A Formal Specification of dMARS

November, 1997

Technical Note 72

THE
AUSTRALIAN
ARTIFICIAL
INTELLIGENCE
INSTITUTE

By:

Mark d'Inverno
Cavendish School of Computer Science, Westminster University
London W1M 8JS, UK

David Kinny
Australian Artificial Intelligence Institute, 171 Latrobe Street,
Melbourne 3000, Australia
dnk@aaii.oz.au

Michael Luck
Department of Computer Science, University of Warwick, CV4 7AL, UK

Michael Wooldridge
Dept. of Electronic Engineering, Queen Mary & Westfield College,
London E1 4NS, UK

This paper appears in *Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages, ATAL'97*

171 La Trobe Street
Melbourne
Victoria 3000
Australia

Telephone
(+61 3) 9663 7922

Facsimile
(+61 3) 9663 7937

Abstract

The Procedural Reasoning System (PRS) is the best established agent architecture currently available. It has been deployed in many major industrial applications, ranging from fault diagnosis on the space shuttle to air traffic management and business process control. The theory of PRS-like systems has also been widely studied: within the intelligent agents research community, the belief-desire-intention (BDI) model of practical reasoning that underpins PRS is arguably the dominant force in the theoretical foundations of rational agency. Despite the interest in PRS and BDI agents, no complete attempt has yet been made to precisely specify the behaviour of real PRS systems. This has led to the development of a range of systems that claim to conform to the PRS model, but which differ from it in many important respects. Our aim in this paper is to rectify this omission. We provide an abstract formal model of an idealised dMARS system (the most recent implementation of the PRS architecture), which precisely defines the key data structures present within the architecture and the operations that manipulate these structures. We focus in particular on dMARS plans, since these are the key tool for programming dMARS agents. The specification we present will enable other implementations of PRS to be easily developed, and will serve as a benchmark against which future architectural enhancements can be evaluated.

1 Introduction

Since the mid 1980s, many control architectures for practical reasoning agents have been proposed [Wooldridge and Jennings, 1995]. Most of these have been deployed only in limited artificial environments; very few have been applied to realistic problems, and even fewer have led to the development of useful field-tested applications. The most notable exception is the Procedural Reasoning System (PRS). Originally described in 1987 [Georgeff and Lansky, 1987], this architecture has progressed from an experimental LISP version to a fully fledged C++ implementation known as the distributed Multi-Agent Reasoning System (dMARS), which has been applied in perhaps the most significant multi-agent applications to date [Georgeff and Rao, 1996]. The PRS architecture has its conceptual roots in the belief-desire-intention (BDI) model of practical reasoning developed by Michael Bratman and colleagues [Bratman *et al.*, 1988], and in tandem with the evolution of the PRS architecture into an industrial-strength production architecture, the theoretical foundations of the BDI model have also been closely investigated (see, e.g., [Rao and Georgeff, 1995a] for a survey).

Despite the success of the PRS architecture, in terms of both its demonstrable applicability to real-world problems and its theoretical foundations, there has to date been no systematic attempt to unambiguously define its operation. There have, however, been several attempts in this direction. For example, in [Rao and Georgeff, 1992], Rao and Georgeff give an abstract specification of the architecture, and informally discuss the extent to which an embodiment of it could be said to satisfy various possible axioms of BDI theory [Rao and Georgeff, 1995a]. However, that specification is (quite deliberately) at a high level, and does not lend itself to direct implementation. Another related attempt is embodied by the AgentSpeak(L) language developed by Rao [Rao, 1996]. AgentSpeak(L) is a programming language based on an abstraction of the PRS architecture; irrelevant implementation detail is removed, and PRS is stripped to its bare essentials. Building on this work, d'Inverno and Luck have constructed a formal specification (in Z [Spivey, 1992b]) of AgentSpeak(L) [d'Inverno and Luck, Forthcoming]. This specification reformalises Rao's original description so that it is couched in terms of state and operations on state that can be easily refined into an implemented system. In addition, being based on a simplified version of dMARS, the specification provides a starting point for actual specifications of these more sophisticated systems.

In this paper, we continue and extend that work, by giving an abstract formal specification of dMARS: the system upon which AgentSpeak(L) is based. In so doing, we provide an operational semantics for dMARS, and thus provide a benchmark against which future BDI systems and PRS-like implementations can be compared. The specification is *abstract* in that important aspects of the dMARS system are included, but unnecessary implementation-specific details are omitted. This approach is very similar to that of [Wooldridge, 1995], in which a formal specification of the MyWorld architecture was developed using VDM, a formal specification language closely related to Z.

The remainder of this paper is structured as follows. First, in Section 2 we present an overview of the dMARS system. In Section 3, we describe the basic types and primitive components of the system, and in Section 4 we proceed to specify more complex components including plans. The next section specifies the dMARS agent and its state, followed by a description of its cycle of operation. At the end of the paper we summarise the contribution made by this specification, its relation to previous work, and prospects for the future.

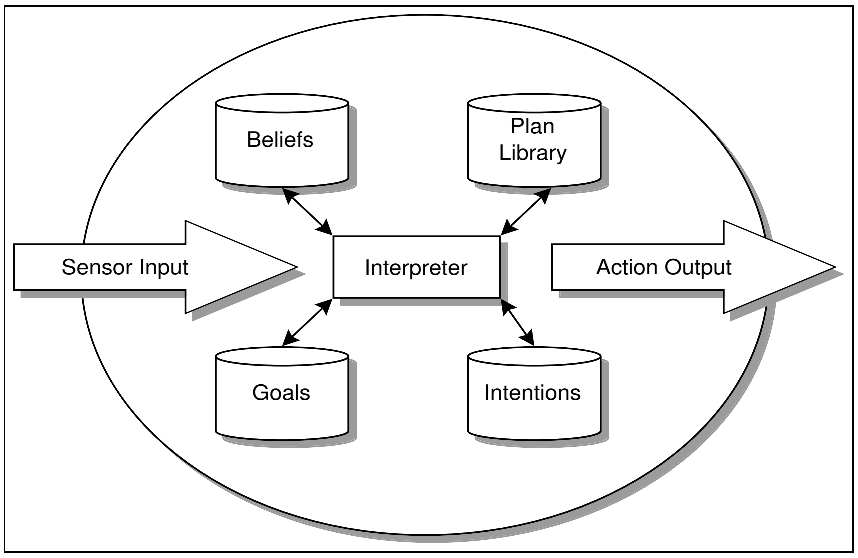

Figure 1: A BDI Agent Architecture: PRS

Notation

The specification below is presented using the Z language [Spivey, 1992b]. Z is a model-oriented formal specification language based on set theory and first-order logic. The key components of a Z specification are definitions of the *state space* of a system and the possible *operations* that transform it from one state to another. Because of space constraints, some auxiliary function definitions are omitted, and only a very brief account of binding is given. A more complete account of a related system can be found in [d'Inverno and Luck, Forthcoming], which provides both an introduction to Z and more explanation of several of the aspects not covered here. The full dMARS specification is available on request from the first author.

2 An Overview of dMARS

The *Procedural Reasoning System (PRS)* developed by Georgeff and Lansky [Georgeff and Lansky, 1987] is perhaps the best-known agent architecture. Both PRS and its successor *dMARS* are examples of a currently popular paradigm known as the *belief-desire-intention* (BDI) approach [Bratman *et al.*, 1988]. As Figure 1 shows, a BDI architecture typically contains four key data structures: beliefs, goals, intentions and a plan library.

An agent's *beliefs* correspond to information the agent has about the world, which may be incomplete or incorrect. Beliefs may be as simple as variables (in the sense of, e.g., PASCAL programs) but implemented BDI agents typically represent beliefs symbolically (e.g., as PROLOG-like facts [Georgeff and Lansky, 1987]). An agent's *desires* (or goals, in the system) intuitively correspond to the tasks allocated to it. (Implemented BDI agents require that desires be logically consistent, although *human* desires often fail in this respect.) The intuition with BDI systems is that an agent will not, in general, be able to achieve *all* its desires, even if these desires *are* consistent. Agents must therefore fix upon some subset of available desires and commit resources to achieving them. These chosen desires are *intentions*, and an agent will typically continue to try to achieve an intention until either it believes the intention is satisfied, or it believes the intention is no longer achievable [Cohen and Levesque, 1990].

The BDI model is operationalised in dMARS agents by *plans*. Each agent has a *plan library*, which is a set of plans, or *recipes*, specifying courses of action that may be undertaken by an agent in order to achieve its intentions. An agent's plan library represents its *procedural knowledge*, or *know-how*: knowledge about how to bring about states of affairs.

Each plan contains several components. The *trigger* or *invocation condition* for a plan specifies the circumstances under which the plan should be considered, usually specified in terms of events. For example, the plan "make tea" may be triggered by the event "thirsty". In addition, a plan has a *context*, or *pre-condition*, specifying the circumstances under which the execution of the plan may commence. For example, the plan "make tea" might have the context "have tea-bags". A plan may also have a *maintenance condition*, which characterises the circumstances that must remain true while the plan is executing. Finally, a plan has a *body*, defining a potentially quite complex course of action, which may consist of both goals (or subgoals) and primitive actions. Our "tea" plan might have the body *get boiling water; add tea-bag to cup; add water to cup*. Here, *get boiling water* is a subgoal, (something that must be achieved when plan execution reaches this point in the plan), whereas *add tea-bag to cup* and *add water to cup* are primitive actions, i.e., actions that can be performed directly by the agent. Primitive actions can be thought of as procedure calls.

dMARS agents monitor both the world and their own internal state, and any events that are perceived are placed on an *event queue*. The *interpreter* in Figure 1 is responsible for managing the overall operation of the agent. It continually executes the following cycle:

- observe the world and the agent's internal state, and update the event queue to reflect the events that have been observed;
- generate new possible desires (tasks), by finding plans whose trigger event matches an event in the event queue;
- select from this set of matching plans one for execution (an *intended means*);
- push the intended means onto an existing or new intention stack, according to whether or not the event is a subgoal; and
- select an intention stack, take the topmost plan (intended means), and execute the next step of this current plan: if the step is an action, perform it; otherwise, if it is a subgoal, post this subgoal on the event queue.

In this way, when a plan starts executing, its subgoals will be posted on the event queue which, in turn, will cause plans that achieve this subgoal to become active, and so on. This is the basic execution model of dMARS agents. Note that agents do no first-principles planning at all, as all plans must be generated by the agent programmer at design time. The planning done by agents consists entirely of context-sensitive subgoal expansion, which is deferred until a point in time at which the subgoal is selected for execution.

Other efforts to give a formal semantics to BDI architectures include a range of *BDI logics* that have been developed by Rao and Georgeff [Rao and Georgeff, 1995a]. These logics are extensions to the branching time logic CTL* [Emerson and Halpern, 1986], which also contain normal modal connectives for representing beliefs, desires, and intentions. Most work on BDI logics has focussed on possible relationships between the three 'mental states' [Rao and Georgeff, 1991] and, more recently, on developing proof methods for restricted forms of the logics [Rao and Georgeff, 1995b]. In future work we will investigate the relationship between this work and the operational semantics described in this paper.

3 Beliefs, Goals, and Actions

We begin our specification by defining the allowable *beliefs* of an agent. Beliefs in dMARS are rather like PROLOG facts: they are essentially ground literals of classical first-order logic (i.e., positive or negative atomic formulae containing no variables). In order to define atomic formulae, we need a stock of variables, function and predicate symbols. We are not concerned with the contents of these sets, and hence we parachute them into our specification.

$$[\mathit{Var}, \mathit{FunSym}, \mathit{PredSym}]$$

A *term* is either a variable or a function symbol applied to a (possibly empty) sequence of terms.

$$\mathit{Term} ::= \mathit{var}\langle\langle \mathit{Var} \rangle\rangle \mid \mathit{functor}\langle\langle \mathit{FunSym} \times \mathrm{seq}\ \mathit{Term} \rangle\rangle$$

An *atom* is a predicate symbol applied to a (possibly empty) sequence of terms.

$$\begin{array}{|l}
\textit{Atom} \\ \hline
\mathit{head} : \mathit{PredSym} \\
\mathit{terms} : \mathrm{seq}\ \mathit{Term} \\ \hline
\end{array}$$

A *belief formula* is then either an atom or the negation of an atom.

$$\mathit{BeliefFormula} ::= \mathit{pos}\langle\langle \mathit{Atom} \rangle\rangle \mid \mathit{not}\langle\langle \mathit{Atom} \rangle\rangle$$

The set of *beliefs* is the set of all ground belief formulae (i.e. those containing no variables).

$$\mathit{Belief} == \{ b : \mathit{BeliefFormula} \mid \mathit{belvars}\ b = \varnothing \bullet b \}$$

An auxiliary function *belvars* is assumed which, given a belief formula, returns the set of variables it contains.

dMARS allows an agent's *goals* to be specified in terms of a simple temporal modal language with two unary connectives in addition to the connectives of classical logic. The operators are "!" and "?", for "achieve" and "query" respectively, so that a formula $!\phi$ in dMARS is read "achieve ϕ". Thus an agent with goal $!\phi$ has a goal of performing some (possibly empty) sequence of actions, such that after these actions are performed, ϕ will be true. Similarly, a formula "$?\phi$" means "query ϕ". Thus an agent with goal $?\phi$ has a goal of performing some (possibly empty) sequence of actions, such that after it performs these actions, it will know whether or not ϕ is true. In order to define these additional connectives, we must first define *situation formulae*: these are expressions whose truth can be evaluated with respect to a set of beliefs, and are thus not temporal.

$$\begin{array}{rl}
\mathit{SituationFormula} ::= & \mathit{belform}\langle\langle \mathit{BeliefFormula} \rangle\rangle \\
\mid & \mathit{and}\langle\langle \mathit{SituationFormula} \times \mathit{SituationFormula} \rangle\rangle \\
\mid & \mathit{or}\langle\langle \mathit{SituationFormula} \times \mathit{SituationFormula} \rangle\rangle \\
\mid & \mathit{true} \\
\mid & \mathit{false}
\end{array}$$

A *temporal formula*, known as a *goal* is then a belief formula prefixed with an achieve operator or a situation formula prefixed with a query operator. Thus an agent can have a goal either of achieving a state of affairs or of determining whether the state of affairs holds.

$$Goal ::= achieve \langle\!\langle BeliefFormula \rangle\!\rangle \mid query \langle\!\langle SituationFormula \rangle\!\rangle$$

The types of action that agents can perform may be classified as either *external* (in which case the domain of the action is the environment outside the agent) or *internal* (in which case the domain of the action is the agent itself). External actions are specified as if they are procedure calls or method invocations (and in reality, from the agent programmer's perspective, they usually are). An external action thus comprises an external action symbol (cf. the procedure name) taken from the set [*ActionSym*], and a sequence of terms (cf. the parameters of the procedure).

$$\begin{array}{|l} \hline \textit{ExtAction} \\ name : ActionSym \\ terms : \mathrm{seq}\ Term \\ \hline \end{array}$$

Internal actions may be one of two types: add or remove a belief from the data base (cf. the PROLOG `assert` and `retract` clauses). Note that it is not possible to add or remove an atom that contains variables.

$$IntAction ::= add \langle\!\langle BeliefFormula \rangle\!\rangle \mid remove \langle\!\langle BeliefFormula \rangle\!\rangle$$

4 Plans

Plans are *adopted* by agents, in the way we describe below. Once adopted, plans constrain an agent's behaviour and act as *intentions*. Plans consists of six components: an *invocation condition* (or *triggering event*); an optional *context* (a situation formula) that defines the pre-conditions of the plan, i.e., what must be believed by the agent for a plan to be executable; the *plan body*, which is a tree representing a kind of flow-graph of actions to perform; a *maintenance condition* that must be true for the plan to continue executing; a set of *internal actions* that are performed if the plan succeeds; and finally, a set of *internal actions* that are performed if the plan fails. The tree representing the body has states as nodes, and arcs (branches) representing either a goal, an internal action or an external action as defined below. Executing a plan successfully involves traversing the tree from the root to any leaf node.

First, we define trigger events. A trigger event is one that causes a plan to be adopted. Four types of events are allowable as triggers: the acquisition of a new belief; the removal of a belief; the receipt of a message; or the acquisition of a new goal. This last type of trigger event allows goal-driven as well as event-driven processing.

$$\begin{aligned} TriggerEvent ::= {} & addbelevent \langle\!\langle Belief \rangle\!\rangle \\ \mid {} & rembelevent \langle\!\langle Belief \rangle\!\rangle \\ \mid {} & toldevent \langle\!\langle Atom \rangle\!\rangle \\ \mid {} & goalevent \langle\!\langle Goal \rangle\!\rangle \end{aligned}$$

As we noted above, plan bodies are trees in which arcs are labelled with either goals or actions and states are place holders. Since states are not important in themselves, we define them using the given set [*State*]. An arc (branch) within a plan body may be labelled with either an internal or external action, or a subgoal.

$$
\begin{aligned}
Branch ::= {} & extaction\langle\!\langle ExtAction\rangle\!\rangle \\
\mid {} & intaction\langle\!\langle IntAction\rangle\!\rangle \\
\mid {} & subgoal\langle\!\langle Goal\rangle\!\rangle
\end{aligned}
$$

Next, we define plan bodies. A dMARS plan body is either an *end tip* containing a state, or a *fork* containing a state and a non-empty set of branches each leading to another tree.

$$Body ::= End\langle\!\langle State\rangle\!\rangle \mid Fork\langle\!\langle \mathbb{P}_1(State \times Branch \times Body)\rangle\!\rangle$$

We can bring these components together into the definition of a plan. The formal definition of the *optional* type and related components, which are non-standard Z, can be found in Appendix A.

$$
\begin{array}{|l}
\textit{Plan} \\
\hline
inv : TriggerEvent \\
context : optional[SituationFormula] \\
body : Body \\
maint : SituationFormula \\
succ : \mathrm{seq}\ IntAction \\
fail : \mathrm{seq}\ IntAction \\
\hline
\end{array}
$$

Plans with no body are called *primitive plans*.

$$PrimitivePlan == \{p : Plan \mid p.body \in (\mathrm{ran}\ End) \bullet p\}$$

4.1 Instantiating Plans

The basic execution mechanism for dMARS agents, described in Section 2, involves an agent matching the trigger and context of each plan against the chosen event in the event queue and the current set of beliefs, respectively, and then generating a set of candidate, matching plans, selecting one, and making a *plan instance* for it. A plan instance contains a copy of the original plan and, in addition: the *environment* of the plan (i.e., any bindings that have been generated in the course of executing the plan); the current state reached in the plan (initially the root of the plan body); the set of branches it can attempt to traverse from this state; the branch it *is* attempting to traverse; an identifier to uniquely identify the plan instance to the agent owner from the set $[PlanInstanceId]$ of all such identifiers; and finally, the *status* of the plan (either "active", indicating that the plan is part of an intention, or "inactive", indicating that the plan has temporarily been suspended).

When a branch cannot be traversed (e.g., because an action or subgoal fails), then the branch itself fails and is removed from the set of possible branches. If the branch that the agent is attempting to traverse is defined, the agent has chosen which branch to attempt next, but if it is undefined, no such choice has been made.

In what follows, the *Substitution* type represents the set of all *substitutions* (i.e., bindings from variables to terms) A function prefixed by *AS* applies a substitution to a dMARS expression. If s and t are substitutions then $s \ddagger t$ denotes the *composition* of s and t. Finally, a function prefixed by mgu is a function which returns the most general unifier of two expressions. A brief description and some relevant definitions can be found in Appendix B.

A plan instance is thus formally defined as follows.

$$Status ::= active \mid suspended$$

$$
\begin{array}{|l}
\hline
\textit{PlanInstance} \\
origplan : Plan \\
env : Substitution \\
state : State \\
nextbranches : \mathbb{P}\, Branch \\
branch : optional[Branch] \\
status : Status \\
id : PlanInstanceId \\
\hline
state \in PlanStates\ origplan \\
state \in \operatorname{dom} End \Rightarrow nextbranches = \varnothing \\
nextbranches \subseteq PlanNextBranches\ origplan\ state \\
branch \subseteq nextbranches \\
\hline
\end{array}
$$

In this schema, we use the auxiliary functions, *PlanNextBranches*, to identify the set of possible next branches from a given state in a plan and *PlanStates*, to give all the states of a plan. The specification that follows also uses the functions, *PlanNextState* and *PlanStartState*, which give the next state in a plan when applied to the current state and the branch traversed, and determine the start state of a plan respectively.

When a plan is first selected, the current state is the first state in the plan. A plan is said to have *succeeded* when it reaches its end state, and it is said to have *failed* if it is not in the end state and there are no available branches (i.e., it has failed if it has tried each branch and none have been successful).

$$
\begin{aligned}
&InitialInstance == \{p : PlanInstance \mid \\
&\qquad p.state = PlanStartState\ p.origplan \land p.status = active\} \\
&SucceedInstance == \{p : PlanInstance \mid p.state \in (\operatorname{dom} End)\} \\
&FailedInstance == \{p : PlanInstance \mid \\
&\qquad p.state \notin (\operatorname{dom} End) \land p.nextbranches = \{\}\}
\end{aligned}
$$

4.2 Intentions

An intention in dMARS is just a sequence of plan instances. In response to an external event, an intention is created containing the generated plan instance. If this plan, in turn, creates an internal event to which the agent responds with another plan, the new plan is concatenated to the intention. In this way, the plan at the top of the intention stack is the plan that will be executed first in any intention.

$$Intention == \operatorname{seq} PlanInstance$$

An event consists of the triggering event and, optionally, a plan instance identifier that identifies the event-generating plan, an environment, and a set of plan instances that may already have failed (and may not be retried).

$$
\begin{array}{|l}
\hline
\textit{Event} \\
trig : TriggerEvent \\
id : optional[PlanInstanceId] \\
env : optional[Substitution] \\
failures : optional[\mathbb{P}\, PlanInstance] \\
\hline
\end{array}
$$

An external event is one that is not associated with an existing plan instance when it first enters the buffer though it will become so when a plan instance is generated for it. By contrast, a subgoal event is an internal event that occurs when the branch of an executing intention is an achieve goal that cannot be achieved immediately. In this case, the variables of the event will all be defined with the constraint that the domain of the environment contains only variables that are contained in the event trigger.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ExternalEvent}} \\
\mathit{Event} \\
\hline
\mathit{trig} \notin \operatorname{ran} \mathit{goalevent} \\
\mathit{undefined\ env} \\
\mathit{undefined\ failures} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{SubgoalEvent}} \\
\mathit{Event} \\
\hline
\mathit{trig} \in \operatorname{ran} \mathit{goalevent} \\
\mathit{defined\ env} \\
\mathit{defined\ failures} \\
\operatorname{dom}(\mathit{the\ env}) \subseteq \mathit{goalatomvars}\ (\mathit{goalevent}^{\sim} \mathit{trig}) \\
\hline
\end{array}$$

5 An Operational Semantics for dMARS Agents

The operation of dMARS agents is driven by the interaction of intentions and events. Events, (which may be the addition or deletion of beliefs, or the generation of new goals or subgoals), provide triggers to execute appropriate plans in the agent's plan library. As events are posted on the agent's event queue, so plans are selected from the agent's plan library that are relevant and applicable to the event. Determining whether a plan is relevant and applicable to an event reduces to attempting to unify the invocation condition and context with the event. From the set of applicable plans found by such unification, the agent chooses one plan, and from it generates a plan instance that is then added to the current intentions of the agent. This plan is thus an *intended means*.

Plans in dMARS are sequences of actions and goals with choice points so that, at any point, there may be more than one path to traverse in order to complete the plan. Intentions, which are those plans currently executing, determine which actions the agent takes, and may also give rise to the generation of new subgoals, both of which occur in the course of the agent's efforts to carry out the plan.

The following formal model specifies how relevant and applicable plans are determined initially, how one is chosen, and then how it is used. Essentially, an event generates either a new intention, or adds to an existing one. An agent then selects an intention to execute and, depending on the current component of the plan, different courses of behaviour are required. Actions may be executed directly and may lead to the posting of new events if the database is modified as a result, while goals either lead to the further instantiation of plans, or to the posting of new events (subgoals to be achieved) and the suspension of the currently executing plan.

This section provides a detailed specification of the dMARS agent operation, covering the agent and agent state, the generation of relevant and applicable plans, the way in which events are processed, the execution of intentions, and finally the achievement and failure of plans.

5.1 The dMARS Agent State

As in other BDI architectures, a dMARS agent consists of a plan library, an intention-selection function, an event-selection function and a plan-selection function. It also has a substitution-selection function for choosing between possible alternative bindings, and a function for selecting which branch in a plan should be attempted next.

$$
\begin{array}{|l}
\textit{dMARSAgent} \\
\hline
\textit{planlibrary} : \mathbb{P}\ \textit{Plan} \\
\textit{intentionselect} : \mathbb{P}_1\ \textit{Intention} \rightarrow \textit{Intention} \\
\textit{planselect} : \mathbb{P}_1\ \textit{Plan} \rightarrow \textit{Plan} \\
\textit{eventselect} : \mathrm{seq}_1\ \textit{Event} \rightarrow \textit{Event} \\
\textit{substitutionselect} : \mathbb{P}_1\ \textit{Substitution} \rightarrow \textit{Substitution} \\
\textit{selectbranch} : \textit{PlanInstance} \nrightarrow \textit{Branch} \\
\hline
\end{array}
$$

In specifying the state of the agent, we indicate which aspects may change over time. These components are the agents' beliefs (which are ground belief formulae), intentions, and events yet to be processed (represented as a sequence).

$$
\begin{array}{|l}
\textit{dMARSAgentState} \\
\hline
\textit{dMARSAgent} \\
\textit{beliefs} : \mathbb{P}\ \textit{Belief} \\
\textit{intentions} : \mathbb{P}\ \textit{Intention} \\
\textit{events} : \mathrm{seq}\ \textit{Event} \\
\hline
\end{array}
$$

An operation only affects the state of the dMARS agent rather than the agent itself.

$$
\begin{array}{|l}
\Delta \textit{dMARSAgentState} \\
\hline
\textit{dMARSAgentState} \\
\textit{dMARSAgentState}' \\
\Xi \textit{dMARSAgentState} \\
\hline
\end{array}
$$

Initially, the agent is provided with an event queue and sets of beliefs and intentions that "pump prime" its subsequent intention generation and action.

$$
\begin{array}{|l}
\textit{InitdMarsAgentState} \\
\hline
\Delta \textit{dMARSAgentState} \\
\textit{initBel}? : \mathbb{P}\ \textit{Belief} \\
\textit{initInt}? : \mathbb{P}\ \textit{Intention} \\
\textit{initEv}? : \mathrm{seq}\ \textit{Event} \\
\hline
\textit{beliefs}' = \textit{initBel}? \\
\textit{intentions}' = \textit{initInt}? \\
\textit{events}' = \textit{initEv}? \\
\hline
\end{array}
$$

Agents can perceive external events which are placed at the end of the event buffer.

$$
\begin{array}{l}
\textit{NewExternalEvent} \\
\hline
\Delta dMARSAgentState \\
newevent? : Event \\
\hline
events' = events \frown \langle newevent? \rangle
\end{array}
$$

5.2 Relevant and Applicable Plans

A plan is *relevant* with respect to an event if there exists a *most general unifier* (mgu) to bind the triggering events of the plan and the event so that they are equal. This is specified in the function *genrelplans*, which takes an event e and a set of plans ps, and returns a set of plan/substitution pairs, such that if (p, σ) is returned, then p is a relevant plan in ps for the event e, and σ is the most general unifier for p. The signature of the functions defining most general unifiers are given in Appendix B. If the event is a subgoal event and therefore contains a substitution environment, it must be applied to the triggering event before the relevant plans are generated.

$$
\begin{array}{l}
genrelplans : Event \rightarrow \mathbb{P}\, Plan \rightarrow \mathbb{P}(Plan \times Substitution) \\
\hline
\forall\, e : Event;\ lib : \mathbb{P}\, Plan \bullet \\
\quad undefined\ e.env \Rightarrow genrelplans\ e\ lib = \\
\quad\quad \{p : lib;\ \sigma : Substitution \mid mguevents\ (e.trig, p.inv) = \sigma \bullet (p, \sigma)\} \land \\
\quad defined\ e.env \Rightarrow genrelplans\ e\ lib = \\
\quad\quad \{p : lib;\ \sigma : Substitution \mid \\
\quad\quad\quad mguevents\ (ASTrigEvent\ (the\ e.env)\ e.trig, p.inv) = \sigma \bullet (p, \sigma)\}
\end{array}
$$

A relevant plan is applicable if its context is a logical consequence of the beliefs of the agent. Thus, we can define a predicate, *dMarsLogCons*, to hold between a situation formula and a belief base if the situation formula is a logical consequence of the belief base.

$$
dMarsLogCons\_ : \mathbb{P}(SituationFormula \times \mathbb{P}\, BeliefFormula)
$$

Using this logical consequence relation, we define an *applicable plan* relation to hold between a relevant plan, a substitution and a current set of beliefs. This is specified in the function, *genapplplans*, which takes a set of plans (and the substitutions which make them relevant), and the current beliefs, and returns the *applicable* plans and updated substitutions.

$$
\begin{array}{l}
genapplplans : \mathbb{P}(Plan \times Substitution) \rightarrow \\
\quad\quad\quad (\mathbb{P}\, BeliefFormula) \rightarrow \\
\quad\quad\quad\quad \mathbb{P}(Plan \times Substitution) \\
\hline
\forall\, relsubs : \mathbb{P}(Plan \times Substitution); \\
\quad\quad bels : \mathbb{P}\, BeliefFormula \bullet \\
\quad genapplplans\ relsubs\ bels = \\
\quad\quad \{rel : Plan;\ \sigma, \psi : Substitution \mid \\
\quad\quad\quad (rel, \sigma) \in relsubs \land \\
\quad\quad\quad dMarsLogCons(ASSitForm\ (\sigma \ddagger \psi)\ (the\ rel.context), bels) \bullet \\
\quad\quad\quad\quad (rel, \sigma \ddagger \psi)\}
\end{array}
$$

5.3 Processing Events

With the dMARS agent and its state specified, we can define the dMARS operation cycle. There are two possible modes of operation, depending on whether the event buffer is empty or not. If the event buffer is not empty, an event is selected from it (typically the first element) and relevant plans and, in turn, applicable plans are determined. An applicable plan is selected and used to generate a plan instance.

With an external event, a new intention containing just the plan instance as a singleton sequence is created. With an internal event, the plan instance is pushed onto the intention stack that generated that (subgoal) event. In addition, we specify that a failed plan instance cannot be re-selected for an internal event. The auxiliary function *CreatePlanInstance* takes a plan and a substitution, and creates a plan instance in its initial state. If the event is external then it must be updated to include the id of the new planinstance.

$$
\begin{array}{|l}
\textit{NewPlanInstance} \\
\hline
\Delta \textit{dMARSAgentState} \\
\hline
\textit{events} \neq \langle\rangle \\
\mathsf{Let}\ \textit{event} == \textit{eventselect events} \bullet \\
\quad \mathsf{Let}\ \textit{applplans} == \textit{genapplplans}\ (\textit{genrelplans event planlibrary})\ \textit{beliefs} \bullet \\
\quad\quad \mathsf{Let}\ \textit{selectedplan} == \textit{planselect}\ (\mathrm{dom}\, \textit{applplans}) \bullet \\
\quad\quad\quad \mathsf{Let}\ \textit{applunifier} == \textit{applplans selectedplan} \bullet \\
\quad\quad\quad\quad \mathsf{Let}\ \textit{instance} == \textit{CreatePlanInstance selectedplan applunifier} \bullet \\
\quad\quad\quad\quad\quad \textit{event} \in \textit{ExternalEvent} \Rightarrow \\
\quad\quad\quad\quad\quad\quad \textit{instance} \notin (\textit{the event.failures}) \wedge \\
\quad\quad\quad\quad\quad\quad \textit{intentions}' = \textit{intentions} \cup \{\langle \textit{instance} \rangle\} \wedge \\
\quad\quad\quad\quad\quad\quad \textit{events}' = (\textit{events} \rhd \{\textit{event}\}) \cup \\
\quad\quad\quad\quad\quad\quad\quad \{(\textit{events}^{\sim} \textit{event}, \textit{MakeEvent}(\textit{event.trig}, \{\textit{instance.id}\}, \varnothing, \varnothing))\} \wedge \\
\quad\quad\quad\quad\quad \textit{event} \in \textit{SubgoalEvent} \Rightarrow \\
\quad\quad\quad\quad\quad\quad (\mathsf{Let}\ \textit{trigint} == (\mu\, i : \textit{intentions} \mid (\textit{head}\ i).\textit{id} = (\textit{the event.id})) \bullet \\
\quad\quad\quad\quad\quad\quad\quad \textit{intentions}' = \textit{intentions} \setminus \{\textit{trigint}\} \cup \{\langle \textit{instance} \rangle \frown \textit{trigint}\})
\end{array}
$$

5.4 Executing Intentions

The remainder of this section addresses the agent operation when the event buffer is empty. We refer to this as the *intention execution operation.* The variables included in the schema below enable the specification of intention execution to be written more elegantly, but do not define the state, and are reset on every operation cycle. When the event buffer becomes empty, all these variables are set to be undefined.

$$
\begin{array}{|l}
\textit{AgentIntExecutionOperationState} \\
\hline
\textit{dMARSAgentState} \\
\textit{selectedintention} : \textit{optional}[\textit{Intention}] \\
\textit{executingplan} : \textit{optional}[\textit{PlanInstance}] \\
\textit{executingbranch} : \textit{optional}[\textit{Branch}] \\
\hline
\end{array}
$$

The first step is to select an intention, *selectedintention*$'$, identify the executing plan, *executingplan*$'$, at the top of this intention stack such that the plan is active, and select the branch of the plan to execute, *executingbranch*$'$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad SelectIntention \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta AgentIntExecutionOperationState \\
\hline
events = \langle\rangle \\
the\ selectedintention' = intentionselect\ intentions \\
the\ executingplan' = head(the\ selectedintention') \\
(the\ executingplan').status = active \\
(the\ executingbranch') = selectbranch\ (the\ executingplan') \\
\hline
\end{array}
$$

Before considering the different cases arising from the different types of selected branch, we must introduce two schemas to specify a move to the next state if the branch is successful, and to delete a branch if it fails. The auxiliary function, *AchieveBranch*, takes a plan instance and moves it on to the next state determined by the *branch* variable of the executing plan.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad BranchSucceed \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta AgentIntExecutionOperationState \\
\hline
the\ executingplan' = AchieveBranch\ (the\ executingplan) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad BranchFail \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta AgentIntExecutionOperationState \\
\hline
(the\ executingplan').nextbranches = \\
\qquad\qquad (the\ executingplan).nextbranches \setminus executingbranch \\
\hline
\end{array}
$$

There are then four cases, depending on whether the branch is an external action, an internal action, a query goal, or an achieve goal.

External Actions:

If the branch is an external action, then it is executed immediately. Its success or failure is modelled by the function *executeaction*, which takes a plan instance with a selected branch that is an external action, and returns the binding that succeeded. If it is not in the domain, the function models the action failing.

$$
\begin{array}{|l}
executeaction : PlanInstance \nrightarrow Substitution
\end{array}
$$

With a successful branch, the binding of the action is *composed* with the substitution environment.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad BranchExtActionSucceed \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta AgentIntExecutionOperationState \\
\hline
the\ executingbranch \in \operatorname{ran} extaction \\
the\ executingplan \in \operatorname{dom} executeaction \\
(the\ executingplan').env = \\
\qquad\qquad (the\ executingplan).env \ddagger executeaction\ (the\ executingplan) \\
\hline
\end{array}
$$

The branch is then traversed to reach the next state, specified by the *BranchSucceed* schema above. The operation of achieving an external action and so moving onto the next state as defined by the tree is therefore defined as the composition of two operations as follows.

$$BranchExtActionSuceed \mathbin{;} BranchSuceed$$

An unsuccessful branch fails and there is no state change.

$$
\begin{array}{|l}
\textit{BranchExtActionFail} \\
\hline
\Xi AgentIntExecutionOperationState \\
\hline
the\ executingbranch \in \operatorname{ran} extaction \\
the\ executingplan \notin (\operatorname{dom} executeaction) \\
\hline
\end{array}
$$

After this occurs the branch must be removed.

$$BranchExtActionFail \mathbin{;} BranchFail$$

Internal Actions:

If the branch is an internal action (denoted by the local variable *action*), the database is modified according to that action. If this action results in a change to the database, an event is added to the set of events.

$$
\begin{array}{|l}
performintaction : (\mathbb{P}\, Belief) \rightarrow IntAction \rightarrow (\mathbb{P}\, Belief) \\
\hline
\forall\, b : Belief;\ i : IntAction;\ bs : \mathbb{P}\, Belief \bullet \\
\quad i = add\ b \Rightarrow performintaction\ bs\ i = bs \cup \{b\} \land \\
\quad i = remove\ b \Rightarrow performintaction\ bs\ i = bs \setminus \{b\}
\end{array}
$$

$$
\begin{array}{|l}
\textit{BranchIntAction} \\
\hline
\Delta AgentIntExecutionOperationState \\
\hline
(the\ executingbranch) \in (\operatorname{ran} intaction) \\
\mathsf{Let}\ action == (intaction^{\sim}(the\ executingbranch)) \bullet \\
\quad beliefs' = performintaction\ beliefs\ action \land \\
\quad action \in (\operatorname{ran} add) \land beliefs' \neq beliefs \Rightarrow \\
\qquad events' = events ^\frown \\
\qquad\quad \langle MakeEvent(addbelevent\ (add^{\sim} action), \emptyset, \emptyset, \emptyset)\rangle \land \\
\quad action \in (\operatorname{ran} remove) \land beliefs' \neq beliefs \Rightarrow \\
\qquad events' = events ^\frown \\
\qquad\quad \langle MakeEvent(rembelevent\ (remove^{\sim} action), \emptyset, \emptyset, \emptyset)\rangle \\
\hline
\end{array}
$$

The auxiliary function, *MakeEvent*, in the schema above, simply constructs an event from its constituent components. This operation is then composed with the operation, *BranchSucceed*, as before.

"Query" Goals:

In the case of a query goal, *qgoal*, if the environment applied to the goal can be unified with the set of beliefs, the most general unifiers are generated and one is chosen (*sub*). This binding is composed with the substitution environment and the next state is reached. The *unifiquery* relation holds between a goal and a set of beliefs if the goal can be unified with the beliefs.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad BranchQueryGoalSucc \quad}} \\
\Delta AgentIntExecutionOperationState \\
\hline
the\ executingbranch \in \operatorname{ran} subgoal \\
subgoal^{\sim}(the\ executingbranch) \in \operatorname{ran} query \\
\mathsf{Let}\ qgoal == (subgoal^{\sim}(the\ executingbranch)); \\
\qquad env == (the\ executingplan).env \bullet \\
\quad \exists\, s : Substitution \bullet unifiquery\ (s, (ASGoal\ env\ qgoal, beliefs)) \land \\
\quad (\mathsf{Let}\ sub == \ mguquery\ (ASGoal\ env\ qgoal, beliefs) \bullet \\
\qquad (the\ executingplan').env = env \ddagger sub) \\
\hline
\end{array}
$$

Where no such unification is possible, the branch fails.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad BranchQueryGoalFail \quad}} \\
\Delta AgentIntExecutionOperationState \\
\hline
the\ executingbranch \in \operatorname{ran} subgoal \\
subgoal^{\sim}(the\ executingbranch) \in \operatorname{ran} query \\
\mathsf{Let}\ qgoal == (subgoal^{\sim}(the\ executingbranch)); \\
\qquad env == (the\ executingplan).env \bullet \\
\quad \neg\, (\exists\, s : Substitution \bullet unifiquery\ (s, (ASGoal\ env\ qgoal, beliefs))) \\
\hline
\end{array}
$$

"Achieve" Goals:

Finally, with an achieve goal, *achievegoal*, that can be unified with the beliefs, the rest of the executing plan is unified as in the previous case, and the branch succeeds. If the goal cannot be unified, the goal achieve event is posted, the executing plan is suspended by setting the status parameter, and the set of tried instances becomes defined as the set containing the empty set. In addition, the identifier of the new internal event is set to the current executing plan.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad BranchAchieveGoal \quad}} \\
\Delta AgentIntExecutionOperationState \\
\hline
(the\ executingbranch) \in \operatorname{ran} subgoal \\
subgoal^{\sim}(the\ executingbranch) \in \operatorname{ran} achieve \\
(the\ executingplan').status = suspended \\
\mathsf{Let}\ achievegoal == subgoal^{\sim}(the\ executingbranch); \\
\qquad env == (the\ executingplan).env \bullet \\
\qquad events' = events \frown \langle MakeEvent\ ((goalevent\ achievegoal), \\
\qquad\qquad\qquad\qquad \{(the\ executingplan).id\}, \{env\}, \{\varnothing\})\rangle \\
\hline
\end{array}
$$

Once an achieve goal is posted, the execution cycle can restart, otherwise further operations are performed as follows.

5.5 Achieving and Failing Plans

A successful branch leads to a new state that is either not an end state, in which case execution of another branch ensues, or is an end state, in which case the plan *succeeds*. In the latter possibility, the substitution environment, $(the\ executingplan).env$, is applied to the success conditions, $(the\ executingplan).origplan.succ$, to give a sequence of ground internal actions, *groundsuccacts*. Then, the database is updated by performing these ground actions one at a time on the current set of beliefs to give the new set of beliefs, $beliefs'$. The auxiliary definition *fold* is given in Appendix A.

$$
\begin{array}{|l}
\textit{AchievePlan} \\
\hline
\Delta AgentIntExecutionOperationState \\
\hline
the\ executingplan \in SucceedInstance \\
\mathsf{Let}\ succacts == (the\ executingplan).origplan.succ; \\
\quad\quad env == (the\ executingplan).env \bullet \\
\quad \mathsf{Let}\ groundsuccacts == map\ (ASIntAction\ env)\ succacts \bullet \\
\quad beliefs' = fold\ performintaction\ beliefs\ groundsuccacts \\
\hline
\end{array}
$$

Two further cases arise if a plan succeeds. If there are more plans in the intention, the current substitution environment, $(the\ executingplan).env$, is updated to include the appropriate bindings from both the achieved plan, *executingplan*, and the environment of the next plan in the stack, *secondplan.env*. The successful plan instance is then removed from the top of the selected intention so that the new executing plan, which is re-activated, is the second in the original stack. *TEVars*, returns the set of variables of a trigger event. Also the internal event which generated the completed plan is removed.

$$
\begin{array}{|l}
\textit{AchievePlanOnly} \\
\hline
AchievePlan \\
\hline
\#(the\ selectedintention) > 1 \\
\mathsf{Let}\ secondplan == (the\ selectedintention)\ 2 \bullet \\
\quad \mathsf{Let}\ newenv == \\
\quad\quad ((TEVars\ (the\ executingplan).origplan.inv) \lhd secondplan.env) \ddagger \\
\quad\quad\quad\quad\quad ((the\ executingplan).env \oplus secondplan.env) \bullet \\
\quad\quad the\ selectedintention' = tail\ (the\ selectedintention) \wedge \\
\quad\quad (the\ executingplan').env = newenv \wedge \\
\quad\quad (the\ executingplan').status = active \\
\quad\quad \mathrm{ran}\ events' = \mathrm{ran}\ events \setminus \{(\mu\, e : SubgoalEvent \mid e \in \mathrm{ran}\ events \wedge \\
\quad\quad\quad\quad\quad\quad\quad the\ e.id = (the\ executingplan).id)\} \\
\hline
\end{array}
$$

If there are no more plans, the intention has succeeded and can be removed as can the external event which generated it.

$$
\begin{array}{|l}
\textit{AchievePlanAndIntention} \\
\hline
AchievePlan \\
\hline
\#(the\ selectedintention) = 1 \\
intentions' = intentions \setminus selectedintention \\
\mathrm{ran}\ events' = \mathrm{ran}\ events \setminus \{(\mu\, e : ExternalEvent \mid e \in \mathrm{ran}\ events \wedge \\
\quad\quad\quad\quad\quad\quad\quad the\ e.id = (the\ executingplan).id)\} \\
\hline
\end{array}
$$

Finally, if a branch fails but more branches remain, these may then be attempted. If there are no further alternatives, however, the plan *fails*. When this is the only plan on the stack, the intention fails completely (which is not specified here), otherwise the substitution environment is applied to the plan's fail conditions, *failacts*, and the ground fail internal actions, *groundfailacts'*, are performed. Since it is not the only plan on the stack, it must have been triggered by an existing *goal* event in the event queue, *origevent*, which is then found and updated to record the failed plan instance so that it is not retried. The status of the second plan remains suspended.

$$
\begin{array}{|l}
\textit{FailPlan} \\
\hline
\Delta \textit{AgentIntExecutionOperationState} \\
\hline
\textit{the executingplan} \in \textit{FailedInstance} \\
\mathsf{Let}\ \textit{origevent} == (\mu\, e : \textit{Event} \mid \textit{the e.id} = (\textit{the executingplan}).\textit{id}); \\
\quad \textit{env} == (\textit{the executingplan}).\textit{env} \bullet \\
\;\mathsf{Let}\ \textit{failacts} == (\textit{the executingplan}).\textit{origplan.fail} \bullet \\
\;\;\mathsf{Let}\ \textit{groundfailacts} == \textit{map}\ (\textit{ASIntAction env})\ \textit{failacts} \bullet \\
\;\;\textit{beliefs}' = \textit{fold performintaction beliefs failacts} \land \\
\;\;\mathrm{ran}\ \textit{events}' = (\mathrm{ran}\ \textit{events} \setminus \{\textit{origevent}\}) \cup \\
\qquad\quad \{\textit{MakeEvent}(\textit{origevent.trig}, \textit{origevent.id}, \textit{origevent.env}, \\
\qquad\qquad\qquad\qquad (\textit{origevent.failures} \cup \{\textit{executingplan}\}))\} \\
\textit{the selectedintention}' = \textit{tail}(\textit{the selectedintention}) \\
\hline
\end{array}
$$

6 Concluding Remarks

The BDI model that underpins dMARS is similar to other computational models used in agent programming environments. In particular, it is closely related to the Concurrent MetateM programming language, as described in [Mulder *et al.*,]. In Concurrent MetateM, an agent is programmed by giving it an *executable specification* of its behaviour, where such a specification is expressed as a set of temporal logic formulae of the form $past \Rightarrow future$. Execution of these rules proceeds by matching the past time antecedents of temporal logic rules against future time consequents; any rules that fire then become *commitments*, which the agent must subsequently attempt to satisfy. Perhaps the main conceptual difference between Concurrent MetateM and the dMARS model is that in dMARS, control structures are explicitly coded in plans; in Concurrent MetateM, a run-time execution algorithm is responsible for determining control, in that it must attempt to find an execution that simultaneously satisfies its commitments. In [Mulder *et al.*,], the relationship between Concurrent MetateM and dMARS is used to encode a dMARS-like interpreter as a set of Concurrent MetateM rules. The same paper also provides an encoding of an abstract BDI interpreter using the DESIRE system (essentially an executable specification framework for knowledge-based systems).

In addition, a new abstract programming language [Hindricks *et al.*,] with a well-defined formal semantics in terms of a transition system has been based on the BDI model. This language uses features of both logic programming and imperative programming, and captures some of the features of other BDI-based languages such as AGENT-0 and AgentSpeak(L). The key distinction between this language and the operation of a dMARS agent is that it contains no notion of events and, indeed, the authors suggest that events are not necessary for agent languages that attempt to capture the *intuitions* of the BDI model. However, in comprison with the dMARS formalisation contained in our paper, which provides a strong, computational model of the operation of a dMARS agent (and from which we claim systems can be implemented), it is not clear how such a strong relation between the semantics and a possible implementation might be made in this other work.

As the technology of intelligent agents matures further, we can expect to see a progression from the "scruffiness" of early investigative work to the "neatness" of rigour and formality. In this paper, we have contributed to the growing body of "neat" intelligent agent research, by presenting a complete formal specification of the best-known and most important agent architecture developed to date.

The specification we have presented in this paper is significant for a number of reasons. First, we need to understand clearly how an architecture works in order that we can evaluate it against others. Implementations are too low-level to allow such evaluations to take place. Formal specifications, using standard software engineering tools like the widely used Z language, are an ideal medium through which to communicate the operation of an architecture (e.g. [d'Inverno *et al.*, June 1997]).

Second, there are understood methods for moving from an abstract specification in Z to an implementation, through a systematic process of refinement and reification. Such a process is not possible from a natural language description. Reimplementation and evaluation of the PRS architecture in different languages and environments is therefore a realistic possibility.

Finally, by understanding the model-theoretic foundations of PRS, (through rigorously defining the data structures and operations on those structures that constitute the architecture), we make it possible to develop a proof theory for the architecture. Such a proof theory has been developed for the MyWorld architecture [Wooldridge, 1995], and also for Rao's AgentSpeak(L) [Rao, 1996], which is itself a restricted version of PRS. Once such an axiomatisation is available, there will exist a straight line from the implementation of PRS to its theory, making it possible to compare the actual behaviour of the architecture against the philosophical idealisations of it that have been developed by BDI theorists [Rao and Georgeff, 1991]. In future work, we hope to investigate such axiomatisations.

Acknowledgements:

Many thanks to Michael Georgeff and Anand Rao who provided many illuminations and insights in discussions with the first author during development of the specification contained in this paper. Thanks to the University of Westminster and the Australian Artificial Intelligence Institute, for supporting and hosting the first author during the development of this work. The specification contained in this document has been checked for type correctness using the fuzz package [Spivey, 1992a]. Thanks also to Sorabain de Lioncourt who identified an error in the original specification.

References

[Bratman *et al.*, 1988] M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349–355, 1988.

[Cohen and Levesque, 1990] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.

[d'Inverno and Luck, Forthcoming] M. d'Inverno and M. Luck. A formal specification of AgentSpeak(L). *Journal of Logic and Computation*, Forthcoming.

[d'Inverno *et al.*, June 1997] M. d'Inverno, M. Priestley, and M. Luck. A formal framework for hypertext systems. *IEE Proceedings - Software Engineering Journal*, 144(3):175–184, June, 1997.

[Emerson and Halpern, 1986] E. A. Emerson and J. Y. Halpern. 'Sometimes' and 'not never' revisited: on branching time versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[Genesereth and Nilsson, 1987] M. R. Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers: San Mateo, CA, 1987.

[Georgeff and Lansky, 1987] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA, 1987.

[Georgeff and Rao, 1996] M. P. Georgeff and A. S. Rao. A profile of the Australian AI Institute. *IEEE Expert*, 11(6):89–92, December 1996.

[Hindricks *et al.*,] K. Hindricks, F. de Boer, W. van der Hoek, and J. Meyer, J. Formal semantics for an abstract agent programming language. In this volume.

[Mulder *et al.*,] M. Mulder, J. Treur, and M. Fisher. Agent modelling in concurrent METATEM and DESIRE. In this volume.

[Rao and Georgeff, 1991] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In R. Fikes and E. Sandewall, editors, *Proceedings of Knowledge Representation and Reasoning (KR&R-91)*, pages 473–484. Morgan Kaufmann Publishers: San Mateo, CA, April 1991.

[Rao and Georgeff, 1992] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In C. Rich, W. Swartout, and B. Nebel, editors, *Proceedings of Knowledge Representation and Reasoning (KR&R-92)*, pages 439–449, 1992.

[Rao and Georgeff, 1995a] A. S. Rao and M. Georgeff. BDI Agents: from theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, San Francisco, CA, June 1995.

[Rao and Georgeff, 1995b] A. S. Rao and M. P. Georgeff. Formal models and decision procedures for multi-agent systems. Technical Note 61, Australian AI Institute, Level 6, 171 La Trobe Street, Melbourne, Australia, June 1995.

[Rao, 1996] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (LNAI Volume 1038)*, pages 42–55. Springer-Verlag: Heidelberg, Germany, 1996.

[Spivey, 1992a] J. M. Spivey. *The fUZZ Manual*. Computing Science Consultancy, 2 Willow Close, Garsington, Oxford OX9 9AN, UK, 2nd edition, 1992.

[Spivey, 1992b] M. Spivey. *The Z Notation (second edition)*. Prentice Hall International: Hemel Hempstead, England, 1992.

[Wooldridge and Jennings, 1995] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[Wooldridge, 1995] M. Wooldridge. This is MyWorld: The logic of an agent-oriented testbed for DAI. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI Volume 890)*, pages 160–178. Springer-Verlag: Heidelberg, Germany, January 1995.

A Auxiliary Z Definitions

The function, *fold*, takes a function, an initial value and a sequence and applies each element in the sequence to the initial value in turn. The function, *map*, takes another function and applies it to every element in a list. Similarly, *mapset*, applies a function to every element in a set.

$$[X, Y]$$

$$\begin{array}{l}
fold : (X \rightarrow Y \rightarrow X) \rightarrow X \rightarrow (\mathrm{seq}\ Y) \rightarrow X \\
map : (X \rightarrow Y) \rightarrow (\mathrm{seq}\ X) \rightarrow (\mathrm{seq}\ Y) \\
mapset : (X \rightarrow Y) \rightarrow (\mathbb{P}\ X) \rightarrow (\mathbb{P}\ Y) \\
\hline
\forall f : (X \rightarrow Y \rightarrow X);\ x : X;\ y : Y;\ ys : \mathrm{seq}\ Y \bullet \\
\qquad fold\ f\ x\ \langle\rangle = x \land \\
\qquad fold\ f\ x\ (\langle y \rangle \frown ys) = fold\ f\ (f\ x\ y)\ ys \\
\forall f : X \rightarrow Y;\ x : X;\ xs : \mathrm{seq}\ X \bullet \\
\qquad map\ f\ \langle\rangle = \langle\rangle \land \\
\qquad map\ f\ \langle x \rangle = \langle f\ x \rangle \land \\
\qquad map\ f\ (xs \frown ys) = map\ f\ xs \frown map\ f\ ys \\
\forall f : X \rightarrow Y;\ xs : \mathbb{P}\ X \bullet \\
\qquad mapset\ f\ xs = \{x : xs \bullet f\ x\}
\end{array}$$

It is useful to be able to assert that an element is optional. The following definitions provide for a new type, *optional*$[T]$, for any existing type, T, along with the predicates, *defined* and *undefined*, which test whether an element of *optional*$[T]$ is defined or not. The function, *the*, extracts the element from a defined member of *optional*$[T]$.

$$optional[X] == \{xs : \mathbb{P}\ X \mid \#\ xs\ \leq\ 1\}$$

$$[X]$$

$$\begin{array}{l}
defined_,\ undefined_ : \mathbb{P}(optional[X]) \\
the : optional[X] \rightarrow\!\!\!\!\!+ X \\
\hline
\forall xs : optional[X] \bullet\ defined\ xs \Leftrightarrow \#\ xs\ =\ 1 \land \\
\qquad\qquad\qquad\qquad undefined\ xs \Leftrightarrow \#\ xs\ =\ 0 \\
\forall xs : optional[X] \mid\ defined\ xs \bullet \\
\qquad\qquad\qquad\qquad the\ xs = (\mu\, x : X \mid x \in xs)
\end{array}$$

B Binding

The standard definition of a substitution is a mapping from variables to terms such that no variable contained in any of the terms is in the domain of the mapping [Genesereth and

Nilsson, 1987]. This is represented as a partial function between variables and terms since, in general, only some variables will be mapped to a term.

$$
\begin{array}{l}
Substitution == \\
\quad \{f : Var \rightarrowtail\!\!\!\!\!\!+\; Term \mid (\operatorname{dom} f) \cap (\bigcup(mapset\ termvars\ (\operatorname{ran} f))) = \varnothing\}
\end{array}
$$

The function *ApplySubTerm* applies either the identity mapping to a variable if the variable is not in the domain of the substitution, or applies the substitution if it is in the domain.

$$
\begin{array}{|l}
ASVar : Substitution \rightarrow Var \rightarrow Term \\
\hline
\forall \psi : Substitution;\ v : Var \bullet \\
\quad ASVar\ \psi\ v = (\{x : Var \bullet (x, var\ x)\} \oplus \psi)\ v
\end{array}
$$

We can then define what it means for a substitution to be applied to a term, internal action, a situation formula, a plan, a goal, a belief formula and a trigger event, as given by *ASTerm*, *ASIntAction*, *ASSitForm*, *ASPlan*, *ASGoal*, *ASBeliefFormula* and *ASTrigEvent*, respectively.

Consider two substitutions τ and σ such that no variable bound in σ appears anywhere in τ. The composition of τ with σ, written $\tau \ddagger \sigma$, is obtained by applying τ to the terms in σ and combining these with the bindings from τ. For example if $\tau = \{x/A, y/B, z/C\}$ and $\sigma = \{u/A, v/F(x, y, z)\}$ then, since none of the variables bound in σ (u, v) appear in τ, it is meaningful to compose τ with σ. In this case $\tau \ddagger \sigma = \{u/A, v/F(A, B, C), x/A, y/B, z/C\}$.

$$
\begin{array}{|l}
[X, Y] \\
\_ \ddagger \_ : Substitution \times Substitution \rightarrow Substitution \\
\hline
\forall \tau, \sigma : Substitution \mid \\
\quad (\operatorname{dom} \sigma) \cap ((\operatorname{dom} \tau) \quad \cup \quad \bigcup(mapset\ termvars\ (\operatorname{ran} \tau))) = \varnothing \bullet \\
\quad\quad \tau \ddagger \sigma = (\tau \cup \{x : Var;\ t : Term \mid (x, t) \in \sigma \bullet (x, ASTerm\ \tau\ t)\}) \\
\hline
\end{array}
$$

A substitution is a *unifier* for two expressions if the substitution, applied to both of them, makes them equal. A substitution is *more general* than another substitution if there exists a third substitution which, when composed with the first, gives the second. The most general unifier of two expressions is a substitution which unifies the expressions such that there is no other unifier that is more general. Here we define the signatures for the most general unifier of two trigger events, a goal with a set of beliefs, and a trigger event with a goal.

$$
\begin{array}{|l}
mguevents\ : (TriggerEvent \times TriggerEvent) \rightarrowtail\!\!\!\!\!\!+\; Substitution \\
mguquery\ : (Goal \times \mathbb{P}\, Belief) \rightarrowtail\!\!\!\!\!\!+\; Substitution \\
mgugoal\ : (TriggerEvent \times Goal) \rightarrowtail\!\!\!\!\!\!+\; Substitution
\end{array}
$$