Chapter 9

Using the Maude Term Rewriting Language for Agent Development with Formal Foundations

M.B. van Riemsdijk, L. Aştefănoaei, and F.S. de Boer

Abstract We advocate the use of the Maude term rewriting language and its supporting tools for prototyping, model-checking, and testing agent programming languages and agent programs. One of the main advantages of Maude is that it provides a single framework in which the use of a wide range of formal methods is facilitated. We use the agent programming language BUpL (Belief Update programming Language) for illustration.

M.B. van Riemsdijk
Delft University of Technology, The Netherlands e-mail: m.b.vanriemsdijk@tudelft.nl

L. Astefanoaei
CWI (Centrum voor Wiskunde en Informatica), The Netherlands e-mail: L.Astefanoaei@cwi.nl

F. de Boer
CWI (Centrum voor Wiskunde en Informatica), The Netherlands e-mail: F.S.de.Boer@cwi.nl

M. Dastani et al. (eds.), *Specification and Verification of Multi-agent Systems*,
DOI 10.1007/978-1-4419-6984-2_9,

9.1 Introduction

An important line of research in the agent systems field is research on *agent programming languages* [64]. The guiding idea behind these languages is that programming languages based on agent-specific concepts such as beliefs, goals, and plans facilitate the programming of agents.

Several agent programming languages have been developed with an emphasis on the use of *formal methods*. In particular, structural operational semantics [340] is often used for formally defining the semantics of the languages. The semantics is used as a basis for prototyping and implementing the languages, and for verification. Several tools and techniques can be used for implementation and verification, such as Java for writing an interpreter and IDE, and the Java PathFinder[1] or SPIN [236] model-checkers for verification [61].

In this chapter, we advocate the use of the *Maude* language [104] and its supporting tools for prototyping, verifying, and testing agent programming languages and agent programs. One of the main advantages of Maude is that it provides a single framework in which the use of a wide range of formal methods is facilitated. Maude is a high-performance reflective language and system supporting *equational and rewriting logic specification and programming*. The language has been shown to be suitable both as a logical framework in which many other logics can be represented, and as a semantic framework through which programming languages with an operational semantics can be implemented in a rigorous way [301]. Maude comes with an LTL model-checker [155], which allows for verification. Moreover, Maude facilitates the specification of strategies for controlling the application of rewrite rules [154].

We will demonstrate how these features of Maude can specifically be applied for developing *agent* programming languages and programs based on solid formal foundations. We use the agent programming language BUpL (Belief Update programming Language) [19] for illustration. BUpL is a simple language that resembles the first version of 3APL [223].

The outline of this chapter is as follows. We present BUpL in Section 9.2, and then use BUpL to illustrate how Maude can be used for prototyping (Section 9.3), model-checking (Section 9.4), and testing (Section 9.5). We conclude in Section 9.6. The complete Maude source code of the implementations discussed in this chapter can be downloaded from `http://homepages.cwi.nl/~astefano/agents/bupl-strategies.php`.

[1] `http://javapathfinder.sourceforge.net/`

9.2 The BUpL Language

In this section, we briefly present the syntax and semantics of BUpL for ease of reference. We refer to Chapter [2] for more details and explanation. A BUpL agent has an initial belief base and an initial plan. A belief base is a collection of ground (first-order) atomic formulas which we refer to as beliefs. The agent is supposed to execute its initial plan, which is a sequential composition and/or a non-deterministic choice of actions or composed plans. The semantics of actions is defined using preconditions and effects (postconditions). An action can be executed if the precondition of the action matches the belief base. The belief base is then updated by adding or removing the elements specified in the effect. If the precondition does not match the belief base, we say the execution of the action (or the plan of which it is a part) fails. In this case repair rules can be applied, and this results in replacing the plan that failed by another.

9.2.1 Syntax

BUpL is based on a simple logical language $\mathcal{L}$ with typical element φ, which is defined as follows. $\mathcal{F}$ and $\mathcal{P}red$ are infinite sets of function, respectively predicate symbols, with typical element f, respectively P. Variables are denoted by the symbol x. As usual, a term t is either a variable or a function symbol with terms as parameters. Predicate symbols with terms as parameters form the *atoms* of $\mathcal{L}$, and atoms or negated atoms are called *literals*, denoted as l. Atoms are also called *positive literals* and negated atoms are called *negative literals*. The negation of a negative literal yields its positive variant. Nullary functions form the constants of the language. $\mathcal{L}$ does not contain quantifiers to bind variables. A formula or term without variables is called *ground*. Formulas from $\mathcal{L}$ are in disjunctive normal form (DNF), i.e., they consist of disjunctions of conjunctions (denoted as c) of literals. A *belief base* $\mathcal{B}$ is a set of ground atoms from $\mathcal{L}$.

$$
\begin{aligned}
t &::= x \mid f(t,\ldots,t) \\
l &::= P(t,\ldots,t) \mid \neg P(t,\ldots,t) \\
c &::= l \mid c \wedge c \\
\varphi &::= c \mid c \vee c
\end{aligned}
$$

Basic actions are defined as functions $a(x_1,\ldots,x_n) =_{def} (\varphi,\xi)$, where $\varphi \in \mathcal{L}$ is a formula which we call *precondition*, and ξ is a set of literals from $\mathcal{L}$ which we call *effect*. The following inclusions are required:

$$Var(\xi) \subseteq Var(\varphi) = \{x_1,\ldots,x_n\}.^{2}$$

[2] These inclusions thus specify that the variables occurring in the precondition and the effect also have to be in the parameter of the action. This has to do with the way actions are implemented

We use the symbol $\mathcal{A}$ for the set of basic actions. We use *Act* to refer to the set of basic action *names*, with typical element a. We say that a function call $a(t_1,\ldots,t_n)$ is a *basic action term*, and will sometimes denote it as α.

Plans, typically denoted as p, are defined as follows, where Π is a set of plan names with typical element π and $a(t,\ldots,t)$ is a basic action term:

$$p ::= a(t,\ldots,t) \mid \pi(t,\ldots,t) \mid a(t,\ldots,t); p \mid p + p.$$

Here, ';' is the *sequential composition* operator and '+' is the *choice* operator, with a lower priority than ';'. The construct $\pi(t,\ldots,t)$ is called *abstract plan*. Abstract plans should be understood as procedure calls in imperative languages, with corresponding procedures of the form $\pi(x_1,\ldots,x_n) = p$. The set of procedures is denoted as $\mathcal{P}$.

Repair rules have the form $\varphi \leftarrow p$, and can be applied if a plan has failed and φ matches the belief base. Then the failed plan is substituted by p. The set of repair rules is denoted as $\mathcal{R}$.

A BUpL agent is a tuple $(\mathcal{B}_0, p_0, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where $\mathcal{B}_0$ is the initial belief base, p_0 is the initial plan, $\mathcal{A}$ are the actions, $\mathcal{P}$ are the procedures, and $\mathcal{R}$ are the repair rules. The initial belief base and plan form the initial *mental state* of the agent.

To illustrate the above syntax, we take as an example a BUpL agent that solves the *tower of blocks* problem, i.e., the agent has to build towers of blocks. We represent blocks by natural numbers. Assume the following initial arrangement of three blocks 1,2, and 3: blocks 1 and 2 are on the table (denoted as block 0), and 3 is on top of 1. The agent has to rearrange them such that they form the tower 321 (1 is on 0, 2 on top of 1 and 3 on top of 2). The only action an agent can execute is $move(x,y,z)$ to move a block x from another block y onto z, if both x and z are clear (i.e., have no blocks on top of them). Blocks can always be moved onto the table, i.e., the table is always clear.

$$\mathcal{B}_0 = \{\, on(3,1), on(1,0), on(2,0), clear(2), clear(3), clear(0) \,\}$$

$$p_0 = build$$

$$\mathcal{A} = \{\, move(x,y,z) = (on(x,y) \wedge clear(x) \wedge clear(z), \{on(x,z), \neg on(x,y), \neg clear(z), clear(0)\}\ )\ \}$$

$$\mathcal{P} = \{build = move(2,0,1); move(3,0,2)\}$$

$$\mathcal{R} = \{\, on(x,y) \leftarrow move(x,y,0); build \,\}$$

Fig. 9.1 A BUpL Blocks World Agent

The BUpL agent from Figure 9.1 is modeled such that it illustrates the use of repair rules: we explicitly mimic a failure by intentionally telling the agent to move

in Maude. It may be relaxed, but for simplicity, we do not do it here. In other languages such as 2APL [122], variables may occur in the precondition that are not in the parameters of the action. These variables are instantiated when matching the precondition against the belief base.

block 2 onto 1. This is not possible, since block 3 is already on top of 1. Similar scenarios can easily arise in multi-agent systems: imagine that initially 3 is on the table, and the agent decides to move 2 onto 1; imagine also that another agent comes and moves 3 on top of 1, thus moving 2 onto 1 will fail. The failure is handled by the repair rule $on(x,y) \leftarrow move(x,y,0); p$. Choosing $[x/3][y/1]$ as a substitution, this enables the agent to move block 3 onto the table and then the initial plan can be restarted.

9.2.2 Semantics

First, we define the satisfaction relation of formulas φ with respect to a belief base $\mathcal{B}$. For this, we consider the usual notion of *substitution* as a set that defines how to replace variables with terms. A substitution is denoted by $[x_0/t_0]\ldots[x_n/t_n]$, which expresses that x_i is replaced by t_i for $0 \leq i \leq n$. A substitution θ can be applied to a formula φ, written as $\varphi\theta$, which yields the formula φ in which variables are simultaneously replaced by terms as specified by θ. If θ and θ' are substitutions and φ is a formula, we use $\varphi\theta\theta'$ to denote $(\varphi\theta)\theta'$. A *ground substitution* is a substitution in which all t_i are ground terms. In the sequel, we will assume all substitutions to be ground, unless indicated otherwise. For technical convenience, we assume any conjunction c has the form $l_0 \wedge \ldots \wedge l_m \wedge l_{m+1} \wedge \ldots \wedge l_n$ where $l_0,\ldots,l_m$ are positive literals and $l_{m+1},\ldots,l_n$ are negative literals. We use $Var(\varphi)$ to denote the variables occurring in φ and $dom(\theta)$ to denote the set of variables forming the domain of θ. The satisfaction relation of a formula φ with respect to a belief base is defined relative to a substitution θ, denoted as $\models_\theta$, and is defined as follows, where $\models$ is the usual entailment relation for ground formulas:

$$
\begin{array}{ll}
\mathcal{B} \models_\emptyset P(t_0,\ldots,t_n) & \text{iff } P(t_0,\ldots,t_n) \text{ is ground and } \mathcal{B} \models P(t_0,\ldots,t_n) \\
\mathcal{B} \models_\emptyset c & \text{iff } c \text{ is ground and } \mathcal{B} \models c \\
\mathcal{B} \models_\theta P(t_0,\ldots,t_n) & \text{iff } \mathcal{B} \models_\emptyset P(t_0,\ldots,t_n)\theta \text{ and } Var(P(t_0,\ldots,t_n)) = dom(\theta) \\
\mathcal{B} \models_\theta c & \text{iff } \mathcal{B} \models_\emptyset (l_0 \wedge \ldots \wedge l_m)\theta \text{ and } Var(l_0 \wedge \ldots \wedge l_m) = dom(\theta) \text{ and} \\
 & \quad \neg\exists\theta' : (\mathcal{B} \models_{\theta'} \neg l_{m+1}\theta \text{ and } Var(l_{m+1}\theta) = dom(\theta')) \\
 & \quad \ldots \\
 & \quad \neg\exists\theta' : (\mathcal{B} \models_{\theta'} \neg l_n\theta \text{ and } Var(l_n\theta) = dom(\theta')) \\
\mathcal{B} \models_\theta c \vee c' & \text{iff } \mathcal{B} \models_\theta c \text{ or } \mathcal{B} \models_\theta c'
\end{array}
$$

We use $Sols(\mathcal{B},\varphi) = \{\theta \mid \mathcal{B} \models_\theta \varphi\}$ to denote the set of all substitutions for which φ follows from the belief base. Note that if $Sols(\mathcal{B},\varphi) = \emptyset$, φ does not follow from $\mathcal{B}$. If φ follows from $\mathcal{B}$ under the empty substitution, we have $Sols(\mathcal{B},\varphi) = \{\emptyset\}$.

We now continue to define what it means to execute an action. Let $a(x_1,\ldots,x_n) =_{def} (\varphi,\xi) \in \mathcal{A}$ be a basic action definition. A function call $a(t_1,\ldots,t_n)$ yields the pair $(\varphi,\xi)\theta$ where $\theta = [x_1/t_1]\ldots[x_n/t_n]$, which does not have to be ground. Let $a(t_1,\ldots,t_n) = (\varphi',\xi')$ be the result of applying the function to the terms $t_1,\ldots,t_n$, and let $\theta' \in Sols(\mathcal{B},\varphi')$. Then the effect on $\mathcal{B}$ of executing $a(t_1,\ldots,t_n)$ is that $\mathcal{B}$ is

updated by adding or removing (ground) atoms occurring in the set $\xi'\theta'$:

$$\begin{aligned} \mathcal{B} \uplus l\theta' &= \mathcal{B} \cup l\theta' \quad \text{if } l \in \xi \text{ and } l \text{ a positive literal} \\ \mathcal{B} \uplus l\theta' &= \mathcal{B} \setminus \neg l\theta' \quad \text{if } l \in \xi \text{ and } l \text{ a negative literal} \end{aligned}$$

We write $\mathcal{B} \uplus \xi'\theta'$ to represent the result of updating $\mathcal{B}$ with the instantiated effect of an action $\xi'\theta'$, which performs the update operation as specified above on $\mathcal{B}$ for each literal in ξ'. This update is guaranteed to yield a consistent belief base since we add only positive literals.

The operational semantics of a language is usually defined in terms of labeled transition systems [340]. A *labeled transition system* (LTS) is a tuple $(\Sigma, s_0, L, \rightarrow)$, where Σ is a set of states, s_0 is an initial state, L is a set of labels, and $\rightarrow \subseteq \Sigma \times L \times \Sigma$ describes all possible transitions between states, and associates a label to the transition. The notation $s \xrightarrow{\alpha} s'$ expresses that $(s, \alpha, s') \in \rightarrow$, and it intuitively means that "s becomes s' by performing action α". Invisible transitions are denoted by the label τ.

The operational semantics for a BUpL agent is defined as follows. Let $(\mathcal{B}_0, p_0, \mathcal{A}, \mathcal{P}, \mathcal{R})$ be a BUpL agent. Then the associated LTS is $(\Sigma, (\mathcal{B}_0, p_0), L, \rightarrow)$, where:

- Σ is the set of states, which are BUpL mental states
- $(\mathcal{B}_0, p_0)$ is the initial state
- L is a set of labels, which are either ground basic action terms or τ
- $\rightarrow$ is the transition relation induced by the transition rules given in Table 9.1.

$$\frac{a(x_1,\ldots,x_n) =_{def} (\varphi,\xi) \in \mathcal{A} \qquad a(t_1,\ldots,t_n) = (\varphi',\xi') \qquad \theta \in Sols(\mathcal{B},\varphi')}{(\mathcal{B}, a(t_1,\ldots,t_n); p') \xrightarrow{a(t_1,\ldots,t_n)\theta} (\mathcal{B} \uplus \xi'\theta, p'\theta)} \ (act)$$

$$\frac{(\mathcal{B}, p_i) \xrightarrow{\mu} (\mathcal{B}', p')}{(\mathcal{B}, (p_1 + p_2)) \xrightarrow{\mu} (\mathcal{B}', p')} \ (sum_i)$$

$$\frac{(\mathcal{B}, \alpha; p) \not\downarrow \alpha\theta' \rightarrow \qquad \varphi \leftarrow p' \in \mathcal{R} \qquad \theta \in Sols(\mathcal{B},\varphi)}{(\mathcal{B}, p) \xrightarrow{\tau} (\mathcal{B}, p'\theta)} \ (fail)$$

$$\frac{\pi(x_1,\ldots,x_n) = p \in \mathcal{P}}{(\mathcal{B}, \pi(t_1,\ldots,t_n)) \xrightarrow{\tau} (\mathcal{B}, p(t_1,\ldots,t_n))} \ (\pi)$$

Table 9.1 BUpL transition rules

In rule (sum_i), p_i is either p_1 or p_2, and μ can be either a ground basic action term or a silent transition τ, in which case $\mathcal{B}' = \mathcal{B}$, and p' is a valid repair plan. In rule (π), $p(t_1,\ldots,t_n)$ stands for $p[x_1/t_1]\ldots[x_n/t_n]$.

9.3 Prototyping

In this section, we describe how the operational semantics of agent programming languages can be implemented in Maude. The main advantage of using Maude for this is that the translation of operational semantics into Maude is direct [390], ensuring a *faithful implementation*. Because of this, it is relatively easy to experiment with different kinds of semantics, making Maude suitable for rapid *prototyping* of agent programming languages. This is also facilitated by the fact that Maude supports user-definable syntax, offering prototype parsers for free. Another advantage of using Maude for prototyping specifically *logic-based* agent programming languages is that Maude has been shown to be suitable not only as a semantic framework, but also as a logical framework in which many other logics can be represented.

We use BUpL to illustrate the implementation of agent programming languages in Maude. BUpL has beliefs and plan revision features, but no goals. We refer to [370] for a description of the Maude implementation of a similar agent programming language that does have goals. While the language of [370] is based on propositional logic, BUpL allows the use of variables, facilitating experimentation with more realistic programming examples. An implementation of the agent programming language AgentSpeak in Maude is briefly described in [170].

9.3.1 Introduction to Maude

A rewriting logic specification or rewrite theory is a tuple $\langle \Sigma, E, R \rangle$, where Σ is a signature consisting of types and function symbols, E is a set of equations and R is a set of rewrite rules. The signature describes the *terms* that form the state of the system. These terms can be rewritten using equations and rewrite rules. Rewrite rules are used to model the dynamics of the system, i.e., they describe transitions between states. Equations form the functional part of a rewrite theory, and are used to reduce terms to their "normal form" before they are rewritten using rewrite rules. The application of rewrite rules is intrinsically non-deterministic, which makes rewriting logic a good candidate for modeling concurrency.

In what follows, we briefly present the basic syntax of Maude, as needed for understanding the remainder of this section. Please refer to [104] for complete information. Maude programs are built from *modules*. A module consists of a *syntax declaration* and *statements*. The syntax declaration forms the signature and consists of declarations for *sorts*, which give names for the types of data, *subsorts*, which impose orderings on data types, and *operators*, which provide names for the operations acting upon the data. Statements are either equations or rewrite rules. Modules containing no rewrite rules but only equations are called *functional modules*, and they define equational theories $\langle \Sigma, E \rangle$. Modules that contain also rules are called *system modules* and they define rewrite theories $\langle \Sigma, E, R \rangle$. Functional modules (system modules) are declared as follows:

```
fmod (mod) <ModuleName> is

          <DeclarationsAndStatements>

endfm (endm)
```

Modules can import other modules, which helps in building up modular applications from short modules, making it easy to debug, maintain or extend.

One or multiple sorts are declared using the keywords `sort` and `sorts`, respectively, and subsorts are similarly declared using `subsort` and `subsorts`. The following defines the sorts `Action` and `Plan` and their subsort relation, which is used for specifying the BUpL syntax.

```
sorts Action Plan . subsort Action < Plan .
```

We can further declare operators (functions) defined on sorts (types) as follows:

```
op <OpName> : <Sort-1> ... <Sort-k>  -> <Sort>
                                    [<OperatorAttributes>] .
```

where `k` is the arity of the operator. For example, the operator declaration below is used to define the BUpL construct plan repair rule. The operator `((_<-_))` takes a query of sort `Query` that should be tested on the belief base, and a plan, and yields a term of sort `PRrule`. The operator is in mixfix form, where the underscores indicate the positions of its parameters. This also illustrates how Maude can be used to define the syntax of a BUpL language construct.

```
op ((_<-_)) : Query Plan -> PRrule .
```

Equations and rewrite rules specify how to transform terms. Terms are variables, constants, or the result of the application of an operator to a list of argument terms. Variables are declared using the keywords `var` and `vars`. For example, `var R : PRrule` declares a variable `R` of sort `PRrule`. Equations can be unconditional or conditional and are declared as follows, respectively:

```
eq  [<Label>] : <Term-1> = <Term-2> .
ceq [<Label>] : <Term-1> = <Term-2>
                         if <Cond-1> /\ ... /\ <Cond-k> .
```

where `Cond-i` is a condition which can be an ordinary equation `t = t'`, a matching equation `t := t'` (which is true only if the two terms match), a Boolean equation (which contains, e.g., the built-in (in)equality =/=, ==, and/or logical combinators such as `not`, `and`, `or`), or a membership equation `t : S` (which means that `t` is a member of sort `S`).

For example, the following conditional equation is part of a module for specifying when a formula logically follows from the belief base. The belief base is defined as a commutative sequence of ground belief atoms of sort `Belief`, separated by `#`. The conditional equation specifies that matching term `T` against a belief base containing belief `B` yields substitution `S`, if `match(T, B)` yields a substitution `S` that is different from `noMatch`, the built-in Maude constant to indicate that no substitution has been found.

```
var B : Belief .
var BB : BeliefBase .
var T : Term .
var S : Substitution .

ceq match(T, B # BB) = S if S := match(T, B) /\ S =/= noMatch .
```

Operationally, equations can be applied to a term from left to right. Equations in Maude are assumed to be terminating and confluent,[3] i.e., there is no infinite derivation from a term `t` using the equations, and if `t` can be reduced to different terms `t1` and `t2`, there is always a term `u` to which both `t1` and `t2` can be reduced. This means that any term has a *unique normal form*, to which it can be reduced using equations in a finite number of steps.

Finally, we introduce rewrite rules. Like equations, rewrite rules can also be unconditional or conditional, and are declared as follows:

```
rl  [<Label>] : <Term-1> => <Term-2> .
crl [<Label>] : <Term-1> => <Term-2>
                          if <Cond-1> /\ ... /\ <Cond-k>  .
```

where `Cond-i` can involve equations, memberships (which specify terms as having a given sort) and other rewrites. We will present several examples in the next section.

9.3.2 Implementing BUpL: Syntax

In this section, we use BUpL to illustrate how the syntax of agent programming languages can be implemented in Maude. We make a distinction between the logical parts of the language and the non-logical parts.

9.3.2.1 Logical Part

First, we have to define the logical language on which BUpL is based. Logical formulas occur in the belief base (ground atoms), in actions specifications (a formula as precondition, and a set of literals as effects), and in repair rules (a formula as the application condition). For the representation of atoms, the Maude built-in sorts `GroundTerm` and `Term` are used. That is, any Maude (ground) term can be used as an atom of our logical base language. In addition, we define the following sorts to represent also negated (ground) terms and (ground) sets of literals.

```
sorts NegGroundTerm NegTerm GroundLitSet LitSet .
```

[3] If this is not the case, the operational semantics of Maude does not correspond with its mathematical semantics.

The following subsort relations are defined on these sorts. Note that `GroundTerm < GroundLitSet` specifies that any Maude ground term can be a (set of) ground literals, and similarly for `Term < LitSet`.

```
subsorts GroundTerm GroundTermList < GroundLitSet .
subsorts Term NegTerm GroundLitSet < LitSet .
subsort  NegGroundTerm < NegTerm .
```

`GroundLitSet` is defined as a supersort of the Maude built-in sort `GroundTermList`, since we use its constant `empty` to represent an empty set of ground literals. The sorts `Belief` and `BeliefBase` are introduced with the subsort relations

```
subsorts Belief < GroundTerm GroundTermList < BeliefBase
                                            < GroundLitSet .
```

to represent beliefs. The following operators are introduced to syntactically represented (ground) literal sets, belief bases, and negated (ground) terms. The attributes `assoc comm id: empty` declare that the operator is associative and commutative with identity the empty set. The attribute `ctor` declares that the operator is a constructor, which means that it is used to construct terms rather than to apply it as a function and calculate the result. We overload the operator `#`, using it for representing both (ground) literal sets and belief bases. The attribute `ditto` specifies that an overloaded operator has the same attributes as the first declaration of the operator (excluding `ctor`).

```
op _#_ : LitSet LitSet -> LitSet [ctor assoc comm id: empty] .
op _#_ : GroundLitSet GroundLitSet -> GroundLitSet [ctor ditto] .
op _#_ : BeliefBase BeliefBase -> BeliefBase [ctor ditto] .

op neg_ : Term -> NegTerm [ctor] .
op neg_ : GroundTerm -> NegGroundTerm [ctor] .
```

We call formulas that are evaluated on the belief base *queries*. The query language is defined over terms as follows. The definition is more general than the DNF of Section 9.2.1. However, when defining the semantics, formulas are first transformed into DNF.

```
sort Query .
subsort Term < Query .

ops top bot : -> GroundTerm .
op ~_ : Query -> Query [ctor] .
op _/\_ : Query Query -> Query [assoc] .
op _\/_ : Query Query -> Query [assoc] .
```

This completes the specification of the syntax of the logical part of BUpL.

It is important to note that Maude is suitable as a framework in which many logics can be represented, using equations to axiomatize the logic and using rewrite rules as inference rules. This facilitates experimentation with different logics for representing agent beliefs, making the framework flexible.

9.3.2.2 Non-Logical Part

The non-logical part consists of the specification of actions, plans, procedures, and repair rules. We distinguish between internal and observable actions. This is useful for testing. Actions are specified as functions using equations. The action name is the function name specified as an operator, and applying the equation yields the precondition and effect of the action. Preconditions and effects are defined using the operators `o[_,_]` and `i[_,_]` for observable and internal actions, respectively. `nilA` is the "empty" action, used to define an empty plan. The code below shows an example specification of the `move` action from the tower of blocks example of Figure 9.1.[4] The sort `Nat` represents natural numbers.

```
sorts Action I-Action O-Action .
subsorts I-Action O-Action < Action .

ops nilA : -> Action .
op o[_,_] : Query LitSet -> O-Action .
op i[_,_] : Query LitSet -> I-Action .

op on : Nat Nat -> Belief .
op clear : Nat -> Belief .

op move : Nat Nat Nat -> O-Action .

ceq [act] : move(X, Y, Z) = o[on(X, Y) /\ clear(X) /\ clear(Z),
                              neg on(X, Y) # on(X, Z) # clear(Y)
                              # neg clear(Z) # clear(0)]
            if X =/= Z .
```

Plans are built from actions, procedure calls (at the end of a plan), sequential composition (`pre`), and non-deterministic choice (`sum`). The operators `pre` and `sum` are declared to be constructors, reflecting the fact that they are used to construct plans. Procedure names are introduced as operators, and a procedure is defined as an equation that yields the plan forming the body of the procedure. For example, the procedure `build` as declared below is used for building a tower of three blocks (`321`).

```
sort Plan .
subsort Action < Plan .

op pre : Action Plan -> Plan [ctor id: nilA strat (1 0)] .
op sum : Plan Plan -> Plan [ctor comm] .

op build : -> Plan .
eq build = pre(move(2, 0, 1), move(3, 0, 2)) .
```

Note that the operator `pre` has the attribute `strat (1 0)`. This specifies that only its first argument (an action) can be normalized using equations (expressed by the

[4] Note that in the specification of the `move` action in Maude, we have added the condition `X =/= Z`, which is easily done using conditional equations.

1), before any equations are applied on the operator `pre` itself (expressed by placing `1` before `0`).[5] The second argument (a plan) is not normalized using equations. Using this attribute thus changes what a normal form is for the operator `pre`: the normal form is obtained by normalizing the operator's first argument and then normalizing the operator itself at top level, while leaving the second argument intact. This prevents the continuous application of equations, which would lead to a stack overflow in case a non-terminating procedure is specified. For example, if we would specify a recursive procedure `build` using the equation

```
eq build = pre(move(2, 0, 1), pre(move(2, 1, 0), build)) .
```

without using `strat` in the declaration of `pre`, the continuous application of the equation to normalize `build` as occurring in the right-hand side of the equation would lead to a stack overflow.

Repair rules are defined similarly to procedures, using equations. An operator is introduced to define the name and parameters of the repair rule, and the equation yields the repair rule itself. On the basis of the equations, repair rules can be collected into a repair rule base (of sort `PRbase`). The example repair rule `pr` shown below can be used to deal with a failing `move(X,Y,Z)` action. The action fails if `Y` or `Z` are not clear. In this case the repair rule can be applied to move a block to the table (clearing the block on which it was placed), after which it is tried again to build the tower.

```
sorts PRrule PRbase .
subsort PRrule < PRbase .

op ((_<-_)) : Query Plan -> PRrule .
op empty-prb : -> PRbase .
op __ : PRbase PRbase -> PRbase [assoc comm id: empty-prb] .

ops pr : Nat Nat -> PRrule .
eq [pr] : pr(X, Y) = ((on(X, Y) /\ Y > 0 <-
                                    pre(move(X, Y, 0), build))) .
```

Finally, we define an operator for representing BUpL mental states. The operator takes a label, belief base and plan, and yields a term of sort `LBpMentalState`. The label represents the label of the transitions in the transition system, i.e., it represents which actions have been executed.

```
op <<_,_,_>> : Label BeliefBase Plan -> LBpMentalState .
```

9.3.3 Example BUpL Program

Using the implementation of the BUpL syntax in Maude, one can easily specify BUpL programs in Maude. An example is the following tower building agent, which

[5] In our implementation, no equations are specified for normalizing `pre` itself.

represents the example agent from Figure 9.1 in Maude. The `move` action and the procedure and plan repair rule have already been introduced above. In addition, the program specifies the initial belief base `bb`, which expresses where blocks are positioned initially and which blocks are clear. Moreover, the initial mental state of the builder agent is specified using the operator `builder`. The initial plan is `build`. Since no actions have been executed yet in the initial mental state, its label is empty. The equation `module-name` is specified to obtain a reference to the module in which the BUpL program is written. This will be used when implementing the semantics.

```
mod AGENT-DATA
    protecting BUPL-SYNTAX .
    protecting NAT .

    eq module-name = 'AGENT-DATA .

    op on : Nat Nat -> Belief .
    op clear : Nat -> Belief .

    op bb : -> BeliefBase .
    eq bb = on(3, 1) # on(1, 0) # on(2, 0) # clear(0) #
                                          clear(3) # clear(2) .

    op move : Nat Nat Nat -> O-Action .
    vars X Y Z : Nat .
    ceq [act] : move(X, Y, Z) =
                          o[on(X, Y) /\ clear(X) /\ clear(Z),
                            neg on(X, Y) # on(X, Z) # clear(Y)
                            # neg clear(Z) # clear(0)]
              if X =/= Z .

    op build : -> Plan .
    eq build = pre(move(2, 0, 1), move(3, 0, 2)) .

    ops pr : Nat Nat -> PRrule .
    eq [pr] : pr(X, Y) = ((on(X, Y) /\ Y > 0 <-
                                  pre(move(X, Y, 0), build))) .

    op builder : -> LBpMentalState .
    eq builder = << bLabel(empty), bb, build >> .
endm
```

9.3.4 Implementing BUpL: Semantics

The implementation of the semantics of BUpL in Maude can again be divided into the implementation of the logical part and of the non-logical part.

9.3.4.1 Logical Part

Implementing the semantics of the logical part means implementing matching a query against a belief base. Matching takes place both to determine whether an action can be executed, as well as to determine whether a repair rule can be applied. It is defined using the operator `match : Query BeliefBase -> Substitution`, which takes a query and a belief base, and yields a substitution in case the query matches the belief base, and the special substitution `noMatch` otherwise.

This operator is defined by making use of Maude's *reflective* capabilities [103]. Maude is a reflective logic since important aspects of its meta-theory can be represented at the object level, so that the object level correctly simulates the meta-theoretic aspects. The meta-theoretic aspect that we use here, is matching two terms. Maude continually matches terms when using equations and rewrite rules. This meta-level functionality can be conveniently used to match a term against a belief.

The meta-level operator that can be used for implementing this, is `metaMatch`. This operator takes the meta-representation of a module and two terms, and tries to match these terms in the module. If the matching attempt is successful, the result is the corresponding substitution. Otherwise, `noMatch` is returned. Obtaining the meta-representation of modules and terms can be done using the operators `upModule` and `upTerm`, respectively. The module that we use for this is the module containing the BUpL program, since the belief base is defined there. The name of the module is obtained by defining an equation for the operator `module-name`, as shown in the example program of Section 9.3.3. The sort `Qid` is a predefined Maude sort for identifiers. The base case for the operator `match`, where a term is matched against a belief, is defined using `metaMatch` as follows.

```
var T : Term .
var B : Belief .

op module-name : -> Qid .

eq match(T, B) =
       metaMatch(upModule(module-name), upTerm(T), upTerm(B)) .
```

Matching a term against a belief base is then defined by making use of the former equation.

```
var S : Substitution .
var BB : BeliefBase .

ceq match(T, B # BB) = S if S := match(T, B) /\ S =/= noMatch .
eq  match(T, B # BB) = noMatch [owise] .
```

For reasons of space, we omit the additional equations for matching composite formulas against a belief base.

9.3.4.2 Non-Logical Part

As proposed in [424], the general idea of implementing transition rules of an operational semantics in Maude, is to implement them as (conditional) rewrite rules. The premises of a transition rule then form the conditions of the corresponding rewrite rule, and the conclusion forms the rewrite itself.

We illustrate the implementation of transition rules using those for action execution and repair rule application. The transition rule for action execution

$$\frac{a(x_1,\ldots,x_n) =_{def} (\varphi,\xi) \in \mathcal{A} \qquad a(t_1,\ldots,t_n) = (\varphi',\xi') \qquad \theta \in Sols(\mathcal{B},\varphi')}{(\mathcal{B}, a(t_1,\ldots,t_n); p') \xrightarrow{a(t_1,\ldots,t_n)\theta} (\mathcal{B} \uplus \xi'\theta, p'\theta)} \ (act)$$

is implemented in Maude as two rewrite rules: one for internal actions and one for observable actions. Here, we present only the rule for observable actions.

```
ops eqSC : -> EquationSet .
eq eqSC = upEqs(module-name, false) .

var OA : O-Action .

crl [exec-OA] : << L:Label, BB, pre(OA, P) >> =>
 << oLabel(getName(OA, eqSC)),
    update(BB, downTerm(substitute(upTerm(effect(OA)), S), 'err)),
    downTerm(substitute(upTerm(P), S), 'err) >>
 if S := match(prec(OA), BB) /\ S =/= noMatch .
```

Recall that equations are used to map actions to their specification in terms of preconditions and effects (expressed using the operator `o[_,_]` in case of observable actions). Before Maude applies rewrite rules to a term, it first reduces the term to its normal form using equations. This means that all actions in a plan of a mental state that is rewritten, are first replaced by their preconditions and effects. Any substitutions that are calculated while executing the plan, are therefore applied to these preconditions and effects. This implements the first two conditions of the corresponding transition rule.

In order to implement the third condition, an auxiliary operator `prec` is used, which yields the precondition of an action. The precondition is then matched against the belief base to yield a substitution. The rule can only be applied if a substitution is indeed found, i.e., if the precondition matches the belief base.

Updating the belief base according to the effect of the action is done using the operator `update : BeliefBase GroundLitSet -> BeliefBase`. The ground set of literals, which forms a parameter of this operator, is obtained from applying the calculated substitution `S` to the effect of the action using the operator `substitute : Term Substitution -> Term`. This operator is general in that it applies a substitution to any term of sort `Term`. In this case, we want to apply the substitution to the effect of an action. This can be done using the operator `upTerm` to obtain the meta-representation of the effect of the action, which is of sort `Term`,

and after applying the substitution transforming the term again into its object-level variant using `downTerm`. In a similar way, the calculated substitution is applied to the rest of the plan, according to the transition rule. The operator `getName`, which is used for obtaining the label of the new mental state, retrieves the name of the action (including instantiated parameters) from its precondition/effect specification and the action equations of the BUpL program (obtained using the meta-level built-in Maude function `upEqs`).

The transition rule for applying a plan repair rule

$$\frac{(\mathcal{B}, \alpha; p) \not\downarrow \alpha\theta' \rightarrow \qquad \varphi \leftarrow p' \in \mathcal{R} \qquad \theta \in Sols(\mathcal{B}, \varphi)}{(\mathcal{B}, p) \xrightarrow{\tau} (\mathcal{B}, p'\theta)} \ (fail)$$

is implemented in Maude as the following rewrite rule:

```
crl [exec-fail] : << L:Label, BB, pre(A, P) >> =>
    << tLabel, BB, downTerm(substitute(upTerm(P), S), 'err) >>
    if match(prec(A), BB) == noMatch /\
       (((Q <- P)) PRB) := getPR(eqSC) /\
       S := match(Q,BB) /\ S =/= noMatch .
```

The first condition of the rewrite rule checks that the action that is to be executed, cannot be executed (which is the case if no substitution can be found when the precondition of the action is matched against the belief base). This implements the first condition of the transition rule.

The second condition of the rewrite rule implements the second condition of the transition rule as follows. Since repair rules are implemented as equations that yield a repair rule (see Section 9.3.2.2), we need an operator to collect the rules into a repair rule base. This is done by `getPR : EquationSet -> PRbase`, which takes the equations corresponding to the repair rules and yields a repair rule base consisting of the rules as defined by the equations.

The third and fourth conditions of the rewrite rule implement matching the condition of the repair rule to the belief base, corresponding to the third condition of the transition rule. The resulting substitution is applied to the plan of the repair rule, which becomes the plan of the next mental state.

9.3.5 Executing an Agent Program

The BUpL example agent from Section 9.3.3 can be executed in Maude using the command `rew builder`. Maude then uses the implemented BUpL semantics to rewrite the term `builder`, which is first reduced to the initial mental state of the builder agent using the equation `eq builder = << bLabel(empty), bb, build >>`, after which other equations and rewrite rules are applied that specify the semantics of BUpl. The Maude output looks as follows.

```
Maude> rew builder .
rewrite in AGENT-DATA : builder .
rewrites: 4722 in 202ms cpu (252ms real) (23264 rewrites/second)
result LBpMentalState:
<< oLabel('move['s_^3['0.Zero],'0.Zero,'s_^2['0.Zero]]),
clear(0) # clear(3) # on(1, 0) # on(2, 1) # on(3, 2), nilA >>
```

This says that the builder finished its execution after moving block 3 onto 2 (the current plan is empty), and that the belief base reflects the current configuration of the blocks, namely the tower 321. The output `'move[...]` is the meta-representation of `move(3, 0, 2)`. For example, `'s_^3['0.Zero]` represents the third successor of zero, i.e., 3.

One can also rewrite the builder step by step. For example, the following shows the resulting mental state after one step of rewriting, namely, a τ transition corresponding to handling the failure of action `move(2, 0, 1)` which cannot be executed since block 3 is on top of 1. We can see that the belief base remains unchanged, and the only change is in the current plan. The application of the repair rule `pr` replaces the failing plan by a plan which consists of first executing the action of moving a block (in our case block 3) onto the floor and then trying `build` again. Note that the action is represented by its precondition and effect in the form `o[precondition,effect]`.

```
Maude> rew [1] builder .
rewrite [1] in AGENT-DATA : builder .
rewrites: 4141 in 181ms cpu (228ms real) (22756 rewrites/second)
result LBpMentalState:
<< tLabel,
clear(0) # clear(2) # clear(3) # on(1, 0) # on(2, 0) # on(3, 1),
pre(o[clear(0) /\ (clear(3) /\ on(3, 1)),
neg clear(0) # neg on(3, 1) # clear(0) # clear(1) # on(3, 0)],
build) >>
```

9.4 Model-Checking

In Section 9.3, we have shown how the syntax and semantics of BUpL can be implemented in Maude, and how an example BUpL program can be defined and executed. One of the main advantages of using Maude for agent development is that it supports software development using formal methods. In this section, we show how the Maude LTL model-checker [156] can be used for verifying agent programs. Verification is important in order to ensure that the final agent program is correct with respect to a given specification or that it satisfies certain properties. Properties are specified in *linear temporal logic* (LTL) [296] and are verified using a model-checking algorithm. Model-checking *only* works for finite state systems.

We briefly recall some of the LTL concepts which we will refer to in the following sections. The basic LTL formulas are the booleans *true* ($\top$) and *false* ($\bot$)

and atomic propositions. Inductively, LTL formulas are built on top of the usual boolean connectives like negation and conjunction. Typical LTL operators are *next* ($\bigcirc$) and *until* ($\mathcal{U}$). The operator $\mathcal{U}$ can be used to define the connective *eventually*, $\Diamond\phi = \top\mathcal{U}\phi$. The connective $\Diamond$ can be used to further define the connective *always*, $\Box\phi = \neg\Diamond\neg\phi$.

The semantics of LTL formulas is defined in the usual way. The satisfaction of an LTL formula ϕ in a finite transition system S with an initial state s is defined as follows:

$$S, s \models \phi \text{ iff } (\forall \pi \in Paths(s))(S, \pi \models \phi)$$

which means that the LTL formula ϕ holds in the state s if and only if ϕ holds for any path in $Paths(s)$, the set of paths in S starting at s. Given a path π, the satisfaction relation for a formula ϕ is defined inductively on the structure of ϕ. We present, as an example, the semantics of the operator "next" and of the connective "until":

$$S, \pi \models_{LTL} \bigcirc\phi \quad \text{iff } S, \pi(1) \models_{LTL} \phi$$
$$S, \pi \models_{LTL} \phi\mathcal{U}\psi \text{ iff } (\exists n)(S, \pi(n) \models_{LTL} \psi) \wedge (\forall m < n)(S, \pi(m) \models_{LTL} \phi)$$

where n, m are natural numbers and $\pi(n)$ denotes the subpath of π starting in the "n"-th state on π. Basically, $\bigcirc\phi$ is satisfied in a state if and only if ϕ is satisfied in the successor state. The formula $\phi\mathcal{U}\psi$ holds on a path π if and only if there is a state which makes ψ true and in all the previous states ϕ was true.

Intuitively, a given path π satisfies the temporal formula $\Diamond\phi$ if there exists a state on π which satisfies ϕ. Similarly, π satisfies the temporal formula $\Box\phi$ if there does not exist a state on π which does not satisfy ϕ. By means of these operators, LTL allows specification of properties such as *safety* properties (something "bad" never happens) or *liveness* properties (something "good" eventually happens). These properties relate to the infinite behavior of a system. We will provide concrete examples in the next sections.

9.4.1 Connecting BUpL Agents and Model-Checker

Maude system modules can be seen as specifications at different levels. On the one hand they can specify *systems* (in our case, BUpL agents), on the other hand they can specify *properties* that we want to prove about a given system. The syntax of LTL is defined in the functional module `LTL` (in the file `model-checker.maude`). The following code, which is a part of the module `LTL`, shows the declaration of the temporal operators "until" (`U`), "release" (`R`), "eventually" (`<>`) and "always" (`[]`). It further shows the definitions of `<> f` (resp. `[] f`).

```
fmod LTL is
     protecting Bool .
     sort Formula .

     *** primitive LTL operators
     ops True False : -> Formula [ctor ...] .
     op _U_ : Formula Formula -> Formula [ctor ...] .
     op _R_ : Formula Formula -> Formula [ctor ...] .
     ...
     *** defined LTL operators
     op <>_ : Formula -> Formula [...] .
     op []_ : Formula -> Formula [...] .
     ...
     var f : Formula .
     eq <> f = True U f .
     eq [] f = False R f .
     ...
endfm
```

In order to use the Maude model checker, one needs to do two main things: (i) define which sort represents the states of the system that is to be model-checked, and (ii) define the atomic predicates that can be checked on these states. LTL formulas defined over these atomic predicates are then used to specify the property that is to be model-checked.

In our case, the states are the BUpL mental states of sort `LBpMentalState`. In order to express that these are the states of our system, we need the Maude model-checker module `SATISFACTION`, which is defined as follows.

```
fmod SATISFACTION is
    protecting BOOL .

        sorts State Prop .
        op _|=_ : State Prop -> Bool [frozen] .
endfm
```

We import this module into our own module `BUPL-PREDS` for defining the BUpL atomic predicates, and declare `subsort LBpMentalState < State` to express that BUpL mental states are to be considered the states of the system that is to be model-checked. Moreover, we use the operator `_|=_` for defining the semantics of the atomic state predicates, which are declared as predicates of sort `Prop`. We define the state predicate `fact(B)` to express that ground atom `B` is believed by the BUpL agent.

```
mod BUPL-PREDS is
    including BUPL-SEMANTICS .
    including SATISFACTION .
    including MODEL-CHECKER .
    including LTL-SIMPLIFIER .

    subsort LBpMentalState < State .
    op fact : Belief -> Prop .
```

```
    var B : Belief .
    eq << L:Label, B # BB:BeliefBase, P:Plan >> |= fact(B) = true .
endm
```

In the sequel, we will introduce additional state predicates to specify properties of BUpL agents.

9.4.2 Examples

To run the model-checking procedure we need, after loading in the system the file `model-checker.maude`, to call the operator `modelCheck` with an initial state and a formula, specifying the property that is to be checked, as arguments. The result of the algorithm is either the boolean **`true`** (if the property holds) or a counterexample. The operator `modelCheck` is declared in the system module `MODEL-CHECKER` which is defined in the file `model-checker.maude`.

```
fmod MODEL-CHECKER is
     including SATISFACTION .
     including LTL .
     subsort Prop < Formula .
     ...
     subsort Bool < ModelCheckResult .
     op modelCheck : State Formula ~> ModelCheckResult [...] .
endfm
```

Recall that `State` and `Formula` are sorts we have already seen declared in the modules `Statisfaction` and `LTL`, respectively (Section 9.4.1).

We can use the predicate `fact` (defined in Section 9.4.1) in order to define safety properties. As an example, we model-check that it is never the case that the agent believes the table is on block `3`. The following Maude output shows that the result is the boolean **`true`**.

```
Maude> red  modelCheck(builder, []~ fact(on(0, 3))) .
reduce in AGENT-DATA : modelCheck(builder, []~ fact(on(0, 3))) .
rewrites: 4811 in 196ms cpu (241ms real) (24425 rewrites/second)
result Bool: true
```

The predicate `fact` enables us to express properties of the beliefs of a BUpL agent. In order to express properties of actions, we define another state predicate `taken` using the label of a BUpL state. Recall that the label specifies which action has been executed.

```
mod BUPL-PREDS is
...
op taken : Action -> Prop .
ceq << oLabel(T), BB:BeliefBase, P:Plan >> |= taken(A) = true
    if T := getName(A, eqSC) .
```

The predicate `taken(A)` is true in a state if the label `T` matches `A`. Note that we cannot match `A` and `T` directly, since `T` is an action name with instantiated parameters, while `A` is an action specified by means of a precondition and effect (of the form `o[precondition,effect]`). The operator `getName` is used to obtain the name and instantiated parameters of `A` (see Section 9.3.4.2).

We can use the predicate `taken` to verify that a certain sequence of actions has been executed. For instance, the following Maude output shows that eventually, if block `2` is moved onto block `1` then moving block `3` onto block `2` takes place after this. This is an example of a liveness property.

```
Maude> red  modelCheck(builder,
        <> (taken(move(2, 0, 1)) -> O taken(move(3, 0, 2)))) .
reduce in AGENT-DATA : modelCheck(builder,
        <> (taken(move(2, 0, 1)) -> O taken(move(3, 0, 2)))) .
rewrites: 30 in 1ms cpu (0ms real) (30000 rewrites/second)
result Bool: true
```

We can define more meaningful liveness properties such as *goals* that should be reached from an initial configuration. The equation `g1` defines the predicate `goal321` as being true if the agent believes that block `3` is on block `2` and block `2` is on block `1`, expressing that the agent built the tower `321`.

```
mod AGENT-DATA-PREDS is
    including BUPL-PREDS .
    including AGENT-DATA .

    op goal321 : -> Prop .
    eq [g1] : goal321 = fact(on(3,2)) /\ fact(on(2,1)) .
endm
```

While the generic BUpL predicates `fact` and `taken` were specified in the module `BUPL-PREDS`, the predicate `goal321` is specific to the tower building agent and is consequently specified in the module `AGENT-DATA-PREDS`.

The following Maude output shows that the result of model-checking `[]<>goal321` is **`true`**, meaning that the BUpL agent will always eventually build the tower `321` from the initial configuration.

```
Maude> red  modelCheck(builder, []<> goal321) .
reduce in AGENT-DATA-PREDS : modelCheck(builder, []<> goal321) .
rewrites: 4816 in 245ms cpu (292ms real) (19580 rewrites/second)
result Bool: true
```

We might be interested in knowing not only that `goal321` is reachable from the initial state, but also in the corresponding trace. For this, it suffices to model-check the negation of `goal321`. This returns a counterexample representing the trace that we want.

```
Maude> red  modelCheck(builder, []~ goal321) .
reduce in AGENT-DATA-PREDS : modelCheck(builder, []~ goal321) .
rewrites: 4568 in 188ms cpu (249ms real) (24173 rewrites/second)
result ModelCheckResult: counterexample(
{<< empty-l,..., ... >>,'exec-fail}
{<< tLabel,..., ... >>,'exec-OA}
{<< oLabel('move['s_^3['0.Zero],'s_['0.Zero],'0.Zero]),
                                              ..., ... >>,'exec-OA}
{<< oLabel('move['s_^2['0.Zero],'0.Zero,'s_['0.Zero]]),
                                            ..., ... >>,'exec-OA},

{<< oLabel('move['s_^3['0.Zero],'0.Zero,'s_^2['0.Zero]]),
    clear(0) # clear(3) # on(1, 0) # on(2, 1) # on(3, 2),
                                                nilA >>, deadlock}
)
```

This counterexample should be read as follows. The declaration of the operator `counterexample` is in the predefined module `MODEL-CHECKER`. It is formed by a pair of transition lists:

```
op counterexample : TransitionList TransitionList ->
                                       ModelCheckResult [ctor] .
```

A transition list is composed of transitions, and a transition records a state and the name of the rule which has been applied from that state.

```
subsort Transition < TransitionList .
op {_,_} : State RuleName -> Transition [ctor] .
op __ : TransitionList TransitionList ->
                          TransitionList [ctor assoc id: nil] .
```

The first list of `counterexample` represents the shortest sequence of transitions (which record the states being visited) that leads to the first state of a loop. This loop is represented by the second list from `counterexample`. In our example, the first list consists of four transitions. It shows that first the rewrite rule `exec-fail` has been applied from the initial state (for readability, the belief base and plan are omitted), and consequently the label of the next state denotes a τ step. Then, the rule `exec-OA` is applied, which changes the label of the next state into the meta-representation of the action `move(3,1,0)`. A similar reasoning applies for the next transition.

The second list of the counterexample (after the white line) consists of only one transition. The initial plan has terminated (the action `nilA` is reached) and the belief base reflects that tower `321` is built. The rule name from this last transition is `deadlock`, a predefined constant which is declared in `MODEL-CHECKER`. It means that from the state that the agent reached, no further rewrite rule is applicable. Thus, the system "cycles" in a deadlock state and this is the loop represented by the second transition list. We note that a Maude deadlock state is, in our case, a termination BUpL state.

9.4.3 Fairness

The BUpL agent we have described always terminates, i.e., all execution paths are finite. Infinite behavior can occur due to recursive abstract plans, and because of the non-determinism of the operator `sum`. The reason in the latter case is that it is possible that the choice between a failing and a terminating action goes always in favor of the failing one. We call such behavior *unfair*.

In practice, unfair traces are generally prevented from occurring through scheduling algorithms such as round-robin. However, at the level of prototyping BUpL in Maude we would like to abstract from controlling the non-deterministic choices. Rather, non-determinism is reduced at a later phase of design, at a more concrete implementation level. We stress that it is important to abstract from control issues at the prototype level, since the main concern is to experiment with language definitions rather than scheduling algorithms.

Nevertheless, when model-checking BUpL agents one may want to ignore unfair traces and show that the agent satisfies certain properties assuming fairness. Since we work in a declarative framework, our solution is to model-check only the traces that satisfy certain fairness constraints and to define fairness using LTL. To illustrate this, we first introduce the predicate `enabled`. The proposition `enabled(A)` holds in a state if the action `A` can be executed in that state, i.e., if the action's precondition holds.

```
op enabled : Action -> Prop .
ceq << L:Label, BB, P >> |= enabled(A) = true
    if match(prec(A), BB) =/= noMatch .
```

Following [296], we then define fairness with respect to an action as follows.

```
op fair : Action -> Prop .
eq fair(A) = <>[] enabled(A) -> []<> taken(A) .
```

This says that if an action is continuously enabled it should be infinitely often taken. This requirement casts aside traces where the failing action is always chosen in spite of a terminating action a since such traces are unfair with respect to a.

For a concrete example where fairness is useful, we modify the BUpL example from Section 9.3.3 such that the initial plan of the agent is `p1`, which is defined as a non-deterministic choice (`sum`) between an always failing action and the plan `build`. We further add an always enabled repair rule `pr1` to handle the case where the failing action has been chosen in `p1`.

```
eq p1 = sum(i[bot, empty], build) .
ops pr1 : -> PRrule .
eq [pr1] : pr1 = (( top <- p1 )) .
...
eq builder = << bLabel(empty), bb, p1 >> .
```

It is now the case that achieving `goal321` is no longer always possible, demonstrated by the following counterexample, which is generated when model-checking the property `[]<> goal321`.

```
Maude> red  modelCheck(builder, []<> goal321) .
reduce in AGENT-DATA-PREDS : modelCheck(builder, []<> goal321) .
rewrites: 4875 in 209ms cpu (254ms real) (23217 rewrites/second)
result ModelCheckResult: counterexample(
    {<< empty-l, ..., ... >>,'sum}
    {<< tLabel, ..., ... >>,'exec-fail},

    {<< tLabel, ..., ... >>,'sum}
    {<< tLabel, ..., ... >>,'exec-fail})
```

The counterexample shows that first the failing action was chosen to be executed, which is then handled by the repair rule `pr1`. In this counterexample, this leads to a loop in which over and over the failing action is chosen and then the repair rule is applied. This loop is represented in the second parameter of `counterexample` (below the white line).

However, if we consider the paths which are fair with respect to `move(3,1,0)` then we have that `goal321` is always achieved.

```
Maude > reduce in AGENT-DATA-PREDS :
modelCheck(builder, fair(move(3, 1, 0)) -> []<> goal321) .
rewrites: 9097 in 196ms cpu (231ms real) (46184 rewrites/second)
result Bool: true
```

9.5 Testing

In the previous section, we have illustrated how Maude can be used for model-checking BUpL agents, using the tower builder of Section 9.3.3 as an example. Since the tower builder has a finite number of mental states, verification by model-checking is in principle feasible. However, the state space of agents can also be infinite, making direct model-checking impossible. This issue may be addressed within the context of model-checking, e.g., by investigating abstractions techniques for reducing the state space. In this section, however, we are concerned with a different technique than model-checking, namely *testing*. Testing can be used for identifying failures in infinite state systems or in finite state systems where the state space becomes too large for model-checking. The basic idea behind testing is that it aims at finding failures by showing that the intended and the actual behavior of a system differ through generating and checking individual executions.

In this section, we present two kinds of testing that fit Maude very well. The first is testing for satisfaction of invariants by means of search (Section 9.5.1), and the second is testing through the specification of test cases that express properties of an execution trace of an agent (Sections 9.5.2 to 9.5.4). The latter is implemented by

means of Maude strategies, which are used to control the application of rewrite rules on a meta-level. We refer to Chapter [126] for a related approach to testing agent programs. It is similar in that it also uses a formal specification of test cases. The main differences concern the language used for specifying test cases, and we show how our approach fits into the rewriting framework of this chapter.

The running example that we use in this section is a variant of the tower builder introduced previously. Here we consider a tower builder that should respect the specification "the agent should continually construct towers, the order of the blocks is not relevant, however each tower should use more blocks than the previous, and additionally, the length of the towers must be an even number". Since the agent keeps on building higher towers, its state space is infinite. We assume that the programmer decides to refine the specification and tries to implement a BUpL agent that builds towers where the constituting blocks are assigned consecutive numbers, thus `21` and `4321` are examples of "well-formed" towers.

Initially, there is one block and it is on the table. In order to indicate that the agent has finished building a tower of length `X`, it inserts a predicate `done(X)` in the belief base by means of the action `finish(X,Y)` (where `Y` is added for technical reasons that we do not further explain). For indicating that the next tower that is to be built has length `X`, the agent uses a predicate `max(X)`. The predicate `length(X)` is used to represent the current length `X` of the tower. The builder agent is executed by rewriting a term of the form `builder(X,Y)`, where `X` is the length of the tower that is to be built as the first one, and `Y` is added for technical reasons that we do not further explain. For illustration purposes, we consider two variants of this tower builder: a correct one and a faulty one that builds odd length towers. Since it is not needed for explaining the techniques presented in this section, we do not provide the code for these tower builders.[6]

9.5.1 Searching

Maude provides a `search` command that can be used, among other things, to test for the satisfaction of invariants. Invariants are defined as properties of states. Search is breadth-first, which means that if there is a state where the invariant does not hold, then the search terminates.

Searching in Maude for invariants can be done using the Maude search command with parameters of the following form.

```
search init =>* x:k such that I(x:k) =/= true .
```

Here, `init` is the initial state from which the search starts. It searches for states `x` of sort `k` that are reachable from this initial state through zero or more rewrite steps

[6] It can be downloaded from http://homepages.cwi.nl/~astefano/agents/bupl-strategies.php.

(represented by =>*) and for which the invariant I does not hold. This is helpful when verifying safety properties. For example, an invariant for the BUpL builder is the length of the towers, which should always be even. This invariant can be specified by means of a predicate doneEven as follows.

```
mod BUPL-BUILDER-INVARIANTS is
    including AGENT-DATA .

    op doneEven : LBpMentalState -> Bool .
    ceq doneEven(<< L:Label, done(X) # BB, P:Plan >>) = true
        if (2 divides X) .
    eq doneEven(<< L:Label, done(X) # BB, P:Plan >>) = false
       [owise] .
    var MS : LBpMentalState .
endm
```

When we take the faulty implementation and search for doneEven(MS) =/= **true** with MS being a variable of sort LBpMentalState, we obtain a solution, i.e., a state where the invariant does not hold (done(3) appears in the belief base):

```
search in BUPL-BUILDER-INVARIANTS :
       builder(3, 0) =>* MS such that doneEven(MS) =/= true .

Solution 1 (state 11)
states: 12  rewrites: 21030 in 1220ms cpu (1301ms real)
(17226 rewrites/second)
MS --> << ..., clear(0) # clear(3) # length(3) # max(3) #
               done(3) # on(1, 0) # on(2, 1) # on(3, 2),
          ... >>
```

However, this procedure terminates only when the implementation is faulty, since in the correct implementation no state would be found where the invariant does not hold. A possible solution is to bound the search. This can be done by explicitly giving a depth bound, for example 100, as in the following example where the correct implementation is searched.

```
search [1, 100] in BUPL-BUILDER-INVARIANTS :
       builder(3, 0) =>* MS such that doneEven(MS) =/= true .

No solution.
states: 10  rewrites: 15266 in 779ms cpu (821ms real)
(19574 rewrites/second)
```

9.5.2 Formalizing Test Cases

Searching as treated in the previous section can be viewed as an ad hoc way of testing. While it may work for certain cases, it has several drawbacks. As for model-checking, state space explosion may be a problem since the whole state space is

considered (if no bound is used on the search). Moreover, it works with invariants expressed over the states of the system, while one may also want to test other properties such as the execution of certain sequences of actions. In this section, we present a formal language for the specification of test cases that does allow to specify this. We use so-called *rewriting strategies* [154] for implementing these tests in Maude. In Section 9.5.3, we introduce rewriting strategies and in Section 9.5.4 we show how these are used to implement a mechanism in Maude for checking whether a BUpL agent passes the tests.

Our test case format is based on one of the main BUpL concepts, namely actions. Our test case format is a kind of black box testing, aimed at testing the *observable behavior* of agents. For this reason, we have made a distinction between *internal and observable actions*. The idea is that the execution of observable actions is visible from outside the agent. Observable actions can be actions the agent executes in the environment in which it operates. In the sequel, we will sometimes omit the adjective "observable" if it is clear from the context. Black box testing as we do in this section can be contrasted with searching (Section 9.5.1), which focuses on testing properties of the belief base of agents and consequently can be viewed as a kind of white box testing.

We introduce a general test case format that allows to test whether certain sequences of observable actions can be executed. Sequences of actions are defined as regular expressions. The idea is that the action expression of a test is used to generate execution traces satisfying the action expression.[7] The action expression thus controls the execution of the agent in the sense that only those actions are executed that are in conformance with the action expression. This is crucial for reducing the state space, and makes this approach essentially different from searching.

In order to distinguish between internal and observable actions, we adapt the BUpL syntax slightly and distinguish internal and observable actions names Act_{int} and Act_{obs}, respectively, where $Act = Act_{int} \cup Act_{obs}$ and $Act_{int} \cap Act_{obs} = \emptyset$. The following BNF grammar defines the language $\mathcal{T}$ of test cases, where $a \in Act_{obs}$ denotes a ground observable action. $\mathcal{T}$ defines regular expressions over actions.

$$\mathcal{T} ::= a \mid \mathcal{T};\mathcal{T} \mid \mathcal{T} + \mathcal{T} \mid \mathcal{T}^*$$

We now define formally what it means to apply a test to a BUpL agent. For this, we adapt the operational semantics of Section 9.2.2 slightly to account for the distinction we make between internal and observable actions. In particular, instead of one transition rule for actions (*act*) we need two: one for internal actions and one for observable actions. The transition rule for internal actions is as the rule (*act*) of Section 9.2.2, except that it becomes a τ transition. This accounts for the fact that

[7] The formalism can be extended to include tests on the belief base of the agent that can be expressed using temporal logic (see [20]). These can be checked on the traces generated by testing for the execution of sequences of observable actions. However, for reasons of simplicity, we do not elaborate on this here. We refer to Chapter [126] for an approach that also uses temporal logic for expressing tests on agent behavior.

these actions are not observable. The transition rule for observable actions is as the rule (*act*) of Section 9.2.2, except that the action is an observable action.

We denote the application of a test $\mathcal{T}$ to an initial BUpL mental state ms_0 as $\mathcal{T}@ms_0$. The semantics is defined such that it yields the set of final states reachable through executing the agent restricted by the test, i.e., only those actions are executed that comply with the test. This means that an agent with initial mental state ms_0 satisfies a test $\mathcal{T}$ if $\mathcal{T}@ms \neq \emptyset$, in which case we say that a test $\mathcal{T}$ is *successful*. Since one usually tests for the absence of "bad" execution paths, we say that a BUpL agent with initial mental state ms_0 is *safe with respect to a test* $\mathcal{T}$ if the application of the test fails, i.e., $\mathcal{T}@ms_0 = \emptyset$. The operator @ (which applies a test to a single mental state) is lifted to its application to a set of mental states in the usual way, by taking the union of its application to each mental state in the set. Note that this means that $\mathcal{T}@\emptyset = \emptyset$. We define the semantics of tests as follows.

$$\mathcal{T}@ms_0 = \begin{cases} \{ms \mid ms_0 \overset{a}{\Rightarrow} ms\}, & \mathcal{T} = a \\ \mathcal{T}^1@ms_0 \cup \mathcal{T}^2@ms_0, & \mathcal{T} = \mathcal{T}^1 + \mathcal{T}^2 \\ \mathcal{T}^2@(\mathcal{T}^1@ms_0), & \mathcal{T} = \mathcal{T}^1;\mathcal{T}^2 \\ \{ms_0\} \cup \bigcup_{i\geq 1}(\mathcal{T}')^i@ms_0, & \mathcal{T} = (\mathcal{T}')^* \end{cases}$$

The arrow $\overset{a}{\Rightarrow}$ stands for $\Rightarrow\overset{a}{\rightarrow}$, where $\Rightarrow$ denotes the reflexive and transitive closure of $\overset{\tau}{\rightarrow}$.

We explain the semantics of $a@ms_0$ in some more detail. The idea is that the test should be successful for ms_0 if action a can be executed in ms_0. The result is then the set of mental states resulting from the execution of a, as defined by $\{ms \mid ms_0 \overset{a}{\Rightarrow} ms\}$. We need to keep those mental states to allow a compositional definition of the semantics. In particular, when defining the semantics of $\mathcal{T}^1;\mathcal{T}^2$ we need the mental states resulting from applying the test $\mathcal{T}^1$, since those are the mental states in which we then apply the test $\mathcal{T}^2$, as defined by $\mathcal{T}^2@(\mathcal{T}^1@ms_0)$.

9.5.3 Introduction to Maude Strategies

We choose to implement tests in Maude using rewriting strategies [154]. In this section we motivate this choice and introduce Maude rewriting strategies. Rewriting strategies are understood as a way to reduce the non-determinism of rewrite theories. Non-determinism is reduced since a strategy controls the application of rewrite rules. Strategies are related to tests as defined in the previous section, by viewing tests as a kind of strategies. Executing a BUpL agent under a test should restrict its execution such that only those actions are executed that are in conformance with the test. Take, for example, the test a. As we have previously defined it, the application of this test to a mental state ms is the set of all mental states which can be reached from ms by executing the observable action a (after possibly executing τ steps corresponding to internal actions, applying repair rules or making choices).

We are only interested in those rewritings that finally make it possible to execute a. Using strategies has the advantage of a clear separation between *execution* (by rewriting) at the object level and *control* (of rewriting) at the meta-level. In our case, we can add strategies to control the execution of the agent without making changes to the operational semantics of BUpL.

A strategy can be specified in a strategy language. Maude comes with such a strategy language, which we briefly describe now. For further details, please see [154] which introduces strategies as a language in Maude. A strategy language $\mathcal{S}$ can be viewed as a transformation of a rewrite theory $\mathcal{R}$ into $\mathcal{S}(\mathcal{R})$ such that the latter represents the execution of $\mathcal{R}$ in a controlled way. Given a term t in a rewrite theory $\mathcal{R}$ and a strategy s in the theory $\mathcal{S}(\mathcal{R})$, the application of s to t is denoted by $s@t$. The semantics of $s@t$ is the set of successors which result by rewriting t in $\mathcal{S}(\mathcal{R})$.

The simplest strategies are the constants `idle` and `fail`: `idle` @ $t = \{t\}$, `fail` @ $t = \emptyset$. Basic strategies consist of applying to a term t a rule (identified by a label) possibly with instantiating some variables appearing in the rule. The semantics of $l@t$, where l is a rule label, is the set of all terms to which t rewrites in one step using the rule labeled l anywhere it matches and satisfies the rule's condition.

Strategies can be combined under typical regular expression constructions: concatenation (`;`), union (`|`), and iteration of zero or more, or one or more steps (* or +). If E, E' are strategies, then $(E\,;\,E')@t = E'@(E@t)$, $(E \mid E')@t = (E@t) \cup (E'@t)$, $E+\ @t = \bigcup_{i \geq 1}(E^i@t)$ with $E^1 = E$ and $E^n = E^{n-1}\ ;\ E$, and $E* = \text{idle} \mid E+$.

It is also possible to define *if-then-else* strategies of the form E `?` E' `:` E'', which means that if the strategy E is successful when evaluated in a given state term, then the strategy E' is evaluated in the resulting states, otherwise E'' is evaluated in the *initial* state:

$$(E \texttt{ ? } E' \texttt{ : } E'') \ @\ t = \text{if } (E@t) \neq \emptyset \text{ then } E'@(E@t) \text{ else } E''@t \text{ fi}.$$

The if-then-else combinator is used to define strategies like `not`(E), which is defined as E `? fail : idle` , meaning that it reverses the result of applying E. A useful strategy is E`!`, which means "repeat until the end" and is defined as E`* ; not`(E).

In our case, state terms t are BUpL mental states. In order to rewrite `builder(3, 0)` using a strategy `E`, we only need to input the command `srew builder(3,0) using E` after loading the Maude file where the strategy language is defined (usually this is `maude-strat.maude`). If `E` is a rule name, for example, `exec-IA`, then the result is the mental state after performing an internal action, in this case setting `max(3)` which corresponds to the first parameter of `builder(3, 0)`.

```
Maude> (srew builder(3, 0) using exec-IA .)
rewrites: 1384 in 30ms cpu (55ms real) (44652 rewrites/second)
rewrite with strategy :
result LBpMentalState :
        << iLabel('set-max['s_^3['0.Zero],'0.Zero]),
        clear(0) # done(0) # length(1) # max(3) # on(1,0),
        ... >>
```

Strategies are declared and defined only in strategy modules. Strategy modules have the following syntax:

```
smod <STRAT-MODULE-NAME> is
     protecting <M> .
     including <STRAT-MODULE-NAME1> . ...
     including <STRAT-MODULE-NAMEk> .
     <DeclarationsAndDefinitionOfStrategies>
endsm
```

where `M` is the module containing the terms we want to rewrite using strategies and `STRAT-MODULE-NAME1`, ..., `STRAT-MODULE-NAMEk` are imported strategy modules.

Similarly to the declaration of operators, strategies are declared using the following format:

```
strat <STRAT-NAME> : <Sort-1> ... <Sort-m> @ <Sort> .
```

where `Sort` is the sort of the term which will be rewritten using the strategy `STRAT-NAME`. Like equations, strategies can be unconditional or conditional and are defined using the following syntax:

```
sd <STRAT-NAME>(<P1>, ..., <Pm>) := <Exp> .
csd <STRAT-NAME>(<P1>, ..., <Pm>) := <Exp> if <Cond> .
```

with `Pi` being the parameters of the strategy `STRAT-NAME` and `Exp` being a strategy expression.

9.5.4 Using Maude Strategies for Implementing Test Cases

We now illustrate how the test definitions of Section 9.5.2 can be implemented by means of Maude strategies. First, we show how the syntax of tests can be specified as a Maude functional module. We then describe a generic strategy `test2Strat` which associates to each test a corresponding strategy that implements the test. Finally, we focus on the implementation of the basic test a.

The following module defines the syntax of tests, in correspondence with the BNF grammar for tests of Section 9.5.2.

```
fmod TEST-SYNTAX is
        protecting SYNTACTICAL-DEFS .
        sort TestA .
        subsort 0-Action < TestA .
        op _;a_ : TestA TestA -> TestA .
        op _+a_ : TestA TestA -> TestA .
        op _*a : TestA -> TestA .
endfm
```

The code shows that we first declare a sort `TestA` for denoting tests. In order to express that any observable action is a test we use the subsort relation `subsort O-Action < TestA`. Further, we declare regular expression operators to construct new tests. We use the index `a` in their declaration in order to distinguish them from the regular expression operators defined for Maude strategies.

Now that we have defined the syntax of tests as above, we can define the strategy `test2Strat` inductively on the structure of tests:

```
strat test2Strat : Test @ LBpMentalState .
var Oa : O-Action . vars Ta1 Ta2 : TestA .
sd test2Strat(Oa) := do(Oa) .
sd test2Strat(Ta1 ;a Ta2) := test2Strat(Ta1) ; test2Strat(Ta2) .
sd test2Strat(Ta1 +a Ta2) := test2Strat(Ta1) | test2Strat(Ta2) .
sd test2Strat(Ta1 *a) := test2Strat(Ta1) * .
```

The strategy `do` is meant to implement the basic test a. Note the natural mapping from tests to the corresponding strategy.

We now focus on describing how to implement the basic test a, i.e., the strategy `do`. We recall that, when applied to a mental state ms, this test succeeds only if after performing some internal steps (corresponding to internal actions, repair rules, and choices among plans) the agent reaches a state where a is enabled. This means that we need to implement a strategy, `tauClosure`, for computing the transitive closure of τ steps. A simple[8] way to do this is as follows:

```
strat tauClosure : @ LBpMentalState .
sd tauClosure := (sum | exec-fail | exec-IA)! .
```

that is, by non-deterministically applying one of the rules which correspond to τ steps until no longer possible. Given that we have the strategy `tauClosure`, the implementation of the test a is straightforward:

```
strat do : O-Action @ LBpMentalState .
sd do(Oa) := tauClosure ; exec-OA[OA <- Oa] .
```

where `exec-OA[OA <- Oa]` applies `exec-OA` with the variable `OA` from the definition of the rewrite rule being instantiated by the argument `Oa` of the strategy. Note that the strategy `tauClosure` returns precisely those states from which no τ steps are possible, that is, the states where the head of the current plan is an observable action. If this observable action is the one given as argument to the strategy `do` then it succeeds and computes again the transitive closure. Otherwise, it fails. To see how this strategy works in practice, we strategically execute `builder(3, X:Nat)` using `do(move(2, 0, 1))`. This means that we test whether the agent executes `move(2, 0, 1)` as the first observable action.

[8] The strategy described here does not always terminate. One immediate solution is to bind the number of iterations. For a more detailed discussion, we refer to [20].

```
Maude> (srew builder(3,X:Nat) using do(move(2,0,1)) . )
rewrites: 18463 in 1415ms cpu (1417ms real) (13040 rewrites/second)
rewrite with strategy :
result LBpMentalState :
 << oLabel('move['s_^2['0.Zero],'0.Zero,'s_['0.Zero]]),
    clear(0)# clear(2)# clear(3)# done(0)# length(1)# max(3)#
    on(1,0)# on(2,1)# on(3,0), ...>>
Maude> (next .)
rewrites: 1210 in 10ms cpu (11ms real) (110020 rewrites/second)
next solution rewriting with strategy :
No more solutions .
```

What we obtain is a state reflecting that the agent moved block 2 onto block 1. This can be seen either from the label of the resulting mental state, or from the fact that `on(2,1)` is in the current belief base. Furthermore, we can also notice that this is the only possible resulting mental state since the command `(next .)` for obtaining other solutions returns `No more solutions`.

We recall that our purpose is to test whether "bad" states are reachable from the initial configuration of `builder` and that "bad" means odd length towers in our case. Thus, a suitable test is $move(2,0,1);move(3,0,2);finish(3,0)$, meaning that we test whether the agent (in its faulty variant) executes the action `finish(3,0)` after moving block 2 onto 1 and block 3 onto 2:

```
Maude> (srew builder(3,X:Nat) using
        test2Strat(move(2,0,1) ;a move(3,0,2) ;a finish(3, 0)) .)
rewrites: 50421 in 2069ms cpu (2082ms real) (24361 rewrites/second)
rewrite with strategy :
result LBpMentalState :
 << oLabel('finish['s_^3['0.Zero],'0.Zero]),
    clear(0)# clear(3)# done(3)# length(3)# max(3)#
    on(1,0)# on(2,1)# on(3,2), ...>>
```

The output shows that this is indeed the case, meaning that the agent is not safe to this test. Performing the same test on the correct builder yields no possible rewriting, and from this we can conclude that the correct builder agent is safe with respect to the test.

9.6 Conclusion

In this chapter, we have shown how the Maude term rewriting language can be used for agent development with formal foundations. We have shown how agent programming languages can be prototyped, and how agent programs can be executed, model-checked and tested using Maude and its accompanying tools. We maintain that one of the main advantages of Maude is that it provides a single framework in which the use of a wide range of formal methods is facilitated. This means that the implementation of the semantics of an agent programming language in Maude can

be used for executing agent programs, as well as for model-checking and testing them.

We see several main areas for future research. First, model-checking as described in this chapter applies the model-checker that comes with Maude. This means that it does not include state space abstraction techniques that are specific to agent programming languages. We see the investigation of such techniques and how they can be used in Maude as an important area for future research. Moreover, with respect to testing, the definition of the language to express test cases needs to be further investigated and experimented with to identify exactly which features are useful in practice. Another aspect related to the use of our testing framework in practice is the issue of how to come up with suitable test cases. It will need to be investigated, for example, whether it would be possible to do automatic test case generation.

Acknowledgements

We would like to thank Mehdi Dastani and John-Jules Ch. Meyer for their contributions to work on which this chapter is partly based.