntax is given in Table 4.2. Path formulas are evaluated over infinite trajectories of a module. Consider an infinite trajectory $\overline{s} = s_0 s_1 \ldots$ of states. Then,

- The formula `N` p holds in $\overline{s}$, for a state formula p, if the state s_1 satisfies p.
- The formula `F` p holds in $\overline{s}$, for a state formula p, if, for some $i \geq 0$, the state s_i satisfies p.
- The formula `G` p holds in $\overline{s}$, for a state formula p, if, for all $i \geq 0$, the state s_i satisfies p.
- The formula (p `U` q) holds in $\overline{s}$, for state formulas p and q, if, there exists $i \geq 0$ such that the state s_i satisfies q and for all $0 \leq j < i$, the state s_j satisfies p.
- The formula (p `W` q) holds in $\overline{s}$, for state formulas p and q, if either all the states s_i satisfy p, or there exists $i \geq 0$ such that the state s_i satisfies q and for all $0 \leq j < i$, the state s_j satisfies p.

Observe that, as in CTL, arguments of temporal operators are state formulas, rather than path formulas (e.g. `A G F` p is not an ATL formula).

State formulas. A state formula is either an atomic proposition, or a boolean combination of state formulas, or an application of a path quantifier to a path formula. While constructing boolean combinations, *binary_boolean_op* is one of the following: `&`, `|`, `<=>`, `=>`. Expressions of boolean types are considered to be state formulas, not atomic propositions. As a result, the operator for equality test for boolean expressions should be `<=>`, not `=`.

In Reactive Modules, each agent corresponds to an atom. For each external variable, there is an extra agent which controls it. The *path_quantifier* construct is given in Table 4.2 where *names* are a list of *full_identifier*s separated by comma. All the names must refer to identifiers of the same type: either they are all module names, or the full atom names. If module names are given, MOCHA internally break

them down into the comprising atoms during model checking. A space is required between the *path_quantifier* and *path_formula* . For example, `A G` should be used instead of `AG`. The meaning of the path quantifiers is explained below

- `E` is existential path quantifier (as in CTL). A state s satisfies the formula `E` φ, for a path formula φ, if there is an infinite trajectory $\overline{s}$ whose first state is s and which satisfies the formula φ.

- `A` is universal path quantifier (as in CTL). A state s satisfies the formula `A` φ, for a path formula φ, if for every infinite trajectory $\overline{s}$ whose first state is s, $\overline{s}$ satisfies the formula φ.

- `<<` *names* `>>` φ, for a path formula φ, means that the listed agents have a strategy to produce a trajectory satisfying φ, no matter how the remaining agents behave. The satisfaction is formally defined via two-player games. To evaluate the formula at a state s, consider the following game between a protagonist and an antagonist. At every step the protagonist executes the atoms parameterizing the path quantifier, while the antagonist executes the remaining atoms. The game proceeds for infinitely many rounds resulting in a trajectory. The protagonist wins if the resulting trajectory satisfies the path formula φ. The initial state s satisfies the formula `<<` *names* `>>` φ if the protagonist has a winning strategy in this game.

  Therefore, the CTL path quantifier `A` is equivalent to the ATL path quantifier `<< >>`.

- The path quantifier `[[ ]]` is the dual of `<< >>`: `[[` *names* `]]` φ, for a path formula φ, means that the agents that are *not* listed in the quantifier have a strategy to produce a trajectory satisfying φ, no matter how the listed agents behave.

  Therefore, the CTL path quantifier `E` is equivalent to the ATL path quantifier `[[ ]]`.

The ATL formula `A G` p means that all reachable states satisfy p. For instance, the mutual exclusion requirement for `Pete`can be specified by the formula

```
A G ~((pc1 = inCS) & (pc2 = inCS))
```

The ATL formula `A F` p means that all trajectories contain a p-state. The deadlock freedom requirement for *Pete* can be specified by the formula:

```
A G ((pc1 = reqCS) => A F (pc1 = inCS))
```

The ATL fomula

```
<< P1 >> G (pc1 = outCS)
```

spec_line	::=	**inv** *formula_name inv_formula* ;
		| **atl** *formula_name path_formula* ;
formula_name	::=	*identifier*

Table 4.3: Specification syntax

means that the module `P1` has a strategy to produce a trajectory in which `pc1` always equals `outCS`. This says that the choice of `P1` to stay outside is under its own control, and does not require any cooperation from $P2$. This formula is not expressible in CTL, and is stronger than existential CTL (or ATL) formula:

```
E G (pc1 = outCS)
```

The ATL formula

```
<< P1 >> F (pc1 = inCS)
```

says that the module $P1$ has a strategy to enter the critical section no matter how the other module behaves. Sample ATL specifications of the railroad controller example are given in Section 7.4.

The model checking problem for ATL is to determine whether a given module satisfies a given ATL formula. The ATL model checking problem is solved by generalizing the symbolic fixpoint computation procedure for CTL model checking. It should be noted that, while ATL is more expressive than CTL, its model checking problem is no harder.

Specification file. A specification file contains a list of specifications. Each specification is an invariant or an ATL formula. The syntax of each line in the specification file is given in Table 4.3 where *formula_name* is a string used to name the formula (and refer to it, while model checking the formula) and either *inv_formula* or *path_formula* are the actual specification formulae. Note that each line needs to be terminated using a semi-colon.

4.3 Refinement

Specifications can also be given in terms of abstract modules. In this case, both the model and the specification are given as reactive modules, and MOCHA can be used to check if the model is a refinement of the specification.

The execution of a module results in a trace of observations. Reactive modules are related via a trace semantics: roughly speaking, one module *implements* (or *refines*) another module if all possible traces of the former, more detailed module are also possible traces of the latter, more abstract module.

4.4 The trace language of a module

Let P be a reactive module. As indicated earlier, a state of P is a valuation for the set X_{P} of module variables. We write Σ_{P} for the set of states of P.

A state s of the module P is *initial* if it can be obtained by executing all initial actions of P in a consistent order. We write $Init_{\mathrm{P}}$ for the set of initial states of the module P. The set $Init_{\mathrm{P}}$ is nonempty, because all initial actions are executable. For two states s and t of P, the state t is a *successor* of s, written $s \rightarrow_{\mathrm{P}} t$, if t can be obtained from s by executing all update actions of P in a consistent order. The binary relation $\rightarrow_{\mathrm{P}}$ over the state space Σ_{P} is called the *transition relation* of the module P. The transition relation $\rightarrow_{\mathrm{P}}$ is serial (i.e., every state has at least one successor), because all update actions are executable. Moreover, a module does not constrain the behavior of the external variables and interacts with its environment in a nonblocking way.

In this way, the module P defines a state-transition graph with the state space Σ_{P}, the initial states $Init_{\mathrm{P}}$, and the transition relation $\rightarrow_{\mathrm{P}}$. The initialized paths of this graph are called the trajectories of the module: a *trajectory* of P is a finite sequence $s_0 \ldots s_n$ of states of P such that (1) the first state s_0 is initial and (2) for all $0 \leq i < n$, the state s_{i+1} is a successor of s_i. If s is a valuation to a set of variables, we use $[s]_{\mathrm{P}}$ to denote the set of valuations from s restricted to the observable variables of P. If $\overline{s} = s_0 \ldots s_n$ is a trajectory of P, then the corresponding sequence $[\overline{s}]_{\mathrm{P}} = [s_0]_{\mathrm{P}} \ldots [s_n]_{\mathrm{P}}$ of observations is called a *trace* of P. Thus, a trace records the sequence of observations that may result from executing the module for finitely many steps. The *trace language* of the module P, denoted L_{P}, is the set of traces of P. By definition, every prefix of a trajectory is also a trajectory, and hence, every prefix of a trace is also a trace. Since the set of initial states is nonempty, and the transition relation is serial, every trajectory of a module, and hence also every trace, can be extended. It follows that a module cannot deadlock. In modeling, therefore, a deadlock situation must be represented by a special state with a single outgoing transition back to itself.

4.5 The implementation preorder between modules

The semantics of the module P consists of the trace language L_{P}, as well as all information that is necessary for describing the possible interactions of P with the environment: the set $intf X_{\mathrm{P}}$ of interface variables, the set $extl X_{\mathrm{P}}$ of external variables, and the await dependencies $\succ_{\mathrm{P}} \cap (intf X_{\mathrm{P}} \times obs X_{\mathrm{P}})$ between interface variables and observable variables (there cannot be any await dependencies between external variables and other variables).

Definition 4.1 [**Refinability**] *The module Q is* refinable *by the module P, if the following conditions are met: (1) every interface variable of Q is an interface variable of P; (2) every external variable of Q is an observable variable of P; and (3) for*

all observable variables x of Q and all interface variables y of Q, if $y \succ_Q x$, then $y \succ_P x$

Definition 4.2 [**Implementation**] *The module P* implements *the module Q, written $P \preceq Q$, if the following conditions are met: (1) Q is refinable by P; and (2) if $\overline{s}$ is a trace of P, then the projection $[\overline{s}]_Q$ is a trace of Q.*

Refinability ensures that the compatibility constraints imposed by P on its environment are at least as strong as those imposed by Q. The second condition for implementation is conventional trace containment. Intuitively, if P $\preceq$ Q, then the module P is *as detailed as* the module Q: the implementation P has possibly more interface and external variables than the specification Q; some external variables of Q may be interface variables of P, and thus are more constrained in P; the implementation P has possibly more await dependencies among its observable variables than the specification Q; and P has possibly fewer traces than Q, and thus more constraints on its execution. It is easy to check that every module P implements itself, and that if a module P implements another module Q, which, in turn, implements a third module R, then P also implements R. Hence, the implementation relation $\preceq$ is a preorder (i.e., reflexive and transitive).

Simulation is a stronger notion of one module refining another.

Definition 4.3 [**Simulation relation**] *Let P and Q be modules such that Q is refinable by P. A relation $H \subseteq \Sigma_P \times \Sigma_Q$ is a* simulation *if the following conditions are met: (1) For all states $s \in \Sigma_P$ and $t \in \Sigma_Q$, if $H(s,t)$ then $[s]_Q = [t]_Q$; (2) For all states $s \in \Sigma_P$ and $t \in \Sigma_Q$, if $H(s,t)$ and $s \rightarrow_P s'$ then there is a state t' of Q such that $t \rightarrow_Q t'$ and $H(s',t')$.*

Definition 4.4 [**Simulation**] *Module Q* simulates *module P (written $P \preceq_s Q$) if the following conditions are met: (1) Q is refinable by P; and (2) There is a simulation relation $H \subseteq \Sigma_P \times \Sigma_Q$ such that, for every s that is an initial state of P there is an initial state t of Q satisfying $H(s,t)$.*

Simulation is a sufficient (but not necessary) condition for implementation, as stated by the following proposition:

Proposition 4.1 *[Simulation and implementation] For any two modules P and Q, if $P \preceq_s Q$ then $P \preceq Q$.*

The problem of checking if P $\preceq$ Q is PSPACE-hard in the state space of Q. The problem of checking if P $\preceq_s$ Q can be checked in time that is linear on the state spaces of P and Q. For the special case in which all variables of Q are observable, the notions of implementation and simulation coincide. In such cases, we just say that P refines Q.

MOCHA provides automatic procedures to check if one module refines the other or if one module simulates the other. To cope up with large modules, MOCHA also

full_identifier ::= *identifier* | *module_name*/*full_identifier*

Table 4.4: Syntax of identifiers for variables (atoms) in invariant and ATL formulae

supports a compositional refinement methodology. The user commands to support refinement and simulation checks are described in Chapter 5. Compositional refinement is discussed in Chapter 6.

4.6 Referencing variables and atoms: the naming convention

In specifying invariants or atomic propositions in ATL formulae, variables in modules need to be referenced. Also, in ATL formulae the path quantifiers may be parametized by atoms. The question arises as to how to refer to variables and atoms, especially in the case of composite modules formed by parallel composition and hiding, etc.

One option is to use the module browser and obtain the name of the variable or atom. The variables and atoms can also be systematically given hierarchical names provided certain restrictions are observed in forming composite modules (if these restrictions are not oberved, the module browser is the only option to find out variable and atom names).

If `M` is a module, its visible variables—interface and external—are referenced by their names. If `M` is a native module, i.e., that which is not formed by hiding or renaming of other modules, its atoms are again referenced by their names, if they were named, and otherwise their names have to be obtained from the browser.

If `M2` is a module as below:

```
M2 := hide x_1, x_2 in M1
```

then the private variables `x_1` and `x_2` of `M2` can be referenced with the names `M2/x_1` and `M2/x_2`, respectively. Variables and atoms in `M_1` are then referenced inductively with names prefixed by `M2/M1`. Syntactically, the identifier of a variable or an atom in a specification is a *full_identifier*. The formal syntax of *full_identifier* expressions is given in Table 4.4.

A similar rule is invoked to name the variables of

```
M3 := hide x_1, x_2 in (M1 || M2)
```

For the above defined rules to uniquely name variables and atoms the following operations are disallowed:

1. Parallel composition of a module with itself. For instance, `P || P`, even if they only have external variables, is not allowed.

2. There can be at most one hide in a module expression. For instance, `X := (hide x in P ) || (hide x in Q)`, is not allowed also.

Chapter 5

User Commands

In this section we illustrate the user level commands of MOCHA. We use the modules described in earlier sections as examples. These examples can be found in `examples/` directory of the MOCHA distribution.

5.1 Parsing modules

The `read_module` command is used to read in a module description. We use Peterson's mutual exclusion protocol from Figure 2.5, which can be found in the MOCHA distribution at `examples/pete`. On a `read_module` command, MOCHA displays the names of the modules that were successfully parsed. In the case of a parse error, an appropriate message is displayed.

```
mocha:  read_module pete.rm
Module P1 is composed and checked in.
Module P2 is composed and checked in.
Module Pete is composed and checked in.
parse successful.
```

The command `reinit` is used to reinitialize MOCHA if there is a parse error. It clears all the type and module definitions. In case of a parse error, fix the error, execute `reinit` and parse the module file again. Once a module file has been read in, a number of commands can be executed to get information about the modules in memory. The command `show_mdls` lists the modules that have been read in.

```
mocha:  show_mdls
P1
P2
Pete
```

The command `show_atoms` lists the atoms of a module.

```
mocha:  show_atoms Pete
Pete/P1/ATM0 Pete/P2/ATM0
```

The command `show_types` lists the various types.

```
mocha:  show_types
Built-in :  bool, int, nat, event
Enumerative :  ctype
Range :  no range type defined.
Bitvector :  no bitvector type defined.
Array :  no array type defined.
```

The command `show_vars` lists the different types of variables for a module—history free, history dependent, event or all the variables.

```
mocha:  show_vars -vALL Pete
pc1
Pete/x1
pc2
Pete/x2
```

```
mocha:  show_vars -vHD Pete
pc1
Pete/x1
pc2
Pete/x2
```

Commands such as `isPrivateVariable`, `isHistoryFree`, `isInterfaceVariable` return 1 or 0.

```
mocha:  isPrivateVariable Pete Pete/x1
1
```

```
mocha:  isHistoryFree Pete pc1
0
```

5.2 Executing modules

In MOCHA, the user can perform three kinds of execution—manual, random, and game, on any module. MOCHA provides a TK-based graphical user interface for interacting with the tool and viewing the execution trace. To execute a module, first read in the file containing the textual description of the module. Then, using the module browser that's obtained from the "File" pull-down menu, select a module

for execution by selecting it and pressing the open button. The ReactiveModules code for the module is then displayed on a new window. Press the "Execute" button. A new window will then pop up offering three types of execution. Choose one.

- Manual execution. Initially, the GUI displays all possible initial states of the module in the upper section of the window. The user can select any one of them, and press the "Go!" button, whereupon the tool generates all possible next states. The user can again select any next state to continue the execution.
- Random execution. User can specify the number of rounds that it wants to execute the module for. The choices for the initial and successor states at each step are made randomly.
- 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 much better way of performing guided execution. In each case, the user has to press the "Go!" button to advance the execution, no matter who's turn it is.

By default, the interface displays the value of only the observable variables of a module. The GUI also lets the user modify the set of variables being displayed, as well as change the format of the display of values of the variables, by choosing the right options under the "Option" menu.

5.3 Invariant Checking

Suppose we want to check if the module `Pete` satisfies mutual exclusion. This is specified as an invariant in the file `examples/pete/pete.spec`:

```
inv "mutex" ~(pc1 = inCS & pc2 = inCS);
```

We first read the invariant using the `read_spec` command and then check the invariant using the `inv_check` command.

```
mocha:  read_spec pete.spec
mutex
mocha:  inv_check Pete mutex
Typechecking invariant mutex...
Typechecking successful
No sym_info..  building it(using sym_trans)
Ordering variables using sym_static_order
Transition relation computed :  2 conjuncts
Calling Dynamic Reordering with sift
Done initializing image info...
Initial Region Computed...
```

```
Step 1:  image mdd size = 3 |states| = 4
reached set mdd size = 6 |states| = 8
Step 2:  image mdd size = 12 |states| = 4
reached set mdd size = 14 |states| = 12
Step 3:  image mdd size = 10 |states| = 6
reached set mdd size = 14 |states| = 16
Step 4:  image mdd size = 12 |states| = 4
Done reached set computation...
reached set mdd size = 14 number of states = 16
Invariant mutex passed
```

If the invariant fails, MOCHA will display an error trace. Let us create a bug in `Pete` by deleting a negation in line 12 of `pete.rm`. The buggy version is found in `petebug.rm`. Let us check the invariant on the buggy model.

```
mocha:  read_module petebug.rm
Module P1 is composed and checked in.
Module P2 is composed and checked in.
Module Pete is composed and checked in.
parse successful.
mocha:  read_spec pete.spec
mutex
mocha:  inv_check Pete mutex
Typechecking invariant mutex...
Typechecking successful
No sym_info..  building it(using sym_trans)
Ordering variables using sym_static_order
Transition relation computed :  2 conjuncts
Calling Dynamic Reordering with sift
Done initializing image info...
Initial Region Computed...
Step 1:  image mdd size = 3 |states| = 4
reached set mdd size = 6 |states| = 8
Step 2:  image mdd size = 12 |states| = 4
reached set mdd size = 12 |states| = 10
Invariant mutex has been violated
Invariant mutex failed in step 2
Counterexample for invariant mutex
pc1=outCS $x1_0=1 pc2=outCS $x2_0=0
pc1=reqCS $x1_0=0 pc2=reqCS $x2_0=0
pc1=inCS $x1_0=0 pc2=inCS $x2_0=0
```

In this case MOCHA has produced an error trace of length 3. In the first round both process are outside the critical section; in the second round they both request

access to the critical section; and, in the third round both processes enter the critical section.

5.4 ATL model checking

We consider the train controller example from Section 3.7.2. The files for this example can be found in `examples/train_control`. There are three agents in the system: the eastbound train `TrainE`, the westbound train `TrainW`, and the signal controller `Controller`. We are interested in verifying the following properties:

1. The system is safe: the eastbound train and the westbound train are not on the bridge at the same time. This requirement is written as: `atl A G ~(pcE = bridge & pcW = bridge);` Note that the same can be specified as an invariant (Section 4.1).

2. The trains have the discretion to stay away from the bridge. No other agents can force it to do otherwise. For the eastbound train, this property can be written as: `atl A G (~(pcE = bridge) => << TrainE >> G ~(pcE = bridge));`

3. Since there is no fairness constraints imposed on the system, once a train is granted access to the bridge, it has the discretion to leave the bridge at any time–no other agents can force it to leave. For the westbound train, this property can be written as: `atl A G ((pcW = bridge) => << TrainW >> G (pcW = bridge));`

These requirements can be found in the file `train_control.spec`. After reading the ReactiveModules description file `train_control.rm` with the `read_module` command, read the specifcations into Mocha:

```
mocha:  read_spec train_control.spec
safetyInvariant
safety
atl0
atl1
```

Mocha read in the specifications. The `show_spec` can now be used to give the list of read formulas. Supplying the `-l` option lists the formulas and the names of the specifications

```
mocha:  show_spec -l
atl specifications:
atl0
<< >> G((!((pcE = bridge)) => << TrainE >> G(!((pcE = bridge)))))

atl1
```

```
<< >> G(((pcW = bridge) => << TrainW >> G((pcW = bridge))))

safety
<< >> G(!(((pcE = bridge) & (pcW = bridge))))


inv specifications:
safetyInvariant
!(((pcW = bridge) & (pcE = bridge)))
```

Note that the specifications are displayed under two sections: atl and inv.

Invariants are checked with the `inv_check` command and ATL formulas are model checked with the `atl_check` command. Both commands have to be followed by two arguments: the module-name and the formula name. Formula that weren't given a name will be assigned a name that can be obtained by the `show_spec` command.

There are two modules in the file `train_control.rm`. `System1` is the defective one and `System2` should satisfy all the properties (invariants and ATL formulae).

Try them. Here is what you should get if you try `atl_check System2 atl1`.

```
mocha: atl_check System2 atl1
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 : 5 conjuncts
Calling Dynamic Reordering with sift
Done initializing image info...
Building the initial region of the module...
Model-checking formula "atl1"
ATL_CHECK: formula "atl1" passed
mocha:
```

The last line of the output before the MOCHA prompt says the formula passed. If the property is not satisfied, the last line will indicate the failure of the property.

Counter-example generation. Currently, the ATL model-checker does not have any mechanism to generate counter-examples. We plan to integrate the game execution facility described in Section 5.2 with the ATL model checker to provide counter-examples and witnesses: when an ATL specification fails, the ATL model checker synthesizes and outputs a *winning* strategy as a counter-example, according to which the simulator will play a game with the user. The user tries to win the game by finding an execution sequence that satisfies the specification. We believe that by

playing a *losing* game, the user can be convinced that their model is incorrect and subsequently discover the bug in their model.

The invariant check will automatically produce a counter-example, a path to a state where the invariant is violated, for a failed invariant. Try:

```
inv_check System1 safetyInvariant
```

5.5 Refinement checking

Consider the module `Sync3BitCounter` (look for it in the file `counter.rm` in the `examples/counter` directory of the MOCHA distribution. It is a 3-bit counter built out of gates. A behavioral specification for it is given below.

```
module Sync3BitCounterSpec
  external start, inc:  bool
  interface out0, out1, out2, done:  bool
  private count:  bitvector3
  atom controls count reads count start awaits inc
  update
    [] start & ~inc' -> count' := 0
    [] start & inc'  -> count' := 1
    [] ~start & inc' -> count' := count + 1
  endatom
  atom controls out0, out1, out2 awaits count
  init update
    [] true -> out0' := count'[0]; out1' := count'[1]; out2' := count'[2]
  endatom
  atom controls done reads count, start awaits count
  update
    [] ~start & count'=count + 1 & count'=0 -> done' := true
    [] default                              -> done' := false
  endatom
endmodule
```

In MOCHA, the notion of refinement is language containment. As is well-known, simulation is a sufficient check for language containment. There are two commands available in MOCHA for checking refinement — `check_refine` and `check_simulation`. `check_refine` is typically more efficient but it can be used only if there are no hidden variables in the specification. If there are private variables in the specification as in this case, then the command `check_simulation` that checks for simulation is used. In this case, the specification has a private variable called count. Hence, we use the command `check_simulation`.

```
mocha: read_module counter.rm
Module And is composed and checked in.
Module Or is composed and checked in.
Module Not is composed and checked in.
Module Xor is composed and checked in.
Module Latch is composed and checked in.
Module Sync1BitCounter is composed and checked in.
Module Sync3BitCounter is composed and checked in.
Module Witness is composed and checked in.
Module Foo is composed and checked in.
Module Sync3BitCounterSpec is composed and checked in.
parse successful.
mocha: check_simulation Sync3BitCounter Sync3BitCounterSpec
Building transition relation for module Sync3BitCounter
Ordering variables using sym_static_order
Transition relation computed : 24 conjuncts
Calling Dynamic Reordering with sift
Done initializing image info...
Writing order into imporder.dat
Building transition relation for module Sync3BitCounterSpec
Ordering variables using sym_static_order
Transition relation computed : 3 conjuncts
Calling Dynamic Reordering with sift
Done initializing image info...
Dynamic variable ordering is enabled with method sift.

 ****************************************************************
 Reached fixpoint after  3 steps
Yes: There is a simulation from Sync3BitCounter to Sync3BitCounterSpec
Sync3BitCounter is simulated by Sync3BitCounterSpec
Freeing syminfo for both specification and implementation
mocha:
```

Chapter 6

Verification Methodology

In this chapter we describe some techniques and the support in MOCHA that enable the verification of large systems. Section 6.1 describes how to verify more general safety properties than invariants by using monitors. Section 6.2 describes how to circumvent the intractability of language inclusion by specifying witness modules for refinement checking. Sections 6.3 and 6.4 describe abstraction modules and assume-guarantee reasoning for modular verification.

6.1 Monitors

Invariants can distinguish between two trajectories only if one of the trajectories contains a state that does not occur on the other trajectory. Hence there are requirements on the behavior of a reactive module P that cannot be phrased as invariants of P. However, many such requirements can be phrased as invariants of the compound module $\mathrm{P}\|M$, for a monitor M of P. The module M is a *monitor* of P if M is compatible with P and $intf X_M \cap extl X_{\mathrm{P}} = \emptyset$. If M is a monitor of P, then in each round, M may record the values of the observable variables of P, but M must not modify any external variables of P. Thus the monitor M can observe but not interfere with the behavior of P. In particular, the monitor M may check if P meets a requirement, and it may signal every violation of the requirement by sounding an observable alarm. The module P then meets the given requirement iff the compound module $\mathrm{P}\|M$ has the invariant that no alarm is sounded by the monitor M.

Figure 3.19 presents an asynchronous railroad controller that enforces the train-safety requirement. Yet the module `Controller` is not a satisfactory railroad controller, because it may keep a train waiting at a red signal while the other train is allowed to cross the bridge repeatedly. In particular, the resulting railroad system does not meet the *equal-opportunity requirement* that, while a train is waiting at a red signal, it is not possible that the signal at the opposite entrance to the bridge turns from green to red and back to green. Since the equal-opportunity requirement is violated by trajectories, and not by individual states, we need to employ monitors. The module

```
module EqOppMonitor
  interface alert :  {0,1,2,3}
  external pc :  {away,wait,bridge}; signal1,signal2 :  {green,red}
  atom controls alert reads alert, pc, signal1, signal2
  init
    [] true -> alert' := 0
  update
    [] alert=0 & pc=wait & signal1=red & signal2=green -> alert' := 1
    [] alert=1 & signal1=green                         -> alert' := 0
    [] alert=1 & signal1=red & signal2=red             -> alert' := 2
    [] alert=2 & signal1=green                         -> alert' := 0
    [] alert=2 & signal1=red & signal2=green           -> alert' := 3
```

Figure 6.1: Monitoring equal opportunity

pcW	pcE	signalW	signalE	alertW	alertE
away	away	red	red	0	0
wait	away	red	red	0	0
wait	wait	red	red	0	0
wait	wait	red	green	0	0
wait	bridge	red	green	1	0
wait	away	red	red	1	0
wait	wait	red	red	2	0
wait	wait	red	green	2	0
wait	bridge	red	green	3	0

Figure 6.2: Error trajectory that violates equal opportunity

```
EqOppMonitorW :=
  EqOppMonitor[alert,pc,signal1,signal2:= alertW,pcW,signalW,signalE]
```

monitors the equal-opportunity requirement for the train that travels clockwise, where `EqOppMonitor` is shown in Figure 6.1. The monitor has four levels of alertness. The alertness level is 0 as long as the train is not waiting at a red signal while the other signal is green, in which case the alertness level rises to 1. The alertness level rises to 2 when the other signal turns red, and to 3 when the other signal turns green again, while the train is still waiting at a red signal. An alertness level of 3 sounds an alarm that indicates a violation of the equal-opportunity requirement for the train that travels clockwise. The equal-opportunity requirement for the train that travels counterclockwise is monitored by the module

```
EqOppMonitorE :=
  EqOppMonitor[alert,pc,signal1,signal2:= alertE,pcE,signalE,signalW]
```

in the same manner. The module `Controller` then meets the equal-opportunity requirement iff the observation predicate

$$\sim\texttt{(alertW = 3 | alertE = 3)}$$

is an invariant of the compound module

```
RailroadSystem || EqOppMonitorW || EqOppMonitorE
```

The error trajectory of Figure 6.2 shows that this is not the case.

6.2 Witness modules for refinement checking

The problem of checking if P $\preceq$ Q is PSPACE-hard in the state space of Q. However, the refinement check is simpler in the special case in which all variables of Q are observable. The module Q is *projection refinable* by the module P if (1) Q is refinable by P, and (2) Q has no private variables. If Q is projection refinable by P, then every variable of Q is observable in both P and Q. Therefore, checking if P $\preceq$ Q reduces to checking if for every trajectory $\overline{s}$ of P, the projection $[\overline{s}]_Q$ is a *trajectory* of Q. According to the following proposition, this can be done by a transition-invariant check, whose complexity is linear in the state spaces of both P and Q.

Proposition 6.1 *[Projection refinement] Consider two modules P and Q, where Q is projection refinable by P. Then P $\preceq$ Q iff (1) if s is an initial state of P, then $[s]_Q$ is an initial state of Q, and (2) if s is a reachable state of P and $s \rightarrow_P t$, then $[s]_Q \rightarrow_Q [t]_Q$.*

We make use of this proposition as follows. Suppose that Q is refinable by P, but not projection refinable. This means that there are some private variables in Q. Define Q^u to be the module obtained by making every private variable of Q an interface variable. If we compose P with a module W whose interface variables include the

```
module WitnessMsg0
  interface msg0 :  msgType
  external pcR :  recCtrlType; doneP, doneC :  event; msgP :  msgType
  atom controls msg0 reads msg0, pcR, doneP, doneC, msgP
      awaits pcR, doneP, doneC
  update
    [] pcR'=consume & doneP?  & ~doneC? -> msg0' := msgP
  endatom
endmodule

module WitnessSpecCtrl
  interface pc :  specCtrlType
  external pcS :  sendCtrlType; pcR :  recCtrlType
  atom controls pc awaits pcR, pcS
  init update
    [] pcS'=produce & pcR'=receive -> pc' := produce
    [] pcS'=send & pcR'=consume    -> pc' := consume
    [] pcS'=send & pcR'=receive    -> pc' := produce_consume
    [] pcS'=produce & pcR'=consume -> pc' := produce_consume
  endatom
endmodule
```

Figure 6.3: Witness modules for `msg0` and `pc`

private variables of Q, then Q^u is projection refinable by the composition P‖W. Moreover, if W does not constrain any external variables of P, then P‖W $\preceq$ Q^u implies P $\preceq$ Q (in fact, P is simulated by Q). Such a module W is called a *witness* to the refinement P $\preceq$ Q. The following proposition states that in order to check refinement, it is sufficient to first find a witness module and then check projection refinement.

Proposition 6.2 *[Witness modules] Consider two modules P and Q such that Q is refinable by P. Let W be a module such that (1) W is compatible with P, and (2) the interface variables of W include the private variables of Q, and are disjoint from the external variables of P. Then (1) Q^u is projection refinable by $P \| W$, and (2) $P \| W \preceq Q^u$ implies $P \preceq Q$.*

Furthermore, it can be shown that if P does not have any private variables, and P is simulated by Q, then a witness to the refinement P $\preceq$ Q does exist. In summary, the creativity required from the human verification expert is the construction of a suitable witness module, which makes explicit how the private state of the specification Q depends on the state of the implementation P.

Consider the `SendRecImpl` and `SendRecSpec` modules described earlier. They can be found in `examples/msg`. Note that `SendRecSpec` is refinable by `SendRecImpl`,

but not projection refinable. We can use `check_simulation` command to check that:

$$\texttt{SendRecImpl} \preceq \texttt{SendRecSpec}$$

In order to carry out a refinement check, we need to write witnesses for the `pc` and `msg0` variables in `SendRecSpec`. First, the `Sender` and `Receiver` modules from Figure 3.18 need to be changed so as to make their private variables visible. Let us call the new modules as `UnhideSender` and `UnhideReceiver` respectively. To avoid a name collision, the `pc` variables of `UnhideSender` and `UnhideReceiver` are renamed to `pcS` and `pcR` respectively. The witness modules for `WitnessMsg0` and `WitnessSpecCtrl`, shown in Figure 6.3 make use of these variables. Now, we can compose the witnesses with the sender and receiver modules:

```
UnhideSendRecImpl :=
  UnhideSender || UnhideReceiver || WitnessMsg0 || WitnessSpecCtrl
```

Finally, we change the private variable `pc` in `SendRecSpec` to an external variable, and obtain `UnhideSendRecSpec`. The `check_refine` command can now be used to prove that `UnhideSendRecImpl` refines `UnhideSendRecSpec`. The witness modules and the "unhidden" modules can be found in `examples/msg/msgRef.rm`

```
mocha:  read_module msgRef.rm
Module UnhideSyncSender is composed and checked in.
Module UnhideReceiver is composed and checked in.
Module WitnessMsg0 is composed and checked in.
Module WitnessSpecCtrl is composed and checked in.
Module UnhideSendRecImpl is composed and checked in.
Module UnhideSendRecSpec is composed and checked in.
parse successful.

mocha:  check_refine UnhideSendRecImpl UnhideSendRecSpec
Building transition relation for module UnhideSendRecImpl
Ordering variables using sym_static_order
Transition relation computed :  7 conjuncts
Calling Dynamic Reordering with sift
Done initializing image info...
Writing order into imporder.dat
Building transition relation for module UnhideSendRecSpec
Ordering variables using sym_static_order
Transition relation computed :  1 conjuncts
Calling Dynamic Reordering with sift
Done initializing image info...
Dynamic variable ordering is enabled with method sift.

...

Done reached set computation...
```

```
reached set mdd size = 78 number of states = 186
UnhideSendRecImpl refines UnhideSendRecSpec
Freeing syminfo for both specification and implementation
mocha:
```

6.3 Abstraction modules

Consider two modules P_1 and P_2. Suppose, we want to show that $P_1 \| P_2 \preceq Q$. Sometimes it is possible to construct modules A_1 and A_2 such that A_1 and A_2 are more abstract that P_1 and P_2 respectively. Then the following proof rule is correct.

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

Example.

P_1 = `Sync3BitCounter [ done := z ]`
P_2 = `Sync3BitCounter [ inc, out0, out1, out2 := z, out3, out4, out5 ]`

The specification Q for $P_1 \| P_2$ is the following module.

```
module Sync6BitCounterSpec
  external start, inc :  bool
  interface out0, out1, out2, out3, out4, out5, done :  bool
  private count :  bitvector6
  atom controls count reads count, start awaits inc
  update
    [] start & ~inc' -> count' := 0
    [] start & inc'  -> count' := 1
    [] ~start & inc' -> count' := count + 1
  endatom
  atom controls out0, out1, out2 awaits count
  init update
  [] true -> out0' := count'[0]; out1' := count'[1]; out2' := count'[2];
    out3' := count'[3]; out4' := count'[4]; out5' := count'[5]
  endatom
  atom controls done reads count, start awaits count
  update
    [] ~start & count'=count + 1 &count'=0 -> done' := true
    [] default                             -> done' := false
  endatom
endmodule
```

Then, we have the following abstraction modules.

A_1 = `Sync3BitCounterSpec [ done := z ]`
A_2 = `Sync3BitCounterSpec [ inc, out0, out1, out2 := z, out3, out4, out5 ]`

Here, A_1 and A_2 are more abstract descriptions of P_1 and P_2. By inspection, we can see that the proof rule described above is applicable. We can show using `check_simulation` that $P_1 \preceq A_1$ and $P_2 \preceq A_2$. Hence, $P_1 \| P_2 \preceq A_1 \| A_2$. Therefore, we have that

$$\texttt{hide z in (}P_1 \texttt{ || } P_2\texttt{) endhide} \preceq \texttt{hide z in (}A_1 \texttt{ || } A_2\texttt{) endhide}$$

Now, we can use `check_simulation` to show that

$$\texttt{hide z in (}A_1 \texttt{ || } A_2\texttt{) endhide} \preceq \texttt{Sync6BitCounterSpec}$$

6.4 Assume-guarantee reasoning

The state space of a module may be exponential in the size of the module description. Consequently, even checking projection refinement may not be feasible. However, typically both the implementation P and the specification Q consist of the parallel composition of several modules, in which case it may be possible to reduce the problem of checking if P $\preceq$ Q to several subproblems that involve smaller state spaces. The assume-guarantee rule for reactive modules [AH96] allows us to conclude P $\preceq$ Q as long as each component of the specification Q is refined by the corresponding components of the implementation P within a suitable environment. The following proposition gives a slightly generalized account of the assume-guarantee rule.

Proposition 6.3 *[Assume-guarantee rule] Consider two composite modules* $P = P_1 \| \cdots \| P_m$ *and* $Q = Q_1 \| \cdots \| Q_n$*, where* Q *is refinable by* P*. For* $i \in \{1, \ldots, n\}$*, let* Γ_i *be the composition of arbitrary compatible components from* P *and* Q *with the exception of* Q_i*. If* $\Gamma_i \preceq Q_i$ *for every* $i \in \{1, \ldots, n\}$*, then* $P \preceq Q$*.*

We make use of this proposition as follows. First we decompose the specification Q into its components $Q_1 \| \cdots \| Q_n$. Then we find for each component Q_i of the specification a suitable module Γ_i (called an *obligation module*) and check that $\Gamma_i \preceq Q_i$. This is beneficial if the state space of Γ_i is smaller than the state space of P. The module Γ_i is the parallel composition of two kinds of modules—*essential modules* and *constraining modules*. The essential modules are chosen from the implementation P so that every interface variable of Q_i is an interface variable of some essential module. There may, however, be some external variables of Q_i that are not observable for the essential modules. In this case, to ensure that Q_i is refinable by Γ_i, we need to choose constraining modules either from the implementation P or from the specification Q (other than Q_i). Once Q_i is refinable by Γ_i, if the refinement check $\Gamma_i \preceq Q_i$ goes through, then we are done. Typically, however, the external variables of Γ_i need to be constrained in order for the refinement check to go through. Until this is achieved, we must add further constraining modules to Γ_i.

It is preferable to choose constraining modules from the specification, which is less detailed than the implementation and therefore gives rise to smaller state spaces (in the undesirable limit, if we choose $\Gamma_i = P$, then the proof obligation $\Gamma_i \preceq Q_i$ involves the state space of P and is no simpler than the original proof obligation $P \preceq Q$). Unfortunately, due to lack of detail, the specification often does not supply a suitable choice of constraining modules. According to the following simple property of the refinement relation, however, we can arbitrarily "enrich" the specification by composing it with new modules.

Proposition 6.4 *[Abstraction modules] For all modules P, Q, and A, if $P \preceq Q\|A$ and Q is refinable by P, then $P \preceq Q$.*

So, before applying the assume-guarantee rule, we may add modules to the specification and prove $P \preceq Q\|A_1\|\cdots\|A_k$ instead of $P \preceq Q$. The new modules $A_1, \ldots, A_k$ are called *abstraction modules*, as they usually give high-level descriptions for some implementation components, in order to provide a sufficient supply of constraining modules. In summary, the creativity required from the human verification expert is the construction of suitable abstraction modules, which on one hand, need to be as detailed as required to serve as constraining modules in assume-guarantee reasoning, and on the other hand, should be as abstract as possible to minimize their state spaces.

MOCHA provides support for assume-guarantee reasoning. To be able to operate at a finer granularity, MOCHA decomposes a refinement proof at the level of atoms (i.e, we treat P and Q as single modules and use the atoms of P and Q as P_i's and Q_i's in the above proof rule). The command used to carry out one step in the assume-guarantee proof is `check_refine_atom`. The following is extracted from the command documentation for `check_refine_atom`:

```
check_refine_atom [-e] [-f <size or varlist> ] [-h] [-i <varlist> ]
  [-k] [-o <fname>] [-r] [-v] <impl> <spec> <spec_interface_varname>
```

Do one step in the compositional refinement proof of "`Impl` refines `Spec`". The step done corresponds to the atom that controls variable named `spec_interface_varname` in the specification (let us call this atom atom1). The command does the following:

Given an implementation, specification modules and an interface variable in the specification that controls an atom (say `atom1`), construct new modules "new specification" and "new implementation" such that (1) new specification contains `atom1` only, and (2) new implementation contains heuristically chosen atoms from specification and implementation that control variables controlled by `atom1`, but do not include `atom1` itself

A refinement check between the new specification and the new implementation is performed First, new specification and implementation modules are created. The new specification contains just `atom1`. The new implementation contains chosen atoms from specification and implementation that control variables controlled by

`atom1`, but do not include `atom1` itself. At this point the new implementation contains only "essential atoms" to do a refinement check.

Further atoms may need to be added to the new implementation to constrain its environment. There are two ways to do this:

1. **Automatic**: If the `-f 0` (size) option is chosen, then the implementation module is not grown any further. For the `-f 1` and `-f 2` options, progressively larger "constraining" environments are chosen for the new implementation module. (The default if no explicit `-f` option is used is `-f 2`)

2. **Manual**: The user can also force specific variables to be controlled in the new implementation by simply supplying a list of variables in the -f option, e.g. `-f {var1 var2 }`.

Preference is always given to choose atoms from the specification for the new implementation whenever possible. However, the `-i` option can be used to choose specific atoms from the implementation.

Note 1: Since atom names are mangled by MOCHA (to uniquify atom names during parallel composition), we identify an atom by any variable that is controlled by the atom.

Note 2: List of variables are specified by enclosing them in curly braces, e.g. {`var1 var2`} .

Note 3: You also have to use curly braces if your variable name has array indices, e.g. {`foo2[0][1]`} .

Examples: Suppose the specification module is `Spec` and implementation module is `Impl`. To do the sub-proof corresponding to a specific atom in the Spec that controls variable `myVar` use: `check_refine_atom Impl Spec myVar`

To force variables `foo1` and `foo2[0]` to be controlled use:
`check_refine_atom -f {foo1 foo2[0]} Impl Spec myVar`

In addition, to force `foo2[0]` to be constrained by an atom from Impl, use:
`check_refine_atom -f {foo1 foo2[0]} -i {foo2[0]} Impl Spec myVar`

Command Options:

- `-e` : do transition invariant check only at the end (after completing reachability)
- `-f 0|1|2|var_name_list` :
  - 0 : only the "essential atoms" are chosen
  - 1 : chooses a bigger set of atoms than size=0. Guaranteed to be comparable with the new specification

  - 2 : heuristically chooses a bigger set of atoms than size=1
  - `var_name_list` : forces atoms controlling variables in `var_name_list` to be chosen in the new implementation

- `-h` : print usage
- `-i var_name_list` : if the atom controlling variable in `var_name_list`is chosen, choose it from the implementation
- `-k` : keep the new modules (for debugging purposes: default behavior if you do not use this option is to delete them)
- `-o fname` : write out the mdd variable ordering in file specified by `fname`
- `-r` : do not perform refinement check on the new modules (for debugging purposes)
- `-v` : verbose mode

We will use the resource manager example from Figure 3.7.3 and Figure 3.22 to illustrate the use of `check_refine_atom`. We use `rmanager.rm` from `examples/resource` First, the module is read in as usual

```
mocha:  read_module rmanager.rm
Module Spec is composed and checked in.
Module Impl is composed and checked in.
parse successful.
```

First, we check if the `ALLOC` atom in `RManager`is refined correctly by the corresponding atom in `RManagerImpl`.

```
mocha:  check_refine_atom Impl Spec alloc

New specification module @M1 created

Adding IMPLEMENTATION atom Impl/ALLOC_0
Adding SPECIFICATION atom Spec/GRANT_INDEX_0
Adding SPECIFICATION atom Spec/GRANT_0
Adding SPECIFICATION atom Spec/HALF_EMPTY_0
New Implementation module @M9 created

Building transition relation for module @M9
Ordering variables using sym_static_order
Transiton relation computed :  4 conjuncts
Calling Dynamic Reordering with sift
Done initializing image info...
Writing order into imporder.dat
```

```
Building transition relation for module @M1
Ordering variables using sym_static_order
Transiton relation computed :  1 conjuncts
Calling Dynamic Reordering with sift
Done initializing image info...
Dynamic variable ordering is enabled with method sift.

...

Done reached set computation...
reached set mdd size = 1 number of states = 16
@M9 refines @M1
Freeing syminfo for both specification and implementation
Deleting intermediate specification and implementation
Compositional refinement step successful
```

Next, we check if the HALFEMPTY atom in RManager is refined correctly by RManagerImpl.

```
mocha:  check_refine_atom Impl Spec half_empty

New specification module @M11 created

Adding IMPLEMENTATION atom Impl/HALF_EMPTY_0
Adding SPECIFICATION atom Spec/ALLOC_0
Adding IMPLEMENTATION atom Impl/SUM_0
Adding SPECIFICATION atom Spec/GRANT_0
Adding SPECIFICATION atom Spec/GRANT_INDEX_0
New Implementation module @M21 created

Building transition relation for module @M21
Ordering variables using sym_static_order
Transiton relation computed :  5 conjuncts
Calling Dynamic Reordering with sift
Done initializing image info...
Writing order into imporder.dat
Building transition relation for module @M11
Ordering variables using sym_static_order
Transiton relation computed :  1 conjuncts
Calling Dynamic Reordering with sift
Done initializing image info...
Dynamic variable ordering is enabled with method sift.

...

Done reached set computation...
```

```
reached set mdd size = 18 number of states = 16
@M21 refines @M11
Freeing syminfo for both specification and implementation
Deleting intermediate specification and implementation
Compositional refinement step successful
```

A similar proof can be done for the `GRANTINDEX` atom. Finally, we are left to verifying if the `GRANT` atom in `RManager` is correctly refined in `RManagerImpl`. For all the above proofs, MOCHA was able to choose the constraining environments automatically. In this proof, however, it is necessary to manually force `grant_index` to be chosen from the implementation. We encourage the reader to try the following command without the `-i` option and interpret the resulting error trace.

```
mocha:  check_refine_atom -i grant_index Impl Spec grant

New specification module @M33 created

Adding IMPLEMENTATION atom Impl/GRANT_0
Adding SPECIFICATION atom Spec/ALLOC_0
Adding IMPLEMENTATION atom Impl/GRANT_INDEX_0
Adding SPECIFICATION atom Spec/HALF_EMPTY_0
Adding IMPLEMENTATION atom Impl/SUM_0
New Implementation module @M43 created

Building transition relation for module @M43
Ordering variables using sym_static_order
Transiton relation computed :  5 conjuncts
Calling Dynamic Reordering with sift
Done initializing image info...
Writing order into imporder.dat
Building transition relation for module @M33
Ordering variables using sym_static_order
Transiton relation computed :  1 conjuncts
Calling Dynamic Reordering with sift
Done initializing image info...
Dynamic variable ordering is enabled with method sift.

...

Done reached set computation...
reached set mdd size = 18 number of states = 16
@M43 refines @M33
Freeing syminfo for both specification and implementation
Deleting intermediate specification and implementation
Compositional refinement step successful
```

Chapter 7

Real-Time Modules

7.1 Describing Systems with Real-Time Modules

MOCHA supports reachability analysis and invariant checking of real-time systems that are described in the form of *timed reactive modules* as defined in [AH97]. In addition to the discrete-valued variables of reactive modules, a timed module makes use real-valued ($\mathbb{R}$) *clock variables*. The purpose of clock variables is to keep track of elapsed time. Guarded commands can assign integer values to clock variables, and guards can depend on values of clock variables.

The keyword `clock` denotes the clock variable type. Clock variables are initialized to 0 by default[1]. For each clock x, guards can include positive Boolean combinations of inequalities of the form $x \leq C$, $x \geq C$ and $x = C$, where $C \in \mathbb{N}$. Assignments can assign integer values to clocks in the usual fashion: $x := C$, where $C \in \mathbb{N}$.

In addition to `init` and `update` commands, a timed module has a set of `wait` commands which describe the passage of time. When a `wait` command is executed, all non-clock variables remain the same, and all clock variables are incremented by the same amount. A typical `wait` command has the following form

```
a=b & x1 <= 3 & x2 <= 5 -> x1':<= 3; x2' :<= 5;
```

a and b are non-clock variables and $x1$ and $x2$ are clocks. Instead of `a=b`, an arbitrary predicate on non-clock variables could be used. The interpretation of this `wait` statement is as follows: If `a=b` evaluates to `true` and the values of the clocks satisfy the inequalities $x1 \leq 3$ and $x2 \leq 5$, then a time period of δ can elapse while the values of all non-clock variables remain constant. δ must satisfy the conditions $x1 + \delta = x1' \leq 3$ and $x2 + \delta = x2' \leq 5$. In this way, `wait` statements are used to specify upper bounds on the time elapsed in a given state. The guards of `wait` commands are sometimes referred to as "clock invariants" because the invariants on the clocks specified by the guard must hold for the module to remain in that state.

[1] An error is signaled if there is a non-zero initial assignment to a clock variable.

A timed module makes progress by executing either an `update` command or a `wait` command. If no `wait` command can be executed, then this forces an `update` command to be executed[2].

```
module RealTimeTrain
  interface pc :  {far,near,gate}; arrive :  bool
  private x :  clock
  atom controls pc, x, arrive reads pc, x, arrive
  init
    [] true -> pc' := far; arrive' := false
  update
    [] pc=far         -> pc' := near; arrive' := true; x' := 0
    [] pc=near & x>=3 -> pc' := gate; x' := 0
    [] pc=gate & x>=1 -> pc' := far; arrive' := false
  wait
    [] pc=far         ->
    [] pc=near & x<=5 -> x' :<= 5
    [] pc=gate & x<=2 -> x' :<= 2
  endatom
endmodule
```

Figure 7.1: Real-time module for the train

A simple example of a real-time module is presented in Figure 7.1. This module describes the behavior of a train approaching a gate. The interface variable `pc` initially has the value `far`, indicating that the train is far from the gate, and the variable `arrive` is set to `false`. `pc` can have the value `far` indefinitely, as indicated by the first `wait` command, or can take on the value `near`. When the train moves to `near`, the timer x is reset to 0 and the interface variable `arrive` is set to `true`. The second `wait` command puts an upper bound of 5 on x while `pc` = `near`. While `pc` = `near`, if $x \geq 3$, then the guard of the second `update` command is satisfied, which means that after spending 3 time units at `pc` = `near`, `pc` can move to `far`. After 5 time units at `pc` = `near`, the guard of the second `wait` statement is no longer satisfied, which forces the second update command to be executed. After 5 time units, `pc` *must* move to `far`.

The timed reactive module description for a gate controller given in Figure 7.2 operates in a similar fashion. The system consisting of the train and the controller is then given by

```
RealTimeTrainSystem :=
  hide arrive in RealTimeTrain || RealTimeGate endhide
```

Transition Relations of Real-Time Modules

Currently, MOCHA restricts guards on clocks and clock invariants to be positive Boolean combinations of inequalities of the form $x \leq c$ and $x \geq c$, where $c \in \mathbb{N}$.

[2]For a precise treatment of the semantics of timed modules, refer to [AH97]

```
module RealTimeGate
  external arrive :  bool
  interface pc :  {open,closing,closed}
  private y :  clock
  atom controls pc, y reads pc, y, arrive awaits arrive
  init
    [] true -> pc' := open
  update
    [] pc=open & arrive' -> pc' := closing; y' := 0
    [] pc=closing & y>=1 -> pc' := closed; y' := 0
    [] pc=closed & y>=7  -> pc' := open
  wait
    [] pc=open & ~arrive' ->
    [] pc=closing & y<=2  -> y' :<= 2
    [] pc=closed & y<=7   -> y' :<= 7
  endatom
endmodule
```

Figure 7.2: Real-time railroad gate controller

This is adequate for modeling the behavior of physical systems. In [HMP92], it is proven that, with this restriction, for each trace γ of a timed module, there exists a trace $[\gamma]$ such that (i) the sequence values that discrete variables take on is the same for γ and $[\gamma]$ and (ii) all updates of discrete variables take place at integer-valued points in time. This enables clocks to be modeled as integer-valued variables that increase at the same rate. Timed modules are converted by MOCHA into (untimed) modules, equivalent to the original ones in the sense described above.

7.2 Verification with Real-Time Modules

The examples used in this section are located in the `examples/` directory of the MOCHA distribution.

7.2.1 Parsing Real-Time Modules

As is the case with untimed modules, real-time modules are read in using the `read_module` command. In the rest of this section, we use the real-time train example, which can be found in `examples/rt_train_control`. On a `read_module` command, MOCHA displays the names of the modules that were successfully parsed. In the case of a parse error, an appropriate message is displayed.

```
mocha:  read_module rt_train_control.rm
Module RealTimeGate is composed and checked in.
Module RealTimeTrain is composed and checked in.
Module System is composed and checked in.
```

```
parse successful.
```

All the commands described in Section 5.1 can be used with real-time modules in an identical manner.

7.2.2 Invariant Checking

The simplest form of invariant checking is computing the set of reached states of a real-time module. This set can be computed and stored in a region using the `rtm_search` command. First, the transition relation of the module needs to be computed using `rtm_trans`.

```
mocha:  rtm_trans System
Ordering variables using rtm_static_order
Transition relation computed :  2 conjuncts
Done initializing image info...
mocha:  rtm_search System
Initial Region Computed...
Step 1:  image mdd size = 20 |states| = 6
reached set mdd size = 20 |states| = 3
Step 2:  image mdd size = 25 |states| = 8
reached set mdd size = 26 |states| = 5
Step 3:  image mdd size = 29 |states| = 12

...

Step 24:  image mdd size = 16 |states| = 28
reached set mdd size = 40 |states| = 168
Step 25:  image mdd size = 14 |states| = 36
Done reached set computation...
reached set mdd size = 40 number of states = 168
System.r0
```

`System.r0` is the name of the region containing the set of reached states for the module `System`. The region corresponding to the set of initial states can be obtained using the `rtm_init` command.

Invariants need to be specified in a `.spec` file, as is the case with untimed modules. The file `examples/rt_train_control/rt_train_control.spec` contains the following specification:

```
inv "safe" ~((pcTrain = gate) & ~(pcGate = closed));
```

The invariant `safe` is read in and checked as follows:

```
mocha:  read_spec rt_train_control.spec
safe
```

```
mocha:  inv_check -r System safe
Typechecking invariant safe...
Typechecking successful
Initial Region Computed...
Step 1:  image mdd size = 20 |states| = 6
reached set mdd size = 20 |states| = 3
Step 2:  image mdd size = 25 |states| = 8
reached set mdd size = 26 |states| = 5

...

Step 25:  image mdd size = 14 |states| = 36
Done reached set computation...
reached set mdd size = 40 number of states = 168

Invariant safe failed in step 14

Counterexample for invariant safe
arrive=0 pcTrain=far $x_0=0 pcGate=open $y_0=0 timeIncrement=0
arrive=1 pcTrain=near $x_0=0 pcGate=closing $y_0=0 timeIncrement=1
arrive=1 pcTrain=near $x_0=1 pcGate=closing $y_0=1 timeIncrement=0
arrive=1 pcTrain=near $x_0=1 pcGate=closed $y_0=0 timeIncrement=1
arrive=1 pcTrain=near $x_0=2 pcGate=closed $y_0=1 timeIncrement=1
arrive=1 pcTrain=near $x_0=3 pcGate=closed $y_0=2 timeIncrement=1
arrive=1 pcTrain=near $x_0=4 pcGate=closed $y_0=3 timeIncrement=0
arrive=1 pcTrain=gate $x_0=0 pcGate=closed $y_0=3 timeIncrement=1
arrive=1 pcTrain=gate $x_0=1 pcGate=closed $y_0=4 timeIncrement=0
arrive=0 pcTrain=far $x_0=1 pcGate=closed $y_0=4 timeIncrement=0
arrive=1 pcTrain=near $x_0=0 pcGate=closed $y_0=4 timeIncrement=1
arrive=1 pcTrain=near $x_0=1 pcGate=closed $y_0=5 timeIncrement=1
arrive=1 pcTrain=near $x_0=2 pcGate=closed $y_0=6 timeIncrement=1
arrive=1 pcTrain=near $x_0=3 pcGate=closed $y_0=7 timeIncrement=0
arrive=1 pcTrain=gate $x_0=0 pcGate=open $y_0=7
```

Note that the command `inv_check` needs to be used with the `-r` option for MOCHA to use the real-time interpretation on a given module. Otherwise the `inv_check` command functionality remains the same. The variable `timeIncrement` is a variable internal to MOCHA and indicates how much time elapses while the discrete-valued variables keep their current values.

The invariant `safe` in the example above would have passed if the train were allowed to approach the gate only once, because the gate closes in at most 2 time units after detecting the *arrive* signal, whereas the train takes at least 3 time units to reach the gate after *arrive* becomes true. However, the invariant `safe` has failed because the train is allowed to enter the gate a second time before the gate has had the time to close.

Bibliography

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

[AH97] R. Alur and T.A. Henzinger. Modularity for timed and hybrid systems. In *CONCUR 97: Proceedings of the 8th International Conference on Concurrency Theory*, Lecture Notes in Computer Science 1243, pages 74–88. Springer-Verlag, 1997.

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

[AHR98] R. Alur, T.A. Henzinger, and S.K. Rajamani. Symbolic exploration of transition hierarchies. In B. Steffen, editor, *TACAS 98: Tools and Algorithms for Construction and Analysis of Systems*, Lecture Notes in Computer Science 1384, pages 330–344. Springer-Verlag, 1998.

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

[BHSV+96] R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, and T. Villa. VIS: A System for Verification and Synthesis. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 428–432. Springer-Verlag, 1996.

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

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, 1988.

[Dil96] D. L. Dill. The Murϕ Verification System. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 390–393. Springer-Verlag, 1996.

[Exp97] Expert Interface Technologies. *Tix Home Page*, 1997. `http://www.xpi.com/tix/index.html`.

[HMP92] T.A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *ICALP '92: 19th International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science 623, pages 545–558. Springer-Verlag, 1992.

[HQR98] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee: methodology and case studies. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science, pages 440–451. Springer-Verlag, 1998.

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

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

Index