Prototyping 3APL in the Maude Term Rewriting Language

M. Birna van Riemsdijk[1] Frank S. de Boer[1,2,3] Mehdi Dastani[1]
John-Jules Ch. Meyer[1]

[1] ICS, Utrecht University, The Netherlands
[2] CWI, Amsterdam, The Netherlands
[3] LIACS, Leiden University, The Netherlands

Abstract. This paper presents an implementation of (a simplified version of) the cognitive agent programming language 3APL in the Maude term rewriting language. Maude is based on the mathematical theory of rewriting logic. 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. We explore the usage of Maude in the context of agent programming languages, and argue that, since agent programming languages such as 3APL have both a logical and a semantic component, Maude is very well suited for prototyping such languages. Further, we show that, since Maude is reflective, 3APL's meta-level reasoning cycle or deliberation cycle can be implemented very naturally in Maude. Moreover, although we have implemented a simplified version of 3APL, we argue that Maude is very well suited for implementing various extensions of this implemented version. An important advantage of Maude, besides the fact that it is well suited for prototyping agent programming languages, is that it can be used for verification as it comes with an LTL model checker. Although this paper does not focus on model checking 3APL, the fact that Maude provides these verification facilities is an important motivation for our effort of implementing 3APL in Maude.

1 Introduction

An important line of research in the agent systems field is research on agent programming languages [3]. This type of research is concerned with an investigation of what kind of programming constructs an agent programming language should contain, and what exactly the meaning of these constructs should be. In order to test whether these constructs indeed facilitate the programming of agents in an effective way, the programming language has to be implemented.

This can be done using Java, which was for example used for implementing the agent programming language 3APL [13]. Java has several advantages, such as its platform independence, its support for building graphical user interfaces, and the extensive standard Java libraries. A disadvantage is however that the

translation of the formal semantics of an agent programming language such as 3APL into Java is not very direct. It can therefore be difficult to ascertain that such an implementation is a faithful implementation of the semantics of the agent programming language, and experimenting with different language constructs and semantics can be quite cumbersome.

As an alternative to the use of Java, we explore in this paper the usage of the Maude term rewriting language [6] for prototyping 3APL. Maude is based on the mathematical theory of rewriting logic. 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 [17]. We argue that, since agent programming languages such as 3APL have both a logical and a semantic component, Maude is very well suited for prototyping such languages (see Section 5.1). Further, we show that, since Maude is reflective, 3APL's meta-level reasoning cycle or deliberation cycle can be implemented very naturally in Maude (Section 4.2).

An important advantage of Maude is that it can be used for verification as it comes with an LTL model checker [12]. This paper does not focus on model checking 3APL using Maude, for reasons of space and since the usage of Maude's model checker is relatively easy, given the implementation of 3APL in Maude. The fact that Maude provides these verification facilities however, is an important, and was in fact, our original, motivation for our effort of implementing 3APL in Maude.

The outline of this paper is as follows. We present (a simplified version of) 3APL in Section 2, and we briefly explain Maude in Section 3. We explain how we have implemented this simplified version of 3APL in Maude in Section 4. In Section 5, we discuss in more detail the advantages of Maude for the implementation of agent programming languages such as 3APL, and we address related work.

2 3APL

In this section, we present the version of 3APL that we have implemented in Maude. This version comes closest to the one presented in [26]. It is single-agent and builds on propositional logic. We refer to [9,10] for first order, multi-agent, and otherwise extended versions. We have implemented this simple version of 3APL to serve as a proof-of-concept of the usage of Maude for prototyping languages such as 3APL. In Section 5.2, we discuss the possible implementation of various extensions of the version of 3APL as defined in this section, although implementing these is left for future research. It is beyond the scope of this paper to elaborate on the motivations for the various language constructs of 3APL. For this, we refer to the cited papers, which also include example programs.

3APL, which was first introduced by Hindriks [13], is a cognitive agent programming language. This means that it has explicit constructs for representing the high-level mental attitudes of an agent. A 3APL agent has beliefs, a plan,

and goals. Beliefs represent the current state of the world and information internal to the agent. Goals represent the desired state of the world, and plans are the means to achieve the goals. Further, a 3APL agent has rules for selecting a plan to achieve a certain goal given a certain belief, and it has rules for revising its plan during execution. Sections 2.1 and 2.2 formally define the syntax and semantics of the language constructs.

2.1 Syntax

The version of 3APL as presented in this paper takes a simple language, consisting of a set of propositional atoms, as the basis for representing beliefs and goals.

Definition 1 *(base language)* The base language is a set of atoms Atom.

As specified in Definition 7, the belief base and goal base are sets of atoms from Atom.

Below, we define the language of plans. A plan is a sequence of basic actions. Informally, basic actions can change the beliefs of an agent if executed. This simple language of plans could be extended with, e.g., if-then-else and while constructs as was done in [26], but these are straightforward extensions and the language as given here suffices for the purpose of this paper. Abstract plans can be modeled as basic actions that are never executable, i.e., that have a precondition that is always false.

Definition 2 *(plan)* Let $\mathsf{BasicAction}$ with typical element a be the set of basic actions. The set of plans Plan with typical element π is then defined as follows.

$$\pi ::= a \mid \pi_1; \pi_2$$

We use ϵ to denote the empty plan and identify $\epsilon; \pi$ and $\pi; \epsilon$ with π.

In order to be able to test whether an agent has a certain belief or goal, we use belief and goal query languages. These languages are built from the atoms $\mathbf{B}(p)$ and $\mathbf{G}(p)$ for expressing that the agent believes p and has p as a goal, respectively, and negation, disjunction and conjunction. Implication could be defined in terms of these, but this is not used a lot in practice. Therefore, we omit it here.

Definition 3 *(belief and goal query language)* Let $p \in \mathsf{Atom}$. The belief query language $\mathcal{L}_{\mathsf{B}}$ with typical element β, and the goal query language $\mathcal{L}_{\mathsf{G}}$ with typical element κ, are then defined as follows.

$$\beta ::= \top \mid \mathbf{B}(p) \mid \neg\beta \mid \beta_1 \wedge \beta_2 \mid \beta_1 \vee \beta_2$$
$$\kappa ::= \top \mid \mathbf{G}(p) \mid \neg\kappa \mid \kappa_1 \wedge \kappa_2 \mid \kappa_1 \vee \kappa_2$$

The actions of an agent's plan update the agent's beliefs, if executed. In order to specify how actions should update the beliefs, we use so-called action specifications. In papers on 3APL, often a belief update function $\mathcal{T}$ is assumed for this purpose, i.e., the exact definition of $\mathcal{T}$ is usually omitted. Since in this paper we are concerned with implementing 3APL, we also have to be specific about the implementation of belief update through actions.

An action specification is of the form $\{\beta\}a\{Add, Del\}$. Here, a represents the action name, β is a belief query that represents the precondition of the action, and Add and Del are sets of atoms, that should be added to and removed from the belief base, respectively, if a is executed. This way of specifying how actions update beliefs corresponds closely with the way it is implemented in the Java version of 3APL.

Definition 4 *(action specification)* The set of action specifications $\mathcal{AS}$ is defined as follows: $\mathcal{AS} = \{\{\beta\}a\{Add, Del\} \ : \ \beta \in \mathcal{L}_{\mathsf{B}}, a \in \mathsf{BasicAction}, Add \subseteq \mathsf{Atom}, Del \subseteq \mathsf{Atom}\}$.[4]

Plan selection rules are used for selecting an appropriate plan for a certain goal. A plan selection rule is of the form $\beta, \kappa \Rightarrow \pi$. This rule represents that it is appropriate to select plan π for the goals as represented through κ, if the agent believes β.[5]

Definition 5 *(plan selection rule)* The set of plan selection rules $\mathcal{R}_{\mathsf{PS}}$ is defined as follows: $\mathcal{R}_{\mathsf{PS}} = \{\beta, \kappa \Rightarrow \pi \ : \ \beta \in \mathcal{L}_{\mathsf{B}}, \kappa \in \mathcal{L}_{\mathsf{G}}, \pi \in \mathsf{Plan}\}$.

Plan revision rules are used to revise an agent's plan during execution. These rules facilitate the programming of flexible agents which can operate in dynamic domains. A plan revision rule $\pi_h \mid \beta \rightsquigarrow \pi_b$ represents that in case the agent believes β, it can replace the plan π_h by the plan π_b.

Definition 6 *(plan revision rule)* The set of plan revision rules $\mathcal{R}_{\mathsf{PR}}$ is defined as follows: $\mathcal{R}_{\mathsf{PR}} = \{\pi_h \mid \beta \rightsquigarrow \pi_b \ : \ \beta \in \mathcal{L}_{\mathsf{B}}, \pi_h, \pi_b \in \mathsf{Plan}\}$.

The notion of a configuration is used to represent the state of a 3APL agent at each point during computation. A configuration consists of a belief base σ and a goal base γ which are both sets of atoms, a plan, and sets of plan selection rules, plan revision rules, and action specifications.

Definition 7 *(configuration)* A 3APL configuration is a tuple $\langle \sigma, \pi, \gamma, \mathsf{PS}, \mathsf{PR}, \mathsf{AS} \rangle$ where $\sigma \subseteq \mathsf{Atom}$ is the belief base, $\pi \in \mathsf{Plan}$ is the plan, $\gamma \subseteq \mathsf{Atom}$ is the goal base, $\mathsf{PS} \subseteq \mathcal{R}_{\mathsf{PS}}$ is a set of plan selection rules, $\mathsf{PR} \subseteq \mathcal{R}_{\mathsf{PR}}$ is a set of plan revision rules, and $\mathsf{AS} \subseteq \mathcal{AS}$ is a set of action specifications.

Programming a 3APL agent comes down to specifying its initial configuration.

[4] We use the notation $\{\ldots \ : \ \ldots\}$ instead of $\{\ldots \mid \ldots\}$ to define sets, to prevent confusing usage of the symbol | in Definition 6.

[5] Note that it is up to the programmer to specify appropriate plans for a certain goal. 3APL agents do not do planning from first principles.

2.2 Semantics

The semantics of 3APL agents is defined by means of a transition system [22]. A transition system for a programming language consists of a set of axioms and derivation rules for deriving transitions for this language. A transition is a transformation of one configuration into another and it corresponds to a single computation step. In the configurations of the transitions below, we omit the sets of plan revision rules PR, plan selection rules PS, and action specifications AS for reasons of presentation, i.e., we use configurations of the form $\langle \sigma, \pi, \gamma \rangle$ instead of $\langle \sigma, \pi, \gamma, \mathsf{PS}, \mathsf{PR}, \mathsf{AS} \rangle$. This is not problematic, since these sets do not change during execution of the agent.

Before moving on to defining the transition rules for 3APL, we define the semantics of belief and goal queries. The satisfaction relations $\models_{\mathcal{L}_\mathsf{B}}$ and $\models_{\mathcal{L}_\mathsf{G}}$ are used for this purpose. Belief and goal queries are evaluated in a configuration. A formula $\mathbf{B}(p)$ is true in a configuration iff p is in the belief base, and $\mathbf{G}(p)$ is true iff p is in the goal base. The semantics of negation, disjunction, and conjunction are defined in the obvious way, which we omit for reasons of space.

Definition 8 *(belief and goal queries)*

$$\langle \sigma, \pi, \gamma \rangle \models_{\mathcal{L}_\mathsf{B}} \mathbf{B}(p) \Leftrightarrow p \in \sigma$$
$$\langle \sigma, \pi, \gamma \rangle \models_{\mathcal{L}_\mathsf{G}} \mathbf{G}(p) \Leftrightarrow p \in \gamma$$

The first transition rule as specified below is used to derive a transition for action execution. An action a that is the first action of the plan, can be executed if there is an action specification for a, and the precondition of this action as specified in the action specification holds. The belief base σ is updated such that the atoms of Add are added to, and the atoms of Del are removed from σ. Further, the atoms that have been added to the belief base should be removed from the goal base, as the agent believes these goals to be achieved. Also, the action is removed from the plan.

Definition 9 *(action execution)* The transition for action execution is defined as follows:

$$\frac{\{\beta\} a \{Add, Del\} \in \mathsf{AS} \qquad \langle \sigma, a; \pi, \gamma \rangle \models_{\mathcal{L}_\mathsf{B}} \beta}{\langle \sigma, a; \pi, \gamma \rangle \rightarrow \langle \sigma', \pi, \gamma' \rangle}$$

where $\sigma' = (\sigma \cup Add) \setminus Del$, and $\gamma' = \gamma \setminus Add$.

An agent can apply a plan selection rule $\beta, \kappa \Rightarrow \pi$ if it has an empty plan. The idea is that the agent can only select a new plan, if it has completed the execution of a previous plan. Further, the conditions β and κ have to hold in order for the rule to be applicable. If the rule is applied, the plan π becomes the plan of the agent.

Definition 10 *(plan selection rule application)*

$$\frac{\beta, \kappa \Rightarrow \pi \in \mathsf{PS} \qquad \langle \sigma, \epsilon, \gamma \rangle \models_{\mathcal{L}_\mathsf{B}} \beta \qquad \langle \sigma, \epsilon, \gamma \rangle \models_{\mathcal{L}_\mathsf{G}} \kappa}{\langle \sigma, \epsilon, \gamma \rangle \rightarrow \langle \sigma, \pi, \gamma \rangle}$$

The transition below specifies the application of a plan revision rule of the form $\pi_h \mid \beta \Rightarrow \pi_b$ to a plan of the form $\pi_h; \pi$. The rule can be applied if β holds. If the rule is applied, the plan π_h is replaced by the body of the rule, yielding the plan $\pi_b; \pi$.

Definition 11 *(plan revision rule application)*

$$\frac{\pi_h \mid \beta \rightsquigarrow \pi_b \in \mathsf{PR} \qquad \langle \sigma, \pi_h; \pi, \gamma \rangle \models_{\mathcal{L}_\mathsf{B}} \beta}{\langle \sigma, \pi_h; \pi, \gamma \rangle \rightarrow \langle \sigma, \pi_b; \pi, \gamma \rangle}$$

3 Maude

We cite from [21]: "Maude is a formal declarative programming language based on the mathematical theory of rewriting logic [18]. Maude and rewriting logic were both developed by José Meseguer. Maude is a state-of-the-art formal method in the fields of algebraic specification [28] and modeling of concurrent systems. The Maude language specifies rewriting logic theories. Data types are defined algebraically by equations and the dynamic behavior of a system is defined by rewrite rules which describe how a part of the state can change in one step."

A rewriting logic specification consists of a *signature*, a set of *equations*, and a set of *rewrite rules*. The signature specifies the *terms* that can be rewritten using the equations and the rules. Maude supports membership equational logic [19], which is an extension of order-sorted equational logic, which is in turn an extension of many-sorted equational logic. For this paper, it suffices to treat only the many-sorted subset of Maude. A signature in many-sorted equational logic consists of a set of *sorts*, used to distinguish different types of values, and a set of *function symbols* declared on these sorts.

In Maude, sorts are declared using the keyword `sort`, for example as follows: `sort List`. Function symbols can be declared as below, using the keywords `op` and `ops`.

```
op app : Nat List -> List .
ops 0 1 2 3 : -> Nat .
op nil : -> List .
```

The function `app`, expressing that natural numbers can be appended to form a list, takes an argument of sort `Nat` and an argument of sort `List`, and the resulting term is again of sort `List`. The functions `0`, `1`, `2` and `3` are nullary functions, i.e., constants, of sort `Nat`. The nullary function `nil` represents the empty list. An example of a term (of sort `List`) over this signature is `app(1,app(2,app(3,nil)))`.

In order to define functions declared in the signature, one can use equations. An equation in Maude has the general form `eq` ⟨`Term-1`⟩ `=` ⟨`Term-2`⟩. Assume a function declaration `op sum : List -> Nat`, and a function `+` for adding natural numbers (declared as `op _+_ : Nat Nat -> Nat`, in which the underscores are used express infix use of `+`). Further, assume variable declarations `var N`

`: Nat` and `var L : List`, expressing that `N` and `L` are variables of sorts `Nat` and `List` respectively. The equations `eq sum(app(N,L)) = N + sum(L)` and `eq sum(nil) = 0` can then be used to define the function `sum`.

Maude also supports conditional equations, which have the following general form.

```
ceq ⟨Term-1⟩ = ⟨Term-2⟩
   if ⟨EqCond-1⟩ /\ ... /\ ⟨EqCond-n⟩
```

A condition can be either an ordinary equation of the form `t = t'`, a matching equation of the form `t := t'`, or an abbreviated boolean equation of the form `t`, which abbreviates `t = true`. An example of the use of a matching equation as the condition of a conditional equation is `ceq head(L) = N if app(N,L') := L`. This equation defines the function `head`, which is used to extract the first element of a list of natural numbers. The matching equation `app(N,L') := L` expresses that `L`, as used in the lefthand side of the equation, has to be of the form `app(N,L')`, thereby binding the first element of `L` to `N`, which is then used in the righthand side of the equation.

Operationally, equations can be applied to a term from left to right. Equations in Maude are assumed to be terminating and confluent,[6] 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. A rewrite rule in Maude has the general form `rl [⟨Label⟩] : ⟨Term-1⟩ => ⟨Term-2⟩`, expressing that term `Term-1` can be rewritten into term `Term-2`. Conditional rewrite rules have the following general form.

```
crl [⟨Label⟩] ⟨Term-1⟩ => ⟨Term-2⟩
   if ⟨Cond-1⟩ /\ ... /\ ⟨Cond-n⟩
```

Conditions can be of the type as used in conditional equations, or of the form `t => t'`, which expresses that it is possible to rewrite term `t` to term `t'`. An example of a rewrite rule is `rl [duplicate] : app(N,L) => app(N,app(N,L))`, which expresses that a list with head `N` can be rewritten into a new list with `N` duplicated. The term `app(1,app(2,app(3,nil)))` can for example be rewritten to the term `app(1,app(1,app(2,app(3,nil))))` using this rule. The former term can however also be rewritten into `app(1,app(2,app(2,app(3,nil))))`, because rewrite rules (and equations alike) can be applied to subterms.

The way the Maude interpreter executes rewriting logic specifications, is as follows [21]. Given a term, Maude tries to apply equations from left to right to this term, until no equation can be applied, thereby computing the normal form of a term. Then, an applicable rewrite rule is arbitrarily chosen and applied (also from left to right). This process continues, until no rules can be applied.

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

Equations are thus applied to reduce each intermediate term to its normal form before a rewrite rule is applied.

Finally, we remark that in Maude, rewriting logic specifications are grouped into modules with the following syntax: `mod` ⟨`Module-Name`⟩ `is` ⟨`Body`⟩ `endm`. Here, ⟨`Body`⟩ contains the sort and variable declarations and the (conditional) equations and rewrite rules.

4 Implementation of 3APL in Maude

In this section, we describe how we have implemented 3APL in Maude. We distinguish the implementation of 3APL as defined in Section 2, which we will refer to as object-level 3APL (Section 4.1), and the implementation of a meta-level reasoning cycle (Section 4.2).

4.1 Object-Level

The general idea of the implementation of 3APL in Maude, is that 3APL configurations are represented as terms in Maude, and the transition rules of 3APL are mapped onto rewrite rules of Maude. This idea is taken from [27], in which, among others, implementations in Maude of the operational semantics of a simple functional language and an imperative language are discussed. In this section we describe in some detail how we have implemented 3APL, thereby highlighting 3APL-specific issues.

Syntax Each component of 3APL's syntax as specified in Definitions 1 through 6 is mapped onto a module of Maude. As an example, we present the definition of the module for the belief query language, corresponding with Definition 3.

```
mod BELIEF-QUERY-LANGUAGE is
   including BASE-LANGUAGE .

   sort BQuery .

   op B : LAtom -> BQuery .
   op top : -> BQuery .
   op  ~_  : BQuery -> BQuery .
   op  _/\_  : BQuery BQuery -> BQuery .
   op  _\/_  : BQuery BQuery -> BQuery .

endm
```

The module `BELIEF-QUERY-LANGUAGE` imports the module used to define the base language of Definition 1. A sort `BQuery` is declared, representing elements from the belief query language. The sort `LAtom` is declared in the module `BASE-LANGUAGE`, and represents atoms from the base language. Five operators

are defined for building belief query formulas, which correspond with the operators of Definition 3. The other syntax modules are defined in a similar way. Note that only sort and function declarations are used in syntax modules. None of the syntax modules contain equations or rewrite rules.

The notion of configuration as specified in Definition 7 is also mapped onto a Maude module. This module imports the other syntax modules, and declares a sort `Conf` and an operator `op <_,_,_,_,_,_> : BeliefBase Plan GoalBase PSbase PRbase ASpecs -> Conf`.

Semantics The implementation of the semantics of 3APL in Maude can be divided into the implementation of the *logical* part, i.e., the belief and goal queries as specified in Definition 8, and the *operational* part, i.e., the transition rules of Definitions 9 through 11. The logical part, i.e., the semantics of the satisfaction relations $\models_{\mathcal{L}_B}$ and $\models_{\mathcal{L}_G}$, is modelled as equational specifications, whereas the transition rules of the operational part are translated into rewrite rules.

As an example of the modeling of the logical part, we present part of the module for the semantics of $\models_{\mathcal{L}_B}$ below. Here `[owise]` is a built-in Maude construct that stands for "otherwise".

```
mod BELIEF-QUERY-SEMANTICS is
   including BELIEF-QUERY-LANGUAGE .

   op  _|=LB_  : BeliefBase BQuery -> Bool .

   var p : LAtom .
   vars BB BB' : BeliefBase .
   vars BQ : BQuery .

   ceq BB |=LB B(p) = true if p BB' := BB .
   eq BB |=LB B(p) = false [owise] .

   ceq BB |=LB ~BQ = true if not BB |=LB BQ .
   eq BB |=LB ~BQ = false [owise] .
   ...

endm
```

The relation $\models_{\mathcal{L}_B}$ is modeled as a function `|=LB`, which takes a belief base of sort `BeliefBase` (a sort from the base language module), and a belief query of sort `BQuery`, and yields a boolean, i.e., `true` or `false`. Although the semantics of belief queries as specified in Definition 8 is defined on configurations rather than on belief bases, it is in fact only the belief base part of the configuration that is used in the semantic definition. For ease of specification we thus define the function `|=LB` on belief bases, rather than on configurations.

The first pair of (conditional) equations defines the semantics of a belief query `B(p)`. Belief bases are defined as associative and commutative space-separated sequences of atoms. The matching equation `p BB' := BB` expresses that belief base `BB` is of the form `p BB'` for some belief base `BB'`, i.e., that the atom `p` is part of `BB`. The second pair of (conditional) equations specifies the semantics of a negative query `~BQ`. The term `not BB |=LB BQ` is an abbreviated boolean equation, i.e., it abbreviates `not BB |=LB BQ = true`, and `not` is a built-in boolean connective. The module for the semantics of goal query formulas is defined in a similar way.

We now move on to the implementation of the operational part. Below, we present the rewrite rule for action execution, corresponding with the transition rule of Definition 9. The variables `B`, `B'` and `B''` are of sort `BeliefBase`, `A` is of sort `Action`, `P` is of sort `Plan`, and `G` and `G'` are of sort `GoalBase`. Moreover, `PSB`, `PRB`, and `AS` are respectively of sorts `PSbase`, `PRbase`, and `ASpecs`. Further, `Pre` is of sort `BQuery` and `Add` and `Del` are of sort `AtomList`.

```
crl [exec] : < B, A ; P, G, PSB, PRB, AS > =>
             < B', P, G', PSB, PRB, AS >
   if {Pre} A {Add,Del} AS' := AS  /\  B |=LB Pre /\ B'' := B U Add  /\
      B' := B'' \ Del  /\  G' := G \ Add .
```

The transition as specified in the conclusion of the transition rule of Definition 9 is mapped directly to the rewrite part of the conditional rewrite rule.[7] The conditions of the transition rule, and the specification of how belief base and goal base should be changed, are mapped onto the conditions of the rewrite rule.

The first condition of the rewrite rule corresponds with the first condition of the transition rule. It specifies that if action `A` is to be executed, there should be an action specification for `A` in the set of action specifications `AS`. The second condition of the rewrite rule corresponds with the second condition of the transition rule, and specifies that the precondition of the action should hold. Note that the previously defined satisfaction relation `|=LB` is used here.

The third and fourth conditions of the rewrite rule specify how the belief base is changed, if the action `A` is executed. For this, a function `U` (union) has been defined using equations, which we omit here for reasons of space. This function takes a belief base and a list of atoms, and adds the atoms of the list to the belief base, thereby making sure that no duplicate atoms are introduced in the belief base. The function `\` for deleting atoms is defined in a similar way, and is also used for updating the goal base as specified in the last condition.

The translation of the transition rules for plan selection and plan revision rule application is done in a similar way. As an illustration, we present the rewrite rule for plan revision, corresponding with the transition rule of Definition 11. The variables `Ph` and `Pb` are of sort `Plan`, and `PRB'` is of sort `PRbase`. The syntax `(Ph | BQ -> Pb)` is used for representing a plan revision rule of the form $\pi_h \mid \beta \rightsquigarrow \pi_b$.

[7] Recall that the plan selection and plan revision rule bases and the action specifications were omitted from Definitions 9 through 11 for reasons of presentation.

```
crl [apply-pr] : < B, Ph ; P, G, PSB, PRB, AS > =>
                    < B, Pb ; P, G, PSB, PRB, AS >
  if (Ph | BQ -> Pb) PRB' := PRB  /\  B |=LB BQ .
```

As was the case for action execution, the transition as specified in the conclusion of the transition rule of Definition 11 is mapped directly onto the rewrite part of the conditional rewrite rule. The conditions of the transition rule furthermore correspond to the conditions of the rewrite rule.

Above, we have discussed the Maude modules for specifying the syntax and semantics of 3APL. In order to run a concrete 3APL program using Maude, one has to create another module for this program. In this module, one needs to specify the initial belief base, goal base, etc. For this, the atoms as can be used in, e.g., the belief base have to be declared as (nullary) operators of sort `LAtom`. Also, the possible basic actions have to be declared. Then, the initial configuration has to be specified. This can be conveniently done by declaring an operator for each component of the configuration, and specifying the value of that component using an equation. An initial belief base containing the atoms p and q can for example be specified using `eq bb = p q`, where `bb` is a nullary operator of sort `BeliefBase`, and `p` and `q` are atoms. In a similar way, the initial plan, goal base, rule bases, and action specifications can be defined. The 3APL program as thus specified can be executed by calling Maude with the command `rewrite <bb, plan, gb, psb, prb, as>`, where `<bb, plan, gb, psb, prb, as>` is the initial configuration.

4.2 Meta-Level

Given the transition system of 3APL as defined in Section 2.2, different possible executions might be derivable, given a certain initial configuration. It might for example be possible to execute an action in a certain configuration, as well as to apply a plan revision rule. The transition system does not specify which transition to choose during the execution. An implementation of 3APL corresponding with this transition system might non-deterministically choose a possible transition. The implementation of 3APL in Maude does just this, as Maude arbitrarily chooses an applicable rewrite rule for application.

In some cases however, it can be desirable to have more control over the execution. This can be achieved by making it possible to specify more precisely which transition should be chosen, if multiple transitions are possible. In the case of 3APL, meta-level languages have been introduced for this purpose (see [13,8]). These meta-languages have constructs for specifying that an action should be executed or that a rule should be applied. Using a meta-language, various so-called *deliberation cycles* can be programmed.

A deliberation cycle can for example specify that the following process should be repeated: first apply a plan selection rule (if possible), then apply a plan revision rule, and then execute an action. Alternatively, a deliberation cycle could for example specify that a plan revision rule can only be applied if it is

not possible to execute an action. It might depend on the application which is an appropriate deliberation cycle.

It turns out that this kind of meta-programming can be modeled very naturally in Maude, since rewriting logic is *reflective* [7]. "Informally, a reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object level representation correctly simulates the relevant metatheoretic aspects." [6, Chapter 10]

In order to perform meta-level computation, terms and modules of the object-level have to be represented as Maude terms on the meta-level, i.e., they have to be *meta-represented*. For this, Maude has predefined modules, that include the functions `upTerm` and `upModule` for meta-representing terms and modules, and the function `metaXapply` which defines the meta-level application of a rewrite rule[8].

The function `upTerm` takes an (object-level) term and yields the meta-representation of this term, i.e., a term of sort `Term`. The function `upModule` takes the meta-representation of the name of a module, i.e., the name of a module with a quote prefixed to it, and yields the meta-representation of the module with this name, i.e., a term of sort `Module`. The function `metaXapply` takes the meta-representation of a module, the meta-representation of a term, the meta-representation of a rule label (i.e., the rule label with a quote prefixed to it), and some more arguments which we do not go into here, as they are not relevant for understanding the general idea. The function tries to rewrite the term represented by its second argument using the rule as represented by its third argument. A rule with the label as given as the third argument of the function, should be part of the module as represented by the function's first argument. The function returns a term of sort `Result4Tuple?`. If the rule application was successful, i.e., if the rule could be applied to the term, the function returns a 4-tuple of sort `Result4Tuple`,[9] which contains, among other information, the term resulting from the rule application. This term can be retrieved from the tuple using the function `getTerm`, which returns the meta-representation of the term of sort `Term` resulting from the rewrite rule application.

Meta-level function calls, such as the application of a certain rewrite rule through `metaXapply`, can be combined to form so-called *strategies* [6]. These strategies can be used to define the execution of a system at the meta-level. Deliberation cycles of 3APL can be programmed as these strategies.

An example of a deliberation cycle implemented in Maude is specified below using the function `one-cycle`. It first tries to apply a plan selection rule, then to execute an action and then to apply a plan revision rule. The function `one-cycle` only specifies one sequence of applications of reasoning rules and action executions. It is repeated to form a deliberation cycle using the function `cycle`, which is specified below.

---

[8] Maude also has a function `metaApply` for this purpose with a slightly different meaning [6]. It is however beyond the scope of this paper to explain the difference.

[9] Note that the difference with the sort `Result4Tuple?` is the question mark. The sort `Result4Tuple` is a subsort of the sort `Result4Tuple?`.

```
ceq one-cycle(Meta-Conf, Meta-Prog) = Meta-Conf'
   if Meta-Conf' := try-meta-apply-pr(try-meta-exec(try-meta-apply-ps(
                    Meta-Conf, Meta-Prog), Meta-Prog), Meta-Prog) .
```

Here, `Meta-Conf` and `Meta-Conf'` are variables of sort `Term` which stand for the meta-representations of 3APL configurations, and `Meta-Prog` is a variable of sort `Module`, which should be instantiated with the meta-representation of the module with Maude code of a 3APL program[10]. The variable `Meta-Conf` is input to the function `one-cycle`, and `Meta-Conf'` represents the result of applying the function `one-cycle` to `Meta-Conf` and `Meta-Prog`. This module imports the syntax and semantics modules, which are also meta-represented in this way. The functions `try-meta-apply-pr`, `try-meta-exec`, and `try-meta-apply-ps` try to apply the (object-level) rewrite rules for plan revision, action execution, and plan selection, respectively. In the definitions of these functions, the pre-defined function `metaXapply` is called, with the names of the respective object-level rewrite rules as one of its arguments, i.e., with, respectively, `apply-pr`, `exec`, and `apply-ps`.

Before giving an example of these functions, we show how the function `one-cycle` can be iterated to form a deliberation cycle. The function `cycle` applies the function `one-cycle` repeatedly to `Meta-Conf` and `Meta-Prog`, until the application of `one-cycle` does not result in a change to the configuration `Meta-Conf`, which means the 3APL program has terminated.

```
ceq cycle(Meta-Conf, Meta-Prog) = cycle(Meta-Conf', Meta-Prog)
   if Meta-Conf' := one-cycle(Meta-Conf, Meta-Prog)  /\
                    Meta-Conf' =/= Meta-Conf .

eq cycle(Meta-Conf,Meta-Prog) = Meta-Conf [owise] .
```

As an example of the functions for applying object-level rewrite rules, we present the definition of the function `try-meta-apply-pr`.

```
ceq try-meta-apply-pr(Meta-Conf, Meta-Prog) =
        if Result? :: Result4Tuple
        then getTerm(Result?)
        else Meta-Conf
        fi
   if Result? := metaXapply(Meta-Prog, Meta-Conf, 'apply-pr, ...) .
```

The variable `Result?` is of sort `Result4Tuple?`. The function `metaXapply` takes the meta-representation of a module representing a 3APL program,[11] the meta-representation of a configuration, and the meta-representation of the label of

[10] Note that by "3APL program", we mean the Maude representation of a concrete 3APL program, i.e., consisting of specific sets of plan selection rules, plan revision rules, etc.

[11] The modules defining the syntax and semantics of 3APL are imported by this module, and are therefore also meta-represented.

the plan revision rewrite rule, i.e., `'apply-pr`, and yields the result of applying the plan revision rewrite rule to the configuration. If the rule application was successful, i.e., if `Result?` is of sort `Result4Tuple`, the term of the resulting 4-tuple which meta-represents the new configuration, is returned. Otherwise, the original unmodified configuration is returned. Note that there is only one object-level rule for plan revision (see Section 4.1), and that this is the one referred to in the definition of the function `try-meta-apply-pr`. Nevertheless, there might be multiple ways of applying this rule, since potentially multiple plan revision rules are applicable in a configuration. The function `metaXapply` then takes the first instance it finds.

A 3APL program can be executed through a deliberation cycle by calling Maude with the command.[12]

```
rewrite cycle(upTerm(conf),upModule('3APL-PROGRAM)) .
```

The term `conf` represents the initial configuration of the 3APL program, and `'3APL-PROGRAM` is the meta-representation of the name of the module containing the 3APL program.

5 Discussion and Related Work

5.1 Experiences in Using Maude

Based on our experience with the implementation of 3APL in Maude as elaborated on in Section 4, we argue in this section that Maude is well suited as a prototyping and analysis tool for logic based cognitive agent programming languages.

Advantages of Maude In [17], it is argued that rewriting logic is suitable both as a *logical* framework in which many other logics can be represented, and as a *semantic* framework.[13] The paper shows how to map Horn logic and linear logic in various ways to rewriting logic, and, among other things, it is observed that operational semantics can be naturally expressed in rewriting logic. The latter has been demonstrated from a more practical perspective in [27], by demonstrating how simple functional, imperative, and concurrent languages can be implemented in Maude.

In this paper, we show how (a simple version of) 3APL can be implemented in Maude. We observe that cognitive agent programming languages such as 3APL have a logical *as well as* a semantic component: the logical part consists of the belief and goal query languages (together with their respective satisfaction relations), and the semantic part consists of the transition system. Since Maude supports both the logical and the semantic component, the implementation of

[12] We omit some details for reasons of clarity.

[13] Obviously, logics often have semantics, but the notion of a *semantic framework* used in the cited paper refers to semantics of programming languages.

languages like 3APL in Maude is very natural, and the integration of the two components is seamless.

We observe that the direct mapping of transition rules of 3APL into rewrite rules of Maude ensures a *faithful* implementation of the operational semantics of 3APL in Maude. This direct mapping is a big advantage compared with the implementation of a 3APL interpreter in a general purpose language such as Java, in which the implementation is less direct. In particular, in Java one needs to program a mechanism for applying the specified transition rules in appropriate ways, whereas in the case of Maude the term rewriting engine takes care of this. As another approach of implementing a cognitive agent programming language in Java, one might consider to implement the plans of the agent as methods in Java, which is for example done in the Jadex framework [23]. Since Java does not have support for revision of programs, implementing 3APL plans as methods in Java is not possible. We refer to [25] for a theoretical treatment of the issues with respect to semantics of plan revision.

A faithful implementation of 3APL's semantics in Maude is very important with regard to our main original motivation for this work, i.e., to use the Maude LTL model checker to do formal verification for 3APL. The natural and transparent way in which the operational semantics of 3APL can be mapped to Maude, is a big advantage compared with the use of, e.g., the PROMELA language [14] in combination with the SPIN model checker [15].

SPIN is a generic verification system which supports the design and verification of asynchronous process systems. SPIN verification models are focused on proving the correctness of process interactions, and they attempt to abstract as much as possible from internal sequential computations. The language PROMELA is a high level language for specifying abstractions of distributed systems which can be used by SPIN, and it's main data structure is the message channel. In [4], an implementation of the cognitive agent programming language AgentSpeak(F) - the finite state version of AgentSpeak(L) [20] - in PROMELA is described, for usage with SPIN. Most of the effort is devoted to translating AgentSpeak(F) into the PROMELA data structures. It is shown how to translate the data structures of AgentSpeak(F) into PROMELA channels. It is however not shown that this translation is correct, i.e., that the obtained PROMELA program correctly simulates the AgentSpeak(F) semantics. In contrast with the correctness of the implementation of 3APL in Maude, the correctness of the AgentSpeak(F) implementation in PROMELA is not obvious, because of the big gap between AgentSpeak(F) data structures and semantics, and PROMELA data structures and semantics.

In [12], it is demonstrated that the performance of the Maude model checker "is comparable to that of current explicit-state model checkers" such as SPIN. The cited paper evaluates the performance of the Maude model checker against the performance of SPIN by taking a number of given systems specified in PROMELA, and implementing these in Maude. Then, for a given model checking problem, the running times as well as memory consumptions of SPIN and of the Maude model checker were compared on the respective specifications.

A further important advantage of Maude is that deliberation cycles can be programmed very naturally as strategies, using reflection. A related advantage is that a clear separation of the object-level and meta-level semantics can be maintained in Maude *because* the meta-level reasoning cycle can be implemented separately from the object-level semantics, using reflection. A 3APL program can be executed without making use of a deliberation cycle, while it can equally easily be executed *with* a deliberation cycle.

Finally, we have found that learning Maude and implementing this simplified version of 3APL in Maude can be done in a relatively short amount of time (approximately two weeks). We cannot compare this effort with the implementation of the 3APL platform in Java where the implementation time is concerned. This is because the platform in Java implements an interpreter for full 3APL [9] and provides a number of other features, such as graphic user interfaces. Nevertheless, we believe that the implementation of the simplified version of 3APL in Maude can be extended very naturally in various ways. This is discussed in Section 5.2.

Advantages of Java over Maude As stated in the introduction, Java has several advantages over Maude such as its platform independence, its support for building graphical user interfaces, and the extensive standard Java libraries. Support for building, e.g., graphical user interfaces, is very important when it comes to building a platform of which the most important aim is to allow the implementation of agent systems in a certain (agent programming) language. Such a platform should ideally implement a (relatively) stable version of a programming language. However, in the process of designing a language, it is important to be able to implement it rapidly in order to be able to test it. Maude is very well suited for this, since logics as well as operational semantics can be translated naturally into Maude, providing a prototype that faithfully implements the designed language.

We thus advocate the usage of Maude primarily for rapid prototyping of cognitive agent programming languages. Once the agent programming language has reached a point where it is reasonably stable, it might be desirable to implement a supporting platform in a language such as Java. While such a platform should be well suited for *testing* an agent program, one may again want to use Maude with its accompanying model checker when it comes to *verifying* this program. One might have to abstract over certain aspects of the agent programming language as implemented in Java, such as calls to Java from plans in the case of 3APL, although there is recent work describing an implementation of an operational semantics of Java in Maude [1].

5.2 Extending the Implementation

As was explained in Section 2, this paper presents an implementation of a simplified version of 3APL in Maude. We however argue that the features of Maude are very well suited to support the implementation of various extensions of this version of 3APL. Implementing these extensions is left for future research.

In particular, an extension of a single-agent to a *multi-agent* version will be naturally implementable, since, from a computational point of view, rewriting logic is intrinsically concurrent [17]. It was in fact the search for a general concurrency model that would help unify the heterogenity of existing models, that provided the original impetus for the first investigations on rewriting logic [18].

Further, a more practically useful implementation will have to be *first-order*, rather than propositional. Although the implementation of a first-order version will be more involved, it can essentially be implemented in the same way as the current version, i.e., by mapping transition rules to rewrite rules. In [9], the transition rules for a first-order version of 3APL are presented. Configurations in this setting have an extra substitution component, which records the assignment of values to variables. An implementation of this version in Maude will involve extending the notion of a configuration with such a substitution component, as specified in the cited paper.

Finally, we aim to extend the *logical part* in various ways, for which, as already pointed out, Maude is very well suited. Regarding this propositional version, one could think of extending the belief base and goal base to arbitrary sets of propositional formulas, rather than just sets of atoms. Also, the belief and goal query languages could be extended to query arbitrary propositional formulas. The satisfaction relations for queries could then be implemented using, e.g., tableau methods as suggested in [17], for checking whether a propositional formula follows from the belief or goal base. Further, when considering a first-order version of 3APL, the belief base can be implemented as a set of Horn clauses, or even as a Prolog program. In the current Java implementation of 3APL, the belief base is implemented as a Prolog program. How to define standard Prolog in rewriting logic has been described in [16]. Finally, we aim to experiment with the implementation of more sophisticated specifications of the goals of 3APL agents and their accompanying satisfaction relations, such as proposed in [24].

An aspect of 3APL as implemented in Java that is not easily implemented in Maude, are the actions by means of which a Java method can be called from the plan of a 3APL agent. The execution of a method may return a value to the 3APL program. A similar mechanism could be implemented in Maude by introducing the ability to access built-in Maude functions from the plans of the agent.

5.3 Related Work

Besides the related work as already discussed in Sections 5.1 and 5.2, we mention a number of papers on Maude and agents. To the best of our knowledge, Maude has not been used widely in the agent community, and in particular not in the area of agent programming languages. Nevertheless, we found a small number of papers describing the usage of Maude in the agent systems field, which we will briefly discuss in this section.

A recent paper describes the usage of Maude for the specification of DIMA multi-agent models [5]. In that paper, the previously not formalized DIMA model of agency is formalized using Maude. This work thus differs from our approach

in that it does not implement an agent *programming language* which already has a formal semantics, independent of Maude. Consequently, its techniques for implementation are not based on the idea that transition rules can be translated into rewrite rules.

Further, Maude has been used in the mobile agent area for checking fault-tolerant agent-based protocols used in the DaAgent system [2]. Protocols in the DaAgent system are related to mobility issues, such as detection of node failure. The authors remark that the Java implementation for testing their protocols has proved to be "extremely time-consuming and inflexible". Using Maude, the protocol specifications are formalized and they can be debugged using the Maude model checker. Another example of the usage of Maude in the mobile agent area is presented in [11]. In that paper, Mobile Maude is presented, which is a mobile agent language extending Maude, and supporting mobile computation.

References

1. W. Ahrendt, A. Roth, and R. Sasse. Automatic validation of transformation rules for Java verification against a rewriting semantics. In G. Sutcliffe and A. Voronkov, editors, *Proceedings, 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Montego Bay, Jamaica*, volume 3835 of *LNCS*, pages 412–426. Springer, Dec 2005.
2. J. V. Baalen, J. L. Caldwell, and S. Mishra. Specifying and checking fault-tolerant agent-based protocols using Maude. In *FAABS '00: Proceedings of the First International Workshop on Formal Approaches to Agent-Based Systems-Revised Papers*, volume 1871 of *LNCS*, pages 180–193, London, UK, 2001. Springer-Verlag.
3. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin, 2005.
4. R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking AgentSpeak. In *Proceedings of the second international joint conference on autonomous agents and multiagent systems (AAMAS'03)*, pages 409–416, Melbourne, 2003.
5. N. Boudiaf, F. Mokhati, M. Badri, and L. Badri. Specifying DIMA multi-agent models using Maude. In *Intelligent Agents and Multi-Agent Systems, 7th Pacific Rim International Workshop on Multi-Agents (PRIMA 2004)*, volume 3371 of *LNCS*, pages 29–42. Springer, Berlin, 2005.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude manual (version 2.1.1). 2005.
7. M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. *Electronic Notes in Theoretical Computer Science*, 4:125–147, 1996.
8. M. Dastani, F. S. de Boer, F. Dignum, and J.-J. Ch. Meyer. Programming agent deliberation – an approach illustrated using the 3APL language. In *Proceedings of the second international joint conference on autonomous agents and multiagent systems (AAMAS'03)*, pages 97–104, Melbourne, 2003.
9. M. Dastani, M. B. van Riemsdijk, F. Dignum, and J.-J. Ch. Meyer. A programming language for cognitive agents: goal directed 3APL. In *Programming multiagent systems, first international workshop (ProMAS'03)*, volume 3067 of *LNAI*, pages 111–130. Springer, Berlin, 2004.

10. M. Dastani, M. B. van Riemsdijk, and J.-J. Ch. Meyer. Programming multi-agent systems in 3APL. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin, 2005.
11. F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of mobile Maude. In *ASA/MA 2000: Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*, volume 1882 of *LNCS*, pages 73–85, London, UK, 2000. Springer-Verlag.
12. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gaducci and U. Montanari, editors, *Proceedings of the 4th International Workshop on Rewriting Logic and Its Applications (WRLA 2002)*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.
13. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in 3APL. *Int. J. of Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
14. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, New Jersey, 1991.
15. G. Holzmann. The model checker SPIN. *IEEE Trans. Software Engineering*, 23(5):279–295, 1997.
16. M. Kulas and C. Beierle. Defining standard Prolog in rewriting logic. In *Electronic Notes in Theoretical Computer Science*, volume 36. Elsevier Science Publishers, 2000.
17. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.
18. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
19. J. Meseguer. Membership algebra as a logical framework for equational specification. In *WADT '97: Selected papers from the 12th International Workshop on Recent Trends in Algebraic Development Techniques*, pages 18–61, London, UK, 1997. Springer-Verlag.
20. A. Moreira and R. Bordini. An operational semantics for a BDI agent-oriented programming language. In *Proceedings of the Workshop on Logics for Agent-Based Systems (LABS'02)*, 2002.
21. P. C. Ölveczky. Formal modeling and analysis of distributed systems in Maude. Lecture Notes, 2005.
22. G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
23. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: a BDI reasoning engine. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin, 2005.
24. M. B. van Riemsdijk, M. Dastani, and J.-J. Ch. Meyer. Semantics of declarative goals in agent programming. In *Proceedings of the fourth international joint conference on autonomous agents and multiagent systems (AAMAS'05)*, pages 133–140, Utrecht, 2005.
25. M. B. van Riemsdijk, J.-J. Ch. Meyer, and F. S. de Boer. Semantics of plan revision in intelligent agents. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Proceedings of the 10th International Conference on Algebraic Methodology And Software Technology (AMAST04)*, volume 3116 of *LNCS*, pages 426–442. Springer-Verlag, 2004.

26. M. B. van Riemsdijk, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in Dribble: from beliefs to goals using plans. In *Proceedings of the second international joint conference on autonomous agents and multiagent systems (AAMAS'03)*, pages 393–400, Melbourne, 2003.
27. A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. Technical report, Universidad Complutense de Madrid, Madrid, 2003.
28. M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 675–788. Elsevier, Amsterdam, 1990.