MCK 1.1.0
User Manual

August 12, 2014

Credits

Development of MCK has been funded by several research grants (all with Chief Investigator, Ron van der Meyden.) Initial development of the system was funded by an ARC Discovery grant (2001-2003), and current development is supported by Australian Research Council Linkage grant LP0882961 and Defence Research and Development Canada (Valcartier) contract W7701-082453/001/QCV.

A first version of the MCK system was released in 2003, with development done by Peter Gammie. This version included symbolic model checking for observational, clock and synchronous perfect recall semantics for knowledge, based on a binding to Long's BDD library using the HBDD library which provides a generic interface allowing bindings to other BDD packages. Kai Baukus did initial development of the asynchronous perfect recall functionality. Jeremy Lee developed the binding of HBDD to CUDD and BuDDy, and also contributed initial versions of the explicit state model checker and the game debugging interface. These were revised by Ron van der Meyden, and completed by Cheng Luo and Xiaowei Huang, who also added bounded model checking, probabilistic knowledge and the graphical user interface. Nicholas Grasevski made improvements to the graphical user interface.

Contents

Chapter 1

Introduction

This manual explains how to use MCK, a prototype model checker for temporal epistemic specifications. It assumes some familiarity with the idea of model checking [9, 10, 18], and temporal and epistemic logics [12], but is otherwise self-contained.

This chapter introduces the general scenario to which the MCK system may be applied (Section 1.1) and describes an abstract model that underlies the system (Section 1.2). The semantics of the constructs of the MCK systems is most easily understood with respect to this model. The MCK system itself uses a more concrete syntax designed to facilitate the encoding of examples. The remainder of the manual describes this concrete syntax and its associated semantics. Chapter 2 discusses the language used to model a scenario in MCK. The language used to describe the specifications that MCK checks in these scenarios is discussed in Chapter 3. MCK implements a variety of model checking algorithms, which are described in Chapter 4. In addition to epistemic logic specifications, the system now also supports probabilistic knowledge; this capability is described in Chapter 5. Chapter 6 presents a number of examples that can be analysed using the system. The system supports debugging of models and specifications by means of a counter-example game and visualization capability, that is described in Chapter 7. The description of syntax and semantics in these chapters is semi-formal; a formal abstract syntax for the inputs to MCK is presented in Appendix A, and Appendix B provides a formal operational semantics for MCK programs.

1.1 The Model Checking Scenario

The overall scenario that can be analysed using the system has the following general structure. We consider that we are modelling a situation where some number of *agents* (which might be players in a game, actors in an economic setting, or processes, programs or components in a computational setting) interact in the context of an *environment*. A *state* of the system consists of a state of the environment together with a *local state* for each of the agents. The agents have the capacity to perform certain *actions* in this environment. The effect of actions is to change the state of the system. Each of the agents performs these actions acccording to a *protocol*, or set of rules, which describes the allowable choices of the next action at each point of time. The agents have *incomplete information* about the state of the system: their possible information is limited by the fact that they are able to *observe* only part of the state at each instant of time.

The MCK system can be applied to the analysis of this type of setting by the use of *model checking* techniques. The input to the MCK system consists of a file, or set of files, that describe:

1. the environment in which the agents operate, including:
    - the possible states of the environment,
    - the initial states of the environment,
    - the names of agents operating in this environment,
    - how the agents' actions change the state of the environment,

- (optionally) a set of *fairness* conditions, which constrain the infinitary behaviour of the system (ensuring, e.g. that some agent is not kept waiting forever for a requested event to occur);

2. the protocol by which each of the named agents chooses their sequence of actions, including:
   - the structure of local states maintained by the agent to make such choices and record other useful information,
   - the possible initial values of the agent's local states, and
   - a description of what parts of the state are observable to the agent;
3. a number of *specification formulas*, expressing some property of the way that the agent's knowledge evolves over time.

Both the possible state changes described in the environment and the agents' choices of action may be non-deterministic, which means that the system may evolve over time in a potentially large number of different ways. The output produced by the MCK system is, for each of the specification formulas, an answer to the question of whether, for the scenario modelled, the agents' knowledge is in fact guaranteed to evolve according to the specification, for every possible evolution of the system.

The MCK system currently allows several different approaches to the description of the temporal and epistemic aspects of the specification formulas. In the epistemic dimension, agents may use their observations in a variety of ways to determine what they know. One way (the *observational* interpretation of knowledge) is to make inferences about the state based just on their latest observation. Another way (the *clock* interpretation of knowledge), permitting more information to be extracted, is to compute knowledge using both the current observation and the current clock value. Finally, even more information can be extracted by the agent if it uses a complete record of all its observations to date to determine what it knows (this is called the *perfect recall* interpretation of knowledge, and comes in synchronous and asynchronous variants). In the temporal dimension, the specification formulas may describe the evolution of the system using the temporal logic CTL*, which combines LTL and CTL and can express behaviour along a single computation as well as the branching structure of all possible computations. The system currently supports different combinations of all these parameters to different degrees: in some cases this is because the implementation remains to be undertaken, in others because there are inherent computational reasons why the problem is difficult or impossible to implement.

Figure 1.1 presents an example of the input file to the system, modelling a scenario in which there is a single agent in the environment, a robot called `Robot` (running the protocol `"robot"`) operating in an environment consisting of 8 possible positions, and sensing the position using a noisy sensor, whose values are recorded in the variable `sensor`, which is observable to the agent. The example contains a single specification formula, indicated by the construct **spec_obs**, which indicates that the knowledge operator `Knows` is to be interpreted using the observational interpretation for knowledge. A more elaborate version of this example is discussed at greater length in section 6.1.

In addition to determining whether a specification formula is true or false in a given scenario, the system provides additional forms of support for the analysis of such scenarios, such as permitting the user to check the model by navigating through executions of the scenario, and presenting counter examples when specification formulas are found to be false.

1.2 Background Theory

This section describes an abstract mathematical model that underlies the MCK system. It is closely related to models used in works including [21, 22, 23], which present some of the algorithms and data structures that underlie the analysis performed by the MCK system. (These papers themselves use a variety of formal modellings, and the differences between these papers and the model described here is largely a matter of mathematical presentation.) The remainder of the manual provides a more concrete syntax and semantics for the model developed here.

We first present an abstract view of the semantics of MCK programs that is adequate from the point of view of the main constructs of the specification language used in MCK. From this perspective, a system comprised of a set of interacting agents is, at each point of time, in some *global state*. Write $\mathcal{G}$ for the set of all possible global states of the

system. A *run* is a possible history of such states, and can be modelled by an infinite sequence $r = s_0, s_1, s_2, \ldots$ where each $s_k \in \mathcal{G}$. We write $r(m)$ for the m-th state in this sequence.

The behaviour of a system is typically non-deterministic, as agents may have a choice of what actions to perform at any given point of time, and the environment may also model non-deterministic events such as communication links failures and delays. We may model this non-determinacy by representing the system as a set $\mathcal{R}$ of runs, intuitively all the possible ways that the history of the system may evolve.

In general, agents are not able to observe the entire state of the system. We model this by means of a function $O_i : \mathcal{G} \to O_i$, for each agent i where O_i is the set of observations made by agent i. Intuitively, $O_i(s)$ is the information that is visible to agent i when the system is in the global state s. Based on their observations, agents are able to make inferences about the situation in which they find themselves, i.e., the particular state, time and past and future history. We model such a situation as a *point*, represented as a pair (r, m), where r is a run and $m \in \mathbf{N}$ is a time.

In order to determine what they know, agents may make use of their observations in a variety of ways. We capture the specific way that the agents use their observations for purposes of computing knowledge by assigning them a *local state* with respect to a *view* at each point of the system. We write $r_i^x(m)$ for the local state of agent i at the point (r, m), where x is the view. The simplest view is the observational view `obs`, where the agent uses just its current observation to determine what it knows. The local state in this case is defined by $r_i^{\mathtt{obs}}(m) = O_i(r(m))$. Somewhat more informative to the agent is the *clock view* `clock`, defined by $r_i^{\mathtt{clock}}(m) = (m, O_i(r(m)))$. Here the agent uses both its current observation and the current global clock value to determine what it knows. Most informative is the *synchronous perfect recall view*, defined by $r_i^{\mathtt{spr}}(m) = \langle O_i(r(0)), \ldots, O_i(r(m))\rangle$. Here the agent uses its complete sequence of observations to the current time to determine what it knows. A variant of this, is the *asynchronous perfect recall view*, $r_i^{\mathtt{apr}}(m)$ defined from $r_i^{\mathtt{spr}}(m)$ by replacing every maximal subsequence of consecutive identical observations $O_i(r(k)) = \ldots = O_i(r(m))$ by a single copy of this observation. Intuitively, this captures that while the agent is able to remember all its observations, it does not have access to a global clock, so does not necessarily know for what length of time it was making each observation.

For each view x, we may define a relation of *indistinguishability* on points: two points (r, m) and (r', m') are said to be indistinghuishable to agent i, written $(r, m) \sim_i^x (r', m')$ if the agent has the same local state with respect to the view at those two points, i.e. if $r_i^x(m) = (r')_i^x(m')$. Intuitively, the set of all points that are indistinguishable from a point (r, m) is the set of all points that the agent considers to be possibly the current point, when using the information capture in that view. We may therefore say that agent i *knows* (with respect to view x) that a formula ϕ holds at a point (r, m) if ϕ holds at all points (r', m') such that $(r', m') \sim_i^x (r, m)$. In MCK, the statement that agent i knows ϕ is written as `Knows` i ϕ, with the view indicated at the level of the larger formula of which this statement is a part.

The above definitions suffice to give semantics to the specification language of MCK. (In addition to the knowledge operator just defined, there is a variety of temporal logic operators. These are defined in Chapter 3.) In order to enable the user to describe the system in which a formula is to be checked, MCK provides a systems modelling language that consists of two parts. The set of runs of a system is taken to be generated by the agents each running a *protocol* by which they choose actions available to them in a given *environment*. The global states $\mathcal{G}$ are made up of two components: a state of the environment and a protocol state for each of the agents.

Abstractly, we model the environment as a finite-state transition system, with the transitions labelled by the agents' actions. For each agent $i = 1 \ldots n$ let A_i be a set of *actions* associated with agent i. MCK currently provides a number of concrete action types, including concurrent read and write to shared variables (which are modelled as part of the environment state), as well as a construct that concurrently sends a signal to the environment and updates local variables as a function of local and environment variables.

A *joint action* consist of an action for each agent, i.e., the set of joint actions is the cartesian product $A = A_1 \times \ldots \times A_n$. Define a *finite environment for n agents* to be a tuple $\mathcal{E}$ of the form $\langle S_e, I_e, \tau_e\rangle$ where the components are as follows:

1. S_e is a finite set of *states of the environment*. (Concretely, these are given in MCK by specifying a set of typed variables.)

2. I_e is a subset of S_e, representing the possible *initial states* of the environment. (Concretely, this is specified in MCK by a constraint on the environment variables, or a program that constructs the initial states .)

3. $\tau_e : S_e \times A \to \mathcal{P}(S_e)$ is a function mapping each state of the environment s_e and joint action $\mathbf{a}$ to the set $\tau_e(s_e, \mathbf{a})$

of possibilities for the next state of the environment when **a** is performed in state s_e. (Concretely this is given in MCK by writing a nondeterministic program that computes this state transition function.)

The behaviour of agents is given concretely in MCK by writing a program that describes their choice of action at each point of time. Each such program defines a set of states S_i, given concretely by means of a set of variables, which includes the *program counter*, a special variable whose value is the position in the program at which control resides at the given point in time. Together, the states of the environment and these protocol states determine the set of global states $\mathcal{G}$: these consist of tuples $\langle s_e, s_1, \ldots, s_n \rangle$ comprised of a state s_e in S_e and a state s_i in S_i for each agent i.

Abstractly, agent i's program defines not just the protocol states S_i, but in fact a tuple $\langle S_i, I_i, P_i, \mu_i \rangle$, where

1. I_i is the set of possible initial states of the protocol.
2. $P_i : \mathcal{G} \to \mathcal{P}(A_i)$ is a function mapping global states to a set of possible actions for the agent.
3. $\tau_i : \mathcal{G} \times A \to \mathcal{P}(S_i)$ is a function that describes how the protocol state (including the program counter) is updated when a joint action occurs.[1]

Using these components, we may now define a *global initial state* to be a global state $\langle s_e, s_1, \ldots, s_n \rangle$ such that $s_e \in I_e$ and $s_i \in I_i$ for each agent i. Moreover, we define a *global state transition relation* T on global states as follows. If there are n agents, sTs', where $s = \langle s_e, s_1, \ldots, s_n \rangle$ and $s' = \langle s'_e, s'_1, \ldots, s'_n \rangle$ if there is a joint action $\mathbf{a} = \langle a_1, \ldots, a_n \rangle$ such that for each agent i, $a_i \in P_i(s)$ is an action that may be selected by agent i's protocol, the state $s'_e \in \tau_e(s_e, \mathbf{a})$ is one of the possible outcomes of performing the joint action, and for each agent i, the state $s'_i \in \tau_i(s, \mathbf{a})$ is one of the possible results of updating the protocol state accordingly.

One final component in the abstract systems model for MCK is a (generalized Büchi) *fairness* condition, represented abstractly by a tuple $\langle \alpha_0, ..., \alpha_n \rangle$, where each α_i is a subset of the set of global states $\mathcal{G}$. Concretely, each α_i is given in MCK by a formula satisfied by the states in α_i. Intuitively, in each run, for each i, there is a state $s_i \in \alpha_i$ that occurs infinitely often.

We may now define the set of runs generated when the agents execute their protocols in the given enviroment as the set of all runs $r = s_0, s_1, \ldots$, where the $s_k \in \mathcal{G}$ are global states, such that s_0 is a global initial state, and for for each $k \geq 0$, we have $s_k T s_{k+1}$, i.e, there is a transition from s_k to s_{k+1}, and the fairness conditions $\langle \alpha_0, ..., \alpha_n \rangle$ are satisfied, i.e., for each $0 \leq i \leq n$, there exists a state $s \in \alpha_i$ such that $s_m = s$ for infinitely many m.

1.3 Installation and Invocation

The program is implemented in Haskell and makes use of extensions only implemented by the Glasgow Haskell Compiler (`http://haskell.org/ghc/`). The graphical user interface is built using the Qt framework (`http://qt-project.org/`). In previous versions (version 0.2.0 and earlier), it used David Long's Binary Decision Diagram (BDD) package (`http://www-2.cs.cmu.edu/~modelcheck/bdd.html`). From version 0.3.0 onwards, the program can also use BuDDy (`http://sourceforge.net/projects/buddy/`), and CUDD (`http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`) and is typically distributed with CUDD. Each of the three BDD packages (Long's BDD package, BuDDy and CUDD) has a Haskell binding that is included in the HBDD library included with the distribution. The bounded model checking functionality uses Picosat (`http://fmv.jku.at/picosat/`).

See the file `README` for detailed installation instructions and the distribution license (available using the call `mck -l`) for information concerning license conditions arising from this associated software.

Invocation

The `mck` program currently implements three approaches to model checking, each of which provides a number of model checking algorithms tailored for specific classes of formulas and knowledge semantics (see Chapter 4). The

[1] We remark that these definitions of P_i and τ_i are somewhat more general than MCK currently supports. For example, we do not presently have a construct in MCK that allows an agent's choice of protocol state update to depend directly on protocol states of another agent. Similarly, an agent's choice of action can presently depend only on its current protocol state plus its current observation of the state of the environment. However, we may include more liberal constructs (e.g., handshakes between agents), in later versions.

default approach is by symbolic model checking using binary decision diagrams. Bounded model checking is also supported. The final approach is by explicit state model checking. Explicit state model checking is only likely to terminate on small models and is included primarily to permit visualization and counter-example game playing on such models. These features are also supported for the counter-examples produced by bounded model checking.

The program accepts the following flags, which are displayed when the program is run without input:

- `-b[Int]` or `-bmc[=Int]`: Use bounded model checking, with bound `Int`.
- `-c` or `--counter-examples`: generate counter-example traces
- `-d[Int]` or `--debug[=Int]`: Output BDD debugging info and stats (symbolic model checking only). This is not particularly useful for end users.
- `-e` or `--environment`: Output the environment information.
- `-f` or `--formula`: Output the BDD formulas in human-readable form (symbolic model checking only). This is not particularly useful for end users.
- `-g` or `--game`: Play game on failed specifications (explicit state and bounded model checking only).
- `-k` or `--es-check`: Use explicit state model checking instead of the default symbolic model checking.
- `-l`: print out license terms
- `-m` or `--es-model`: Output explicit states model information (explicit state model checking only).
- `-o File` or `--var-order=File`: Output the BDD variable order to a file (symbolic model checking only).
- `-p` or `--protocol`: Output the pre-processed protocol information.
- `-r[s|w]` or `--reorder[=s|w]`: Enable BDD variable re-ordering (symbolic model checking only). The two methods are sifting and window-based sifting (see the corresponding BDD package's manpage for details).

  Note that, for all but the simplest examples, some kind of variable ordering information must be supplied. If this is unknown, using `-rs` usually greatly reduces memory consumption, although it is somewhat time-inefficient (this may also depend on the BDD package used).
- `-s File` or `--set-var-order=File`: Load BDD variable ordering from a file (symbolic model checking only).
- `-u`: this is for use by the mck GUI, in order to obtain results in XML format.
- `-v` or `--visualize`: Invoke the visualizer. (see section 7.1).
- `-x` or `--es-context`: Output exlicit states model context (explicit states model checking only). This is not particularly useful for end users.

Not all combinations of these flags are supported, and a flag will be ignored if it is given in an incompatible context. For example, the `-g` flag will not take effect when using symbolic model checking. Similarly, the `-f` flag will be ignored when using explicit state model checking.

The program expects the filename of an input script to be supplied. It will search the current directory for files matching the names of any protocols that are used in an input script but not defined there. See Section 6.2 for an example of this.

Graphical User Interface

In addition to the command line invocation, a graphical user interface is available for the system. This is described in detail in Chapter 8.

```
-- There are 8 positions in the world.
-- If the robot is really at position p, then the sensor will have a
-- value ∈ {p − 1, p, p + 1}, for a truncating interpretation of arithmetic.
type Pos = {0..7}

position : Pos
sensor : Pos
halted : Bool

init_cond = position == 0 /\ sensor == 0 /\ neg halted

agent Robot "robot" ( sensor )

-- At each time step, the environment moves the robot one step to the
-- right, and generates a new sensor reading.
transitions
begin
  if Robot.Halt -> halted := True
  fi;
  if neg halted -> position := position + 1
  fi;
  if True -> sensor := position - 1
  [] True -> sensor := position
  [] True -> sensor := position + 1
  fi
end

spec_obs = G (sensor >= 3 <=> Knows Robot position in {2..4})

-- The "car handbrake" protocol.
-- In order to stop moving, the robot only needs to yank it once.
protocol "robot" (sensor : observable Pos)

begin
  do neg (sensor >= 3) -> skip
  [] break             -> <<Halt>>
  od
end
```

Figure 1.1: A simplified version of the robot example.

Chapter 2

The Input Language

This chapter provides an informal description of the language used to describe the model checking scenario (Section 1.1). A more formal description is given in Appendix A (syntax) and Appendix B (semantics).

An input script consists of a collection of environment declarations, one or more specifications, and one or more protocols, as illustrated in Figure 2.1. Both environments and agent protocols are described using a simple programming language. We proceed by describing the lexical conventions, the role of types, the structure of programs, then cover the variety of environment declarations and finally agent protocols. Specifications and fairness constraints are covered in Chapter 3.

2.1 Lexical Structure

The lexical structure of the language follows Haskell [20] closely. Specifically:

Comments begin with '−−' and terminate at the end of the line they appear on.

Block Comments begin with '{−' and are terminated with '−}'.

Constants start with an upper-case letter, and can be followed by any number of a mix of alphanumeric characters and underscores. These are used in enumerated types (Section 2.2), actions and agent names.

Variables start with a lower-case letter followed by any number of a mix of alphanumeric characters and underscores. Program variables and labels belong to this class.

Relational variables start with an underscore followed by any number of a mix of alphanumeric characters and underscores. These are used in μ-calculus specifications (Section 3.5).

2.1.1 Reserved Words

The reserved words in the MCK input language, of which there are many, are spelt out in the formal syntax given in Appendix A.

2.2 Types

All types are finite sets. Types can be introduced by either explicitly enumerating their elements (*enumerated types*), or by specifying a range of integers (*arithmetic types*). For example, in the following,

---

type Int3 = {0..7}
type Weekday = {Mon, Tue, Wed, Thu, Fri, Sat, Sun}

Int3 is an arithmetic type and Weekday is an enumerated type. The only primitive pre-defined type is *Bool*, which behaves as if it were defined by:

$$\textbf{type}\ \text{Bool} = \{\text{False}, \text{True}\}$$

The concrete syntax for type declarations can be found in Section A.1. Lexically, the type name and elements are **Constants**, and elements do not have to have a unique type. Note that this *ad hoc* overloading of constants restricts the allowable structure of expressions (see Sections 2.3 and A.3).

All defined types (but not Bool) are totally-ordered, so can be used in relational expressions in the natural way. The canonical ordering on the elements is the textual order in which they are defined. For example, in the context of the above declarations, we have $2 < 5$ and `Tue` < `Fri`.

To declare that a variable x has a type T, we write $x : T$.

Arrays

Environment variables can be given an array type using the standard C syntax, e.g.

$$x : \mathit{Bool}[3]$$

declares x as an array of length 3. All arrays are indexed from 0. Only one-dimensional arrays are available at present.

Arrays are best thought of as a list of variables, (with names $x[0], x[1], x[2]$ in the above case) since indexing an array by an expression (e.g., $x[i]$, where i is a variable) is not permitted. The only exception to this is the special index `self`, in the context of an agent protocol, which can be thought of as a macro for the agent number associated to that agent. (For example, inside an instance of a protocol, with the agent assigned number 3, x[`self`] refers to $x[3]$.) The utility of this is in abstracting a protocol from the concrete number of agents present in a given scenario. See Section 6.2 for an example.

2.3 Expressions

The expression sub-language is used in formulas, branching statements and the right-hand-sides of assignment statements. The syntax of expressions depends on the type of the value that they represent, and the context in which they appear. There are three types of expressions: Boolean, Numerical and Enumerated expressions. The full syntax is spelt out in Section A.3, and what follows is an overview.

- Boolean expressions are formed from the constants $True, False$ and atomic propositions using the boolean operators /\ (and), \/ (or), => (implication), <=> (bi-implication) and *xor*.
- Numerical expressions are formed from numerical constants and variables using the operators + and −. Since the language has only finite types, the interpretation of these operators is truncating: a value less than the minimum value in the type is interpreted as the minimum value, and similarly any value greater than the maximum value is interpreted as the maximum.
- Enumerated expressions are formed from the elements and variables of enumerated type, together with the monadic operators `next` and `prev`, which refer to successors and predecessors in the linear order on the type, with a rotating interpretation at the endpoints.

Expressions can mention any appropriately typed variables that are defined in the context in which they occur.

In the context of the type definitions in Section 2.2 and x : Int3 with value 3, and a:Weekday with value Sun, basic expressions of the following forms are valid, and are evaluated to the specified value (we write $e \Rightarrow v$ for e evaluates to v):

Operator	Examples
Enumeration	x **in** $\{1,2,4\} \Rightarrow$ False, x **in** $\{3..7\} \Rightarrow$ True.
Equality	$x == 2 \Rightarrow$ False, $x /= 3 \Rightarrow$ False.
Relational	$x > 3 \Rightarrow$ False, $x >= 3 \Rightarrow$ True, $x < 3 \Rightarrow$ False, $x <= 3 \Rightarrow$ True.
Truncating Arithmetic	$x + 1 \Rightarrow 4$, $x - 4 \Rightarrow 0$.
Rotating Adjacency	$\text{next}(a) \Rightarrow$ Mon, $\text{prev}(a) \Rightarrow$ Sat.

It is envisaged that other forms of arithmetic will be useful, and these will be implemented in the future.

As all types are enumerated, we have very liberal overloading rules. Indeed, the type checker simply computes an approximation to what values an expression can take on and ensures it's a subset of **Bool**, in the case of boolean formulas, or of the type of the variable being assigned to otherwise.

2.4 Programs

Programs are used to describe agent protocols, as well as in the environment description to represent transitions and initial states. This section describes the basic constructs of the program language, which is an imperative language with (probabilistic) nondeterminism, reminiscent of Dijkstra's *guarded commands* [8]. The only substantial deviation from his presentation is the addition of a **break** branch in a **do** statement which is executed when the **do** loop terminates (i.e. when all other conditions are false). Why this is important is discussed in more detail in Sections 6.1 and Appendix B.

Not all of the constructs of the language are treated as consuming time when they execute. In particular, evaluation of an expression and selection of a branch of a conditional statement are treated as taking no time. This enables the representation of the selection of the next atomic action to execute by traversal of a nested conditional statement.

Atomic Statements

Atomic statements are the basic unit of the language, corresponding to parts of a state transition in the temporal semantics. In the context of an agent protocol, atomic statements take one unit of time to execute. In the context of an initialization or transition clause, atomic statements are treated as taking no time to execute.

The variables whose values may change on the occurrence of an atomic statement depend on the context. In the context of the environment (in a transitions and initialization clause), only environment variables are affected (but this implicitly causes changes to the agent parameter variables that are aliased to these variables.) In an agent protocol, the agent's local variables may be directly affected by the atomic statement, but the statement also emits a signal to the environment that may cause changes to the environment variables. A signal *sig* emitted by agent *agt* is represented in the environment description by the atomic proposition *agt.sig*.

Skip: The primitive `skip` is an atomic statement that consumes unit time. In the context of a transitions or initialization clause, it changes no variables. In the context of an agent protocol, it changes no local variables, but emits the signal *NilAction* to the environment. (The transition clause in the environment may change environment variables as a result.)

Assignment: The assignment statement evaluates an expression and assigns its value to a variable. This type of statement has the form

$$v := expr$$

where v is variable and *expr* is an expression whose values are of the same type as v.

Nondeterministic Satisfying Assignment: The syntactic structure for a nondeterministic satisfying assignment is as follows:

$$[[var\_list \mid formula]]$$

The *var_list* is a comma separated list of variables. The *formula* is a propositional formula over all variables defined in the context in which this statement occurs, and the *primed* versions of variables in *var_list*. (If v is a variable, then its primed version is another variable written as v'.) The primed variables represent the values of the variables in *var_list* after this statement has executed. Intuitively, this statement says "change only the variables in `var_list`, so as to make the relationship *formula* hold between the current values of all variables and the new values". Since there may be many possible choices of values that satisfy the formula, this statement is nondeterministic. (It may also be interpreted probabilistically, see Chapter **??**.)

For example, when x, y, z are variables of type {1..10},

$$[[x \mid \texttt{True}\ ]]$$

represents a nondeterministic choice of any of the values 1..10 to the variable x, (and no change to any other variable), and

$$[[x, y \mid x' <= y' \ /\backslash \ y' <= z]]$$

chooses new values for both x and y, depending on the current value of z. For example, if the current value of z is 1, then the possible new assignments to the pair (x, y) are $(0, 0), (0, 1), (1, 1)$. If the current value of z is 2, then the possible new assignments to (x, y) are $(0, 0), (0, 1), (1, 1), (0, 2), (1, 2), (2, 2)$.

The nontderministic satisfying assignment operation may be used in initialization and transition clauses, and in agent protocols. In the context of an agent protocol, the variables in *var_list* must be local to that protocol. In the context of an initialization or transition clause, these variables must be global (environment) variables.

It is assumed that the formula has at least one solution for all fair reachable states in the system. The onus is on the user to ensure that this is the case. In a future version of the system, we plan to provide a debugging facility to detect situations in which there is no solution.

Actions: Atomic statements in the context of agent protocols have the dual effect of updating local variables as well as emitting a signal to the environment. Actions are a special form of atomic statement that is used to represent this type of dual effect. (The signal emitted to the environment is used to update the environment variables in a way that is described in Section 2.5.6.)

An action has one of two forms:

$$<< Action >>$$

or

$$<< Action \mid var_1 := expr_1; \ ... \ ; var_n := expr_n >> \ .$$

Here *Action* has lexical type constant, and is the name of the signal that this atomic statement emits to the environment. For each agent *AgentName* that may perform this action (as determined by its protocol), there is a special atomic proposition *AgentName.Action* that may be referred to in the context of the environment. When emitted by agent *AgentName*, the proposition *AgentName.Action* is true during the transition computation performed by the environment.

Note that $var_1, ..., var_n$ must be distinct local variables. The intuitive semantics of the assignments is that the values of the $expr_i$ are computed in the state before the action is performed, and the assignments are carried out in parallel (and simultaneously with the environment transition computation) to produce the agent's protocol state after the action is performed.

Implicit Actions: In the context of protocols, the other types of atomic statements are treated as implicitly generating the special signal *NilAction*. That is, *skip* is treated as if defined by

$$[skip] = << NilAction >>$$

assignments are treated as if defined by

$$[var := expr] = << NilAction \mid var := expr >>$$

and the statement [[*var_list* | *formula*]] is treated as issuing << *NilAction* >> to the environment in addition to updating the variables in *var_list* so as to satisfy *formula*. The simplified action form

$$<< var_1 := expr_1; ...; var_n := expr_n >>$$

is also permitted, this is treated as if defined by

$$[<< var_1 := expr_1; ...; var_n := expr_n >>] = << NilAction \mid var_1 := expr_1; ...; var_n := expr_n >>.$$

Shared Variable Read/Write: The semantics of actions is generally specified in the transitions clause of the environment. A special case of actions is provided, which captures the idiom of concurrent read/write on shared variables.

The read form for actions:

$$<< local\_var \ := \ environment\_var.read() >>$$

or

$$<< local\_var \ := \ environment\_var.read() \mid var_1 := expr_1; \ ... \ ; var_n := expr_n >> \ ,$$

assigns the value of an environment variable to a local variable, taking into account concurrent writes.

The write form:

$$<< environment\_var.write(expr) >>$$

or

$$<< environment\_var.write(expr) \mid var_1 := expr_1; \ ... \ ; var_n := expr_n >>$$

provides a way for an agent to assign a value to a shared variable.

In each of these statements, *environment_var* must be the local name of the environment variable (i.e., the parameter variable that is bound to the environment variable, see Section 2.6.1).

Programming Constructs

Alternatives: The non-deterministic choice statement has the form:

$$\textbf{if } cond_1 \rightarrow C_1 \ ... \ [] \ cond_i \rightarrow C_i \ ... \ [\textbf{otherwise} \rightarrow C_o] \ \textbf{fi}$$

where each command C_i is eligible for execution only if the corresponding condition $cond_i$ evaluates to true in the current state. If, for all i, $cond_i$ evaluates to false, then C_o is executed. If the **otherwise** branch is absent then an implicit **otherwise** → **skip** is introduced.

Traversing the **if**, evaluating the conditions $cond_i$, entering a branch via → and exiting this statement through **fi** are treated as if they consume no time.

Repetition: The non-deterministic iteration statement has the form:

$$\textbf{do } cond_1 \rightarrow C_1 \ ... \ [] \ cond_i \rightarrow C_i \ ... \ [\textbf{otherwise} \rightarrow C_o] \ [\textbf{break} \rightarrow \ C_b \ ] \textbf{od}$$

where each command C_i is eligible for execution only if it's corresponding condition $cond_i$ evaluates to true in the current state, a process which is repeated until all conditions evaluate to false. At this time the C_b statement is executed if the **break** branch is present; otherwise the system implicitly executes the **skip** command.

Traversing the **do**, evaluating the conditions $cond_i$, entering a branch via → and exiting this statement through **od** are treated as if they consume no time.

Sequential Composition: An arbitrary number of statements $C_1, ..., C_n$ to be executed in sequence can be aggregated by writing: **begin** $C_1; ...; C_n$ **end**.

Traversing the **begin** or **end** are treated as if they take no time.

Derived forms

Some other common programming idioms are supported as derived forms. The following give the expansion $[\cdot]$ of these in terms of the core language.

if-then-else: A choice construct derived from **if**. The expansion is as follows:

$$[\textbf{if } cond \textbf{ then } C_1 \textbf{ else } C_2] = \textbf{if } cond \ \rightarrow \ C_1 \ [] \ \textbf{neg } cond \ \rightarrow \ C_2 \ \textbf{fi}$$

while: A repetition construct derived from **do**. The expansion is as follows:

$$[\textbf{while } cond \textbf{ do } C] = \textbf{do } cond \ \rightarrow \ C \ [] \ \textbf{break} \rightarrow \ \textbf{skip od}$$

Intuitively, C is executed until *cond* becomes false. The **break** → **skip** branch means that it takes 1 time step to exit the while loop.

2.5 The Environment

This section describes the environment declarations that specify which agents exist, and the environment through which they communicate. The environment declarations specify a set of types, introduce a set of variables, introduce definitions relating to these variables, describe the initial states of the environment, introduce the agents (naming the protocols they run and their binding to the environment variables), describe how states of the environment are updated using a transitions clause, and give fairness conditions on runs. The formal syntax is presented in Section A.1. Note that these declarations must appear in the script in the same order as they are presented here.

2.5.1 Types

The first set of declarations introduce zero or more types, as specified in Section 2.2.

2.5.2 Shared Variables

A state of the environment is an assignment to a set of variables declared in the environment section. These variables are declared with a sequence of statements of the form

variablename : *typename*

where *typename* specifies a type.

Conceptually, these variables can be viewed as shared by the agents in the system. However, agents are generally not able to observe and update all these variables - the agent declaration statement (described below) specifies which variables each agent is able to access. The transitions section (described below) describes how the agent's actions cause changes in the environment variables.

2.5.3 Definitions

Definitions may be introduced in order to abbreviate boolean expressions in the environment declaration or in agent protocols. The syntax for definitions is

define ⟨*varid*⟩ = ⟨*expression*⟩

where ⟨*expression*⟩ is a boolean expression. Definitions are handled like macros: any occurrence of the defined variable is replaced by its corresponding expression before processing of the script.

In the environment section, definitions are placed after variable declarations and before the initial condition, and ⟨*expression*⟩ may contain environment variables, and agent qualified local variables, actions and labels.

The user is responsible for ensuring that the defined term is valid for any context (specification formula, transitions section, initial condition, fairness condition) where it is used. (E.g., an expression containing agent actions cannot be used in formulas.)

2.5.4 Initial condition

The initial values of the environment variables are constrained by the **initialization** statement. The initial state of the environment need not be unique. In some cases, it is convenient to generate the possible initial states by means of a program, starting from some default values. This is done in a two-phase process, using a statement of the form

initialization
from ⟨*fromPart*⟩
begin
  ⟨*program*⟩
end

Intuitively, the ⟨*fromPart*⟩ specifies a set of *starting* states from which the construction of initial states begins. The actual initial states are then constructed as the states reached after running the program block from these starting states. The ⟨*fromPart*⟩ may be either

- *uniform*: this specifies that each assignment to the variables yields a starting state; (The terminology is due to the fact that in the probabilistic setting, these states are equipped with the uniform distribution, see Chapter 5.)
- *all_init*: this specifies that there is a unique starting state, the state in which every variable takes the least value in its type.

After constructing a set of environment states according to the *from* part, the initialization proceeds by executing ⟨*program*⟩ from each of these starting states, and collecting the resulting final states. Each of these final states is a possible initial state of the environment.

For cases where a formula suffices to describe the initial states, the **initialization** statement can be replaced by the statement

$$\textbf{init\_cond} = formula$$

where $formula$ is a boolean formula over the variables. In this case, every environment state satisfying the formula is initial (with uniform distribution in the probabilistic setting).

The initialization statement (and in particular, the program it contains) is treated as taking no time to run, it is simply an artefact for the definition of initial states.

2.5.5 Agent Bindings

The next set of declarations bind distinct agent names to the protocols they run, and instantiate each protocol's environment parameters. There must be at least one agent in the system. Each declaration has the form

agent *AgentName* "*protocol_name*" (*var_list*)

where *var_list* is a comma separated list of variables that have been previously declared for the environment. This statement introduces an agent named *AgentName* to the environment, which will run the protocol named *protocol_name*, as declared later in the script. The protocol declaration will include a list of parameter variables for this protocol: the number and types of these much match with the variables used in the agent declaration.

The effect of the agent declaration statement is to alias the instances of the parameter variables in the context of the agent to the corresponding environment variables. Note that this introduces the possibility of having multiple names for the same variable. The usual problem of interference when two parameters are bound to the same environment variable does not arise, because updates to environment variables are made only within the environment rather than within agent protocols.

2.5.6 Transition Specification

The next part of the environment description specifies how environment states change over time, using an optional **transitions** clause.

In the execution semantics, at each step of the computation, the agents first simultaneously choose an atomic statement (see Section 2.4) to perform. The statement has both a local effect and sends a signal to the environment. The environment resolves the set of signals it receives from the agents in order to determine the values of the environment variables in the next state. (The local effects and the effect on the environment happen in the same transition and do not interfere with each other.)

The effect on the environment is determined in two phases. The first phase deals with the specialised read/write form for actions, in order to support the perspective that the environment variables are shared variables on which the agents may perform concurrent read/write operations. (This phase is always executed in the same way, and no statement needs to be added for this to be enabled.) The second phase processes the other signals sent by the agents to the environment, using the **transitions** clause, which provides a way to program how the environment variables are updated.

Phase 1: The input to the first phase is a set of signals of the form *AgentName.environment_var.write*(*value*) and *AgentName.environment_var.read*, where *AgentName* indicates the agent performing the action. This set of signals is treated as a set of concurrent read and write actions on the environment variables, with the order unspecified. This makes the semantics nondeterministic: simultaneous reads and writes to a variable are resolved to any of the possible interleavings of these actions. For example, if x has value 0 the set of actions $A1.x.write(1)$, $A2.x.write(2)$, $A3.x.read()$, could result in the value for x that is read by $A3$ being any of 0, 1 or 2, and, independently, the value of x in the successor state being either 1 or 2.

Phase 2: In phase 2, the remaining signals sent by the agents to the environment are processed. This processing is done using a program that is described in the **transitions** clause, which has the form

transitions
begin
⟨*program*⟩
end

Here ⟨*program*⟩ is a program as described in Section 2.4, but with the following syntactic restrictions:

- Only non-looping constructs are allowed (no **do** or **while** loops).
- The guards of conditional statements are boolean expressions formed from atomic propositions over environment variables, agent-qualified actions (*AgentName.Action*) and agent-qualified labels (*AgentName.label*).

An example of the use of this mechanism is the robot example of Section 6.1.

The combined effect on environment variables from the shared variable operations in Phase 1, followed by the program in the transitions clause in Phase 2 is interpreted atomically, as if it occurred in a single step of computation taking unit time. Since the both the shared variable operations and the transitions clause program may be non-deterministic, there may be many successor states for each starting state.

2.5.7 Fairness Constraints

The final section of the environment declaration concerns the fairness constraints in the system. MCK supports fairness constraints in the manner described in [9, Section 7] and [10, Section 6.3]. Briefly, a fairness constraint filters the runs of the system by accepting only those along which a given propositional formula is satisfied infinitely often. This is explained in more detail in Chapter 1.

The fairness declaration section consists of a sequence of statements of the form

fairness = ⟨*formula*⟩

where ⟨*formula*⟩ is a boolean formula. The effect of each such statement is to restrict the runs of the system to those in which the formula is satisfied infinitely often. (Since there may be multiple such statements, the effect is like that of a *generalized Büchi automaton* fairness condition.)

The formula may be constructed from atomic propositions built using environment variables, variables local to agents (in the form *AgentName.variable*) and labels local to agents (in the form *AgentName.label*; see Section 2.6.5.)

Some examples of the use of the fairness statement are give in Section 6.4.

2.6 Agent Protocols

A protocol defines an agent's behaviour. In particular, it specifies how the agent maintains its local variables and emits actions into the environment, as a function of the input it receives from the environment. The behaviour is described using a program of the type defined in Section 2.4.

The basic structure of protocols is illustrated in Figure 2.1, and is more formally spelt out in Section A.2. The structure consists of a header, local variable declaration, definitions, initial condition, and program, described in the following sections.

2.6.1 Protocol Header

The protocol header is a statement of the form

`protocol` "*protocol_name*" (*parameter_list*)

where *parameter_list* is a comma-separated list of variable declarations. Each of these variable declarations is of one of the forms

var : *type*

or

var : `observable` *type* .

The latter form declares that the variable *var* is one of the variables used in the construction of the agent's observation. In order to allow arbitrary length arrays to be passed in from the environment, *type* may be the special type form *basic_type*[], which matches an arbitrary length array. The user is responsible for consistency of indexing in this case. The keyword `self` (see Section 2.2) provides one possible index in this case. (See Section 6.2 for an example of this combination.)

The `agent` declaration in the environment aliases the local instances of the variables in the header to shared variables in the environment. The type of the environment variable and the aliasing protocol header variable should be the same.

Conceptually, the semantics of the header variables is somewhat like "call-by reference", in that fresh copies are not created for the agent-local instances. However, in order to prevent concurrency problems, the body of the protocol may not assign directly to the header variables, so many of the usual subtleties do not arise.

The protocol may "read" the header variables by using them in its expressions. In this case the phased temporal semantics needs to be borne in mind. At each program step, such references to these header variables in the protocol's program section will refer to the "current" values of the aliased variable, i.e., their values in the global state from which the next transition of the system will be taken. The value of the expressions using these current values are used to select the next action that the agent performs, which has the general form

$$<< Action \mid var_1 := expr_1;\ \ldots\ ; var_n := expr_n >>$$

As noted above the expressions $expr_i$ in this construct are also evaluated using the "current" values of the variables they contain. After this, the signal *Action* is emitted to the environment, which uses it to update the environment variables as part of the transition to the next global state of the system. The updated values of the environment variables are then available to the agent in this next state via the aliasing by parameter variables. (In case *Action* is a shared variable read, the result is written to the local variable indicated.)

2.6.2 Local Variable Declaration

The next block of the protocol text is a declaration of the local variables and their types, using a sequence of statements of the forms

var : *type*

or

var : `observable` *type* .

As with parameter variables, the latter type of declaration indicates that the variable is part of what is observable to the agent for the purposes of the epistemic semantics.

2.6.3 Initialization

The local variable declaration is followed by a statement that describes the initial values of the local variables. This can be of the form of a **where** clause or an **initialization** clause.

The former has the form

where ⟨*formula*⟩

where ⟨*formula*⟩ is either a boolean formula over atoms constructed from the local variables, or the construct *all_init*, which abbreviates the formula that states that all local variables have the least value in their corresponding type. The initial state of each instance of the protocol is non-deterministically taken to be one of the assignments satisfying the formula.

Alternately, an **initialization** clause with syntax and semantics as described in Section 2.5.4 may be used, but operating on local variables rather than environment variables.

2.6.4 Definitions

Definitions in the form

define ⟨*var*⟩ = ⟨*expression*⟩

may also be included in a protocol. In this case, ⟨*expression*⟩ may contain variables local to the agent. The defined variable may be referred to in the environment as usual as *Agt*.⟨*var*⟩.

In applying MCK to verifying that a concrete protocol implements a knowledge-based program, one wants to check that the conditions in the protocol correspond to a formula about the agent's knowledge. This leads to a need to refer to the conditions both in the protocol and in their specifications. Definitions help to abbreviate the specification and to keep it aligned to the protocol when changes are made to the protocol. (See Section 6.1 for an example of the use of conditions in knowledge-based program verification.)

2.6.5 Protocol Program

The final part of the protocol text is a program block of the form

begin
⟨*program*⟩
end

The variables occurring in the list of environment parameters, the local variables and the definitions form the context for this program block: no other variables may occur in ⟨*program*⟩. Compared to the transition statement in the environment, this program has some extra flexibility in its syntax: it may contain looping constructs, actions, and make use of labels. These are described in the following sections.

Actions

Actions and the associated execution semantics have already been described above in Section 2.4 and Section 2.5.6. Note that at each step of the temporal semantics, every agent nondeterministicallly selects one of the enabled actions. The corresponding signals are simultaneously emitted to the environment, which uses them to compute the next state of the environment variables. A more precise description of the execution semantics is given in Appendix B.

Labels

All program statements *C* in a protocol can be given a (not necessarily unique) label, which is written as follows:

label :: *C*

where *label* belongs to the **Variable** lexical class. Explication of the semantics of labels requires a distinction to be drawn between state transitions that are *enabled* versus those which are *taken*. For example, when control reaches the following program fragment:

```
do
    True -> l0 :: var := 0;
[] True -> l1 :: var := 1;
[] True -> l2 :: var := 2;
od
```

we have that all branches of the **if** statement are enabled, but only one can be (non-deterministically) taken.

Each label *label* in the protocol being run by agent *AgentName* is associate to an atomic proposition *AgentName.label*, which evaluates to *True* if and only if a statement labelled by *label* is enabled in *AgentName*'s protocol. (Thus, when control reaches the above statement, all of the propositions *AgentName.l*0, *AgentName.l*1 and *AgentName.l*2 are true. If there are several statements with the same label, then the proposition is true if any of them are enabled.

The propositions associated to labels may be used within specifications, fairness constraints and in expressions in the environment transitions specification. Section 6.4 gives an example involving all three types of use.

While it is tempting to try to use labels to specify which branches of **if**, **do** and and **while** should be treated fairly, some care is required, since the truth of a label propositions represents that the program location is enabled, rather than that the corresponding program statement will be taken. Consider, for example, the constraint that the middle branch in the above example is executed infinitely often. It is tempting to write the fairness constraint as:

$$\textbf{fairness} \;=\; \mathit{AgentName.l}1$$

but this merely asserts that the branch is *enabled* infinitely often, and does not rule out runs where it is never actually taken. The simplest solution is to use a variable *var* (inserting a new variable if necessary) that tracks when a branch is taken, and re-formulate the constraint in terms of it. Thus (provided control is guaranteed to reach this statement)

$$\textbf{fairness} \;=\; \mathit{var} \;==\; 1$$

captures the constraint that the middle branch be infinitely often taken. Similarly,

$$\textbf{fairness} \;=\; (\mathit{neg}\; \mathit{l}1) \;\backslash/\; \mathit{var} \;==\; 1$$

says that infinitely often, location *l*1 is not enabled, or the corresponding branch has been taken; this is equivalent to the statement that if location *l*1 is forevermore enabled, then eventually the corresponding branch is taken.

```
-- Types. (zero or more)
type TypeName0 = { ... elements ... }
...
type TypeNameT = { ... elements ... }


-- Shared variables. (zero or more)
varDec0 : Type
...
varDecN : Type

define def0 = .. boolean expression involving shared variables ...
..
define defL = .. boolean expression involving shared variables ...

-- Environment initial condition. (optional)
init_cond = ... boolean expression involving env variables ...

-- Agent bindings. (at least one)
agent AgentName1 "protocol for agent 1" ( ... env variables ... )
...
agent AgentNameM "protocol for agent M" ( ... env variables ... )


-- Transitions clause. (optional)
transitions
begin
  ... statements, no loops ...
end

-- Fairness constraints. (zero or more)
fairness = ... Boolean formula ...


-- Specifications. (at least one)
<specification_type> = ... temporal and knowledge formula ...


-- Protocol declarations. (zero or more - can be in a separate file)
protocol "first protocol name" ( ... env parameters ... )
-- Agent-local variables (zero or more)
localVar0 : Type
...
localVarK : Type
-- Agent-local initial condition. (optional)
  where ... boolean expression involving local variables ...

define def0 = .. boolean expression involving local variables ...
..
define defL = .. boolean expression involving local variables ...

begin
  ... statements ...
end
```

Figure 2.1: The structure of an input script.

Chapter 3

The Specification Language

MCK supports a rich specification language (a combination of CTL* with epistemic and mu-calculus operators) that is able to express a number of aspects of knowledge and time, and admits a range of different semantics for knowledge, depending on the view of the agents (observational, clock or synchronous/asynchronous perfect recall, as described in Section 1.2). In this chapter we describe the syntax of the specification language. The full language is not supported in all cases: depending on the knowledge semantics and the model checking algorithm selected, syntactic restrictions may be required. These restrictions are described in the next chapter, on algorithms.

3.1 General Form of Specifications

Specifications are of the form: ⟨*spectype*⟩ [`"` *Comment* `"`] = ⟨*formula*⟩. The *Comment* is optional; if present, the model checker will output it along with the formula. The precise form of the ⟨*formula*⟩ is specified below.

In general, specification identifiers ⟨*spectype*⟩ have the form `spec_x` or `spec_x_y`, where `x` is a keyword identifying the view on which the semantics for the agent's knowledge is based (either `obs` for observational, `clk` for clock, `apr` for asynchronous perfect recall, or `spr` for synchronous perfect recall), and `y` is a keyword identifying the algorithm to be used for model checking. The options for the algorithm are described in the next chapter.

3.2 The Propositional Core

Basic propositions in this language can be formed in several ways:

Boolean variables can be used directly: environment variables are denoted by their names, and agent variables can be accessed as *AgentName.variable*.

Equality of terms is written as $t_1 == t_2$, where t_1,t_2 are terms of the same type. Inequality is written as $t_1 / = t_2$.

Relational expressions between arithmetic or enumerated terms include $t_1 < t_2$, $t_1 <= t_2$, $t_1 > t_2$, $t_1 >= t_2$, x *in* S where x is a variable or constant and S is a concrete set, e.g. $\{3, 5, 7\}$.

Label references of the form *AgentName.label*, where *label* is a label occurring in the protocol being run by agent *AgentName*. (See Section 2.6.5.)

The usual boolean connectives (Section A.6) can be used to combine propositions.

3.3 Linear Temporal Logic

Linear Temporal Logic is a well-known *linear-time* logic used (for example) in SPIN [14]. Informally, the available operators and semantics are as follows:

Operator	Description
$\mathbf{F}\ f$	eventually f.
$\mathbf{G}\ f$	always f.
$f\ \mathbf{U}\ g$	f until g.
$f\ \mathbf{R}\ g$	f release g.
$\mathbf{X}\ f$	f in the next state.

The formula $f\mathbf{U}g$ requires that g holds at some time in the future, and f holds at all times before that time. The formula $f\mathbf{R}g$ is similar, but by contrast does not require that f holds at some time in the future; it says that g holds (possibly forever) until it is released by an occurrence of f. This can be defined in terms of the other operators as $f\mathbf{R}g = \neg(\neg f\mathbf{U}\neg g) = \mathbf{G}(g) \vee (g\mathbf{U}(f \wedge g))$. LTL can directly encode fairness conditions, but it is also possible to use CTL fairness constraints (Section 2.5.7).

An example of using LTL specifications is the following simple, not-fully-correct mutual exclusion algorithm.

```
turn1 : Bool
turn2 : Bool

-- Non-deterministic choice of who goes first.
init_cond = turn1 xor turn2

agent M1 "mutex" ( turn1, turn2 )
agent M2 "mutex" ( turn2, turn1 )

-- Safety                                                              10
spec_obs = G neg (M1.in_cs /\ M2.in_cs)

-- Non-blocking
spec_obs = G (((F M1.left_cs) /\ neg M1.in_cs) => F M1.trying)
spec_obs = G (((F M2.left_cs) /\ neg M2.in_cs) => F M2.trying)

-- Non-strict sequencing (fails)
spec_obs = EF (M1.in_cs /\ E[M1.in_cs
                              U (neg M1.in_cs /\ E[neg M2.in_cs U M1.in_cs])])
                                                                       20
-- Liveness (need to consider fairness)
spec_obs = G (((F M2.left_cs) /\ M1.trying) => F M1.in_cs)
spec_obs = G (((F M1.left_cs) /\ M2.trying) => F M2.in_cs)

protocol "mutex" ( env_turn1 : observable Bool, env_turn2 : Bool )
in_cs : Bool
  where neg in_cs

begin
  while True do                                                        30
  begin
    while env_turn1 /= True do trying :: skip;
    -- Critical section
    in_cs := True;
    do in_cs -> skip
    [] in_cs -> in_cs := False
    [] break -> skip
    od;
    -- End critical section
    left_cs :: << env_turn1.write(False) >>;                           40
    << env_turn2.write(True) >>
  end
end
```

3.4 Branching Time Temporal Logic

The branching time temporal logic CTL* is obtained by adding a branching operator **A** so that **A**f that says that the formula f holds on all possible futures from the current state. Its dual operator **E** is defined by **E**$f = \neg$**A**$\neg f$, which expresses that there exists a possible future satisfying f.

In CTL*, the branching operator **A** can be arbitrarily nested with the linear time operators of LTL. *Computation Tree Logic* is a well-known fragment of CTL*used (for example) in SMV [10], that restricts the use of the branching operator so that it always occurs directly in combination with a temporal operator. The available operators and an informal semantics are as follows:

Operator	Description
AX f	f in all next states.
EX f	f in at least one next state.
A[f **U** g]	on all paths, f until g.
E[f **U** g]	on at least one path, f until g.
A[f **R** g]	on all paths, f release g.
E[f **R** g]	on at least one path, f release g.
AF f	On all paths, in some future state, f.
EF f	On at least one path, in some future state, f.
AG f	On all paths, in all future states, f.
EG f	On at least one path, in all future states, f.

An example appears in Figure 3.1. In English, the specification might be rendered as "in all reachable states, there exists a future state where *A.var* is the case". The set of reachable states is subject to fairness constraints (see below).

3.5 μ-calculus Constructs

The μ-calculus adds greatest and least fixed-points to the branching-time model that CTL uses, and can indeed express all (unfair) CTL constructs. Concretely, in **spec_obs** specifications the following two operators can be used:

gfp _Z f greatest fixed point of f wrt variable _Z.

lfp _Z f least fixed point of f wrt variable _Z.

with the constraints that all *relational variables* (such as _Z) used in f fall under an even number of negations, and that the specification as a whole is closed with respect to relational variables.

For example, the (unfair) CTL operator **EG**f can be written as **gfp** _Z ($f \wedge$ **EX** _Z).

Note that the evaluation of fixpoints in MCK is quite naive at the moment as the Emerson-Lei algorithm (or a more-recent refinement) isn't implemented. See [10, Chapter 7] for details.

3.6 Knowledge Modalities

The knowledge modality is written **Knows** *Agent formula*. The various semantics for this operator were spelt out in Section 1.2.

For the clock and observational semantics, a *common knowledge* operator is also available, written **CK** {$Agent_1$, ..., $Agent_n$} *formula* (*formula* is common knowledge to the specified agents), or **CK** *formula* (*formula* is common knowledge to all agents).

3.7 Exponentiation of operator prefixes

It is sometimes useful to iterate a sequence of modalities, and this is supported by the notation **O**^n ϕ, where **O** is a sequence of modal operators and n is a positive natural number, which abbreviates **O O ...O** ϕ, where there are n copies

of **O**. For example, one common use of this notation is in formulas **X**^$n(\phi)$, stating that ϕ holds n steps in the future. (Previous versions of MCK supported the notation **X** n ϕ for this: for backwards compatibility, this is still supported, but deprecated.) Another example is (**Knows** A **Knows** B)^2 p, expressing **Knows** A **Knows** B **Knows** A **Knows** B p.

3.8 Local Next

In the context of specifications dealing with agents operating asynchronously, it is of interest to specify what an agent knows after it has made a given number of distinct observations, which cannot be done using the next-time operator **X**. The local next operator **XK** can be used for this purpose. This is used in the syntactic form **XK** `AgentName` ϕ, where ϕ is a formula that may contain agent `AgentName`'s knowledge operators, but no temporal operators or knowledge operators involving other agents. The meaning of this formula is that "if there exists a later time where the agent's local state differs from its current local state, then while the agent is in that next local state, the formula ϕ holds. Note that this means that if there is, on the current run, no future time where the agent's local state differs from its current local state, then the formula is trivially true. This construct can presently only be used in the context of asynchronous perfect recall specifications of the form (**XK** `AgentName`)^n ϕ, asserting that ϕ describes the agent's knowledge in all situations where its asynchronous perfect recall local state consists of $n + 1$ observations.

3.9 Resolution of Ambiguity

MCK will accept ambiguous specification formulas such as **Knows** A p $\wedge$ **Knows** B q, which could be parsed either as (**Knows** A p) $\wedge$ (**Knows** B q), or as **Knows** A (p $\wedge$ **Knows** B q). The system uses operator priorities and associativity rules to resolve these ambiguities; this particular example is parsed as (**Knows** A p) $\wedge$ (**Knows** B q). How the system has resolved an ambiguity can always be seen from the way that the formula is printed out when model checking.

Details of operator priority and associativity rules are given in Appendix A. However, since changes to the language may cause the operator priority rules to change from version to version, we recommend that users avoid relying on these, and prefer to write unambiguous formulas through sufficient use of parentheses.

```
agent A "while" ()

fairness = A.var

spec_obs_ctl = AG AF A.var

protocol "while" ()
var : Bool
  where neg var

begin
  do True -> var := False
  [] True -> var := True
  od
end
```

Figure 3.1: An example of a CTL specification.

Chapter 4

Model Checking Algorithms

MCK supports several different types of model checking algorithms for temporal epistemic model checking. This chapter describes these algorithms, syntactic restrictions and semantic assumptions under which they apply, and additional features supported in some cases. A summary is given in Table 4.1.

Semantics	Flag	Algorithm	Formula Restrictions	Type	Game	Visual.	C-trace
spec_obs		spec_obs_ctl	CTL / lfp / gfp / K / CK	BDD			
		spec_obs_ctls	CTL* / K / CK	BDD			
	-k	spec_obs_es	CTL / K / CK	ESMC	√	√	
	-b Int	spec_obs_bmc_ctl Int	(univ) CTL / K / CK	BMC	√	√	
	-b Int	spec_obs_bmc Int	(univ) CTL* / K / CK	BMC			
spec_clk		spec_clk_xn	leading X^n / K / CK	BDD			√
		spec_clk_ctl_nested	AX / EX / K / CK	BDD			
		spec_clk_nested	A / X / K / CK	BDD			
		spec_clk_g	leading G / K / CK	BDD			√
	-b Int	spec_clk_bmc Int	(univ) A / R / G / X / K / CK	BMC			
spec_spr		spec_spr_xn	leading X^n / K_1	BDD			√
		spec_spr_nested	A / X / K / CK	BDD			
		spec_spr_g	leading G / K_1	BDD			√
	-b Int	spec_spr_bmc Int	(univ) A / R / G / X / K / CK	BMC			
spec_apr		spec_apr_xn	leading XK_1^n / K_1	BDD			√
		spec_apr_g	leading G / K_1	BDD			√

Table 4.1: Algorithms in MCK

The choice of algorithm used can be controlled in one of two ways: either as part of the command line call of MCK, using one of the command line flags (e.g., `mck -k` for explicit state model checking), or as part of the identifier of a specification statement. In general, specification identifiers have the form `spec_x` or `spec_x_y`, where `x` is a keyword identifying the view on which the semantics for the agents' knowledge is based (either `obs` for observational, `clk` for clock, `spr` for synchronous perfect recall, or `apr` for asynchronous perfect recall – see Section 1.2), and `y` is a keyword identifying the algorithm to be used for model checking. The algorithms available are based on a variety of approaches: symbolic model checking using Binary Decision Diagrams (BDD), Explicit State Model Checking (ESMC), or Bounded Model Checking (BMC), but the details for each approach vary with the knowledge semantics and formula type.

When an algorithm is explicitly identified in a specification statement, that algorithm is always used to model check the specification. If the algorithm part `y` is omitted, then the algorithm to be applied is determined by the system according to the following rules. If no flag is given on the command line, then the default approach of BDD-based model checking is used. If an algorithm flag applicable to the specification's knowledge semantics is given on the

command line, then this flag is used to select the algorithm. If several flags applicable to a specification are given on the command line, the first applicable one is selected. If several algorithms are available that have the same selected flag then the first algorithm in the table that is consistent with the syntax of the formula is applied.

In general, the algorithms for the observational semantics are more efficient than those for the clock semantics, which are in turn more efficient than those for the two perfect recall semantics. Since the algorithms for the observational semantics also work on the more expressive temporal syntax, it is recommended to specify properties that do not require epistemic operators using **spec_obs**.

4.1 Explicit State Model Checking

Explicit state model checking works by explicitly constructing all reachable states of the system and then inductively labelling these states with the subformulas of the specification formula that hold at these states. This is essentially the classical approach to model checking [6], extended in the obvious way to handle epistemic operators. This algorithm works only for the observational semantics, and formulas in the language fragment based on CTL with knowledge and common knowledge operators. It is invoked using either the construct `spec_obs_es` or using `spec_obs` together with the command line flag `-k`.

While explicit state model checking can only be expected to be effective on small models, it has the advantage that it generates information that enables the complete state space of the system to be visualized (using the `-v` command line flag), which may be useful for debugging the model. It also produces information for the debugging game (invoked using flag `-g`) to be played on the model. See Chapter 7 for a description of these features.

4.2 Binary Decision Diagram Symbolic Model Checking

MCK's default algorithms (invoked without command line flags) use symbolic model checking techniques based on binary decision diagrams (BDD's). There are several algorithms, depending on the semantics for knowledge.

The BDD based model checking algorithms in MCK do not presently allow either state space visualization or the counter-example game to be used. Support for the counter-example game may be added in a future release. However, several of these algorithms do support a simpler counter-example facility, the generation of a single counter-example trace on which the specification fails (invoked using the **-**c flag).

4.2.1 BDD Algorithms for Observational Semantics

The system has two algorithms for dealing with the combination of knowledge and time under the observational semantics, depending on whether the temporal basis is the branching time logic CTL or the logic CTL*, which includes the linear time logic LTL.

The specification statement `spec_obs_ctl` invokes an extension of the BDD-based model checking algorithm for the temporal logic CTL [3, 19], where epistemic operators for knowledge and common knowledge have been added to the language, as well as mu calculus operators. Fairness constraints are handled following the techniques in [13].

Formulas based on CTL* operators plus knowledge and common knowledge operators are handled using the specification statement `spec_obs_ctls`. The mu-calculus operators may not be used in this case. The associated algorithm first transforms the CTL*-based formula into a CTL-based formula, using techniques from [7], and then applies the CTL-based model checker.

For statements `spec_obs` and an invocation without command line flags, whichever of the above cases applies to the syntax of the formula is invoked.

4.2.2 BDD Agorithms for Clock Semantics

Clock Semantics is handled by two BDD-based model checking algorithms that work for fragments of CTLK$_n$ and LTLK$_n$, respectively.

The specification statement `spec_clk_xn` works on formulas of the form $X\hat{}n\ \phi$, where ϕ is a formula that may contain boolean, knowledge and common knowledge operators (nested arbitrarily), but not temporal operators.

The algorithm works by constructing a BDD representation of the states of the system at time n and then applying BDD-based techniques to model check the formula ϕ on that representation.

The statement `spec_clk_nested` works on formulas in the language fragment based on the temporal operators A, X with boolean, knowledge and common knowledge operators, nested arbitrarily. The algorithm in this case constructs a BDD representation of paths of length equal to the maximal depth of temporal nesting, and uses this as a basis for BDD-based model checking of the formula. (This is a generalization of the algorithm for the construct `spec_clk_ctl_nested` from earlier versions of MCK, which allowed the operators AX, EX and K_i, nested arbitrarily. This is still supported for backward compatibility, and may be more efficient when the formula can be expressed in this fragment.)

The specification statement `spec_clk_g` works on formulas of the form $G\ \phi$, where ϕ is a formula that may contain boolean, knowledge and common knowledge operators (nested arbitrarily), but not temporal operators. The algorithm works by constructing a loop of BDD representations of the states of the system and then applying BDD-based techniques to model check the formula ϕ on all those representations. Each BDD representation is the set of states that is possible on a specific time.

For statements `spec_clk` and an invocation without command line flags, the first of the above cases that applies to the syntax of the formula is invoked.

Counter-example traces are supported for two of these algorithms, `spec_clk_xn` and `spec_clk_g`, and can be invoked by using the **-c** flag (see Section 7.3).

4.2.3 BDD Agorithms for Synchronous Perfect Recall Semantics

The synchronous perfect recall semantics for knowledge is handled by three BDD-based model checking algorithms that work for fragments of CTL*K$_n$.

The specification statement `spec_spr_xn` works on formulas of the form $X\hat{}n\ \phi$, where ϕ is a formula that may contain boolean operators and just *one* agent's knowledge operator, nested arbitrarily, but not temporal operators or common knowledge operators. The algorithm works by constructing a BDD representation of the mapping from sequences of observations to time n for the agent to the sets of states of the system that are consistent with those observations. BDD-based techniques are then used to model check the formula ϕ on that representation. The algorithm is described in more detail in [23].

The specification statement `spec_spr_g` works on formulas of the form $G\ \phi$, where ϕ is a formula that may contain boolean operators and just *one* agent's knowledge operator, nested arbitrarily, but not temporal operators or common knowledge operators. The algorithm in this case constructs all reachable *knowledge states* of the agent in question, where a knowledge state is the set of states that the agent considers to be possible, given its sequence of observations. The knowledge states are represented as BDD's.

The statement `spec_spr_nested` works on formulas in the language fragment based on the boolean operators and modal operators A, X, K_i, nested arbitrarily. The algorithm in this case constructs a BDD representation of paths of length equal to the maximal depth of temporal nesting, and uses this as a basis for BDD-based model checking of the formula.

For statements `spec_spr` and an invocation without command line flags, the first of the above cases that applies to the syntax of the formula is invoked.

Counter-example traces are supported for two of these algorithms, `spec_spr_xn` and `spec_spr_g`, and can be invoked by using the **-c** flag (see Section 7.3).

4.2.4 BDD Algorithms for Asynchronous Perfect Recall

Two algorithms are provided for the asynchronous perfect recall semantics. Counter-example traces are supported for these algorithms, and can be invoked by using the **-c** flag (see Section 7.3).

The specification statement `spec_apr_xn` works on formulas of the form $(XK\ A)^n\ \phi$, where A is an agent and ϕ is a formula that may contain boolean operators and just agent A's knowledge operator, nested arbitrarily, but no temporal operators or common knowledge operators. Here $(XK\ A)$ is the local next-time operator, see Section 3.8. The algorithm works by constructing a BDD representation of the mapping from sequences of observations of length

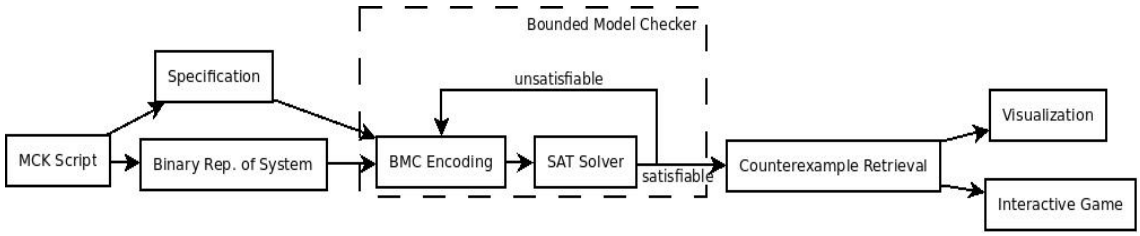

Figure 4.1: The bounded model checking procedure

n for the agent to the sets of states of the system that are consistent with those observations. BDD-based techniques are then used to model check the formula ϕ on that representation.

The specification statement `spec_apr_g` works on formulas of the form $G\ \phi$, where ϕ is a formula that may contain boolean operators and just *one* agent's knowledge operator, nested arbitrarily, but not temporal operators or common knowledge operators. The algorithm in this case constructs all reachable *knowledge states* of the agent in question, where a knowledge state is the set of states that agent considers to be possible, given its sequence of observations. The knowledge states are represented as BDD's. The formula ϕ is then checked for satisfaction in all the reachable knowledge sets.

4.3 Bounded Model Checking

MCK provides bounded model checking for several temporal epistemic logics and knowledge semantics. Bounded model checking, abbreviated as BMC, aims at refuting, instead of justifying, a specification by using a counterexample in the system. Figure 4.1 gives an overview of the workflow when using MCK's bounded model checker.

For bounded model checking, the specification is required to be a formula in the universal fragment of a logic. The universal fragment of a logic requires that the negation operator may apply only to atomic propositions, and all modal operators are universal operators such as AG and K_i.

Bounded model checking relies on the fact that for universal formulas, a small part of the model may suffice to serve as a counter-example that demonstrates that the formula is false. Inspection of this counter-example helps to identify the error in the specification or model. For branching and temporal-epistemic logic, these counter-examples require several fragments (prefixes) of runs of the system.

The procedure for finding a counterexample starts from run fragments of length 1. If the model checker cannot find a counterexample of length $i \geq 1$, it will proceed to length $i + 1$, until a bound k is reached. The bound k should be explictly assigned by the user. This can be done either by using a generic specification statement `spec_x` and the command line flag `-bInt`, where `Int` is a positive integer, or by giving the bound in the specification statement, e.g. `spec_obs_bmc Int`.

A BMC encoding takes as input the binary representation of the system, a specification to be checked and a length $i \leq k$ of run fragments, and outputs an encoded formula. MCK transforms the encoded formula into conjunctive normal form (CNF) and then uses a SAT solver to decide whether it is satisfiable or not. The embedded SAT solver is PicoSAT (see `http://fmv.jku.at/picosat/`).

4.3.1 Bounded Model Checking for Observational Semantics

There are two specification statements `spec_obs_bmc_ctl` and `spec_obs_bmc` for bounded model checking with respect to the observational semantics.

The statement `spec_obs_bmc_ctl` uses a BMC encoding for the universal fragment ACTLK_n defined by the grammar

$$\psi = \ \neg p \mid p \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid K_i\psi \mid C_G\psi \mid AX\psi \mid AG\psi \mid AF\psi \mid A(\psi U \psi) \mid A(\psi R \psi).$$

Several encoding functions have appeared in other bounded model checkers for this temporal epistemic logic. The one implemented in MCK is described in [15], together with theoretical and experimental results on its performance

Semantics	Algorithm	Formula Restrictions	Type	Subsumed by
spec_obs	spec_obs_ltl	LTL / K / CK	BDD	spec_obs_ctls
spec_spr	spec_spr_ltl_nested	X / K	BDD	spec_spr_nested

Table 4.2: Deprecated Algorithms

compared to other implementations. When a `spec_obs_bmc_ctl` specification fails, MCK is able to present the counterexample found by bounded model checking to the user in two ways: visualization (invoked using the `-v` flag) and by playing the interactive game (invoked using the `-g` flag). See Chapter 7 for a description of visualization and the interactive game. We note that in the case of bounded model checking, only the states that occur in the counter-example are visualized, rather than all states of the system, as is done for explicit state model checking.

The statement `spec_obs_bmc` works for the richer universal fragment ACTL*K$_n$, defined by the grammar

$$\psi = \neg p \mid p \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid K_i\psi \mid C_G\psi \mid A\psi \mid X\psi \mid G\psi \mid F\psi \mid \psi U\psi \mid \psi R\psi.$$

Note that here the branching operator A does not need to be directly coupled with a temporal operator. For example $A(Fp \vee Fq)$ is a formula of this language, but not of ACTLK$_n$.

Visualization and the counter-example game are not presently supported in this case.

In case `spec_obs` is used together with command line flag `-b`, `spec_obs_bmc_ctl` is used provided the formula is in the smaller fragment ACTLK$_n$, otherwise `spec_obs_bmc` is used.

4.3.2 Bounded Model Checking for Clock Semantics

Bounded model checking for the clock semantics is supported through the statement `spec_clk_bmc`. The formula must be in the language fragment given by the grammar

$$\psi = \neg p \mid p \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid K_i\psi \mid C_G\psi \mid A\psi \mid X\psi \mid \psi R\psi \mid G\psi$$

in this case. Visualization and the counter-example game are not presently supported.

4.3.3 Bounded Model Checking for Synchronous Perfect Recall Semantics

Bounded model checking for the synchronous perfect recall semantics is supported through the statement `spec_obs_bmc`. The formula must be in the language fragment given by the grammar

$$\psi = \neg p \mid p \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid K_i\psi \mid C_G\psi \mid A\psi \mid X\psi \mid \psi R\psi \mid G\psi$$

in this case. Visualization and the counter-example game are not presently supported.

The system does not presently support bounded model checking for the asynchronous perfect recall semantics.

4.4 Deprecated Specification Statements

The specification statements in Table 4.1 encompass all the functionality in the system. In some cases, the algorithms generalise algorithms from previous versions of MCK, which used specification statements with more restrictive nomenclature. Table 4.2 lists these statements and the new statements under which they are subsumed. For backwards compatibility, these older statements are still supported, but their use is deprecated.

Chapter 5

Model Checking Probabilistic Knowledge

MCK implements an algorithm for the verification of probabilistic knowledge, that works on models in the form of partially observed discrete time Markov chains. (That is, all branching is treated probabilistically when dealing with probabilistic specifications: the combination of nondeterminism and probability is not presently supported.) The underlying theory is described in greater detail in [16]. The present chapter describes the syntactic features involved when dealing with probabilistic models and specifications.

5.1 Probabilistic Input Language

The input language for the probabilistic algorithm is a simple extension on the input language for non-deterministic algorithms, as described in Chapter 2. All non-deterministic statements are interpreted as probabilistic statements with uniform distributions over the choices, and some extra syntax is provided to indicate non-uniform probabilities in the case of branching statements. One restriction is required: no fairness statements may be used in the presence of probabilistic specifications, since the interaction of probability and fairness is problematic when there are unfair transitions with non-zero probability.

5.1.1 Probabilistic Satisfying Assignment

In the context of probabilistic specifications, the nondeterministic satisfying assignment construct is interpreted probabilistically. The nondeterministic interpretation of [[$Vars$ | $formula$]] yields, from each state, a set of possible successor states. The probabilistic interpretation gives each of these possible successors equal probability.

For example, when x, y, z are variables of type {1..10},

$$[[x \mid \texttt{True}\ ]]$$

represents a probabilistic choice of new value to the variable x, each with probability 1/10 (and no change to any other variable), and

$$[[x, y \mid x' <= y' /\backslash y' <= z]]$$

chooses new values for both x and y, depending on the current value of z, with a uniform distribution over the combined choices. Thus, if the current value of z is 1, then the possible new assignments to the pair (x, y) are $(0, 0), (0, 1), (1, 1)$, each with probability 1/3. If the current value of z is 2, then the possible new assignments to (x, y) are $(0, 0), (0, 1), (1, 1), (0, 2), (1, 2), (2, 2)$, each with probability 1/6.

5.1.2 Shared Variables

The interpretation of concurrent read/write on shared variables introduces nondeterminism. The present system provides only limited support for the probabilistic interpretation of this nondeterminism. A model is said to be *conflict-free* if no reachable state enables a simultaneous read and write, or two simultaneous writes, on the same variable. The

inherent nondeterminism of read/write actions does not occur in the case of conflict free models. MCK presently supports only the conflict-free case of read/write actions in the presence of probabilistic specifications. The onus is on the user to ensure that the model is conflict-free when read/write and probabilistic specifications are combined.

5.1.3 Conditional Statements

Conditional statements such as **if** and **do** enable, at each state, a set of choices for the next step of execution. In the probabilistic interpretation, each of these occurs with equal probability. That is, if a condition is false at a state, then the corresponding action occurs with probability zero, and the probability is distributed uniformly across the remaining *enabled* branches.

For example, in the statement

if
$True -> \ x := 1$
$[] \ x < 3 -> \ x := x + 1$
$[] \ x = 1 -> x := x + 2$
if

at a state where $x = 1$, all three branches have probability $1/3$ of executing, and at a state where $x = 2$, the first two branches have probability $1/2$ and the last has probability 0.

In order to represent non-uniform distributions, additional syntax is available that enables weighting of the branches of conditional statements. The weighted probabilistic **if** statement has the form

$$\textbf{if } p_1 : \ cond_1 \rightarrow C_1 \ ... \ [] \ p_i : \ cond_i \rightarrow C_i \ ... \ [\textbf{otherwise} \rightarrow C_o] \ \textbf{fi}$$

and the probabilistic **do** statement has the form

$$\textbf{do } p_1 : \ cond_1 \rightarrow C_1 \ ... \ [] \ p_i : \ cond_i \rightarrow C_i \ ... \ [\textbf{otherwise} \rightarrow C_o] \ [\textbf{break} \rightarrow \ C_b \ ] \ \textbf{od} \ .$$

Here the p_i are real numbered weights that represent the relative likelihood between the enabled branches. The probabilities associated to the enabled branches in a given state are obtained by normalization.

For example, for the statement

if
$1 : \ True -> \ x := 1$
$[] \ 2 : \ x < 3 -> \ x := x + 1$
$[] \ 3.0 : \ x = 1 -> x := x + 2$
if

at a state where $x = 1$, the three branches have probability $1/6$, $2/6$, and $3/6$ of executing, respectively, and at a state where $x = 2$, the first two branches have probability $1/3$ and $2/3$ and the last has probability 0.

Note that if the weights are omitted, then they are implicitly interpreted as equal to 1 on all branches, so that all enabled branches are always equally weighted.

5.1.4 Initialization

The probabilistic interpretation assumes that agents have a *common prior*, i.e., that they all assign the same probability to initial states. The prior on initial states is the product of the probability distributions constructed using the initialization statements in the environment and agent protocols. The form of these statements in the probabilistic case is the same as described in Section 2.5.4 and 2.6.3.

When the *from* part of an initialization statement is `uniform`, the starting states have the uniform distribution, when it is `all_init`, the unique starting state has probability 1. Note that the program part of the initialization statement may contain probabilistic branching statements, which update the probability distribution. The resulting final distribution at the end of executing the program in the initialization statement, starting from the distribution on the starting states, gives the distribution on the initial states of the model. In the case of an **init_cond** or **where** statement, the initial states are given a uniform distribution.

5.2 The Probabilistic Specification Language

The probabilistic specification language extends that of Chapter 3 by adding constructs that talk about the probabilistic knowledge of agents. The *basic probability terms* are given in the following table:

Modality	Description
Prob *Agent f*	*Agent*'s current probability of f.
Prior *Agent f*	*Agent*'s *prior* probability of f.
Prob *Agent* (f | g)	*Agent*'s current probability of f under the condition g.
Prior *Agent* (f | g)	*Agent*'s *prior* probability of f under the condition g.

In each case, f and g are required to be boolean formulas that may refer to local, environment and agent-qualified variables. Each form has the type of a real number in the interval $[0, 1]$. The prior forms refer to the agent's probability at the initial state in the run at which the formula is being evaluated. This is useful for specifications such as $X\hat{}5$ ((**Prob** i ϕ) == (**Prior** i ϕ)), which says that the agent does not increase its degree of belief about ϕ in the first 5 steps of the computation.

Basic probability terms and real numbers may be combined using the operators +, $*$, to form probability terms. Atomic propositions are then derived of the form $t_1 R t_2$, where t_1, t_2 are probability terms and R is one of the relations ==, / =, <, <=, > or >=.

Only a limited temporal semantics is presently supported, however: the specification formula to be checked should be in the form of $X\hat{}k$ ϕ with the only modal operators in ϕ being the knowledge modality K_i and the probability modalities $\texttt{Pr}_i$ and $\texttt{Prior}_i$, for a single, fixed agent i. (These operators may be nested and combined with boolean operators.)

Two epistemic semantics are supported for specifications in this form: synchronous perfect recall and clock semantics. (Observational and asynchronous perfect recall semantics would imply that the combination of nondeterminism and probability need to be dealt with: the system does not yet attempt to support this.) In both cases only one model checking algorithm is available, that is called `xn`. Thus, invocation of the algorithm can be done with either of the specification identifiers: `spec_spr_xn` or `spec_clk_xn`. The forms `spec_spr` or `spec_clk` in which the algorithm is omitted will default to these.

The underlying algorithm is a symbolic model checking algorithm based on OBDD and MTBDD [5], and is an extension of the `xn` algorithm for the cases of `spr` and `clk` semantics. It works by constructing BDD representations of the mapping from sequences of observations to time n for the agent to the sets of states of the system that are consistent with those observations. BDD-based techniques are then used to model check the formula ϕ on that representation. The details of the algorithm and some experimental results are described in the paper [16].

The usual variable reordering heuristics for the BDD-based algorithms are available in the probabilistic case, and the counterexample trace facility works with these operators.

5.3 Precision and Threshold

The probability operators take real values in the range $[0, 1]$. The precision of the computation is 10^{-16} by the default setting of CUDD package. To enable a comparison between two probabilities, a threshold of 10^{-10} is used. Given two probabilities x and y,

- if $|x - y| \leq 10^{-10}$ then we decide that $x = y$,
- if $x - y > 10^{-10}$ then we decide that $x > y$,
- other relational operations are deduced from the above two operations.

Chapter 6

Examples

This chapter illustrates the kinds of properties the model checker can verify, and some of the subtleties that may arise when formalising a system.

6.1 The Robot example

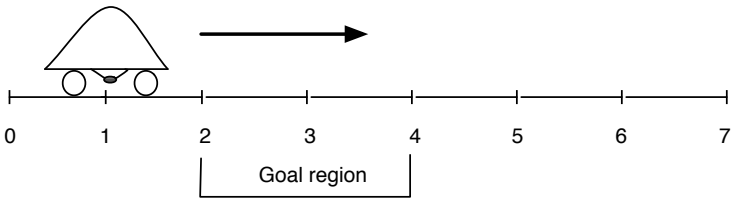

Figure 6.1: Autonomous Robot

This example is taken from [2]. To quote the summary found in [11]:

> A robot travels along an endless corridor, which in this example is identified with the natural numbers. The robot starts at 0 and has the goal of stopping in the goal region $\{2, 3, 4\}$. To judge when to stop the robot has a sensor that reads the current position. (See Figure 6.1.) Unfortunately, the sensor is inaccurate; the readings may be wrong by at most 1. The only action the robot can actively take is halting, the effect of which is instantaneous stopping. Unless this action is taken, the robot may move by steps of length 1 to higher numbers. Unless it has taken its halting action, it is beyond its control whether it moves in a step.

A sound and complete solution to this problem is to do nothing while the sensor has a value of less than 3, and halt as soon as it takes on a value of 3 or more. (The naive solution of halting iff the sensor reads 3 is sound but not complete.)

In order to model check an implementation of the robot's control policy, we need to restrict the environment to a finite number of locations. We have arbitrarily chosen to have 8 distinct locations, but any number greater than 4 is sufficient.

Bearing this restriction in mind, the following script implements the scenario and the proposed protocol:

−− There are 8 positions in the world.
−− If the robot is really at position p, then the sensor will have a
−− value $\in \{p-1, p, p+1\}$*, for a truncating interpretation of arithmetic.*
type Pos = $\{0..7\}$

```
incpos : Bool
position : Pos
sensor : Pos
halted : Bool

init_cond = incpos /\ position == 0 /\ sensor == 0 /\ neg halted

agent Robot "robot" ( sensor )

-- At each time step the environment might move the robot one step to the
-- right, and always generates a new sensor reading.
transitions
begin
  if neg halted /\ neg Robot.Halt ->
    begin
      position := position;
      incpos := False
    end
  [] neg halted /\ neg Robot.Halt ->
    begin
      position := position + 1;
      incpos := True
    end
  [] Robot.Halt -> halted := True
  fi;
  if True -> sensor := position - 1
  [] True -> sensor := position
  [] True -> sensor := position + 1
  fi
end

-- Rule out the traces where the environment stops trying to advance.
fairness = incpos \/ halted

-- Knowledge-based program specification agrees with the implementation.
spec_obs = G (Robot.test <=> Knows Robot position in {2..4})

-- The "car handbrake" protocol.
-- In order to stop moving, the robot only needs to yank it once.
protocol "robot" (sensor : observable Pos)

define test = sensor >= 3

begin
  do neg test -> skip
  [] break    -> <<Halt>>
  od
end
```

Note that timing is critical in this example: the robot must have continuous control over the emitted Actions, and must be able to register its intention to halt with the environment *before* the environment decides to move it any further. These two constraints mean that using a **while** construct, with the implied **skip**-on-exit, is not sufficient for correctness. This "timing gap" is illustrated by this (incorrect) protocol:

```
protocol "robotbroken-agent-protocol.mck" (sensor : observable Pos)

begin
  while neg (sensor >= 3) do skip;
  <<Halt>>
end
```

The sequence of (*position*, *sensor*, *robot action*) values:

$$\langle (0,0,Nil),(1,0,Nil),(2,1,Nil),(3,2,Nil),(\mathbf{4},\mathbf{3},\mathbf{Nil}),(\mathbf{5},\mathbf{4},\mathbf{Halt}),...\rangle$$

is an example of a run where the robot decides it wants to stop when it receives a sensor reading of 3, but doesn't manage to assert the halted action until it has moved past the goal region.

6.2 The Dining Cryptographers

The problem solved by this protocol is framed as follows [4]:

> Three cryptographers are sitting down to dinner at their favorite three-star restaurant. Their waiter informs them that arrangements have been made with the maître d'hôtel for the bill to be paid anonymously. One of the cryptographers might be paying for the dinner, or it might have been the NSA (US National Security Agency). The three cryptographers respect each other's right to make an anonymous payment, but they wonder whether the NSA is paying.

(The details of the model checking of this protocol are given at length in [23].)

Assuming that at most one cryptographer is paying, the following protocol will allow all cryptographers to discover whether the NSA or one of their fellows is paying:

```
protocol "dc-agent-protocol.mck"
(
  paid : observable Bool,
  chan_left : Bool,
  chan_right : Bool,
  said : observable Bool[] -- the broadcast variables.
)

coin_left : observable Bool
coin_right : observable Bool                                                    10
  where all_init

begin
  -- The enviroment tells us whether we paid or not.
  -- The agent decides the coin toss to the right.
  [[coin_right|True]];
  << chan_right.write(coin_right) >>;
  << coin_left := chan_left.read() >>;
  << said[self].write(coin_left xor coin_right xor paid) >>
end                                                                             20
```

The following enviroment implements the scenario for three cryptographers:

```
paid : Bool[3]
chan : Bool[3]
said : Bool[3]

initialization
from all_init
begin
if True -> paid[0]:= True
[] True -> paid[1]:= True
[] True -> paid[2]:= True                                                       10
fi
end


-- Agents are numbered in the order they appear.
agent C0 "dc-agent-protocol.mck" (paid[0], chan[0], chan[1], said)
agent C1 "dc-agent-protocol.mck" (paid[1], chan[1], chan[2], said)
agent C2 "dc-agent-protocol.mck" (paid[2], chan[2], chan[0], said)
```

This protocol illustrates the utility of arrays of environment variables – in this case simply to implement a broadcast. As the number of agents scales, **initialization** statement has the advantage over the **init_cond** statement in this case of being able to express the initial states linearly rather than quadratically.

The specifications for this example can be done either using knowledge operators, or, more precisely, using probabilistic knowledge.

```
-- This talks about the knowledge of the first agent.
spec_spr_xn = X 4
              (neg paid[0]) => ((Knows C1 (neg paid[0])
                                          /\ (neg paid[1])
                                          /\ (neg paid[2]))
                                \/ ((Knows C1 (paid[1] \/ paid[2]))
                                          /\ (neg (Knows C1 paid[1]))
                                          /\ (neg (Knows C1 paid[2]))))

-- A probabilistic version ...
spec_spr_xn = X 4 (neg paid[0]) => (    (Prob C0 paid[1]) == (Prob C0 paid[2]) /\
                                   ( (Prob C0 paid[1]) == 0 \/ (Prob C0 paid[1]) == 0.5 ) )
```

6.3 Shared Variables Example

The following example illustrates what happens when several agents attempt to concurrently read and write different values to an environment variable. All of the specifications in this example hold.

```
type Three = {One, Two, Three}

envVarDef : Three

init_cond = envVarDef == One

agent WF "conflicting_env_write_Two" ( envVarDef )
agent WT "conflicting_env_write_Three" ( envVarDef )
agent R "conflicting_env_write_read" ( envVarDef )

spec_obs = neg EX envVarDef == One
spec_obs = EX envVarDef == Two
spec_obs = EX envVarDef == Three
spec_obs = EX (envVarDef == Two /\ R.var == One)
spec_obs = EX (envVarDef == Two /\ R.var == Two)
spec_obs = EX (envVarDef == Two /\ R.var == Three)
spec_obs = EX (envVarDef == Three /\ R.var == One)
spec_obs = EX (envVarDef == Three /\ R.var == Two)
spec_obs = EX (envVarDef == Three /\ R.var == Three)

protocol "conflicting_env_write_Two" (envVar : Three)
begin
  << envVar.write(Two) >>
end

protocol "conflicting_env_write_Three" (envVar : Three)
begin
  << envVar.write(Three) >>
end

protocol "conflicting_env_write_read" (envVar : Three)
var : Three
  where all_init
```

```
begin
  << var := envVar.read() >>
end
```

In essence, the resulting value of envVarDef is one of the values written to it.

6.4 Fairness

An example of using a fairness declaration to eliminate undesired runs is shown in Figure 3.1. The constraint **fairness** = *A.var* ensures that the runs that, after some finite period of time, forevermore have *var* false are eliminated.

A more complex example is given in Figure 6.2, where we encode a nondeterministic Buchi automaton using the initial condition, transitions specification and fairness constraint to express the property that always, if Action1 is infinitely often enabled, then it is eventually taken.

```
type BuchiState = {S0,S1,S2,S3}

b_l0: BuchiState

init_cond = b_l0 == S0


agent A "prot1" ()

transitions
begin
if
    b_l0 == S0 /\ A.Action1 -> b_l0 := S1
 [] b_l0 == S0  -> skip
 [] b_l0 == S0  -> b_l0 := S2

 -- being in S1 means that we just saw A.Action1
 -- we immediately switch to another state
 [] b_l0 == S1  -> b_l0 := S2
 [] b_l0 == S1  -> b_l0 := S0

 -- once in S2, we must never again have A.l0
 [] b_l0 == S2 /\ A.l0 -> b_l0 := S3
 [] b_l0 == S2 /\ (neg A.l0)    -> skip

 -- once in S3, we cannot satisfy the acceptance condition
 [] b_l0 == S3  -> skip
fi
end


fairness = b_l0 in {S1,S2}


spec_obs_ctls = "Always, if we are at l0 infinitely often, then we eventually do Action1"
                G( (G (F A.l0)) => F A.didAction1 )


protocol "prot1" ()

didAction1: Bool
looping: Bool

where looping /\ (neg didAction1)

begin
-- we can choose to fall out of the loop at any time, in which case
-- we are only finitely often at l0, otherwise we are there infinitely often
while looping do
  begin
    l0:: if True -> <<Action1 | didAction1 := True>>
            [] True -> <<Action2>>
            [] True -> looping := False
         fi;
    -- didAction1 is true for just one step
    didAction1 := False
  end
end
```


Figure 6.2: Encoding a Buchi automaton for a fairness constraint

Chapter 7

Debugging: Visualization and Interactive Game

MCK provides several types of support for debugging of models and specifications, depending on the type of specification and model checking algorithm used. When using explicit state model checking or bounded model checking, one can use visualization of the statespace (or counter-example, in the case of bounded model checking) and an interactive game played between the user and the system in a variant of Hintikka game semantics for modal logic. A more primitive debugging mode is available for perfect recall specifications, where a single path on which a formula fails can be obtained. (We plan to give a more encompassing and unified treatment of these capabilities in later releases of the system.)

7.1 Visualization

The visualization functionality is invoked using `mck -v`. In combination with **spec_obs_es**, the visualization shows the entire statespace of the system, as generated in explicit model checking (so is only useful for systems with small statespaces.) In combination with **spec_obs_bmc_ctl**, the visualization shows the states that take part in a counter-example for a failing specification.

The states visualized are output as a file in the Graphviz[1] graph visualization "dot" format. MCK then invokes a visualization tool on this file. By default this is the `zgrviewer`[2]. The user can override this default in two ways. One is to give the invocation command for the visualization tool using full path name as a parameter, i.e. using `mck -v<tool>`. (For zgrviewer, we would use `<tool>="java -jar <path>/zgrviewer-<version>.jar -Pdot"`.) If this has not been done, then the system will use the invocation command in the environment variable `MCK_VISUALIZER`, if this has been set, otherwise fall back to a search for the default.

7.2 Interactive Game

An interactive game, played between the user and the system, can be generated from each specification formula. The game is structured so that on failing specifications, the system has a winning strategy in the game. The system will play this strategy, so the user will be unable to win the game. (If the user ever wins on a failing specification, this is a bug in `mck` and should be reported!) However, the process of playing the game will draw the user's attention to specific states of the model and specific parts of the specification, by following evaluation of the formula on states of the model. In effect, the system's winning strategy amounts to showing the user a counter-example to the specification.

[1] see `http://www.graphviz.org/`

[2] See `http://zvtm.sourceforge.net/zgrviewer.html`.

Besides helping the user to understand why a specification has failed, the game can also be used to allow the user to explore the structure of the model. For example, the effect of playing the game on the (invalid) specification

$$\textbf{spec\_obs\_es} = EF\ neg\ Knows\ A\ True$$

is to allow the user to construct a path in the model from an initial state, followed by a move from the final state of this path to another state that is indistinguishable to agent A. This allows the user to explore both the temporal and the epistemic structure of the model.

The following gives an informal account of the interactive debugging game as implemented in MCK. For a more formal presentation, see the paper [17].

In the current version, the interactive game functionality can be used in combination with the specification types **spec_obs_es** and **spec_obs_bmc_ctl** (see chapter 4).

The game mode can be invoked by supplying the `-g` flag when invoking the program. When a specification that has been model checked using **spec_obs_es** and **spec_obs_bmc_ctl** fails, the user will be prompted to play game on this specification. If the user answers yes, then game mode is entered. Otherwise, the program continues with the next specification. (Game mode is only entered when a specification fails, but the user may invoke playing of the game on a formula ϕ that holds by model checking the formula $\neg\phi$ using **spec_obs_es**. Note that does not work with **spec_obs_bmc_ctl** because if ϕ is universal then its negation will not be universal.)

When game mode is invoked, information will be output to a file `<input_script>.game.info`. The information in `<input_script>.game.info` is intended for inspection by the user playing the game, and contains the following:

Environment: The environment information includes agent bindings and environment protocol. This is the same information as is output when using the `-e` flag.

Agent Protocols: The pre-processed agent protocol information includes internally assigned labels for tracking protocol statement locations. This is the same information as is output when using the `-p` flag.

Explicit states model or Encoded Model: The explicit states model information includes the internally assigned state numbers, state valuations, transition relations, and agent equivalence classes according to observational semantics. This is the same information as is output when using the `-m` flag. The algorithm **spec_obs_bmc_ctl** does not compute the whole state space, so in this case the information output is the encoded model information, which includes agent information, formula representations of initial condition and transition relation, current and next state variables, types of variables, and fairness constraints. This is the same information as is output when using the `-f` flag.

The user may find it helpful to refer to the game file `<input_script>.game.info` when playing game, since the interactions between the system and the user are in terms of the information stated in this game file.

7.2.1 States of the Game

A typical state of the game is given by (1) a state of the model (called the focus state), (2) a formula (called the focus formula) and (3) a role for the user. The focus formula will be a subformula of the failing specification or one of the atomic propositions $\{Fair, Reach, Init\}$, asserting that the focus state is fair, reachable or initial. When dealing with certain path quantifiers, some more complicated types of game state arise:

1. When dealing with focus formulas EFf, AGf, $E[f_1 U f_2]$, $A[f_1 U f_2]$, $C_G f$, and reachability claims, one of the players will be required to construct a fair path or knowledge chain and show that formulae hold in some states. A fair path $s_1 ... s_n$ is a sequence of fair states such that there are transitions from state s_i to s_{i+1} for all $1 \leq i \leq n-1$. A knowledge chain $s_1 ... s_n$ is a sequence of fair and reachable states such that there are indistinguishability relations for some agent in G from state s_i to s_{i+1} for all $1 \leq i \leq n-1$.

2. When dealing with focus formulas EGf and AFf, and fairness claims, one of the players will be required to construct a fair full-cycle path and show that formulae hold in some states in the path or in some states in the

loop of the path. A full-cycle path $s_1 ... s_n$ is a sequence of fair states such that $s_n = s_i$, for some $1 \leq i \leq n-1$, and there are transitions from state s_j to s_{j+1} for all $1 \leq j \leq i-1$. Moreover, we call the part $s_i ... s_{n-1}$ the loop in this full-cycle path. A full-cyle path is fair if each of the fairness conditions in the model are satisfied at some state in the loop.

7.2.2 The Roles of Players

At each round of the game, one of the players (the system or the user) can make a move, depending on the focus formula and the roles of the players.

The user's role can be **VERIFIER**, in which case the game state corresponds to an assertion by the user that the focus formula is true in the focus state, or **REFUTER**, in which case the game state corresponds to an assertion by the user that the focus formula is false in the focus state. The system will always take on the opposite role.

The game starts at an initial state of the model, with the failing specification as the focus formula, and the user in the role of the **VERIFIER**. The roles of the players change during the game depending on the specific formula being played.

7.2.3 Winning Conditions

The following combinations decide the winner of the game:

Game states at which the focus formula is an atomic proposition p determine the winner, depending on the focus state s:

- If p is true at s, the **VERIFIER** wins. (Thus the user wins if she is the **VERIFIER**, otherwise the system is the **VERIFIER**, so the user loses).
- If p is false at s, the **REFUTER** wins. (Thus the user wins if she is the **REFUTER**, otherwise the system is the **REFUTER**, so the user loses).

7.2.4 Unfair States

It is possible to write programs that have reachable states that can never occur on a fair run. We call such a reachable state an *unfair state*. Since the semantics is based on fair runs, unfair states are ignored during model checking. However the existence of unfair reachable states is typically a bug that the user will want to be able to diagnose. If such states exist when using explicit model checker, before the game starts, the user will be prompted to decide whether they should be included when playing the game. If the user answers yes, unfair states will show up when they are valid next states. Otherwise, unfair states will be invisible to the user.

If the user ever makes a game move leading to an unfair state, the user will become the **VERIFIER** for a fairness claim for this state. This requires constructing a fair full-cycle path from this state.

7.2.5 Unreachable States

When making a move of the game where the user is requested to choose from a set of states, the user also has the option of constructing a state. It is possible for the user to construct an unfair or an unreachable state t. In this case, the game will proceed in a mode where the user is required to justify its fairness or reachability. The reachability of this state is justified by constructing a path from an initial state to the given state. The construction of the path is done backwards, i.e. the first step is to construct a state s such that there is a transition from s to t. Path construction then continues with s, and play would terminate with a win for the user if an initial state were ever reached. (However, since the state is unreachable, this will never happen.)

7.2.6 Game Rules and Interactions

When game mode is first entered, the game starts at an initial state. The user will be prompted to select one of the initial states in which the specification fails.

During the game, when the program asks for input from the user, the user is given several options for the next step of the game, including the option to construct a new state. Additionally, the user is given the opportunity to quit or undo previous moves of the game. There are two ways in which previous moves can be undone. The first is called Backtracking, which returns play to the previous game state. The backtracking option is available to the user in each step during the play. The second is called Skipback, and is available when the user is constructing a path. If Skipback is applied, the play will return to the starting state of path construction. When using either of these options to return to a previously visited game state, the user will be reminded of the choices previously made from this state.

The following describes the game rules and interactions for each type of focus formula. When the game state is not final (there is no winner at that state), each type of formula determines whether it is the **REFUTER** or the **VERIFIER**'s move, and a set of choices available to this player for their next move in the game, each associated with a transition to a new game state. If it is the user's move, the user will be offered a set of choices for the next move of the game. When it is the system's move, the system will display the set of choices available to the system, as well as the choice it has taken.

If unfair states are being displayed, the user will never be able to win the game by choosing an unfair state. If the user chooses an unfair state, he/she will be required to construct a full-cycle path such that all fairness constraints are satisfied in this path. This will be impossible, but the process helps the user understand why it is an unfair state.

Atomic Propositions

If the focus formula is atomic, there is a winner and the game state is final. See the winning conditions above.

$f \wedge g$

It is the **REFUTER**'s move to choose which conjunct the **REFUTER** believes is false. Play proceeds on the same focus state with the chosen subformula as the new focus formula and the players in the same roles.

$f \vee g$

It is the **VERIFIER**'s move to choose which disjunct the **VERIFIER** believes is true. Play proceeds on the same focus state with the chosen subformula as the new focus formula and the players in the same roles.

$\neg f$

Play proceeds on the same focus state with the subformula f as the new focus formula and the players exchanging their roles.

f **xor** g

The formula will be unfolded into $(\neg f \wedge g) \vee (f \wedge \neg g)$.

$f \rightarrow g$

The formula will be unfolded into $\neg f \vee g$.

$f \leftrightarrow g$

The formula will be unfolded into $(f \rightarrow g) \wedge (g \rightarrow f)$.

EXf

The formula **EX**f will be transformed into **AX**$\neg f$ with the players exchanging their roles.

EFf

The formula **EF**f will be transformed into **AG**$\neg f$ with the players exchanging their roles.

E[f**U**g]

It is the **VERIFIER**'s move to construct a fair path to a state where the **VERIFIER** believes that the formula g holds, such that the formula f holds in *all* previous states before the final state in the path. Then, the **REFUTER** should counter-assert this fact by choosing whether the formula g does not hold in the final state or the formula f does not hold in one of the states before the final state in the path or the final state is unfair. Play proceeds on the chosen state as the new focus state with the subformula f or g or atomic proposition *Fair* as the new focus formula and the players' roles staying the same. In case the user is the **VERIFIER**, path construction and counterclaim is handled incrementally. The formula is unfolded into $g \vee (f \wedge \mathbf{EX}(\mathbf{E}[f\mathbf{U}g]))$ and the user offered a choice of disjuncts. The path construction will be terminated by the user's choosing of the left disjunct or the fact that a full-cycle path is completed. The completion of a full-cycle path is regarded as a failure of the **VERIFIER**.

EGf

The formula **EG**f will be transformed into **AF**$\neg f$ with the players exchanging their roles.

AXf

It is the **REFUTER**'s move to construct a successor *fair* state where the **REFUTER** believes the formula f does *not* hold. Then, the **VERIFIER** should counter-assert this fact by choosing whether the formula f holds on that state or the state is unfair. Play proceeds on the constructed state as the new focus state with the subformula $\neg f$ or atomic proposition *Fair* as the new focus formula and the players exchanging their roles.

In case the user is the **REFUTER**, the state may be chosen from a list of valid states, or constructed by the user by assigning values to each variable. In the latter case, the state constructed may be unreachable or unfair, and play continues with the reachability or fairness subgame. See Sections 7.2.4 and 7.2.5.

AFf

It is the **REFUTER**'s move to construct a full-cycle *fair* path, starting from the current focus state, where the **REFUTER** believes the formula f does *not* hold in *all* states in the path. Then, the **VERIFIER** should counter-assert this fact by choosing whether $\neg f$ or one of the fairness constraints fails in the chosen path. If formula $\neg f$ is chosen, then the **VERIFIER** will choose from the path a state where the **VERIFIER** believes the chosen formula does *not* hold. If one of the fairness constraints is chosen, then the **REFUTER** will choose from the *loop* a state where they believe that the chosen fairness constraint holds. Play proceeds on the chosen state as the new focus state with the formula $\neg f$ or one of the fairness constraints as the new formula and the players exchanging their roles. In case the user is the **REFUTER**, path construction and counterclaim is handled incrementally. The formula is unfolded into $(f \vee \mathbf{AX}(\mathbf{AF}f))$ and the user offered a choice of disjuncts. The path construction will be terminated by the user's choosing of the left disjunct or the fact that a full-cycle path is completed.

A[f**U**g]

The formula **A**[f**U**g] will be unfolded into $\neg(\mathbf{EG}\neg g \vee \mathbf{E}[\neg g\mathbf{U}(\neg g \wedge \neg f)])$.

AGf

It is the **REFUTER**'s move to construct a *fair* path, starting from the current focus state, where the **REFUTER** believes that the formula $\neg f$ holds in the final state of the path. Then, the **VERIFIER** should counter-assert this fact by choosing whether the formula f holds on the final state or the final state is unfair. Play proceeds on the final state as the new focus state with the subformula $\neg f$ or atomic proposition *Fair* as the new focus formula and the players

exchanging their roles. In case the user is the **REFUTER**, path construction and counterclaim is handled incrementally. The formula is unfolded into $(f \wedge \mathbf{AX}(\mathbf{AG}f))$ and the user offered a choice of conjuncts. The path construction will be terminated by the user's choosing of the left conjunct or the fact that a full-cycle path is completed. The completion of a full-cycle path is regarded as a failure of the **REFUTER**.

$\mathbf{A}[f\mathbf{R}g]$

The formula $\mathbf{A}[f\mathbf{R}g]$ will be unfolded into $\neg\mathbf{E}[\neg f\mathbf{U}\neg g]$.

$\mathbf{E}[f\mathbf{R}g]$

The formula $\mathbf{E}[f\mathbf{R}g]$ will be unfolded into $\mathbf{EG}g \vee \mathbf{E}[g\mathbf{U}(f \wedge g)]$.

$\mathbf{K}\ A\ f$

It is the **REFUTER**'s move to choose or construct a *fair and reachable* state which agent A is unable to distinguish from the current focus state (i.e. observable variables have the same value in both states) and where the **REFUTER** believes the formula f does *not* hold. Then, the **VERIFIER** should counter-assert this fact by choosing whether the formula f holds on that state or the state is unfair or the state is unreachable. Play proceeds on the chosen state as the new focus state with $\neg f$ or *Fair* or *Reach* as the new focus formula and the players exchanging their roles.

In case the user is the **REFUTER**, the state may be chosen from a list of valid states, or constructed by the user by assigning values to each variable. In the latter case, the state constructed may be unreachable or unfair, and play continues with the reachability or fairness subgame. See Sections 7.2.4 and 7.2.5.

$\mathbf{CK}\ \{A_1, \ldots, A_n\}\ f$

It is the **REFUTER**'s move to construct a knowledge chain $s_0, \ldots, s_k$, with s_0 equal to the current focus state s, and the **REFUTER** believes the formula f does *not* hold in the last state s_k in the sequence. Then, the **VERIFIER** should counter-assert this fact by choosing whether the formula f holds on the final state s_k or one of the states s_j with $0 \leq j \leq k$ on the path is unfair or unreachable. Play proceeds on the state s_k as the new focus state with subformula $\neg f$ as the new focus formula or state s_j as the new focus state with atomic proposition *Fair* or *Reach* as the new focus formula, and the players exchanging their roles. In case the user is the **REFUTER**, the construction and counterclaim is handled incrementally. The formula is unfolded into $(\bigwedge_{i=1}^{n}(\mathbf{K}\ A_i\ (f \wedge \mathbf{CK}\ \{A_1, \ldots, A_n\}\ f)))$ and the user offered a choice of conjuncts. The path construction will be terminated by the user's choosing of the conjunct f or the fact that a full-cycle path is completed. The completion of a full-cycle path is regarded as a failure of the **REFUTER**.

Fair

The atomic proposition *Fair* will be transformed into $\mathbf{EG}true$ with the players' roles staying the same.

Reach

The atomic proposition *Reach* will be transformed into $\mathbf{EF}Init$, played using the *reversed* temporal transition relation, with the players' roles staying the same.

Init

The atomic proposition *Init* will be decided directly, depending on the focus state s. If s is an initial state, then the **VERIFIER** wins, otherwise the **REFUTER** wins.

7.3 Clock and Perfect Recall Counterexamples

The game has not yet been implemented for specifications that use clock or perfect recall semantics: we intend to implement this in the future. For the moment, a simpler debugging facility exists. This is available only for the specification types **spec_clk_xn**, **spec_clk_g**, **spec_spr_xn**, **spec_spr_g**, **spec_apr_xn** and **spec_apr_g**. These specification types admit formulas of the form $X\hat{}n\ \phi$ or $G\phi$ or $(XK\ A)\hat{}n\ \phi$, where ϕ is a formula with knowledge and/or probability operators but not temporal operators. When a specification of one of these types fails, and MCK has been invoked using the **-c** flag, the system will print out an execution such that ϕ fails at the final point of this execution. This does not give a full explanation of the reason for the failure of the epistemic part of the specification, so the user is required to manually analyze the trace to determine the reason for this failure.

Chapter 8

The Graphical User Interface

In addition to the command line invocation, a graphical user interface is available for the system. The interface assumes that a local installation of the Qt framework is available, and requires some configuration before use. See the Readme.txt file in the distribution for details.

The interface is invoked in Linux by calling the program `xmck`. On a Macintosh, click on the `mck.app` icon or, from the command line, call `mck.app/Contents/MacOS/mck`. Due to differences in the implementation of Qt on these platforms, there may be some differences in behaviour of the interface.

8.1 Configuration and Preferences

The entry "Preferences" on the main "System" menu opens a window where some basic installation parameters should be set at first use. These are:

- *mck bin*: the location of the MCK command-line program binary
- *dot bin*: the location of the tool used to visualize dot files (e.g. Graphviz `http://www.graphviz.org`)
- *workspace*: a workspace directory. The interface will store its temporary files as a subdirectory `tmp` of this directory.

8.2 Menu Bar

The top level menu bar provides a main menu for general MCK features as well as one menu corresponding to each of the active interface tabs. The main "System" menu has items including "About" (MCK version and license information for this release), "Preferences" (see above), and "Quit". The "Edit" menu bar contains items relating to edit actions such as "Find" in the currently active window.

The main purpose of the remaining menus is to provide open and save functionality relating to each of the active tabs in the interface. The "File" menu relates to complete MCK scripts (typically with suffix *.mck*, but this is optional). The "Environment", "Protocol" and "Specification" tabs relate to the component parts of such scripts. The interface has a window for each part, and supports loading and saving these parts via the correponding menus. The suffixes used on the corresponding files are *.env*, *.spec*, and *.prot*. Each of these menus provides the operations "New", "Open", "Save", and "Save As" with the usual meanings. The currently active file name is shown with each window. In case the window displays a version of the file that has been edited but not saved, this is indicated by an asterisk ($*$) after the file name and by use of italic font for the filename.

The "Log" menu contains items for saving output from the model checker: "Save Result" for saving model checking results in the Log window and "Save Game Run" for saving a run of the debugging game. The "CounterExample" menu may be used to save the counterexample traces generated by the BDD model checker.

A "Reordering" menu appears if the BDD variable reordering functionality has been invoked, where input and output variable orderings (see below for further explanation of these) may be saved (there is no distinction between "Save" and "Save As" in this case since the file name may already be edited directly in the tab.)

8.3 Tabs

The interface provides various windows, grouped into several tabs, to facilitate the editing of MCK scripts, model checking tasks, and the display of results.

Model Tab

The "Model" tab contains two editable windows for the current model's environment and protocol sections. Line numbers are provided on each, and error messages from the parser may refer to these line numbers. The Environment and Protocol menus are used to save and load the corresponding `.env` and `.spec` files, or the environment and protocols may be saved together with the specifications from the Specification tab in a `.mck` file using the File menu.

Specification Tab

The main area in the Specification tab is a list of specifications, each consisting of a semantics `X`, algorithm `Y` and formula ϕ, corresponding to a statement `spec_X_Y =` ϕ in the input script. In case a specification uses a bounded model checking algorithm, there is also an editable column `Length` for the BMC parameter.

The semantics and algorithm entries may be clicked to activate a menu for selection of the value: some consistency rules are enforced, so the `Semantics` entry of a specification should generally be chosen first. (Due to behaviour of Qt, it may be necessary to click outside of an entry after making a selection to get that selection to register in the system.)

If no algorithm is selected on a specification (equivalently, the `default` value is selected), corresponding to a statement `spec_X =` ϕ, MCK will choose an algorithm to apply using the rules described in Chapter 4. A preferred algorithmic approach for these cases may be set in the "Default Algorithm Type" section, either BDD, BMC (bounded model checking), or Explicit State, corresponding, respectively, to use of no flag, the `-b` flag, or the `-k` flag in the command line program. In the case BMC is selected a default BMC parameter value should be set.

If a default algorithm type of BDD is selected, the option of setting a BDD heuristic is available. There are also selection boxes for variable ordering (for `-o` and `-s` flags) and counter-example generation (`-c flag`). These activate the Reordering and Counter-example tabs (see below).

To the left of the specifications are selection boxes that are used to choose a subset of the specifications to be checked. The model checker is activated using the Start button, and may be halted using the Stop button. The model checking results are presented in two columns to the right of a specification. The first of these indicates the outcome of model checking: "Fails" or "Holds" or, in the case of bounded model checking, "Holds ($\leq n$)" to express that bounded model checker can not find a counterexample up to a BMC parameter length of n. The second column to the right of a specification is used to access debugging information. In case of observational semantics formulas for which the debugging game is available, this contains a button that may be used to load that specification into the Game tab (see below). Otherwise, provided that the BDD-Counterexample option has been selected, and this is available for the specification, a button is provided that switches to the corresponding result in the `Counter-example` tab.

Game Tab

The Game tab provides an interface to the counter-example game and visualization capabilities (see Chapter 7). Failed specifications are loaded into the Game by pressing the corresponding `Game` button in the Specification tab. Before playing a game, we need to identify an algorithm type (Explicit State or BMC) that will be used by the system. (This involves rerunning the model checking computation. Note that the game may be available for specifications checked using a BDD algorithm, but run with a very different performance when using one of the available game algorithms.)

Visualization may also be activated before playing the game. If activated, the state space or a counterexample will be visualized using the visualization tool selected in the mck/Preferences menu. After pressing the "Start" button, the

game is played at the "Game" area. This consists of windows displaying the current game state, and for selecting a move in the game. The selection may be made either by clicking on one of the move buttons or by typing the choice and RETURN into the bottom window. A record of the play of the game may be saved using the Log/SaveGameRun menu item.

Log Tab

This tab contains an area "Result" that displays all the model checking results output produced by the model checker, together with timing information on each model checking run. This log may be saved using the Log/SaveResult menu item.

Counterexample Tab

The counter-example tab displays the counter-examples generated in the case of `clk` and `spr` specifications (see Section 7.3). A specific counter-example can be accessed by clicking the corresponding "C-ex" button in the Specification tab. These results may be saved using the CounterExample menu.

Reordering Tab

This tab displays information relating to variable ordering, and is activated if the user selects default BDD model checking and the Order box in the Specification tab.

The *input* ordering is the variable ordering that is provided to the BDD algorithm at the start of model checking (corresponding to the `-s` flag). A default ordering can be constructed from the input script using the "Generate" button. The BDD heuristics may change this ordering during the computation of a BDD for the model: the *output* ordering refers to the final order after model checking is complete (corresponding to the `-o` flag). An existing ordering may be loaded for use as the input order by using the corresponding Find button. The orderings displayed may be sorted by dragging the variable entries into their new location, and the revised order saved by using the Save options in the Reordering menu. (Note that a revised input order must be saved before it can be used by the model checker, and a revised but unsaved order will be lost when invoking the model checker, which will also overwrite the existing output file.)

The typical use case is to first generate the default ordering, manually reorder this, save into an input order file, specify an output file name and then run model checking. Once this is done, the computed variable order is saved to the specified output file and also displayed in the Output order window.

Bibliography

[1] J. C. M. Baeten. *Applications of Process Algebra*, chapter *An Introduction to Process Algebra* by J.A. Bergstra and J.W. Klop. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[2] Ronen I. Brafman, Jean-Claude Latombe, Yoram Moses, and Yoav Shoham. Applications of a logic of knowledge to motion planning under uncertainty. *JACM*, 44(5), 1997.

[3] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. *Inf. Comput.*, 98(2):142–170, June 1992.

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

[5] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2):149–169, 1997.

[6] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[7] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at ltl model checking. In David L. Dill, editor, *CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 415–427. Springer, 1994.

[8] Edgar W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.

[9] E. Clarke, O. Grumberg, and D. Long. Verification Tools for Finite State Concurrent Systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency-Reflections and Perspectives*, volume 803, pages 124–175, Noordwijkerhout, Netherlands, 1993. Springer-Verlag.

[10] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[11] Kai Engelhardt, Ron van der Meyden, and Yoram Moses. A program refinement framework supporting reasoning about knowledge and time. In Jerzy Tiuryn, editor, *Foundations of Software Science and Computation Structures*, volume 1784 of *Lecture Notes in Computer Science*, pages 114–129. Springer-Verlag, March 2000.

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

[13] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173. ACM, 1980.

[14] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[15] Xiaowei Huang, Cheng Luo, and Ron van der Meyden. Improved bounded model checking for a fair branching-time temporal epistemic logic. In *Sixth Workshop on Model Checking and Artificial Intelligence (MoChArt 2010)*, 2010.

[16] Xiaowei Huang, Cheng Luo, and Ron van der Meyden. Symbolic model checking of probabilistic knowledge. In *the thirteenth conference on Theoretical Aspects of Rationality and Knowledge (TARK XIII)*, pages 177–186, 2011.

[17] Xiaowei Huang and Ron van der Meyden. Model checking games for a fair branching-time temporal epistemic logic. In *Australasian Conference on Artificial Intelligence*, volume 5866 of *LNCS*, pages 11–20. Springer, 2009.

[18] M. Huth and M. Ryan. *Logic in Computer Science: modelling and reasoning about systems*. Cambridge University Press, 2000.

[19] K. McMillan. *Symbolic Model Checking: An Appropach to the State Explosion Problem*. PhD thesis, Department of Computer Science, Carnegie Mellon University, 1992.

[20] Simon Peyton Jones, John Huges, et al. Haskell 98: A non-strict, purely functional language. `http://haskell.org/definition/`, February 1999.

[21] Ron van der Meyden. Knowledge-based programs: on the complexity of perfect recall in finite environments (extended abstract). In *Proceedings of the Conference on Theoretical Aspects of Reasoning about Knowledge*, Renesse, Netherlands, mar 1996.

[22] Ron van der Meyden. Common knowledge and update in finite environments. *Information and Computation*, 140(2), 1998.

[23] Ron van der Meyden and Kaile Su. Symbolic model checking the knowledge of the dining cryptographers. In *CSFW*, pages 280–. IEEE Computer Society, 2004.

Appendix A

Input Script Syntax

This section describes the abstract syntax of an input script in EBNF. The convention is that ⟨*UpperCase*⟩ production names denote non-terminals, and ⟨*lowercase*⟩ names denote lexemes.

There are the following lexical classes (see Section 2.1): ⟨*constant*⟩ and ⟨*typename*⟩ are **Constants**, ⟨*label*⟩ and ⟨*varid*⟩ are **Variables**, and ⟨*muvar*⟩ are **Relational variables**. The class ⟨*int*⟩ signifies integers and ⟨*digit*⟩ signifies decimal digits.

A.1 Joint Protocols and the Environment

⟨*JointProtocol*⟩ ::= ⟨*Environment*⟩ ⟨*Protocol*⟩*

⟨*Environment*⟩ ::= ⟨*TypeDec*⟩*
⟨*EnvVarDec*⟩*
⟨*Definition*⟩*
⟨*EnvInitCond*⟩?
⟨*EnvAgentDec*⟩*
⟨*EnvTransitions*⟩?
⟨*EnvFairness*⟩*
⟨*EnvSpec*⟩+

⟨*TypeDec*⟩ ::= '`type`' ⟨*typename*⟩ '=' '`{`' ⟨*Vartype*⟩ '`}`'

⟨*EnvVarDec*⟩ ::= ⟨*VarDec*⟩

⟨*Definition*⟩ ::= '`define`' ⟨*varid*⟩ '=' ⟨*Expr*⟩

⟨*EnvAgentDec*⟩ ::= '`agent`' ⟨*constant*⟩ ⟨*String*⟩ '`(`' [⟨*VarList*⟩] '`)`'

⟨*EnvInitCond*⟩ ::= ⟨*InitDec*⟩ | '`init_cond`' '=' ⟨*BoolExpr*⟩

⟨*EnvFairness*⟩ ::= '`fairness`' '=' ⟨*BoolTransitionsExpr*⟩'

A.1.1 Initialization

⟨*InitDec*⟩ ::= '`initialization`' ⟨*InitFromPart*⟩ ⟨*InitProgPart*⟩

⟨*InitFromPart*⟩ ::= '`from`' ⟨*InitFromDec*⟩

⟨*InitFromDec*⟩ ::= '`uniform`' | '`all_init`'

⟨*InitProgPart*⟩ ::= ⟨*TransitionsBlock*⟩

A.1.2 Environment Transitions Clause

⟨*EnvTransitions*⟩ ::= 'transitions' ⟨*TransitionsBlock*⟩

⟨*TransitionsBlock*⟩ ::= 'begin' ⟨*TransitionsStatement*⟩ (';' ⟨*TransitionsStatement*⟩)* 'end'

⟨*TransitionsStatement*⟩ ::= ⟨*TransitionsBlock*⟩
| 'if' ⟨*TransitionsClause*⟩ ('[]' ⟨*TransitionsClause*⟩)* [⟨*TransitionsClauseOtherwise*⟩] 'fi'
| 'if' ⟨*BoolTransitionsExpr*⟩ 'then' ⟨*TransitionsStatement*⟩ 'else' ⟨*TransitionsStatement*⟩
| 'skip'
| ⟨*Assignment*⟩
| ⟨*Randomization*⟩

⟨*TransitionsClause*⟩ ::= [⟨*Probability*⟩ ':'] ⟨*BoolTransitionsExpr*⟩ '->' ⟨*TransitionsStatement*⟩

⟨*TransitionsClauseOtherwise*⟩ ::= 'otherwise' '->' ⟨*TransitionsStatement*⟩

⟨*Randomization*⟩ ::= '[[' ⟨*VarList*⟩ '|' ⟨*BoolExpr*⟩ ']]'

⟨*Assignment*⟩ ::= ⟨*varid*⟩ ':=' ⟨*Expr*⟩

A.2 Agent Protocols

⟨*Protocol*⟩ ::= ⟨*ProtocolHeader*⟩ ⟨*EnvVarParams*⟩ ⟨*LocalVars*⟩ ⟨*Definition*⟩* ⟨*Block*⟩

⟨*ProtocolHeader*⟩ ::= 'protocol' ⟨*String*⟩

⟨*EnvVarParams*⟩ ::= '(' [⟨*VarDec*⟩ (',' ⟨*VarDec*⟩)*] ')'

⟨*LocalVars*⟩ ::= [⟨*VarDec*⟩+ [⟨*LocalVarInitCond*⟩]]

⟨*LocalVarInitCond*⟩ ::= 'where' ('all_init' | ⟨*Expr*⟩) | ⟨*InitDec*⟩

⟨*Block*⟩ ::= 'begin' ⟨*LabelledStatement*⟩ (';' ⟨*LabelledStatement*⟩)* 'end'

⟨*LabelledStatement*⟩ ::= [⟨*label*⟩ '::'] ⟨*Statement*⟩

⟨*Statement*⟩ ::= ⟨*Block*⟩
| 'if' ⟨*Clause*⟩ ('[]' ⟨*Clause*⟩)* ['[]' 'otherwise' '->' ⟨*LabelledStatement*⟩] 'fi'
| 'do' ⟨*Clause*⟩ ('[]' ⟨*Clause*⟩)*
['[]' 'otherwise' '->' ⟨*LabelledStatement*⟩]
['[]' 'break' '->' ⟨*LabelledStatement*⟩]
'od'
| 'if' ⟨*BoolExpr*⟩ 'then' ⟨*LabelledStatement*⟩ 'else' ⟨*LabelledStatement*⟩
| 'while' ⟨*BoolExpr*⟩ 'do' ⟨*LabelledStatement*⟩
| 'skip'
| ⟨*Assignment*⟩
| ⟨*Randomization*⟩
| '<<' ⟨*Action*⟩ ['|' ⟨*ActionAssignments*⟩] '>>'

⟨*Clause*⟩ ::= [⟨*Probability*⟩ ':'] ⟨*BoolExpr*⟩ '->' ⟨*LabelledStatement*⟩

⟨*Action*⟩ ::= ⟨*constant*⟩
  | ⟨*Var*⟩ '`.`' '`write`' '`(`' ⟨*Expr*⟩ '`)`'
  | ⟨*varid*⟩ '`:=`' ⟨*Var*⟩ '`.`' '`read`' '`(`' '`)`'

⟨*ActionAssignments*⟩ ::= ⟨*ActionAssignment*⟩ ('`;`' ⟨*ActionAssignment*⟩)*

⟨*ActionAssignment*⟩ ::= ⟨*Var*⟩ '`:=`' ⟨*Expr*⟩

A.3 Expressions

⟨*BoolTransitionsExpr*⟩ ::= ⟨*VarPrime*⟩ | ⟨*constant*⟩
  | ⟨*LocalObject*⟩
  | ⟨*Var*⟩ '`in`' '`{`' ⟨*Vartype*⟩ '`}`'
  | ⟨*ArithExpr*⟩ '`==`' ⟨*ArithExpr*⟩ | ⟨*ArithExpr*⟩ '`/=`' ⟨*ArithExpr*⟩
  | ⟨*EnumExpr*⟩ '`==`' ⟨*EnumExpr*⟩ | ⟨*EnumExpr*⟩ '`/=`' ⟨*EnumExpr*⟩
  | ⟨*ArithExpr*⟩ ⟨*RelOp*⟩ ⟨*ArithExpr*⟩
  | ⟨*EnumExpr*⟩ ⟨*RelOp*⟩ ⟨*EnumExpr*⟩
  | ⟨*BoolTransitionsExpr*⟩ ⟨*BoolBinOp*⟩ ⟨*BoolTransitionsExpr*⟩
  | '`neg`' ⟨*BoolTransitionsExpr*⟩
  | '`(`' ⟨*BoolTransitionsExpr*⟩ '`)`'

⟨*Expr*⟩ ::= ⟨*BoolExpr*⟩ | ⟨*ArithExpr*⟩ | ⟨*EnumExpr*⟩

⟨*BoolExpr*⟩ ::= ⟨*VarPrime*⟩ | ⟨*constant*⟩
  | ⟨*Var*⟩ '`in`' '`{`' ⟨*Vartype*⟩ '`}`'
  | ⟨*ArithExpr*⟩ '`==`' ⟨*ArithExpr*⟩ | ⟨*ArithExpr*⟩ '`/=`' ⟨*ArithExpr*⟩
  | ⟨*EnumExpr*⟩ '`==`' ⟨*EnumExpr*⟩ | ⟨*EnumExpr*⟩ '`/=`' ⟨*EnumExpr*⟩
  | ⟨*ArithExpr*⟩ ⟨*RelOp*⟩ ⟨*ArithExpr*⟩
  | ⟨*EnumExpr*⟩ ⟨*RelOp*⟩ ⟨*EnumExpr*⟩
  | ⟨*BoolExpr*⟩ ⟨*BoolBinOp*⟩ ⟨*BoolExpr*⟩
  | '`neg`' ⟨*BoolExpr*⟩
  | '`(`' ⟨*BoolExpr*⟩ '`)`'

⟨*ArithExpr*⟩ ::= ⟨*VarPrime*⟩ | ⟨*int*⟩
  | ⟨*ArithExpr*⟩ ⟨*ArithOp*⟩ ⟨*ArithExpr*⟩
  | '`(`' ⟨*ArithExpr*⟩ '`)`'

⟨*EnumExpr*⟩ ::= ⟨*VarPrime*⟩ | ⟨*constant*⟩
  | '`prev(`' ⟨*EnumExpr*⟩ '`)`'
  | '`next(`' ⟨*EnumExpr*⟩ '`)`'
  | '`(`' ⟨*EnumExpr*⟩ '`)`'

A.4 Specifications

Some syntactic restrictions apply in the following cases that are not captured in the grammar.

⟨*EnvSpec*⟩ ::= '`spec_obs`' '`=`' [⟨*String*⟩] ⟨*KF*⟩
  | '`spec_clk`' '`=`' [⟨*String*⟩] ⟨*KF*⟩
  | '`spec_spr`' '`=`' [⟨*String*⟩] ⟨*KF*⟩
  | '`spec_apr`' '`=`' [⟨*String*⟩] ⟨*KF*⟩
  | '`spec_obs_ctl`' '`=`' [⟨*String*⟩] ⟨*KF*⟩
  | '`spec_obs_ltl`' '`=`' [⟨*String*⟩] ⟨*KF*⟩

| ‘spec_obs_ctls’ ‘=’ [⟨*String*⟩] ⟨*KF*⟩
| ‘spec_obs_es’ ‘=’ [⟨*String*⟩] ⟨*KF*⟩
| ‘spec_obs_bmc_ctl’ ⟨*int*⟩ ‘=’ [⟨*String*⟩] ⟨*KF*⟩
| ‘spec_obs_bmc’ ⟨*int*⟩ ‘=’ [⟨*String*⟩] ⟨*KF*⟩
| ‘spec_clk_xn’ ‘=’ [⟨*String*⟩] ⟨*KF*⟩
| ‘spec_clk_ctl_nested’ ‘=’ [⟨*String*⟩] ⟨*KF*⟩
| ‘spec_clk_nested’ ‘=’ [⟨*String*⟩] ⟨*KF*⟩
| ‘spec_clk_g’ ‘=’ [⟨*String*⟩] ⟨*KF*⟩
| ‘spec_clk_bmc’ ⟨*int*⟩ ‘=’ [⟨*String*⟩] ⟨*KF*⟩
| ‘spec_spr_xn’ ‘=’ [⟨*String*⟩] ⟨*KF*⟩
| ‘spec_spr_ltl_nested’ ‘=’ [⟨*String*⟩] ⟨*KF*⟩
| ‘spec_spr_nested’ ‘=’ [⟨*String*⟩] ⟨*KF*⟩
| ‘spec_spr_g’ ‘=’ [⟨*String*⟩] ⟨*KF*⟩
| ‘spec_spr_bmc’ ⟨*int*⟩ ‘=’ [⟨*String*⟩] ⟨*KF*⟩
| ‘spec_apr_xn’ ‘=’ [⟨*String*⟩] ⟨*KF*⟩
| ‘spec_apr_g’ ‘=’ [⟨*String*⟩] ⟨*KF*⟩

KF is the CTL*+ mu + knowledge + probability specification language,

⟨*KF*⟩ ::= ⟨*BoolSpecExpr*⟩
| ⟨*KF*⟩ ⟨*BoolBinOp*⟩ ⟨*KF*⟩
| ‘neg’ ⟨*KF*⟩ | ‘(’ ⟨*KF*⟩ ‘)’
| ‘X’ ⟨*KF*⟩ | ‘X’ ⟨*int*⟩ ⟨*KF*⟩
| ‘F’ ⟨*KF*⟩ | ‘G’ ⟨*KF*⟩
| ⟨*KF*⟩ ‘U’ ⟨*KF*⟩
| ⟨*KF*⟩ ‘R’ ⟨*KF*⟩
| ‘A’ ⟨*KF*⟩
| ‘E’ ⟨*KF*⟩
| ⟨*muvar*⟩
| ‘gfp’ ⟨*muvar*⟩ ⟨*KFctl*⟩ | ‘lfp’ ⟨*muvar*⟩ ⟨*KFctl*⟩
| ‘Knows’ ⟨*constant*⟩ ⟨*KF*⟩
| ‘CK’ ⟨*AgentList*⟩ ⟨*KF*⟩ | ‘CK’ ⟨*KF*⟩
| ‘XK’ ⟨*constant*⟩ ⟨*KF*⟩
| ‘(’ ⟨*KFOps*⟩ ‘)’ ‘^’ ⟨*int*⟩ ⟨*KF*⟩

⟨*KFOps*⟩ ::= ⟨*KFOp*⟩
| ⟨*KFOp*⟩ ⟨*KFOps*⟩

⟨*KFOp*⟩ ::= ‘neg’
| ‘X’ | ‘X’ ⟨*int*⟩
| ‘F’ | ‘G’
| ‘A’ | ‘E’
| ‘gfp’ ⟨*muvar*⟩ | ‘lfp’ ⟨*muvar*⟩
| ‘Knows’ ⟨*constant*⟩
| ‘CK’ ⟨*AgentList*⟩ | ‘CK’
| ‘XK’ ⟨*constant*⟩

The basic propositions.

⟨*BoolSpecExpr*⟩ ::= ⟨*Var*⟩ | ⟨*constant*⟩ | ⟨*LocalObject*⟩ | ⟨*constant*⟩ ‘.’ ‘terminated’
| ⟨*Var*⟩ ‘in’ ‘{’ ⟨*Vartype*⟩ ‘}’
| ⟨*ArithSpecExpr*⟩ ‘==’ ⟨*ArithSpecExpr*⟩ | ⟨*ArithSpecExpr*⟩ ‘/=’ ⟨*ArithSpecExpr*⟩

| ⟨*EnumSpecExpr*⟩ '==' ⟨*EnumSpecExpr*⟩ | ⟨*EnumSpecExpr*⟩ '/=' ⟨*EnumSpecExpr*⟩
| ⟨*ProbArithSpecExpr*⟩ '==' ⟨*ProbArithSpecExpr*⟩ | ⟨*ProbArithSpecExpr*⟩ '/=' ⟨*ProbArithSpecExpr*⟩
| ⟨*ArithSpecExpr*⟩ ⟨*RelOp*⟩ ⟨*ArithSpecExpr*⟩
| ⟨*EnumSpecExpr*⟩ ⟨*RelOp*⟩ ⟨*EnumSpecExpr*⟩
| ⟨*ProbArithSpecExpr*⟩ ⟨*RelOp*⟩ ⟨*ProbArithSpecExpr*⟩
| ⟨*BoolSpecExpr*⟩ ⟨*BoolBinOp*⟩ ⟨*BoolSpecExpr*⟩
| 'neg' ⟨*BoolSpecExpr*⟩
| '(' ⟨*BoolSpecExpr*⟩ ')'

⟨*ArithSpecExpr*⟩ ::= ⟨*Var*⟩ | ⟨*int*⟩ | ⟨*LocalObject*⟩
| ⟨*ArithSpecExpr*⟩ ⟨*ArithOp*⟩ ⟨*ArithSpecExpr*⟩
| '(' ⟨*ArithSpecExpr*⟩ ')'

⟨*EnumSpecExpr*⟩ ::= ⟨*Var*⟩ | ⟨*constant*⟩ | ⟨*LocalObject*⟩
| 'prev(' ⟨*EnumSpecExpr*⟩ ')'
| 'next(' ⟨*EnumSpecExpr*⟩ ')'
| '(' ⟨*EnumSpecExpr*⟩ ')'

⟨*ProbArithSpecExpr*⟩ ::= ⟨*Real*⟩ | ⟨*ProbFormula*⟩ | ⟨*ProbArithSpecExpr*⟩ ⟨*ProbArithOp*⟩ ⟨*ProbArithSpecExpr*⟩ | (⟨*ProbArithSpecExpr*⟩)

⟨*ProbFormula*⟩ ::= 'Prob' ⟨*constant*⟩ ⟨*KF*⟩ | 'Prior' ⟨*constant*⟩ ⟨*KF*⟩ | 'Prob' ⟨*constant*⟩ '(' ⟨*KF*⟩ | ⟨*KF*⟩ ')' |
'Prior' ⟨*constant*⟩ '(' ⟨*KF*⟩ | ⟨*KF*⟩ ')'

A.5 Variable Types

⟨*Vartype*⟩ ::= ⟨*Enumeration*⟩ | ⟨*NumericRange*⟩

⟨*Enumeration*⟩ ::= ⟨*Constant*⟩ (',' ⟨*Constant*⟩)*

⟨*NumericRange*⟩ ::= ⟨*int*⟩ '..' ⟨*int*⟩

⟨*VarDec*⟩ ::= ⟨*varid*⟩ ':' ⟨*Type*⟩

⟨*Type*⟩ ::= [⟨*TypeAttr*⟩] ⟨*TypeDesc*⟩

⟨*TypeAttr*⟩ ::= 'observable'

⟨*TypeDesc*⟩ ::= ⟨*typename*⟩ | ⟨*typename*⟩ '[' [⟨*int*⟩] ']'

A.6 Common Productions

⟨*Var*⟩ ::= ⟨*varid*⟩ | ⟨*varid*⟩ '[' ('self' | ⟨*int*⟩) ']'

⟨*VarPrime*⟩ ::= ⟨*Var*⟩''' | ⟨*Var*⟩

⟨*VarList*⟩ ::= ⟨*Var*⟩ (',' ⟨*Var*⟩)*

⟨*LocalObject*⟩ ::= ⟨*constant*⟩ '.' ⟨*constant*⟩

⟨*AgentList*⟩ ::= '{' ⟨*constant*⟩ (',' ⟨*constant*⟩)* '}'

⟨*BoolBinOp*⟩ ::= '/\' | '\/' | '=>' | '<=>' | 'xor'

⟨*ArithOp*⟩ ::= ‘+’ | ‘−’

⟨*ProbArithOp*⟩ ::= ‘+’ | ‘−’ | ‘⋆’

⟨*RelOp*⟩ ::= ‘<’ | ‘<=’ | ‘>’ | ‘>=’

⟨*Constant*⟩ ::= ⟨*int*⟩ | ⟨*constant*⟩

⟨*String*⟩ ::= ‘"’ ⟨*char*⟩* ‘"’

⟨*Probability*⟩ ::= ⟨*Real*⟩

⟨*Real*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩ ‘.’ ⟨*digit*⟩+

A.7 Operator Priorities

The operators are declared with the following priorities (from low to high; operators on the same line have equal priority) and associativity

```
left '=>'
left '<=>'
left '/\'  '\/' 'xor'
left 'U' 'R'
right 'Knows' 'CK' 'XK' 'F' 'G' 'X' 'E' 'A' 'EX' 'AX' 'EF' 'AF' 'EG' 'AG' 'Prob' 'Prior'
right 'neg'
nonassoc "==" "/=" '>' '<' ">=" "<="
left '+' '-'
left '*'
left 'next' 'prev'
```

Appendix B

Operational Semantics

The motivation for providing a detailed operational semantics is to make the timing model precise. We do not attempt to capture all features of the language here. (In particular, we do not incorporate treatment of user-defined labels or the read/write actions.)

There are two major constraints on the design of the language:

1. The agents need to have control over which action they emit at all times (see the Robot example in Section 6.1 for further details).

2. Automata must be naturally expressible.

Also, arbitrary nesting of constructs and a simple, regular timing model are very desirable.

In the following description, let:

- *Var* be the type of variables.

- *Val* be the type of values. (We assume that local variables of agent *Agt* are represented either by the environment variable to which they are aliased, or written in the form *Agt.var* to distinguish them from variables of other agents with the same name.)

- $\Sigma = Var \rightarrow Val$ be the type of valuation functions (giving values to all variables), and $\rho :: \Sigma$.

- $\langle\, elt_1, \ldots, elt_n \,\rangle$ be sequences of arbitrary objects.

- $[x \mapsto val] :: Var \rightarrow Val \rightarrow \Sigma \rightarrow \Sigma$ be a state transfomer:

$$([x \mapsto val]\ \rho)\ y = \begin{cases} val & : \quad x == y \\ \rho\ y & : \quad \text{otherwise} \end{cases}$$

- $f \circ g$ be function composition: $(f \circ g)\ x \equiv f(g(x))$. A summation-style $\bigcirc :: [a \rightarrow a] \rightarrow a \rightarrow a$ is also used, where $\bigcirc\ \langle\rangle = \mathbf{id}$ and $\bigcirc\ \langle\, head \mid tail \,\rangle = (\bigcirc\, tail) \circ head$, where $\langle\, head \mid tail \,\rangle$ is a list of composable functions.

In order to implement the select/resolve model (Chapter 1), the overall state transition relation is split into two sets of relations – agent-local and environment. Both rely on another relation to give meaning to expressions.

Expression Semantics

$\rho : Expr \hookrightarrow Val$ is the expression evaluation function, described in Section B.2.

Transition relations

For each agent A, two relations are defined:

- $\rho : ProgramText \rightsquigarrow_A ProgramText'$ are agent-internal transitions, which become stuck upon encountering a manifest action.
- $\rho : ProgramText \rightarrow_A (Action, \langle Assignment \rangle, ProgramText')$ is the *selection* phase (determines what action the agent performs in this time step).

The idea is that each agent executes as much $ProgramText$ as possible until it is manifest which action is to be performed (at which point the $\rightsquigarrow_A$ relation gets stuck). The $\rightarrow_A$ relation takes care of detaching this action and tracking where the agent is up to in its protocol.

Additionally, we define a global relation $\rightarrow_n$ to capture the synchronous, lock-step aggregate of all the $\rightarrow_A$ agent relations.

The definitions of these relations are given in Section B.3.

Environment Transitions

- $[\![\cdot]\!]_{(\rho,\mathcal{A})}$ generates a state transformer from the **transitions** clause, given the current state and the actions $\mathcal{A}$ the agents have performed.
- $\rho : \langle ProgramText_1, ..., ProgramText_n \rangle \Rightarrow \rho' : \langle ProgramText'_1, ..., ProgramText'_n \rangle$ is the overall single-step state-transformation relation.

The definitions of these relations are given in Section B.4.

B.1 Pre-processing

A pre-processing pass is used to expand derived forms and normalise the programs.

1. Append **while True do skip** to the original program P. This ensures all programs generate infinite runs.
2. Make all variable names unique by qualifying local variables with agent names.
3. Expand all definitions (Section 2.6.4).
4. Eliminate derived forms using the rules in Sections B.2 and **??**.
5. For all **if ... fi** statements: if present, replace an **otherwise** $\rightarrow C$ branch with $\bigwedge_i \neg cond_i \rightarrow C$, or add a $\bigwedge_i \neg cond_i \rightarrow$ **skip** branch if it is absent.

   (Note this expansion also applies to a **transitions** clause, if present.)
6. For all **do ... od** statements: if the **break** branch is absent, add **break** $\rightarrow$ **skip**.
7. Add internal labels to each action statement. Note these are distinct from the source-language labels, and are used by the BDD encoding to track where the $\rightsquigarrow_A$ relation becomes stuck.

B.2 Expression Semantics

$[\![\cdot]\!]_\rho : Expr \rightarrow Val$ is an expression evaluation function, where ρ is the system state when evaluating the expression. All expressions should be well-typed such that all operands are in a same type.

B.2.1 Basic Values

1. $[\![var]\!]_\rho \equiv \rho(var)$ where *var* is a variable
2. $[\![val]\!]_\rho \equiv val$, where *val* is a constant.
3. $[\![var.constant]\!]_\rho \equiv True$, if action $\ll constant \gg$ is activated by agent *var*, and $\equiv False$, otherwise.

B.2.2 Operations for Boolean Type

Let expressions f and g be boolean expressions. We have the following operations.

1. Two basic boolean operations:
$$[\![f \wedge g]\!]_\rho \equiv [\![f]\!]_\rho \wedge [\![g]\!]_\rho$$
$$[\![\neg f]\!]_\rho \equiv \neg [\![f]\!]_\rho$$
2. The standard boolean algebraic identities hold:
$$[\![f \vee g]\!]_\rho \equiv [\![\neg(\neg f \wedge \neg g)]\!]_\rho$$
$$[\![f \rightarrow g]\!]_\rho \equiv [\![\neg f \vee g]\!]_\rho$$
$$[\![f \leftrightarrow g]\!]_\rho \equiv [\![(f \rightarrow g) \wedge (g \rightarrow f)]\!]_\rho$$
$$[\![f \text{ xor } g]\!]_\rho \equiv [\![(\neg x \wedge y) \vee (x \wedge \neg y)]\!]_\rho$$

B.2.3 Truncated Arithmetic Operations for Numeric Types

For truncated arithmetic operators + and −, the type of their operands can be any numeric type, and the two operands should be in the same type. Operations $[\![e_1 + e_2]\!]_\rho$ and $[\![e_1 - e_2]\!]_\rho$ are defined as follows, where e_1 and e_2 are expressions with numeric types T.

1. $[\![e_1 + e_2]\!]_\rho = \begin{cases} max(T) & \text{if } [\![e_1]\!]_\rho + [\![e_2]\!]_\rho \geq max(T) \\ e_1 + e_2 & \text{otherwise} \end{cases}$
2. $[\![e_1 - e_2]\!]_\rho = \begin{cases} min(T) & \text{if } [\![e_1]\!]_\rho - [\![e_2]\!]_\rho \leq min(T) \\ e_1 - e_2 & \text{otherwise} \end{cases}$

B.2.4 Comparison Operations for Numeric Types

We have two basic operators == and <. For two expressions with numeric type T, we have

1. $[\![e_1 == e_2]\!]_\rho = \begin{cases} True & \text{if } [\![e_1]\!]_\rho = [\![e_2]\!]_\rho \\ False & \text{otherwise} \end{cases}$
2. $[\![e_1 < e_2]\!]_\rho = \begin{cases} True & \text{if } [\![e_1]\!]_\rho < [\![e_2]\!]_\rho \\ False & \text{otherwise} \end{cases}$

Then, we have derived operations

1. $[\![e_1 > e_2]\!]_\rho \equiv [\![e_2 < e_1]\!]_\rho$
2. $[\![e_1 >= e_2]\!]_\rho \equiv [\![(e_2 < e_1) \vee (e_1 == e_2)]\!]_\rho$
3. $[\![e_1 <= e_2]\!]_\rho \equiv [\![(e_1 < e_2) \vee (e_1 == e_2)]\!]_\rho$
4. $[\![e_1 / = e_2]\!]_\rho = [\![\neg(e_1 == e_2)]\!]_\rho$

B.2.5 Operations for Enumerated Types

Suppose that expressions $e_1, e_2 : T$ are in the same type $T : \{const_0, ..., const_{m-1}\}$.

1. $[\![next(e_1)]\!]_\rho \equiv const_{i+1 \ mod \ m}$ if $\rho(e_1) = const_i$
2. $[\![prev(e_1)]\!]_\rho \equiv const_{i-1 \ mod \ m}$ if $\rho(e_1) = const_i$
3. $[\![e_1 < e_2]\!]_\rho \equiv i < j$ with $\rho(e_1) = const_i$ and $\rho(e_2) = const_j$.
4. $[\![e_1 == e_2]\!]_\rho \equiv i = j$ with $\rho(e_1) = const_i$ and $\rho(e_2) = const_j$.

Other derived comparison operations based on == and < are defined the same with numeric types.

B.2.6 Set Operation

The set operation is interpreted as a syntax sugar. Let $e, e_1, ..., e_n$ be in the same numeric type or enumerated type.

1. $[\![e \in \{e_1, ..., e_n\}]\!]_\rho \equiv [\![(e == e_1) \vee ... \vee (e == e_n)]\!]_\rho$

B.3 Agent Protocol Semantics

The definition of $\rho : ProgramText \rightsquigarrow_A RestOfProgramText$:

$$\textbf{Sequencing:} \quad \frac{\rho : C_1 \rightsquigarrow_A C_1'}{\rho : C_1; C_2 \rightsquigarrow_A C_1'; C_2}$$

If: For each alternative i:

$$\frac{\rho : cond_i \hookrightarrow True}{\rho : \textbf{if} \ ... \ cond_i \ \rightarrow C_i \ ... \ \textbf{fi} \rightsquigarrow_A C_i}$$

Do: For each alternative i:

$$\frac{\rho : cond_i \hookrightarrow True}{\rho : \textbf{do} \ ... \ cond_i \ \rightarrow C_i \ ... \ \textbf{od} \rightsquigarrow_A C_i; \textbf{do} \ ... \ \textbf{od}}$$

The $\rho : ProgramText \rightsquigarrow_A RestOfProgramText$ relation is the reflexive, transitive closure of the above definitions. This relation is purposefully non-deterministic.

The definition of $\rho : ProgramText \rightarrow_A (Action, \langle Assignment \rangle, ProgramText')$:

$$\frac{\rho : ProgramText \rightsquigarrow_A < Action | \langle Assignment \rangle >; ProgramText'}{\rho : ProgramText \rightarrow_A (Action, \langle Assignment \rangle, ProgramText')}$$

We expect $\rho : C \rightsquigarrow_A C'$ to terminate for all C, and the overall agent relation

$$\rho : ProgramText \rightarrow_A (Action, \langle Assignment \rangle, ProgramText')$$

to not get stuck.

Define:

$$\rho : (ProgramText_1, ..., ProgramText_n) \rightarrow_n$$
$$((Action_1, \langle Assignment \rangle_1, ProgramText_1'), ..., (Action_n, \langle Assignment \rangle_n, ProgramText_n'))$$

to be the composite of the individual transitions

$$\rho : ProgramText_A \rightarrow_A (Action_A, \langle Assignment \rangle_A, ProgramText_A')$$

for each agent $A \in \{1...n\}$.

Note on why 'do' has a 'break' branch

As noted in Section 2.4, the protocol language is non-standard in that it allows the programmer to specify what happens immediately after all guards evaluate to false in a **do** construct. (In Dijkstra's version [8], there is always an implicit **skip** before the programmer regains control.) The reason we require continuous control was set out in Section 6.1.

A superficially plausible alternative semantics is to try to determine an action which will be enabled once the **do** loop terminates, and execute that. Unfortunately, this leads to problems. Take, for example, the following program:

```
do True ->
  do False -> <Action>
  od
od
```

under this semantics. In this case, there simply is no action that can be executed between the evaluation of the two guards, and so the $\rightsquigarrow_A$ relation must be non-terminating.

Given our constraint that guards should be instantaneously evaluated, the most natural thing to do is to add the **break** branch. An alternative solution would be to ban nested looping constructs, or require that the first statement in the body of a **do** statement always produce an action (i.e. be a non-looping construct). This latter approach is standard in process algebras, and is termed *guarded recursion* [1].

B.4 Environment Transitions-Clause Semantics

The manner in which the environment's state is updated is expressed as a program in the non-looping subset of the language used to represent protocols. The most significant departure is that guards can mention agent-qualified actions.

As mentioned in Section 2.5.2, the system currently provides shared-variable **read()** and **write()** actions as primitives, distinct from the mechanism described here. A semantics for these primitives is omitted.

The definition of $[\![\cdot]\!]_{(\rho,\mathcal{A})} :: Statement \to (State, \{Action\}) \to \Sigma \to \Sigma$, giving meaning to a **transitions** clause in terms of state transformers:

$$\frac{\rho' = [\![C_1]\!]_{(\rho,\mathcal{A})}\,\rho}{[\![C_1; C_2]\!]_{(\rho,\mathcal{A})} = [\![C_2]\!]_{(\rho',\mathcal{A})} \circ [\![C_1]\!]_{(\rho,\mathcal{A})}}$$

For each alternative i:

$$\frac{(\rho, \mathcal{A}) \models cond_i}{[\![\textbf{if } ...\ cond_i\ \to C_i\ ...\ \textbf{fi}]\!]_{(\rho,\mathcal{A})} = [\![C_i]\!]_{(\rho,\mathcal{A})}}$$

$$[\![\textbf{skip}]\!]_{(\rho,\mathcal{A})} = \textbf{id}_{\Sigma\to\Sigma}$$

$$\frac{\rho : expr \hookrightarrow val}{[\![x := expr]\!]_{(\rho,\mathcal{A})} = [x \mapsto val]}$$

where $\textbf{id}_{\Sigma\to\Sigma}$ is the identity function for the specified type.

This function is purposefully non-deterministic.

Conditions are evaluated with respect to a set of actions $\mathcal{A}$ and a state ρ. The base cases are as follows:

$$\frac{Action \in \mathcal{A}}{(\rho, \mathcal{A}) \models Action}$$

$$\frac{\rho\ Var = True}{(\rho, \mathcal{A}) \models Var}$$

and the recursive cases (for the logical connectives) are similar to those in Section B.2.

The definition of $\rho : (ProgramText_1, ..., ProgramText_n) \Rightarrow \rho' : (ProgramText'_1, ..., ProgramText'_n)$, taking states and program counters in one state to the next:

$$\rho : (ProgramText_1, ..., ProgramText_n) \rightarrow_n$$
$$((Action_1, \langle Assignment \rangle_1, ProgramText'_1), ..., (Action_n, \langle Assignment \rangle_n, ProgramText'_n))$$
$$st_{env} = [\![TransitionsClause]\!]_{(\rho,\mathcal{A})}$$
$$\text{for each agent } i\text{: } st_i = [\![Assignment_{i_1}]\!]_{(\rho,\mathcal{A})} \circ ... \circ [\![Assignment_{i_k}]\!]_{(\rho,\mathcal{A})}.$$
$$\rho' = (st_{env} \circ \bigcirc \langle st_1, \ldots, st_n \rangle)\, \rho$$

$$\rho : (ProgramText_1, ..., ProgramText_n) \Rightarrow \rho' : (ProgramText'_1, ..., ProgramText'_n)$$

Notes:

- The order of composing the Environment's st_{env} and Agents' state transformers st_i doesn't matter as their domains are disjoint.
- Where there are several assignments in a single action statement, the variables in their right-hand-sides refer to the current state. For example, in a state where $\{x \mapsto True,\ y \mapsto False\}$, the action $<< NilAction \mid x := y;\ y := x >>$ gives rise to a state where $\{x \mapsto False,\ y \mapsto True\}$.

B.4.1 Runs

A *run* is a sequence of global states ρ_i conformant with the overall one-step transition relation $\Rightarrow$:

- The initial state of the system is $\rho^0 : (ProgramText_1, ..., ProgramText_n)$, where ρ^0 must satisfy all local and global initial constraints, and $ProgramText_i$ is the entire program for agent i.
- For each $i \geq 0$, we have (with superscripts indicating position in the run):

$$\rho^i : (ProgramText^i_1, ..., ProgramText^i_n) \Rightarrow \rho^{i+1} : (ProgramText^{i+1}_1, ..., ProgramText^{i+1}_n)$$