MCMAS v0.9.6.2: User Manual

Contents

1 Introduction **1**

1.1 Introduction 1

1.2 For the impatient 2

1.2.1 For the *very* impatient 3

2 Tutorial **4**

2.1 Tutorial 4

2.1.1 How to describe a system of agents? 4

2.1.2 A concrete example: the bit transmission problem and its encoding in ISPL 6

2.1.3 Verification and simulation 10

2.1.4 A more complex example: the protocol of the dining cryptographers 13

3 Reference **16**

3.1 Command line options 16

3.2 ISPL syntax 18

3.2.1 ISPL overview 18

3.2.2 Reserved keywords 23

3.2.3 The grammar 25

3.3 The graphical interface 29

3.4 Theoretical background: the semantics of interpreted systems . . 33

Chapter 1

Introduction

1.1 Introduction

MCMAS is a Model Checker for Multi-Agent Systems (MAS). MCMAS takes in input a MAS specification and a set of formulae to be verified, and it evaluates the truth value of these formulae using algorithms based on Ordered Binary Decision Diagrams (OBDDs [1]). Whenever possible, MCMAS produces counterexamples for false formulae and witnesses for true formulae. MCMAS allows the verification of a number of modalities, including CTL operators, epistemic operators, operators to reason about correct behaviour and strategies, with or without fairness conditions.

MCMAS can also be used to run interactive, step-by-step simulations. Additionally, a graphical interface is provided as an Eclipse plug-in which includes a graphical editor with syntax recognition, a graphical simulator, and a graphical analyser for counterexamples.

Multi-Agent Systems are described in MCMAS using a dedicated programming language derived from the formalism of *interpreted systems* [4]. This language, called ISPL (Interpreted Systems Programming Language), resembles the SMV language in that it characterises agents by means of variables and represents their evolution using Boolean expressions.

IMPORTANT NOTICE: MCMAS is being actively developed and it has to be considered as an academic prototype. Bugs are likely to be present and we welcome all bug reports, which we try to address in the shortest possible time.

The remainder of this document is organised as follows:

- Section 2.1 is a simple tutorial providing a short introduction to the formalism of interpreted systems, a flavour of ISPL and basic MCMAS commands.

- Section 3.1 describes the command line options for the MCMAS executable.

- Section 3.2 contains the complete ISPL syntax.
- Section 3.3 describes the graphical interface.
- Section 3.4 presents a detailed description of the theoretical background of MCMAS.

1.2 For the impatient

System requirements:

- Tested platforms: x86 compatible 32 bit or 64 bit processor; ppc and MacIntel.
- Operating system: Linux, Mac OSX 10.3 and 10.4, Windows using Cygwin;
- Compiler: flex 2.5.4 or higher, GNU bison 2.3 or higher, GNU g++ 4.0.1 or higher;
- Eclipse 3.2 or higher with Java 1.6 (optional, for the graphical interface).

Please feel free to contact us at `f.raimondi@cs.ucl.ac.uk` and/or `hongyang@doc.ic.ac.uk` if you want to run MCMAS on different architectures / configurations.

Installation steps

1. (Windows platform only) Install cygwin and the packages *g++*, *flex* and *bison* on Windows XP/Vista. Detailed instructions can be found from `http://www.cygwin.com/`.
2. Install the CUDD library. This library can be obtained from `http://vlsi.colorado.edu/~fabio/CUDD/`:
   - `tar -xzvf cudd-2.4.1.tar.gz`
   - `cd cudd-2.4.1`
   - Edit Makefile for your architecture (default: Linux x86), then `make`
   - Build the support for C++: `cd obj` then `make testobj`
3. Extract MCMAS sources with `tar`. Modify `Makefile` and change the location of the CUDD library to the correct location on your machine, i.e. change the line `CUDD = /usr/local/cudd-2.4.1/` as appropriate. Then type `make` and you should obtain the executable `mcmas`.
4. (Optional) Install Eclipse plug-in for the graphical editor by copying the file `org.mcmas.ui_1.0.0.jar` files to the `plugin/` directory of your Eclipse installation. Run Eclipse with the option "-clean" for the first time and specify the path to MCMAS in the MCMAS preference.

Running mcmas

- `./mcmas -h` from the command line. See the examples in the `examples` directory.
- Graphical interface: start Eclipse; if the plugin has been recognised, you should be able to create a new ISPL project and create a new ISPL file.

1.2.1 For the *very* impatient

We might be able to provide a pre-compile binary version for your system, please contact us at `f.raimondi@cs.ucl.ac.uk` and/or `hongyang@doc.ic.ac.uk`.

Chapter 2

Tutorial

2.1 Tutorial

2.1.1 How to describe a system of agents?

Various techniques and languages exists to describe a system of agents. MCMAS adopts and extends the formalism of interpreted systems [4] using the dedicated ISPL language. We distinguish between two kinds of agents: “standard” agents, and the environment agent. The environment is used to describe boundary conditions and infrastructures shared by “standard” agents and it is modelled similarly to standard agents (see below).

In brief, in MCMAS each agent (including the environment) is characterized by:

1. A set of local states (for instance the states “ready” or “busy” for a receiver).

2. A set of actions (for instance “sendmessage” or “open channel”).

3. A *rule* describing which action can be performed by an agent in a given local state. We call this rule *a protocol*[1].

4. An evolution function, describing how the local states of the agents evolve based on their current local state and on other agents’ actions.

Local states. Local states are defined in terms of *local variables*: as an example, consider a printer with two sensors, one sensor for toner (which could be high or low), and one sensor for paper (which could be full or empty). In this case, the agent printer has four possible local states corresponding to all the possible combinations of values of toner and paper. Local states are private, i.e., each agent can observe only its own local states, and all the other parameters discussed below (protocol and evolution function) *cannot* refer to other agents’

[1]Not to be confused with the notion of protocol in networking.

local variables. The only exception is the environment agent: for this agent two kind of variables can be defined: "standard" variables and *observable* variables. "Standard" agents can "peek" at the observable variables of the environment and their evolution function can refer to these variables. Additionally, the epistemic accessibility relation of an agent (see Section 3.4) is based on the agent's local states and on the environment local states. Intuitively, an agent "knows" something in a state of the system if this something is true in all the states of the system in which its local states and the observable variables of the environment remain the same.

Actions. Each agent (including the environment) is allowed to perform some actions, for instance send a message. It is assumed that all actions performed are visible by all the other agents.

Protocols. Protocols describe which actions can be performed in a given local state. As local states are defined in terms of variables, the protocol for an agent is expressed as a function from variable assignments to actions. In ISPL protocols are not required to be exhaustive: it is sufficient to specify only the variables assignments relevant to the execution of certain actions, and introduce a catch-all assignment by means of the keyword `Others` (see below). Protocols are not required to be deterministic: it is possible to associate a *set* of actions to a given variable assignment. In this case the action to be performed is selected non-deterministically in this set.

Evolution functions. The evolution function for an agent describes how variable assignments change as a results of the actions performed by all the other agents. For instance, the evolution function for a printer could prescribe that, if the current local state (or a variable composing the local state) is "ready" and an agent performs the action "send print job", then the next local state of the printer is "busy". Formally, the evolution function is a function returning a "next" assignment to the local variables of an agent as a function of the "current" set of assignments to local variables, the observable variables of the environment, and the actions performed by the agents. A global evolution function is computed by taking the conjunction of all the agents' evolution functions.

The description of a MAS using ISPL is completed by the declaration of a set of initial states expressed as assignments to local variables. If more than one state satisfies the assignments, then the initial state is selected randomly. The system evolves from this set of initial states in accordance to the protocols and the evolutions functions, and this process is used to compute the truth value of formulae specified by the user. Fairness conditions can also be specified in ISPL, to rule out unwanted behaviour (e.g. a communication channel being continuously noisy or a printer being locked forever by a single agent)

In the next sections we will provide two concrete examples and their encoding in ISPL. We refer to Section 3.4 for a more formal definition of ISPL and its semantics.

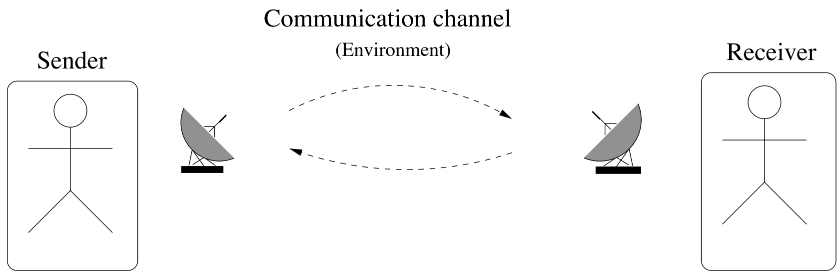

Figure 2.1: The bit transmission problem.

2.1.2 A concrete example: the bit transmission problem and its encoding in ISPL

In the bit-transmission problem [4] a sender S wants to communicate the value of a bit to a receiver R, by using an unreliable communication channel (see Figure 2.1). In this example, the channel may drop messages, but cannot tamper messages; also, at any given time, the channel may transmit messages in one direction but not in the other.

One mechanism to achieve communication is as follows: S immediately starts sending the bit to R, and continues to do so until it receives an acknowledgement from R. R does nothing until it receives the bit; from then on, it sends messages acknowledging the receipt to S. S stops sending the bit to R when it receives the first acknowledgement from R, and the protocol terminates here.

To encode this example in the formalism of interpreted systems we first introduce an Environment agent (notice: this agent is required is every ISPL file), whose ISPL code is as follows:

```
Agent Environment
  Obsvars:
  end Obsvars
  Vars:
    state : {S,R,SR,none};
  end Vars
  RedStates:
  end RedStates
  Actions = {S,SR,R,none};
  Protocol:
    state=S: {S,SR,R,none};
    state=R: {S,SR,R,none};
    state=SR: {S,SR,R,none};
    state=none: {S,SR,R,none};
  end Protocol
  Evolution:
    state=S if (Environment.Action=S);
```

```
    state=R if (Environment.Action=R);
    state=SR if (Environment.Action=SR);
    state=none if (Environment.Action=none);
  end Evolution
end Agent
```

In this case, the Environment does not have observable variables (this section is empty), and it only has one variable `state` representing the availability of the communication channel (e.g. `SR` represents the fact that both directions are open for communication). Thus, the Environment agent has 4 possible local states. The Environment can perform four actions (in this case we use the same names for local states and actions): transmit the message from Sender only, from both Sender and Receiver, from Receiver only, or don't transmit any message. The protocol in this case simply prescribes that in every state any action can be chosen (non-deterministically) by the agent Environment. The Evolution function is defined as follows: take the first line below `Evolution:`, this is read as "the *next* state will be `S` if the (current) Action of the Environment is `SR`". Essentially, the evolution function simply records in the local state of the Environment the last action performed. In general, a line in the evolution function is triggered when the Boolean condition to the right of the `if` keyword becomes true.

We encode the agent `Sender` by means of the following ISPL code:

```
Agent Sender
  Vars:
    bit : { b0, b1}; -- The bit can be either zero or one
    ack : boolean;  -- This is true when the ack has been received
  end Vars
  RedStates:
  end RedStates
  Actions = { sb0,sb1,nothing };
  Protocol:
    bit=b0 and ack=false : {sb0};
    bit=b1 and ack=false : {sb1};
    ack=true : {nothing};
  end Protocol
  Evolution:
    (ack=true) if (ack=false) and
       ( ( (Receiver.Action=sendack) and (Environment.Action=SR) )
        or
       ( (Receiver.Action=sendack) and (Environment.Action=R) )
       );
  end Evolution
end Agent
```

Notice that this is a "standard" agent and no observable variables are present. Two variables are declared in the `Vars` section: an enumeration type `bit` encoding the value of the bit the Sender wants to send, and a Boolean variable `ack` encoding whether or not an acknowledgement has been received (comments can be added by escaping the commented text with the prefix `--`). Therefore, the Sender has four possible local states corresponding to all the possible combination of values of `bit` and `ack`. Three actions are declared for the sender: send bit 0, send bit 1, and do nothing. The Protocol section for the Sender defines how these actions are performed. In general, each line of the protocol starts with a Boolean condition on the values of the variables, followed by a colon, followed by a list of actions that are allowed when the Boolean condition is true. The lines of the protocol do not need to be exhaustive: if they are not, the special keyword `Other` needs to be used to specify what to do when none of the Boolean condition is true (for instance by introducing a "nothing" action as in this case). The evolution function function in this case is straightforward: the Sender changes the value of the variable `ack` only if it is false and an acknowledgement is received from the Receiver (and the variable `bit` does not change its value); notice how other agents' actions are scoped with the syntax construct `AgentName.Action`. If no scoping prefix is added, the value is intended to refer to the agent in which the condition is declared. As in the case of protocols, the list of Boolean conditions does not need to cover all possible cases: MCMAS assumes that, if none of the Boolean conditions is true, then the local state of the agent does not change.

We encode the agent Receiver by means of the following ISPL code:

```
Agent Receiver
  Vars:
    state : { empty, r0, r1 };
  end Vars
  RedStates:
  end RedStates
  Actions = {nothing,sendack};
  Protocol:
    state=empty : {nothing};
    (state=r0 or state=r1): {sendack};
  end Protocol
  Evolution:
    state=r0 if ( ( (Sender.Action=sb0) and (state=empty) and
                    (Environment.Action=SR) ) or
                  ( (Sender.Action=sb0) and (state=empty) and
                    (Environment.Action=S) ) );
    state=r1 if ( ( (Sender.Action=sb1) and (state=empty) and
                    (Environment.Action=SR) ) or
                  ( (Sender.Action=sb1) and (state=empty) and
                    (Environment.Action=S) ) );
  end Evolution
end Agent
```

Only one enumeration variable is declared for this agent, representing whether or not the bit has been received, and its value. Agent Receiver can perform two actions: either do nothing (if state is `empty`), or send an acknowledgement if a bit has been received. Receiver evolves to state `r0` if it was in state `empty` and the sender is performing the action of sending bit 0, and the Environment is enabling transmission either in both direction (`Environment.Action=SR`), or at least from the sender (`Environment.Action=S`). The evolution to state `r1` is similar.

After the declaration of Environment and agents, five more sections are required to complete the ISPL input to MCMAS: `Evaluation`, `InitStates`, `Groups`, `Fairness`, and the list of formulae to be verified:

```
Evaluation
  recbit if ( (Receiver.state=r0) or (Receiver.state=r1) );
  recack if ( ( Sender.ack = true ) );
  bit0 if ( (Sender.bit=b0));
  bit1 if ( (Sender.bit=b1) );
  envworks if ( Environment.state=SR );
end Evaluation

InitStates
  ( (Sender.bit=b0) or (Sender.bit=b1) )
  ( Receiver.state=empty ) and ( Sender.ack=false) and
  ( Environment.state=none );
end InitStates

Groups
  g1 = {Sender,Receiver};
end Groups

Fairness
  envworks;
end Fairness

Formulae
  AF(K(Sender,K(Receiver,bit0) or K(Receiver,bit1)));
  AG(recack -> K(Sender,(K(Receiver,bit0) or K(Receiver,bit1))));
end Formulae
```

The Evaluation section introduces the Boolean variable that are used in Fairness conditions and in the formulae to be verified. These Boolean formulae are defined by Boolean expressions over the local states of the agents. For instance, the proposition `recbit` is true if the local state of the Receiver is `r0` or `r1`.

The section `InitStates` declares the set of initial states by using a Boolean expression over local states. In this case, there are two possible initial states, one where the bit value is `b0` and one where the bit value is `b1`, and with `ack=false` and all the other local states for Receiver and Environment set to their empty value.

The section `Groups` allows for the definition of groups of agents, that can be used in the verification of group modalities in the `Formulae` section.

The section `Fairness` contains a list of Boolean expressions: intuitively, it is required that all the formulae listed in this section must be true infinitely often along all executions. For instance, in the example above it is required that the proposition `envworks` is true infinitely often, meaning that the environment cannot avoid the state `SR` forever.

The section `Formulae` contains the list of formulae to be verified. Formulae are built using CTL temporal operator, epistemic operators, operators to reason about correct behaviour and strategies. In the example listed above, the first formula is read as "along all paths, at some point in the future the sender will know that the receiver knows that the bit value is either 0 o 1". This formula is true in this particular case (see below) because of the fairness condition `envworks`. If this fairness condition is commented, then the formula becomes false (because the Environment could forbid communication indefinitely). The second formula claims that "it is always true that, if an acknowledgement had been received, then the sender knows that the receiver knows the value of the bit". This formula is true even if the fairness condition is removed.

The example presented in this section and additional formulae can be found in the text file `examples/btp.ispl` in the source distribution of MCMAS.

2.1.3 Verification and simulation

In this section we present how to run MCMAS from the command line to perform verification and simulation of the example presented in the previous section.

The minimal MCMAS execution consists in the invocation of the executable followed by the name of the ispl file to be verified:

```
$ ./mcmas examples/btp.ispl
********************************************************************************
                       MCMAS v0.9.5

 This software comes with ABSOLUTELY NO WARRANTY, to the extent
 permited by applicable law.

 Please check
 http://www.cs.ucl.ac.uk/staff/f.raimondi/MCMAS/
 for the latest release.
 Report bugs to <hongyang.qu@imperial.ac.uk> or <f.raimondi@cs.ucl.ac.uk>
********************************************************************************

examples/btp.ispl has been parsed successfully.
Gloabl syntax checking...
Done
```

```
Encoding BDD parameters...
Done.
Checking formulae...
Building set of fair states...
  Formula number 0: (AF K(Sender, (K(Receiver, bit0) ||
       K(Receiver, bit1)))),
      is TRUE in the model
  Formula number 1: (AG (recack -> K(Sender, (K(Receiver, bit0) ||
       K(Receiver, bit1))))),
      is TRUE in the model
done, 2 formulae successfully read and checked
execution time = 0
number of reachable states = 18
BDD memory in use = 4306216
```

In this case, if the syntax is correct, MCMAS simply outputs the result of the evaluation of the formulae found in the `Formulae` section of the ISPL file. MCMAS performs a detailed syntax check of the input file and the verification process is not invoked if a syntax error is present. In case of errors, MCMAS terminates with a warning and details of the error. As an example, if the section `ObsVars` is not declared in the agent Environment, MCMAS terminates with the following error:

```
$ ./mcmas examples/btp.ispl
************************************************************************
                       MCMAS v0.9.5

 This software comes with ABSOLUTELY NO WARRANTY, to the extent
 permited by applicable law.

 Please check
 http://www.cs.ucl.ac.uk/staff/f.raimondi/MCMAS/
 for the latest release.
 Report bugs to <hongyang.qu@imperial.ac.uk> or <f.raimondi@cs.ucl.ac.uk>
************************************************************************

examples/btp.ispl:10.1-4: syntax error, unexpected VARS, expecting OBSVARS
examples/btp.ispl has syntax error(s).
```

A number of options are available to compute counterexamples, to increase the verbosity level, etc. These options are explained in detail in Section 3.1.

One important feature of MCMAS is the possibility of running simulations. The simulation environment is started with the option `-s` from the command line:

```
$ ./mcmas -s examples/btp.ispl
************************************************************************
```

```
                    MCMAS v0.9.5

 This software comes with ABSOLUTELY NO WARRANTY, to the extent
 permited by applicable law.

 Please check
 http://www.cs.ucl.ac.uk/staff/f.raimondi/MCMAS/
 for the latest release.
 Report bugs to <hongyang.qu@imperial.ac.uk> or <f.raimondi@cs.ucl.ac.uk>
********************************************************************************

examples/btp.ispl has been parsed successfully.
Gloabl syntax checking...
Done
Encoding BDD parameters...

--------- Initial state ---------
Agent Environment
  state = none
Agent Sender
  ack = false
  bit = b0
Agent Receiver
  state = empty
----------------------------
Is this the initial state? [Y(es), N(ext), E(xit)]:
```

MCMAS stops at this point waiting for input from the user. It is possible to go through all the possible initial states with the keys N (Next) and P (Previous). User Y to select an initial state:

```
--------- Initial state ---------
Agent Environment
  state = none
Agent Sender
  ack = false
  bit = b0
Agent Receiver
  state = empty
----------------------------
Is this the initial state? [Y(es), N(ext), E(xit)]: Y
Enabled transtions:
1 : Environment : none; Sender : sb0; Receiver : nothing
2 : Environment : SR; Sender : sb0; Receiver : nothing
3 : Environment : S; Sender : sb0; Receiver : nothing
4 : Environment : R; Sender : sb0; Receiver : nothing
Please choose one, or type 0 to backtrack or -1 to quit:
```

When a state is chosen, MCMAS outputs the possible transition from that state. Transitions can be chosen by typing the corresponding number (in the example below transition number two is chosen):

```
Please choose one, or type 0 to backtrack or -1 to quit:
2

--------- Current state ---------
Agent Environment
  state = SR
Agent Sender
  ack = false
  bit = b0
Agent Receiver
  state = r0
----------------------------
Enabled transtions:
1 : Environment : S; Sender : sb0; Receiver : sendack
2 : Environment : R; Sender : sb0; Receiver : sendack
3 : Environment : none; Sender : sb0; Receiver : sendack
4 : Environment : SR; Sender : sb0; Receiver : sendack
Please choose one, or type 0 to backtrack or -1 to quit:
```

When a transition is chosen, MCMAS displays the new state and the transitions available in the new state. Notice that it is always possible to backtrack using 0, or to exit using -1.

2.1.4 A more complex example: the protocol of the dining cryptographers

The protocol of the dining cryptographers was introduced in [2]. The original wording from [2] is as follows:

"*Three cryptographers are sitting down to dinner at their favourite three-star restaurant. Their waiter informs them that arrangements have been made with the maitre d'hotel for the bill to be paid anonymously. One of the cryptographers might be paying for the dinner, or it might have been NSA (U.S. National Security Agency). The three cryptographers respect each other's right to make an anonymous payment, but they wonder if NSA is paying. They resolve their uncertainty fairly by carrying out the following protocol:*

Each cryptographer flips an unbiased coin behind his menu, between him and the cryptographer on his right, so that only the two of them can see the outcome. Each cryptographer then states aloud whether the two coins he can see – the one he flipped and the one his left-hand neighbour flipped – fell on the same side or on different sides. If one of the cryptographers is the payer, he states the opposite of what he sees. An odd number of differences uttered at the table indicates that a cryptographer is paying; an even number indicates *that NSA is paying (assuming that the dinner was paid for only once). Yet if a cryptographer is paying, neither of the other two learns anything from the utterances about which cryptographer it is.*"[2]

Notice that similar versions of the protocol can be defined for any number of cryptographers greater than three.

We model an instance of this example with three cryptographers by introducing three agents C_i ($i = \{1, 2, 3\}$) to model the three cryptographers, in addition to the environment agent.

The environment is used to select non-deterministically the identity of the payer and the results of the coin tosses. We introduce three variables for the environment, one for each coin. Also, we introduce an *observable variable* to record the result of the utterances (even or odd):

```
Agent Environment
  Obsvars:
    numberofodd : { none, even, odd };
  end Obsvars

  Vars:
    coin1 : {head,tail};
    coin2 : {head,tail};
    coin3 : {head,tail};
  end Vars
[...]
```

It is assumed that the environment can perform only one action, the null action. Therefore, the protocol P_E is simply mapping every local state to the null action by means of the `Other` keyword. The evolution function of the environment determines the evolution of the observable variable only to record the result of the utterances.

The local states of a cryptographer are composed by four variables representing, respectively, whether the cryptographer is the payer, the value of the coins, and whether the coins are equal or different. Each cryptographer can perform one of three actions: say "equal", say "different", or do nothing. These actions are performed in accordance with the protocol derived from the informal description above. The evolution function for the cryptographers simply updates the variable recording whether or not the coins that can be seen are equal.

There are 32 possible initial states, corresponding to the possible combinations of coin tosses and payers. In this example no fairness condition is required and MCMAS can be used to check the characteristic properties of this example: if there is an odd number of utterances, then someone at the table paid the bill. In this case, it is also true that a cryptographer did not pay for the dinner, the this cryptographer knows that a cryptographer paid for it, but he does not know who is the actual payer. This is captured by the following formula:

```
( (odd and !c1paid) -> (K(DinCrypt1,(c2paid or c3paid) ) ) and
```

```
              !K(DinCrypt1,c2paid) and !K(DinCrypt1,c3paid) );
```

where `c1paid` is an atomic proposition which is true if the first cryptographer paid the dinner (and similarly for 2 and 3), and `odd` is an atomic proposition true when there is an odd number of utterances.

Conversely, an even number of utterances implies that all the cryptographers know that the company paid for the dinner. The following formula captures that the first cryptographer knows this fact:

```
( (even) -> (K(DinCrypt1,!c2paid) and K(DinCrypt1,!c3paid) ) );
```

The ISPL code for this example can be found in the source distribution under the directory `examples/din-crypt`.

Chapter 3

Reference

3.1 Command line options

The available command line options are displayed by running MCMAS with the `-h` option:

```
$ ./mcmas -h
*********************************************************************
                       MCMAS v0.9.5

 This software comes with ABSOLUTELY NO WARRANTY, to the extent
 permited by applicable law.

 Please check
 http://www.cs.ucl.ac.uk/staff/f.raimondi/MCMAS/
 for the latest release.
 Report bugs to <hongyang.qu@imperial.ac.uk> or <f.raimondi@cs.ucl.ac.uk>
*********************************************************************

Usage: mcmas [OPTIONS] FILE
Example: mcmas -v 3 -bdd_stats myfile.ispl

Options:
  -s              Interactive execution
  -v Number       verbosity level ( 0 -- 5, default 0 )
  -e Number       Choose the approach to do model checking (0 -- 2, default 1)
  -bc             Store some BDDs to speed up verification
  -if             Encode initial states first
  -bdd_stats      Print BDD statistics
  -src            Print parsed ISPL source code and exit
  -gsc            Check the syntax and exit
  -c              Print counterexamples or witness executions
```

```
-cex_prefix    Destination directory for counterexample files
-g             Check formulas globally (i.e., not only on initial states)
-d             Run in debug mode
-h             This screen
```

More in detail:

- `-s`: this option invokes the interactive mode execution (see previous section).
- `-v number`: this option is used to modify the verbosity level. It is particularly useful to detect the bottlenecks in large examples and to investigate unexpected behaviours of MCMAS or bugs.
- `-e number`: this option is used to switch between different strategies for the computation of the transition relation and reachable states. The value 0 pre-computes a monolithic transition relation. Option 1 and 2 postpone this computation and they differ on how reachable states are computed internally.
- `-bc`: this option stores some BDDs in memory to speed up verification at the expenses of more memory consumption.
- `-if`: this option is used to decide when to compute the set of initial states. In a normal execution, initial states are computed at the beginning; however, in some cases (e.g., simulation) it is not necessary to compute this set.
- `-bdd_stats`: this option is used to print statistics on OBDDs at the end of the execution. Using this option it is possible to estimate memory consumption and the compression level.
- `-src`: this option prints a parsed and cleaned version of the input file.
- `-gsc`: this option verifies the correct syntax of the input file and exits without performing verification.
- `-c`: this option is used to compute counterexamples (for false universal formulae) and witnesses (for true existential formulae). For each formula for which such computation is possible, MCMAS emits two files: a .dot file encoding the graphical representation of the counterexample/witness path, and a .info file containing a detailed description of the states in the path. These files are named cexN(.dot/.info), where N is the number of the formula. These files are used by the graphical interface to display a graphical representation of the traces. A textual representation is also printed on screen.
- `-cex_prefix dirname`: this option allows to specify a path where the .dot/.info file are written (the default is to write this file in the directory where MCMAS is launched from).

- `-g`: this option performs verification on the set of all reachable states. The default behaviour is to check formulae in the set of initial states only (if a global check is required, then AG(formula) should be used).

- `-d`: this option is used to run MCMAS in debug mode. This is particularly useful to trace the cause of unexpected behaviour or bugs. Notice: this option produces a lot of output to the screen.

3.2 ISPL syntax

In this section we present the formal syntax of ISPL. Section 3.2.1 provides an overview of the language. Section 3.2.2 lists the reserver keywords which cannot be used as identifiers. Section 3.2.3 reports the formal grammar of ISPL using a bison-like syntax. See Section 2.1.2 for an informal description of the various components of an ISPL file.

3.2.1 ISPL overview

A multi-agent system specified in ISPL is composed of an Environment agent and a set of normal agents. Each agent has a set of local variables and the Environment also has a set of observable variables, which can be "observed" by other agents. The local states of an agent, each of which contains a valuation of its local variables (and observable variables if the agent is the Environment), are partitioned into two sets: the set of green states and the set of red states. The two sets are used to check correct behaviour properties. Every agent also has a set of actions, a protocol function and an evolution function.

The ISPL specification also contains the definition of initial states, propositions, groups, fairness formulae and formulae to be checked.

Below is the general structure of a model.

```
Agent Environment
  Obsvars:
  ...
  end Obsvars
  Vars:
  ...
  end Vars
  RedStates:
  ...
  end RedStates
  Actions = {...};
  Protocol:
  ...
  end Protocol
  Evolution:
  ...
```

```
  end Evolution
end Agent

Agent TestAgent
  Vars:
  ...
  end Vars
  RedStates:
  ...
  end RedStates
  Actions = {...};
  Protocol:
  ...
  end Protocol
  Evolution:
  ...
  end Evolution
end Agent

Evaluation
...
end Evaluation

InitStates
...
end InitStates

Groups
...
end Groups

Fairness
...
end Fairness

Formulae
...
end Formulae
```

Note that all the strings in the structure above (except `TestAgent`, which is the name of a normal agent) are reserved keywords. More agents could be defined similarly to `TestAgent`. The following sections explain the details of each section of an ISPL file.

Definition of variables

Currently, ISPL allows three types of variables: Boolean, enumeration and bounded integer. Suppose `x`, `y` and `y` are variables of Boolean, enumeration and bounded integer respectively. They are be defined as follows:

```
x : boolean;
y : {a, b, c};
z : 1 .. 4;
```

Note that the value of `x` can be `true` or `false`, the value of `y` is one of `a`, `b` and `c`, and the value of `z` can be 1, 2, 3, or 4. The *lower bound* and the *upper bound* of `z` are 1 and 4 respectively. The definition of a local variable and the definition of an observable variable are the same. A comparison over Boolean variables or enumeration variables can only be an equality test, e.g., `x = true`, `y = a`, `x != false` or `y != b`. Arithmetic operations "=", "!=", "<", "<=", ">", ">=" are allowed for bounded integers, e.g., `z < 2` or `z >= z * 2 - 3`.

Definition of red states

The red states of an agent are represented by a Boolean formula over its local variables (and observable variables if the agent is the Environment). That is, all the local states that satisfy the formula are red, while the other local states are green. Allowed Boolean operators are `and`, `or` and ! (for *not*). For example, `x = true and (!(y = a) or z > 3)` is an acceptable Boolean formula for red states.

Definition of actions

All actions of an agent are defined in the section `Actions`:

```
Actions = { a1, b2, c3};
```

Definition of protocol function

A line in a protocol function is composed of a condition, which is a Boolean formula over local states, and a list of actions. The condition represents all local states that satisfy the condition and the list of actions allowed to be performed in local states specified by the condition. In this example:

```
x = true and z < 2 : { a1, a3};
```

`x = true and z < 2` is the condition and {`a1`, `a3`} is the list of actions. The conditions appearing in different lines do not need to be mutually exclusive, i.e., the conjunction of these two conditions needs not to be false. If this is the case, the agent has nondeterministic behaviour and all behaviours are considered possible by MCMAS.

For an agent that has many local states, it might be unrealistic or even impossible to specify actions for every state. MCMAS includes the reserved keyword `Other`:

```
Other : { action-list };
```

This item is optional, but it must be the last one in a protocol function if it is used. The keyword `Other` encodes all states except those specified in any line appearing before it. This keyword is also useful if the same set of actions is allowed in all local states. In this case, simply let the `Other` item be the only one in the protocol function.

Definition of evolution function

A line in an evolution function consists of a set of assignments of local variables (and observable variables for the Environment) and an *enabling condition*, which is a Boolean formula over local variables, observable variables of the Environment, and actions of all agents.

The left hand side (LHS) of an assignment is a local/observable variable being assigned to a new value and the right hand side (RHS) is a truth value or a Boolean local/observable variable if LHS is a Boolean variable, an enumeration value or an enumeration local/observable variable if LHS is an enumerate variable, or an arithmetic expression if LHS is a bounded integer variable. An arithmetic expression can contain local variables and observable variables of bounded integer type. An observable variable must have a prefix "Environment", such as `Environment.x`. Multiple assignments can be connected by the keyword `and`.

In an enabling condition, all observable variable must have the prefix "Environment". A proposition over actions is of the form `XXX.Action = xxx`, where `XXX` is the name of an agent and `xxx` is one of its actions.

This is a possible line of an evolution function:

```
(x = true and z = Environment.z + 1) if (y = b and TestAgent.Action = a1);
```

This is read as: "in the next step, the value of x is true and the value of z is equal to the (current) value of z for the Environment /emphif the current value of y is b and TestAgent is performing action a1"

Definition of evaluation function

An evaluation function consists of a group of atomic propositions, which are defined over global states. Each atomic proposition is associated with a Boolean formula over local variables of all agents and observable variables in the Environment. The proposition is evaluated to true in all the global states that satisfy the Boolean formula. Every variable involved in the formula has a prefix indicating the agent the variable belongs to. An example of defining an atomic proposition is shown below:

```
happy if Environment.x  = true and TestAgent.z < Environment.z;
```

where `happy` is an atomic proposition and `if` is a keyword.

Definition of initial states

Initial states are defined by a Boolean formula over variables, exactly like a Boolean formula for an atomic proposition. However, each proposition in the Boolean formula has only the following form:

```
XXX.x  = xxx
```

where `XXX` is a normal agent or the Environment, `x` is a variable of `XXX` and `xxx` is a truth value, an enumeration value or an integer, depending on the type of the variable. For simplicity, arithmetic expressions are not allowed. Below is an example:

```
Environment.x = false and Environment.y = a and
TestAgent.x = true and TestAgent.z = 1;
```

Definition of groups

Groups are used in formulae involving group modalities. A group includes one or more agents, including the Environment, such as

```
g1 = { TestAgent, Environment };
```

Definition of fairness formulae

A fairness formula is a Boolean formula over atomic propositions defined in Section 3.2.1. Besides Boolean operators `and`, `or` and !, operator $->$ is also allowed in fairness formulae and in formulae defined in Section 3.2.1. Below is an example:

```
happy and ! dead;
```

where `happy` and `dead` are atomic propositions. Notice that this section can contain a list of formulae.

Definition of formulae to be checked

A formula to be verified is defined over atomic proposition. It can have one of the following forms:
formula ::= (formula)
| formula **and** formula
| formula **or** formula
| ! formula
| formula −> formula
| **AG** formula
| **EG** formula
| **AX** formula
| **EX** formula
| **AF** formula

| **EF** formula
| **A** (formula **U** formula)
| **E** (formula **U** formula)
| **K** (`AgentName` , formula)
| **GK** (`GroupName` , formula)
| **GCK** (`GroupName` , formula)
| **O** (`AgentName` , formula)
| **KH** (`AgentName` , `AgentNameOrGroupName` , formula)
| **DKH** (`GroupName` , `AgentNameOrGroupName` , formula)
| **DK** (`GroupName` , formula)
| $<$ `GroupName` $>$ X formula
| $<$ `GroupName` $>$ F formula
| $<$ `GroupName` $>$ G formula
| $<$ `GroupName` $>$ (formula **U** formula)
| `AtomicProposition`

In the above definition, `AgentName` is the name of a normal agent or the Environment, `GroupName` is the name of a group defined in Section 3.2.1, `AgentNameOrGroupName` can be the name of a normal, the Environment, or the name of a group, `AtomicProposition` is an atomic proposition defined in Section 3.2.1.

Notes

1. All sections in the Environment can be left blank if they are not needed;
2. Section `RedStates` in any normal agent can be left blank if all local states are green;
3. Section `Groups` can be left blank if no group is used by any formula being checked;
4. Section Fairness can be left blank if fairness conditions are not required.

3.2.2 Reserved keywords

```
"--".*
"Agent"
"RedStates"
"Actions"
"Action"
"Protocol"
"Evolution"
"Evaluation"
"InitStates"
"Groups"
"Fairness"
"Formulae"
"end"
```

```
"Environment"
"Obsvars"
"Vars"
"boolean"
"true"
"false"
"Other"
"("
")"
"{"
"}"
"<"
">"
"<="
">="
"<>"
"if"
"="
"and"
"or"
"->"
"AG"
"EG"
"AX"
"EX"
"X"
"F"
"G"
"AF"
"EF"
"A"
"E"
"U"
"K"
"GK"
"GCK"
"O"
"KH"
"DKH"
"DK"
"!"
":"
","
"."
";"
".."
```

"-"
"+"
"*"
"/"

3.2.3 The grammar

/* Interpreter System */
is ::= environment agents evaluation istates groups fairformulae formulae

/* AGENT ENVIRONMENT */
environment ::= **Agent Environment** obsvardef envvardef reddef envactiondef envprotdef envevdef **end Agent**

/* Observable variables */
obsvardef ::= **Obsvars** : varidlist **end Obsvars**
| **Obsvars** : **end Obsvars**
varidlist ::= onevardef ; | varlist onevardef ;
onevardef ::= ID : **boolean**
| ID : integer .. integer
| ID : { enumlist }
enumlist ::= ID | enumlist, ID
integer ::= NUMBER | - NUMBER

/* Non-observable variables in Environment */
envvardef ::= **Vars** : varidlist **end Vars**
| **Vars** : **end Vars**

/* Definition of red states */
reddef ::= **RedStates** : lboolcond ; **end RedStates**
| **RedStates** : **end RedStates**

/* ACTIONS in Environment */
envactiondef ::= **Actions** = { actionidlist } ;
| **Actions** = { } ;
actionidlist ::= ID | actionidlist , ID

/* PROTOCOL in Environment */
envprotdef ::= **Protocol** : protdeflist **end Protocol**
| **Protocol** : protdeflist otherbranch **end Protocol**
| **Protocol** : **end Protocol**
| **Protocol** : otherbranch **end Protocol**
protdeflist ::= protline | protdeflist protline
protline ::= lboolcond : { enabledidlist } ;
enabledidlist ::= ID | enabledidlist , ID

otherbranch ::= **Other** : { enabledidlist } ;

/* Boolean conditions for protocols */
lboolcond ::= (lboolcond)
| lboolcond **and** lboolcond
| lboolcond **or** lboolcond
| ! lboolcond
| expr1 logicop expr1

/* Arithmetical expression for Environment */
expr1 ::= expr1 + term1
| expr1 - term1
| term1
term1 ::= term1 * element1
| term1 / element1
| element1
element1 ::= (expr1)
| varvalue1
logicop ::= < | <= | > | >= | = | !=

/* Variable values (not allow prefix like ID. ID) */
varvalue1 ::= boolvalue | ID | integer
boolvalue ::= **true** | **false**

/* EVOLUTION DEFINITION for Environment */
envevdef ::= **Evolution** : envevdeflist **end Evolution**
| **Evolution** : **end Evolution**
envevdeflist ::= envevline | envevdeflist envevline
envevline ::= boolresult **if** eboolcond ;
boolresult ::= (boolresult)
| boolresult **and** boolresult
| ID = expr1
/* Boolean conditions for Environment's evolution function */
eboolcond ::= (eboolcond)
| eboolcond **and** eboolcond
| eboolcond **or** eboolcond
| ! eboolcond
| expr1 logicop expr1
| **Action** = ID
| ID . **Action** = ID

/* Agents */
agents ::= agent | agents agent
agent ::= **Agent** ID vardef reddef actiondef protdef evdef **end Agent**

/* Non-observable variables */

vardef ::= **Vars** : varidlist **end Vars**

/* ACTIONS */
actiondef ::= **Actions** = { actionidlist } ;

/* PROTOCOL */
protdef ::= **Protocol** : protdeflist **end Protocol**
    | **Protocol** : protdeflist otherbranch **end Protocol**
    | **Protocol** : otherbranch **end Protocol**

/* EVOLUTION DEFINITION for normal agents*/
evdef ::= **Evolution** : evdeflist **end Evolution**
evdeflist ::= evline | evdeflist evline
evline ::= boolresult1 **if** gboolcond ;
gboolcond ::= (gboolcond)
    | gboolcond **and** gboolcond
    | gboolcond **or** gboolcond
    | ! gboolcond
    | expr2 logicop expr2
    | **Action** = ID
    | ID . **Action** = ID
    | **Environment** . **Action** = ID
boolresult1 ::= (boolresult1)
    | boolresult1 **and** boolresult1
    | ID = expr2

/* Arithmetical expression for normal agents */
expr2 ::= expr2 + term2
    | expr2 - term2
    | term2
term2 ::= term2 * element2
    | term2 / element2
    | element2
element2 ::= (expr2)
    | varvalue2

/* Variable values (add Environment.ID) */
varvalue2 ::= boolvalue | ID | **Environment** . ID | integer

/* EVALUATION */
evaluation ::= **Evaluation** evalist **end Evaluation**
evalist ::= evaline | evalist evaline
evaline ::= ID **if** evaboolcond ;
evaboolcond ::= (evaboolcond)
    | evaboolcond **and** evaboolcond
    | evaboolcond **or** evaboolcond

| ! evaboolcond
| expr3 logicop expr3

/* Arithmetical expression for evaluation function */
expr3 ::= expr3 + term3
| expr3 - term3
| term3
term3 ::= term3 * element3
| term3 / element3
| element3
element3 ::= (expr3)
| varvalue3

/* Variable values for evaluation function */
varvalue3 ::= boolvalue | ID | ID . ID | **Environment** . ID | integer

/* INITIAL STATES */
istates ::= **InitStates** isboolcond ; **end InitStates**
isboolcond ::= (isboolcond)
| isboolcond **and** isboolcond
| isboolcond **or** isboolcond
| ! isboolcond
| ID . ID = varvalue1
| **Environment** . ID = varvalue1

/* Groups */
groups ::= **Groups** groupdeflist **end Groups**
| **Groups end Groups**
groupdeflist ::= groupline | groupdeflist groupline
groupline ::= ID = { namelist } ;
namelist ::= agentname | namelist , agentname
agentname ::= **Environment** | ID

/* FAIRNESS FORMULAE */
fairformulae ::= **Fairness** fformlist **end Fairness**
| **Fairness end Fairness**
fformlist ::= fformula ; | fformlist fformula ;
fformula ::= (fformula)
| fformula **and** fformula
| fformula **or** fformula
| ! fformula
| fformula −> fformula
| ID

/* FORMULAE TO CHECK */
formulae ::= **Formulae** formlist **end Formulae**

formlist ::= formula ; | formlist formula ;
formula ::= (formula)
| formula **and** formula
| formula **or** formula
| ! formula
| formula −> formula
| **AG** formula
| **EG** formula
| **AX** formula
| **EX** formula
| **AF** formula
| **EF** formula
| **A** (formula **U** formula)
| **E** (formula **U** formula)
| **K** (ID , formula)
| **K** (**Environment** , formula)
| **GK** (ID , formula)
| **GCK** (ID , formula)
| **O** (ID , formula)
| **O** (**Environment** , formula)
| **KH** (ID , ID , formula)
| **KH** (**Environment** , ID , formula)
| **KH** (ID , **Environment** , formula)
| **KH** (**Environment** , **Environment** , formula)
| **DKH** (ID , ID , formula)
| **DKH** (ID , **Environment** , formula)
| **DK** (ID , formula)
| < ID > X formula
| < ID > F formula
| < ID > G formula
| < ID > (formula **U** formula)
| ID

3.3 The graphical interface

The graphical interface is installed by copying the file
`org.mcmas.ui_1.0.0.jar`
into the `plugin/` directory under your Eclipse installation. The version available online has been tested with Eclipse 3.3 and 3.4, with Java 1.5 and 1.6, and with Linux, Mac, and Windows operating systems. If you have problems with the plugin, please contact us.

Initial configuration: once the plugin is installed, it needs to be configured by specifying the directory locations of MCMAS, DOT (`http://www.graphviz.org`) and (optional, only if you are using MCMAS under windows) of Cygwin. This is done by accessing the general “Preferences” of Eclipse, as in Figure 3.1.

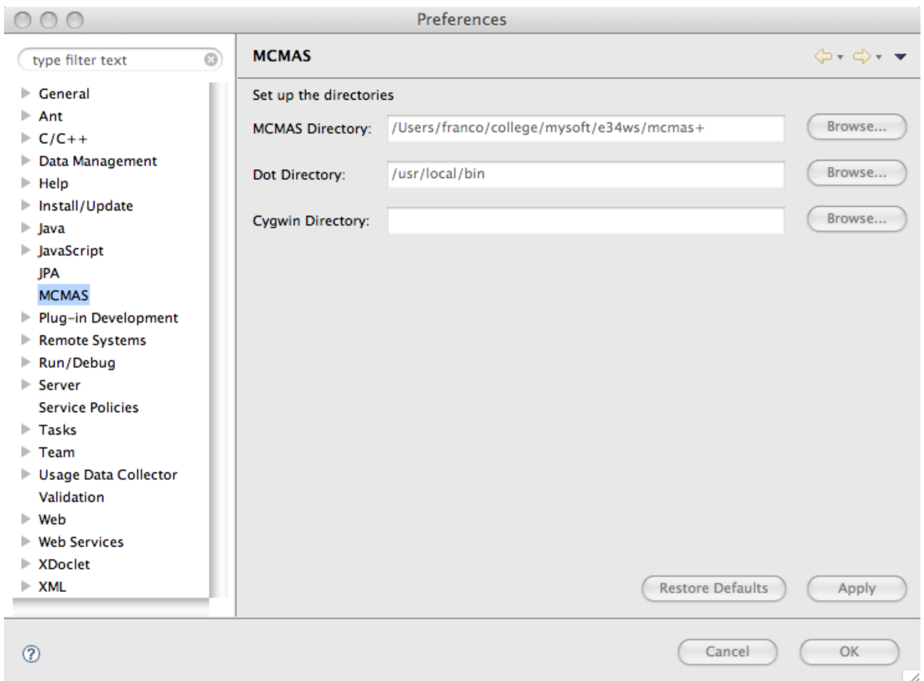

Figure 3.1: The MCMAS preference tab.

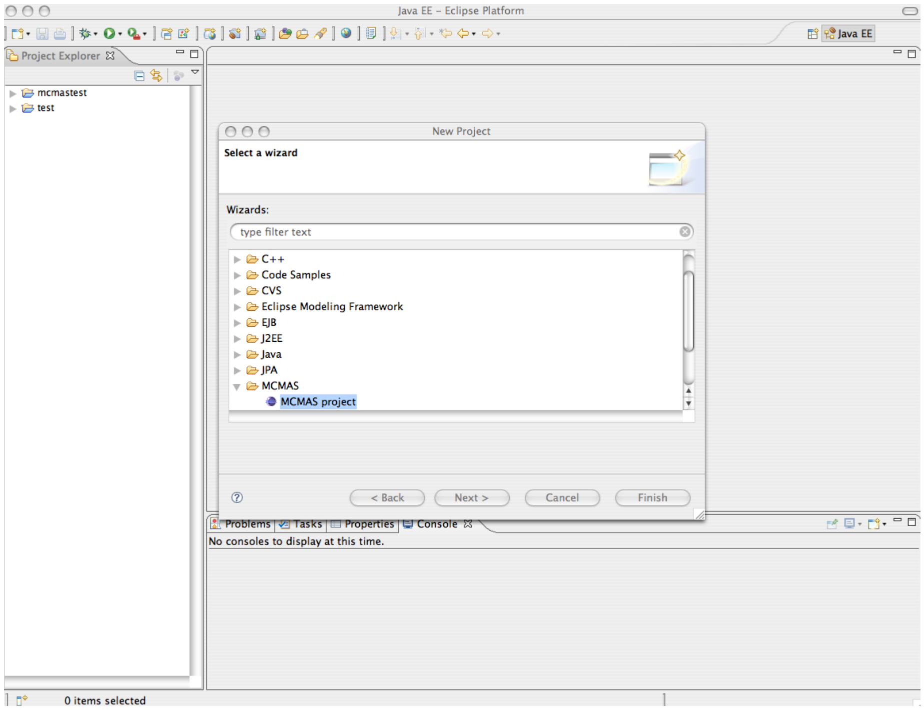

Figure 3.2: The MCMAS project wizard.

Once the plugin is installed correctly, it is possible to create a new MCMAS project using the wizard, by selecting File -¿ New -¿ Other, and then MCMAS project (see Figure 3.2).

The new project creates an empty file with the initial structure of an ISPL file. The file can be renamed and it should be completed with all the necessary information required by the grammar. Syntax errors are underlined and contextual help is provided to fix them (see Figure 3.3).

Verifications, simulations, and counter-examples analysis can be performed from this graphical interface:

Running a simulation. To run a simulation, select the desired method from the drop-down MCMAS menu (see Figure 3.4) (symbolic interactive mode is more appropriate for large examples). Click the appropriate tab under the editor window to access the correct pane and simply follow the instructions displayed to move forward and backward in a simulation.

Performing verification. Verification of the fomulae in an ISPL file is performed by selecting “Launch verification” from the MCMAS drop down menu. The results of the verification are available by clicking on the “Model Checking” tab at the bottom of the editor window (see Figure 3.5).

Analysing counter-examples. It is possible to analyse witnesses and counter-examples by clicking on “Show counterexample/witness” from the ver-

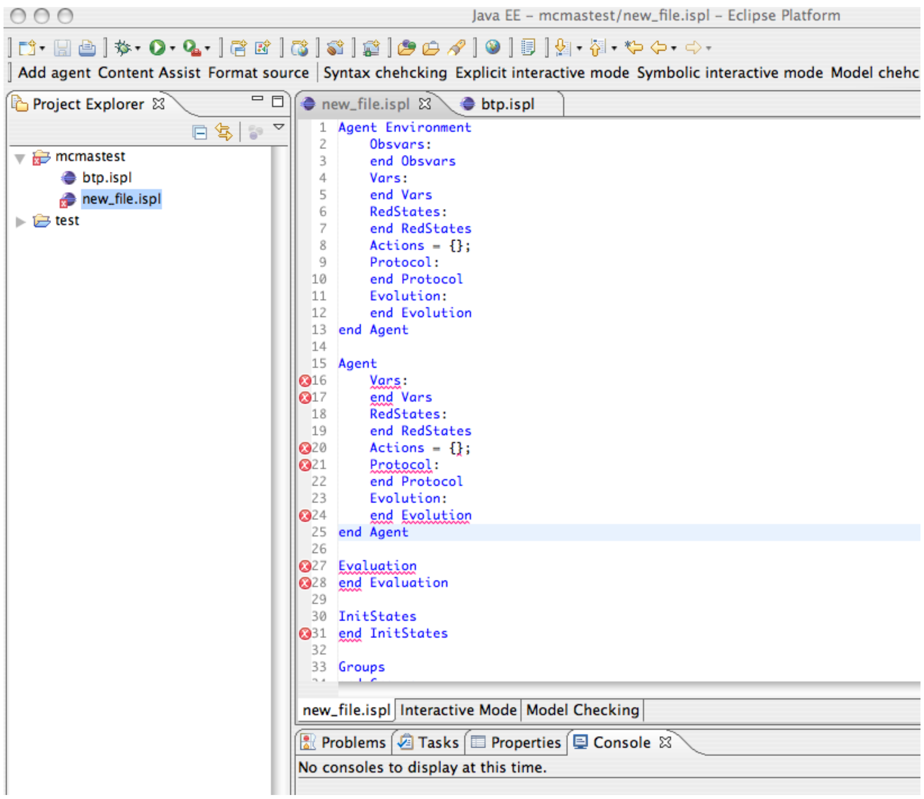

Figure 3.3: The ISPL editor for an empty file.

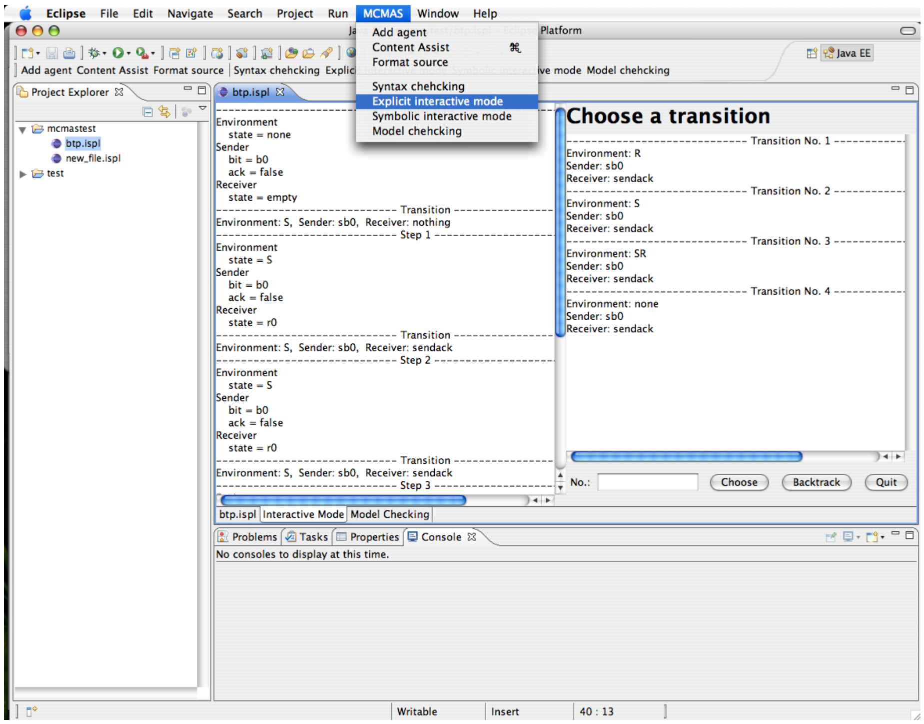

Figure 3.4: Running a simulation.

ification window. A description of the states is available on the right-hand side of the window. Temporal transitions are represented by a black arrow labelled with the joint action performed. Epistemic transitions are represented by a red arrow labelled with the appropriate name of the agent (or group of agents) for the relation.

3.4 Theoretical background: the semantics of interpreted systems

This section is extracted from [5] and only slightly modified to introduce the notion of "public" (or "observable") local states for the environment.

The formalism of *interpreted systems* was introduced in [4] to model a system of agents and to reason about the agents' epistemic and temporal properties. In this formalism, each agent is modelled using a set of *local states*, a set of *actions*, a *protocol*, and an *evolution function*.

- The set of local states for an agent i is denoted by the symbol L_i. Elements of L_i capture the "private" information of an agent and, at any given time, local states represent the state in which an agent is (e.g. `ready` and `busy`

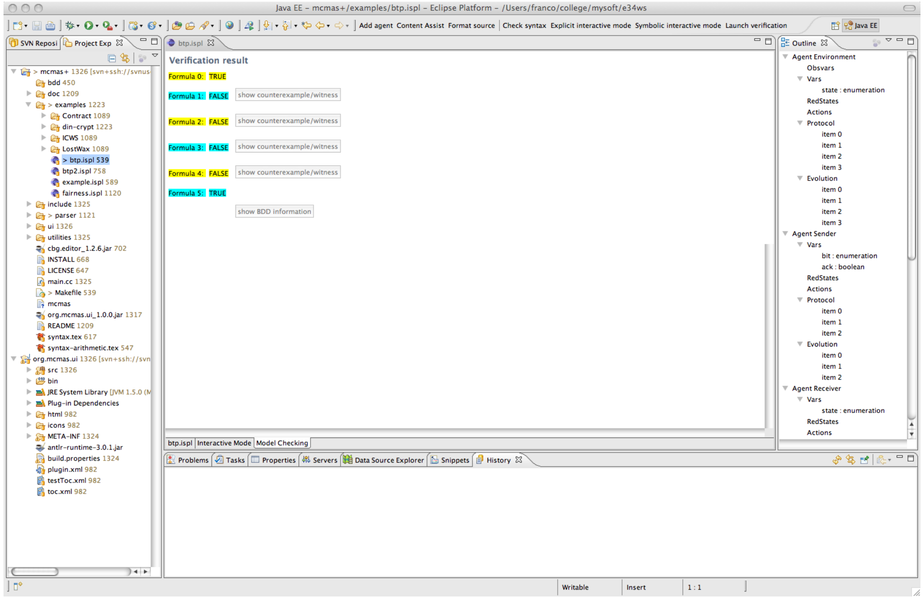

Figure 3.5: Verification results.

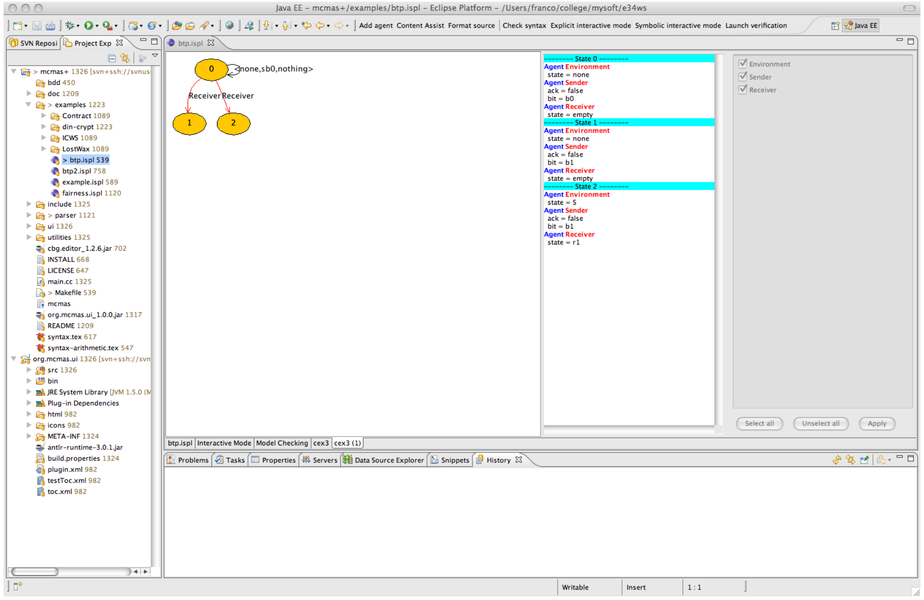

Figure 3.6: A counter-example.

may be elements of L_i). Contrary to [4], it is assumed that the set L_i is finite (this is required by the model checking algorithms).

- The set of actions for an agent i is denoted by the symbol Act_i. Elements of Act_i represent the possible actions that an agent is allowed to perform. Differently from local states, actions are "public". Similarly to local states, here the set Act_i is assumed to be finite.

- The protocol for an agent i is denoted by the symbol P_i. The protocol is a "rule" establishing which actions may be performed in each local state. The protocol P_i is modelled by a function $P_i : L_i \rightarrow 2^{Act_i}$, assigning a set of actions to a local state. Intuitively, this set corresponds to the actions that are enabled in a given local state. Notice that this definition may enable more than one action to be performed for a given local state. When more than one action is enabled, it is assumed that an agent selects *non-deterministically* which action to perform.

- The evolution function for agent i is denoted by the symbol t_i (notice: [4] define a single evolution function t for all the agents, see discussion below). The evolution function determines how local states "evolve", based on the agent's local state, on other agents' actions, and on the public local state of a special agent used to model the environment (see below). The evolution function is modelled by a function $t_i : L_i \times L_{E_P} \times Act_1 \times \ldots \times Act_n \times Act_E \rightarrow L_i$, where n is the number of agents in the system.

A special agent E is used to model the environment in which the agents operate. Similarly to the other agents, E is modelled using a set of local states L_E, a set of actions Act_E, a protocol P_E, and an evolution function t_E. As mentioned above, part of the local states of E are "public", i.e. $L_E = L_{E_P} \times L_{E_R}$, where L_{E_P} denotes the set of "public" local states of E, and L_{E_R} denotes the set of "private" local states of E: all the remaining agents may "peek" at L_{E_P} to determine their temporal evolution, and the epistemic accessibility relations of the agents contain the public information of E as well (see below).

For all agents including the environment, the sets L_i and Act_i are assumed to be non-empty, and the number $n \in I\!N$ of agents is assumed to be finite. For convenience, the symbol Act denotes the Cartesian product of the agents' actions, i.e., $Act = Act_1 \times \ldots \times Act_n \times Act_E$. An element $\alpha \in Act$ is a tuple of actions (one for each agent) and is referred to as a *joint action*. The Cartesian product of the agents' local states is denoted by S, i.e., $S = L_i \times \ldots \times L_n \times L_E$. An element $g \in S$ is called a *global state*; given a global state g, the symbol $l_i(g)$ denotes the local state of agent i in the global state g; we write $l_{E_P}(g)$ to denote the "public" component of $l_E(g)$. It is assumed that, in every state, agents evolve *simultaneously* (notice that this requirement is similar to the definition of Moore synchronous game structures: see [5]).

The definition of a single evolution function $t : S \times Act \rightarrow S$ presented in [4] differs slightly from the definition of $n+1$ evolution functions presented here. The two definitions are, in fact, equivalent: $t(g, a) = g'$ iff, for all $i \in$

$\{1,\ldots,n\}, t_i(l_i(g),a) = l_i(g')$ and $t_E(l_E(g),a) = l_E(g')$ (the decomposition from a single t to $n+1$ "local" transition functions is guaranteed to be possible by the assumptions on t). This choice is motivated by the fact that the definition of an evolution function for each agent helps to keep the description of the system compact.

Given a set of *initial global states* $I \subseteq S$, the protocols and the evolution functions generate a set of *reachable global states* $G \subseteq S$, obtained by all the possible runs of the system. A set of atomic propositions P and an *evaluation relation* $V \subseteq P \times S$ are introduced to complete the description of an interpreted system. Formally, given a set of n agents $\{1,\ldots,n\}$, an interpreted system is a tuple:

$$IS = \left\langle (L_i, Act_i, P_i, t_i)_{i\in\{1,\ldots,n\}}, (L_E, Act_E, P_E, t_E), I, V \right\rangle.$$

It has been shown in [4] that interpreted systems can provide a semantics to reason about time and epistemic properties, by means the following language:

$$\phi \quad ::= \quad p \mid \neg\phi \mid \phi \vee \phi \mid EX\phi \mid EG\phi \mid E[\phi U \psi] \mid \; K_i\phi \mid E_\Gamma\phi \mid C_\Gamma\phi \mid D_\Gamma\phi.$$

In this grammar, $p \in P$ is an atomic proposition, and the operators EX, EG, and EU are the standard **CTL** operators [3]; the remaining **CTL** operators EF, AX, AG, AU, AF can be derived in a standard way. The formula $K_i\phi$ ($i \in \{1,\ldots,n\}$) is read as "agent i knows ϕ". The symbol Γ denotes a group of agents. The formula $E_\Gamma\phi$ is read as "everybody in group Γ knows ϕ"; the formula $C_\Gamma\phi$ is read as "ϕ is *common knowledge* in group Γ" (intuitively, common knowledge of ϕ in a group of agents denotes the fact that everyone knows ϕ, and everyone knows that everybody else knows ϕ); the formula $D_\Gamma\phi$ is read as "ϕ is *distributed* knowledge in group Γ" (intuitively, distributed knowledge in a group of agents is the knowledge obtained by "sharing" all agents' knowledge).

Given an interpreted system IS, it is possible to associate a Kripke model [4] $M_{IS} = (W, R_t, \sim_1, \ldots, \sim_n, V)$ to IS; the model M_{IS} can be used to interpret formulae of the grammar above. The model M_{IS} is obtained as follows:

- The set of possible worlds W is the set G of reachable global states (this is to avoid the epistemic accessibility of states which cannot reached using the temporal relation).

- The temporal relation $R_t \subseteq W \times W$ relating two worlds (i.e., two global states) is defined by the temporal transition t_i. Two worlds w and w' are such that wR_tw' iff there exists a joint action $a \in Act$ such that $t(g,a) = g'$, where t is the transition relation of IS obtained by the composition of the functions t_i, $i \in \{1,\ldots,n\}$ and t_E.

- The epistemic accessibility relations $\sim_i \subseteq W \times W$ are defined by imposing the equality of the local components (for i and for the "public" part of E) of the global states. Formally, two worlds $w, w' \in W$ are such that $w \sim_i w'$ iff $l_i(w) = l_i(w')$ and $l_{E_P}(w) = l_{E_P}(w')$ (i.e., two worlds w and w'

are related via the epistemic relation $\sim_i$ when the local states of agent i in global states w and w' are the same [4], and the "public" or "observable" part of the Environment local states are the same).

- The evaluation relation $V \subseteq AP \times W$ is the evaluation relation of IS.

Similarly to [3], let $\pi = (w_0, w_1, \ldots)$ be an infinite sequence of worlds such that, for all i, $w_i R_t w_{i+1}$, and let $\pi(i)$ denote the i-th world in the sequence (the temporal relation is assumed to be serial and thus all computation paths are infinite). Let $R_\Gamma^E \subseteq W \times W$ denote the relation obtained by taking the union of the epistemic relations for the agents in Γ, i.e., $R_\Gamma^E = \bigcup_{i\in\Gamma} \sim_i$. Let R_Γ^D denote the intersection of the epistemic relations for the agents in Γ, i.e., $R_\Gamma^D = \bigcap_{i\in\Gamma} \sim_i$. Let R_Γ^C denote the transitive closure of R_Γ^E. It is written $M_{IS}, w \models \phi$ when a formula ϕ is true at a world w in the Kripke model M_{IS}, associated with an interpreted system IS. Satisfaction is defined inductively as follows:

$M_{IS}, w \models p$ iff $(p, w) \in V$,
$M_{IS}, w \models \neg\phi$ iff $M_{IS}, w \not\models \phi$,
$M_{IS}, w \models \phi_1 \vee \phi_2$ iff $M_{IS}, w \models \phi_1$ or $M_{IS}, w \models \phi_2$,
$M_{IS}, w \models EX\phi$ iff there exists a path π such that $\pi(0) = w$, and $M_{IS}, \pi(1) \models \phi$,
$M_{IS}, w \models EG\phi$ iff there exists a path π such that $\pi(0) = w$, and $M_{IS}, \pi(i) \models \phi$ for all $i \geq 0$,
$M_{IS}, w \models E[\phi U \psi]$ iff there exists a path π such that $\pi(0) = w$, and there exists $k \geq 0$ such that $M_{IS}, \pi(k) \models \psi$, and $M_{IS}, \pi(j) \models \phi$ for all $0 \leq j < k$,
$M_{IS}, w \models K_i\phi$ iff for all $w' \in W$, $w \sim_i w'$ implies $M_{IS}, w' \models \phi$,
$M_{IS}, w \models E_\Gamma\phi$ iff for all $w' \in W$, $wR_\Gamma^E w'$ implies $M_{IS}, w' \models \phi$,
$M_{IS}, w \models C_\Gamma\phi$ iff for all $w' \in W$, $wR_\Gamma^C w'$ implies $M_{IS}, w' \models \phi$,
$M_{IS}, w \models D_\Gamma\phi$ iff for all $w' \in W$, $wR_\Gamma^D w'$ implies $M_{IS}, w' \models \phi$.

Similarly to standard Kripke models, a formula ϕ is *true in a model*, written $M_{IS} \models \phi$, if $M_{IS}, w \models \phi$ for all $w \in W$.

A formula ϕ *is true in an interpreted system IS*, denoted by $IS \models \phi$, iff it is true in the associated Kripke model ([4], p. 111).

Bibliography

[1] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, 35(8):677–691, 1986.

[2] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[4] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, 1995.

[5] F. Raimondi. *Model Checking Multi-Agent Systems*. PhD thesis, 2006.