Autonomous Agents and Multi-Agent Sytems, 9, 5–53, 2004
© 2004 Kluwer Academic Publishers. Manufactured in The Netherlands.

The dMARS Architecture: A Specification of the Distributed Multi-Agent Reasoning System

MARK D'INVERNO dinverm@westminster.ac.uk
Cavendish School of Computer Science, University of Westminster, London W1M 8JS, UK

MICHAEL LUCK mml@ecs.soton.ac.uk
School of Electronics and Computer Science, University of Southampton S017 1BJ, UK

MICHAEL GEORGEFF michael.georgeff@infotech.monash.edu.au
Faculty of Information Technology Monash University, Victoria 3800, Australia

DAVID KINNY dnk@agentis.net
Agentis Software, Level 2, 33 Lincoln Square South, Carlton, Victoria 3053, Australia

MICHAEL WOOLDRIDGE mjw@csc.liv.ac.uk
Department of Computer Science, University of Liverpool, Liverpool L69 7ZF, UK

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.

Keywords: agent architectures, procedural reasoning system, BDI, formal specification.

1. Introduction

Since the mid 1980s, many control architectures for practical reasoning agents have been proposed [56]. 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 [19], 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 [21].

For example, Oasis is a system for air traffic management that can handle the flow of over 100 aircraft arriving at an airport. It addresses issues of scheduling aircraft, comparing actual progress with established sequences of aircraft, estimating delays, and notifying controllers of was to correct deviations. The prototype implementation comprised several different kind of agents, including aircraft agents and coordinator agents, each of which was based around PRS. Oasis successfully completed operational tests at Sydney Airport in 1995. Similarly, dMARS has been used as the basis of an agent-based simulation system, Swarmm, developed for Australia's Defence Science and Technology Organisation, to simulate air mission dynamics and pilot reasoning. More recent work has sought to apply dMARS to represent different roles in an organisation in more general business software for running call centres or internet services. The Agentis system allowed the generation of fully automated, server-side customer-service applications. The first major installation aimed to use 4000 dMARS agents.

The PRS architecture has its conceptual roots in the belief-desire-intention (BDI) model of practical reasoning developed by Michael Bratman and colleagues [3], 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., [43] for a survey).

A whole range of practical development efforts relating to BDI systems have been undertaken, and can be divided into those that are almost straight-forward reimplementations and refinements of PRS and dMARS on the one hand, and those that are more loosely based on broader BDI principles. The former category includes systems such as UM-PRS developed at the University of Michigan, a freely available reimplementation of PRS [36], PRS-Lite developed at SRI by Myers [41], Huber's JAM system [31], and the commercial JACK system [5], which are respectively increasingly sophisticated. The latter category includes Bratman's IRMA architecture [3], Burmeister and Sundermeyer's COSY [4], and the GRATE* system developed by Jennings [33], all of which are reviewed in [23]. Nevertheless, PRS and dMARS remain as the exemplar BDI systems.

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 [45], 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 [43]. 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 [42]. 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 [49]) of AgentSpeak(L) [13]. 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 summary, there are several logical models of abstracted BDI systems, and several efforts to define particular BDI systems. Although each of these has been successful in terms of its specific aims, the situation remains that the abstract theoretical models are divorced from the realities of the computational architectures, while the specifications of particular systems has largely been point work, concentrating on an accurate representation rather than a consideration of the general architecture.

BDI is regarded by many as a foundational topic in many areas of artificial intelligence and cognitive science, both for its folk psychological basis for understanding and predicting behaviour, and for its use and deployment in systems and applications as described above. Because of the distinct approaches to BDI, the computational and the theoretical, interpretation and understanding of agent architectures tends to be done in only one of these areas.

In this paper, we seek to provide a complete, general, computational model of probably the most widely used, implemented BDI system, dMARS, and show how it relates to other architectures. The particular contributions of this paper are threefold: first, we provide the first complete yet general description of dMARS and its operation; second we show how the the general model may be modified to accommodate architectural variations within the basic model, and how this is facilitated by the formalisation we have provided; third, we show how the formal model may be used as an evaluation framework to provide a means for a stronger and cleaner comparison with other architectures. In this sense the formal model here can be considered a reference specification. On the one hand, it can be put to straight- forward use by developers seeking to build BDI systems. On the other, it can be used for analysis, design and evaluation purposes both by researchers seeking to understand and compare complete architectures and specific architectural variants, and by researchers seeking to develop new architectures, both general models, and those tailored to particular application domains.

The work continues and extends the work begun in providing the reformalisation of Rao's AgentSpeak(L) described above, by giving the first complete description of dMARS (the system upon which AgentSpeak(L) is based) and its operation through an abstract formal specification. In so doing, we provide an operational semantics for dMARS, and thus offer 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. Our approach is very similar to that of [55], 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 Z notation used subsequently to develop the specification of dMARS in the rest of the paper. The specification itself begins in Section 4, in which we describe the basic types and

primitive components of the system, and in Section 5 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.

2. An overview of dMARS

Developed by Georgeff and Lansky [19], the PRS is arguably the best-known agent architecture. It has been widely applied in fault-diagnosis on the space shuttle [32] air traffic control, air mission dynamics and business solutions [21]. Both PRS and its successor *dMARS* are prime examples of the popular agent paradigm introduced above as the BDI approach [3]. 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. In implemented BDI agents the typically representation of an agents beliefs is symbolically (e.g., as propositions [19]). An agent's *desires* (or goals, in the system) intuitively correspond to the tasks allocated to it.

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 [7]. In dMARS agents, the BDI model is operationalised 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

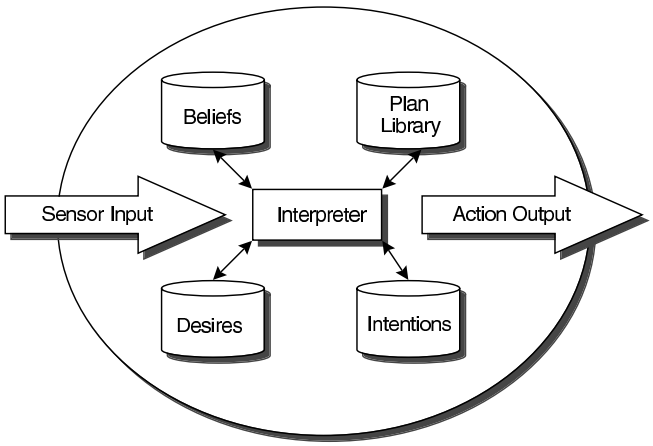


Figure 1. A BDI agent architecture: PRS.

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 *invocation condition* 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. Once the intention stack executes successfully then the event is removed from the event queue. 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 performed 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.

Although PRS and dMARS allow explicit forking and spawning (enabling more than one intention to be pursued at a time), only one intention is used to respond to a goal. This is because *usually* it is only worthwhile pursuing a goal in one way – there is little value in achieving the same goal in more than one way. (Sometimes one does follow two paths at once, to mitigate risk, however. For example, if your goal is to get married, it may be worth dating two people at once to increase your chances of

success). The latter case is relatively rare, and in those situations in which it is desired, it can be explicitly programmed using fork or spawn operations.

The response to events, however, should trigger all matching plans. Since the semantics for event-triggered plans is "whenever some event occurs do something", all plans must be triggered in response to an event. If the user does not wish this, it can be bypassed by having one plan respond to the event and then explicitly handling the others using goal invocation.

In PRS, planning ahead is possible through *meta-plans* (or alternatively decision theory, specialised knowledge, etc). However, in practice, users rarely use meta-plans because this knowledge can always be embedded in the object plan – it just requires the inclusion of an extra step in the plan to call the relevant planning engine (which could also include the set of plans used by dMARS). For example, a plan to go overseas would include a step "plan itinerary". The reason dMARS does not do this automatically is the assumption that in most cases the environment is so uncertain that planning ahead will not help – it is better to try something and if that does not work, try something else. For special cases, a special planning component can be called, as mentioned above. In general, however, the most sensible course of action, as in much human and animal behaviour, is to do anything appropriate to achieving the goal, and reassess or retry on failure.

Other efforts to give a formal semantics to BDI architectures include a range of *BDI logics* that have been developed by Rao and Georgeff [43]. These logics are extensions to the branching time logic CTL* [17], which also contain normal modal connectives for representing beliefs, desires, and intentions. Most work on BDI logics has focused on possible relationships between the three 'mental states' [44] and, more recently, on developing proof methods for restricted forms of the logics. In future work we will investigate the relationship between this work and the operational semantics described in this paper.

3. Notation

3.1. The Z specification language

There is a large and varied number of formal techniques and languages available to specify properties of software systems [11]. These include state-based languages such as VDM [34], Z [49] and B [35], process-based languages such as CCS [39] and CSP [30], temporal logics [17], modal Logics [6], and Statecharts [50]. There are advocates for each of these approaches to modelling various aspects of computer programs, but it is perhaps the most controversial aspect of the use of formal techniques in the agent field is that they do not directly relate strongly enough to the construction of actual software.

In choosing the language Z, therefore, we deliberately adopt a technique which not only enables designs of systems to be developed formally, but allows for the systematic reduction of these specifications to implementations. Furthermore, Z is a specification language that is increasingly being used both in industry and academia, as a strong and elegant means of formal specification, and is supported by a large

array of books (e.g., [2, 25, 47, 53]), industrial case studies (e.g., [8, 10, 51]), well-documented refinement methods [52], and available tools for animating operations (e.g., [26, 46, 48]). In this respect, Z offers just the qualities we need, helping us maintain the critical link between formal models and implemented systems.

Additionally, Z has other benefits.

- It is more *accessible* than many other formalisms since it is based on existing *elementary* components such as set theory and first order predicate calculus.
- It is an extremely expressive language, allowing a consistent, unified and structured account of a computer system and its associated operations.
- As with the related language VDM [24], Z is gaining increasing acceptance as a tool within the artificial intelligence community (e.g., [9, 22, 40, 55]) and is therefore appropriate for the current work in terms of standards and dissemination capabilities.

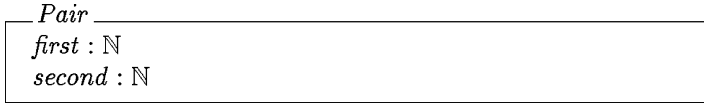
In this section, we introduce the syntax of the Z language by way of example. Much of this will be intuitive to many, but for a more detailed exposition, the authors recommend consulting one of the array of existing Z text books cited above.

3.2. *Syntax*

Z, developed at Oxford University in the late 1970s, is based on set theory and first order predicate calculus. The key syntactic element of Z is the schema, which allows specifications to be structured into manageable modular components. Z schemas consist of two parts, the upper declarative part, which declares variables and their types, and the lower predicate part, which relates and constrains those variables.

If we allow d to stand for a set of declarations, p to be a set of predicates and S to be the name of a schema we have the basic notation for a schema as follows:

It is therefore appropriate to liken the semantics of a schema to that for Cartesian products. For example suppose we define a schema as follows:



This is very similar to the following Cartesian product type.

$$Pair == \mathbb{N} \times \mathbb{N}$$

The difference between these forms is that there is no notion of order in the variables of the schema type. In addition, a schema may have a predicate part that can be used to constrain the state variables. Thus, we can state that the variable, *first*, can never be greater than *second*.

Pair

$$first : \mathbb{N}$$
$$second : \mathbb{N}$$

$$first \leq second$$

Modularity is facilitated in Z by allowing schemas to be included within other schemas. For example, if we wished to include all the information of schema, S, in another schema T along with a further set of declarations, d and predicates, p, we would write the schema below.

T

S
d
---
p

We can select a state variable, *var*, of a schema, *schema*, by writing *schema.var*. For example, it should be clear that *Pair.first* refers to the variable *first* in the schema *Pair*.

Now, operations in a state-based specification language are defined in terms of *changes to the state*. Specifically, an operation relates variables of the state after the operation (denoted by dashed variables) to the value of the variables before the operation (denoted by undashed variables).

ΔS

S
S'

Operations may also have inputs (denoted by variables with question marks), outputs (exclamation marks) and a precondition. In the *GettingCloser* schema below, there is an operation with an input variable, *new*?; if *new*? lies between the variables *first* and *second*, then the value of *first* is replaced with the value of *new*? The original value of *first* is the output as *old*! The $\Delta Pair$ symbol, is an abbreviation for $Pair \wedge Pair'$ and, as such, includes in this schema all the variables and predicates of the state of *Pair* before and after the operation. Note that the equals symbol, $=$, is used both for equality constraint and assignment. Note, also, that because by writing $\Delta Pair$ we are essentially just including all the declarations and predicates (with both

dashed and undashed state variables) from *Pair* we can simply refer to them as *first*, say, rather than *Pair.first*.

$$
\begin{array}{|l}
\hline
\textit{GettingCloser} \\
\textit{new}? : \mathbb{N} \\
\Delta \textit{Pair} \\
\textit{old}! : \mathbb{N} \\
\hline
\textit{first} \leq \textit{new}? \\
\textit{new}? \leq \textit{second} \\
\textit{first}' = \textit{new}? \\
\textit{second}' = \textit{second} \\
\textit{old}! = \textit{first} \\
\hline
\end{array}
$$

Finally, we can declare global, axiomatic definitions, where the value of the variables is *fixed* as follows:

$$
\begin{array}{|l}
d \\
\hline
p \\
\end{array}
$$

where d is a set of declarations and p is a set of predicates. For example, we can declare two global variables that must be strictly less than 10 and strictly more than ten, respectively, as follows:

$$
\begin{array}{|l}
\textit{min}, \textit{max} : \mathbb{N} \\
\hline
\textit{min} < 10 \land \textit{max} > 10 \\
\end{array}
$$

To introduce a type in Z where no information about the elements within that type are known, a *given set* is used. For example, [*TREE*] represents the *type* of all trees without saying anything about the nature of the individual elements within the type. If we wish to state that a variable takes on a value, a set of values, or an ordered pair of values of this type, we write $x : \textit{TREE}$, $x : \mathbb{P}\ \textit{TREE}$ and $x : \textit{TREE} \times \textit{TREE}$, respectively. Suppose we have $xs : \textit{TREE} \times \textit{TREE}$ then the expression *first xs* and *second xs* are the first element and second element of the ordered pair *xs*.

The most important type is the *relation* type, expressing a mapping between *source* and *target* sets. The type of a relation with source X and target Y is $\mathbb{P}(X \times Y)$, and any element of this type (or *relation*) is simply a set of ordered pairs.

The definition of functions is also standard: relations are functions if no element from the source is related to more than one element in the target set. If every element in the source set is related, then the function is *total*; *partial* functions do not relate every source set element. Total functions are represented by ($\rightarrow$) and *partial* functions by ($\nrightarrow$). Sequences are simply special types of function where the domain consists of contiguous natural numbers.

We introduce two examples of relations, *Rel*1 and *Rel*2, by way of illustration. *Rel*1 defines a *function* between trees, while *Rel*2 defines a *sequence* of trees. The size of the source set determines whether *Rel*1 is a partial or a total function; If the only elements of the source set are *tree*1, *tree*2 and *tree*3, then the function is total, otherwise it is partial.

$$Rel1 = \{(tree1, tree2), (tree2, tree3), (tree3, tree2)\}$$

$$Rel2 = \{(3, tree3), (2, tree2), (1, tree4)\}$$

The sequence *Rel*2 is more usually written in Z as $\langle tree4, tree2, tree3\rangle$. Operations on sequences include taking the head, tail and concatenation.

$$head\ \langle tree4, tree2, tree3\rangle = tree4$$
$$tail\ \langle tree4, tree2, tree3\rangle = \langle tree2, tree3\rangle$$
$$\langle a, c, b\rangle \frown \langle b, c, a\rangle = \langle a, c, b, b, c, a\rangle$$

The *domain* of a relation or function is the set of source elements that are related. Similarly, the *range* is the set of target set elements which are related. The *inverse* of a relation is obtained by reversing each of the ordered pairs so that the domain becomes the range and the range becomes the domain. A relation can be restricted to a particular subset of its domain using *domain restriction*. Similarly a relation can be anti-restricted by a set in such a way such that the resulting relation does not contain any ordered pairs whose second element is in the restricting set. This is known as *anti-range restriction*. Lastly, one relation can be updated by another relation using *relational overriding*. The second relation can be thought of as "new" information about its domain elements, overwriting any existing pairs whose first element is in the domain of the second.

Examples of these operators can be seen below.

$$\mathrm{dom}\ Rel1 = \{tree1, tree2, tree3\}$$
$$\mathrm{ran}\ Rel1 = \{tree2, tree3\}$$
$$\mathrm{dom}\ Rel2 = \{1, 2, 3\}$$
$$\mathrm{ran}\ Rel2 = \{tree2, tree3, tree4\}$$
$$Rel1^{\sim} = \{(tree2, tree1), (tree3, tree2), (tree2, tree3)\}$$
$$\{tree2, tree3, tree4\} \lhd Rel1 = \{(tree2, tree3), (tree3, tree2)\}$$
$$Rel1 \rhd \{tree1, tree2\} = \{(tree2, tree3)\}$$
$$Rel1 \oplus \{(tree1, tree3), (tree2, tree2), (tree2, tree3)\} =$$
$$\{(tree1, tree3), (tree2, tree2), (tree2, tree3), (tree3, tree2)\}$$

Sets of elements can be defined using set comprehension. For example, the expression $\{x : \mathbb{N} \mid x < 4 \bullet x * x\}$ denotes the set $\{0, 1, 4, 9\}$. To state that, say, the square of every natural number greater than 10 is greater than 1000, we write $\forall n : \mathbb{N} \mid n > 10 \bullet n * n > 100$. The expression, $\mu\, a : A \mid P$, selects the unique element from the type A, which satisfies the predicate P.

A summary of the notation that is used in this paper is given in Figure 2. As well as the standard Z we also provide some auxiliary Z definitions in Appendix C. As stated earlier, more complete treatments of Z and its formal semantics [31] can be found elsewhere. It should also be noted that whilst this paper does contain formal material we will always attempt to provide intuition, examples and discussions in English to support the mathematical specification of dMARS.

4. Beliefs, goals, and actions

Now in order to illustrate the operation of a dMARS agent, we will build up an example scenario throughout the course of development of the specification. Our example, loosely inspired by the example used by Rao to illustrate AgentSpeak(L) [42], is concerned with the design of an agent for clearing rubbish from a collection of adjacent locations. Some of these locations (which we will call *boxes*) contain rubbish bins that may be either full or empty, and the agent is designed to empty bins into a truck that it can drive away to a dump. A simple diagram of a particular state of this world is illustrated in Figure 3 which, for example, includes the empty *binA* in *box*2.

4.1. *Beliefs and belief formulae*

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 constants, variables, function and predicate symbols. Since we are not concerned with the contents of these sets, we define them as *given sets*.

$$[Const, Var, FunSym, PredSym]$$

A *term* is either a constant, a variable or a function symbol applied to a (non-empty) sequence of terms.

$$Term ::= const\langle\langle Const\rangle\rangle \mid var\langle\langle Var\rangle\rangle \mid fun\langle\langle FunSym \times \mathrm{seq}_1\ Term\rangle\rangle$$

Definitions and declarations

a, b	Identifiers
p, q	Predicates
s, t	Sequences
x, y	Expressions
A, B	Sets
R, S	Relations
$d;\ e$	Declarations
$a == x$	Abbreviated definition
$[a]$	Given set
$A ::= b\langle\langle B\rangle\rangle$	
$\mid c\langle\langle C\rangle\rangle$	Free type declaration
$\mu\, d \mid P$	Definite description
let $a == x$	Local variable definition

Logic

$\neg\, p$	Logical negation
$p \wedge q$	Logical conjunction
$p \vee q$	Logical disjunction
$p \Rightarrow q$	Logical implication
$p \Leftrightarrow q$	Logical equivalence
$\forall X \bullet q$	Universal quantification
$\exists X \bullet q$	Existential quantification

Sets

$x \in y$	Set membership
$\{\}$	Empty set
$A \subseteq B$	Set inclusion
$\{x, y, \ldots\}$	Set of elements
$(x, y, \ldots)$	Ordered tuple
$A \times B \times \ldots$	Cartesian product
$\mathbb{P} A$	Power set
$\mathbb{P}_1 A$	Non-empty power set
$A \cap B$	Set intersection
$A \cup B$	Set union
$A \setminus B$	Set difference
$\bigcup A$	Generalised union
$\# A$	Size of a finite set
$\{d;\ e \ldots \mid p \bullet x\}$	Set Comprehension

Functions

$A \nrightarrow B$	Partial function
$A \rightarrow B$	Total function

Relations

$A \leftrightarrow B$	Relation
dom R	Relation Domain
ran R	Relation Range
$R^{\sim}$	Relational Inverse
$A \lhd R$	Domain restriction
$A \rhd R$	Anti-range restriction
$R \oplus S$	Relational overriding

Sequences

seq A	Sequence
$\text{seq}_1\, A$	Non-empty
$\langle\rangle$	Empty
$\langle x, y, \ldots\rangle$	Sequence
$s \frown t$	Concatenation
$head\ s$	First element
$tail\ s$	All but first

Schema notation

S d --- p	Schema
$[X]$ d --- p	Generic Schema
d --- p	Axiomatic def
T S d --- p	Inclusion
ΔS S S'	Operation
ΞS ΔS --- $\theta S' = \theta S$	No Change of State
$z.a$	Component

Figure 2. Summary of Z notation used.

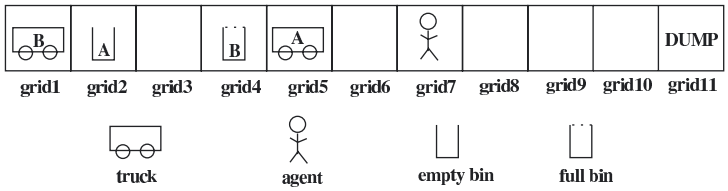


Figure 3. Example scenario of rubbish collection.

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

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

With reference to our example scenario, we can suppose that: the set of constants includes $box1$, $box2$, ..., and $binA$, $binB$, $truckT$; the set of variables includes X and Y; and the set of predicate symbols includes *location*, *adjacent*, *full*, *empty*. The set of atoms can thus include the following expressions for example.

$$location(X, Y), location(binA, Y), full(binB)$$

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

$$BelForm ::= pos\langle\langle Atom\rangle\rangle \mid not\langle\langle Atom\rangle\rangle$$

Now, a critical distinction can be made between belief formulae that contain variables and those that do not. The latter belief formulae are referred to as *beliefs*, and encode the representation an agent has of its environment. In order to define beliefs, however, we must first define an auxiliary function, *belformulavars*, which returns the variables within a belief formula. This relies, in turn, on defining two further functions, which return the variables in terms and atoms respectively. Note here that we must apply the function *termvars* to the terms (*const c*) and (*var v*), we could not apply it to either variables (v) or constants (c).

$$\begin{array}{|l} \textit{belformulavars} : \textit{BelForm} \to (\mathbb{P}\ \textit{Var}) \\ \textit{atomvars} : \textit{Atom} \to (\mathbb{P}\ \textit{Var}) \\ \textit{termvars} : \textit{Term} \to (\mathbb{P}\ \textit{Var}) \\ \hline \forall\, \textit{atom} : \textit{Atom} \bullet \\ \quad \textit{belformulavars}\ (\textit{not atom}) = \textit{atomvars atom} \land \\ \quad \textit{belformulavars}\ (\textit{pos atom}) = \textit{atomvars atom} \land \\ \quad \textit{atomvars atom} = \bigcup(\text{ran}(\textit{map termvars atom.terms})) \\ \forall\, c : \textit{Const};\ v : \textit{Var};\ f : \textit{FunSym};\ ts : \text{seq } \textit{Term} \bullet \\ \quad \textit{termvars}\ (\textit{const } c) = \{\} \land \\ \quad \textit{termvars}\ (\textit{var } v) = \{v\} \land \\ \quad \textit{termvars}\ (\textit{fun}(f, ts)) = \bigcup(\text{ran}(\textit{map termvars } ts)) \end{array}$$

The set of beliefs is thus defined as those containing no variables.

$$Belief == \{b : BelForm \mid belformulavars\ b = \{\} \bullet b\}$$

For example, the expressions *pos location*$(agent, X)$ and *not full*(Y) are belief formulae, but since they contain variables they are not beliefs. Only when the variables are bound, do the belief formulae become actual beliefs about the world. For example, *pos location*$(agent, box7)$ and *not full*$(binA)$ indicate that the agent is at $box7$ and that $binA$ is not full.

4.2. Goals

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.

$$\begin{aligned} SitForm ::= &\ belform\langle\langle BelForm\rangle\rangle \\ \mid &\ and\langle\langle SitForm \times SitForm\rangle\rangle \\ \mid &\ or\langle\langle SitForm \times SitForm\rangle\rangle \\ \mid &\ true \\ \mid &\ false \end{aligned}$$

We need to consider a *temporal* formula because we assume that the achievement of a goal takes place over a sequence of atomic state transitions. Given states $s_1, s_2, s_3, \ldots, s_n$, $!\phi$ is true if and only if ϕ holds in s_n, and $?\phi$ is true if and only if ϕ holds in s_1.

- In the trivial case, if ϕ is already believed, then $!\phi$ and $?\phi$ are both true without the need to execute any plan.
- If $\neg\phi$ is already believed, then $?\phi$ is not true. The expression $!\phi$ can be made true by executing a plan that guarantees, upon successful completion, that $!\phi$ is true in the final state (i.e., ϕ holds in s_n).
- If ϕ is unknown, then $?\phi$ can be made true by executing a plan that guarantees, upon successful completion, that ϕ was indeed true prior to the plan being

executed (i.e., ϕ holds in s_1). Thus, the fact that the plan completes successfully indicates that ϕ was true prior to execution. Equally, $!\phi$ can be made true by executing a plan that guarantees upon successful completion ϕ is true in the final state (i.e., ϕ holds in s_n).

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.

$$Goal ::= achieve\langle\langle BelForm\rangle\rangle \mid query\langle\langle SitForm\rangle\rangle$$

For example, *achieve empty*(*binB*) is the goal to empty *binB*, and the expression *query full*(*binB*) is the goal to determine whether the same bin is full.

4.3. *Actions*

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).

ExtAction

$name$: *ActionSym*
$terms$: seq *Term*

An example of such an action is *move*(*box*7, *box*8) which moves the agent from *box*7 to *box*8.

Internal actions may be one of two types: add or remove a belief from the data base (cf. the PROLOG `assert` and `retract` clauses). As specified later, an agent's database contains only ground atoms; it is therefore not possible to add or remove an atom that contains variables.

$$IntAction ::= add\langle\langle BelForm\rangle\rangle \mid remove\langle\langle BelForm\rangle\rangle$$

Now, if the agent moved as determined by the example above, it would clearly need to perform the following (internal) actions in order to update its beliefs: *add location*(*agent*, *box*8) and *remove location*(*agent*, *box*7). Similarly, if the agent emptied *binB* then it would add the predicate *empty*(*binB*) and remove the predicate *full*(*binB*).

5. Plans

As described earlier, the BDI model is operationalised in dMARS by *plans*, with each agent having its own *plan library* representing its *procedural knowledge*, or *know-how*: knowledge about how to bring about states of affairs.

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 (start state) to any leaf node (end state). This is illustrated in Figure 4 in which there is an invocation condition, a context, a body, maintenance conditions, and success and failure actions. The plan body is a tree of states and branches, in which each branch is either a query goal, an achieve goal or an external action.

Now, in order to specify the invocation condition, we define triggers, which are simply those events that cause 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.

$$\begin{aligned} Trigger ::= & \; addbel\langle\langle Belief \rangle\rangle \\ | & \; removebel\langle\langle Belief \rangle\rangle \\ | & \; toldevent\langle\langle Atom \rangle\rangle \\ | & \; goalevent\langle\langle Goal \rangle\rangle \end{aligned}$$

As 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 $[State]$. Thus we define a branch (or arc) within a plan body as being labelled with either an internal or external action, or a subgoal.

$$\begin{aligned} Branch ::= & \; ext\langle\langle ExtAction \rangle\rangle \\ | & \; int\langle\langle IntAction \rangle\rangle \\ | & \; goal\langle\langle Goal \rangle\rangle \end{aligned}$$

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.

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

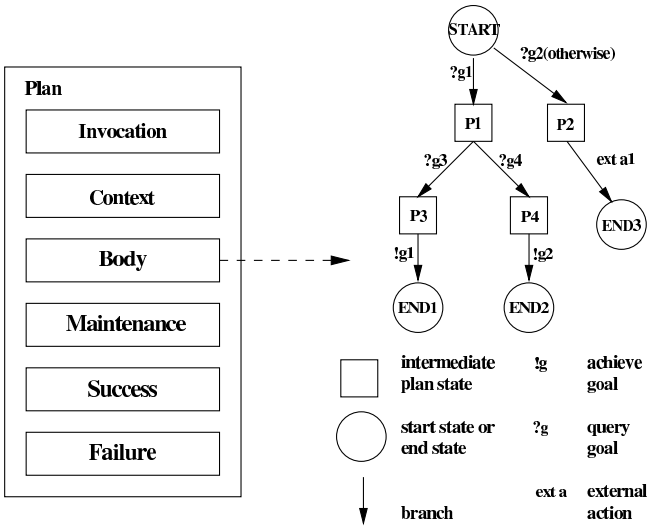


Figure 4. The dMARS plan structure.

All of these components can now be brought together into the definition of a plan as follows: (The definition of the *optional* type used here and other related components, which are non-standard Z, can be found in Appendix C.)

$$
\begin{array}{|l}
\textit{Plan} \\
\hline
inv : Trigger \\
context : optional[SitForm] \\
body : Body \\
maint : SitForm \\
succ, fail : \text{seq } IntAction \\
\hline
\end{array}
$$

If a plan does not have a body, it is called a *primitive plan.*

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

Now, returning to our example, consider a plan which we will call *PlanP*, illustrated in Figure 5, which is triggered whenever an agent perceives a full bin. For future reference we will call this *PlanP*. The plan body, which is not included in this diagram, but is shown in Figure 6, specifies the course of action to take for the agent to empty the bin.

In the example scenario, the invocation condition holds (is made true) by binding the variable *W* to the full *binB*. The context of the plan states that the plan can only be adopted if the agent *believes* there to be a truck which is currently not in use; if such a truck can be found then variable *X* is bound to it. There are no maintenance conditions or fail actions but if the plan succeeds then the agent updates

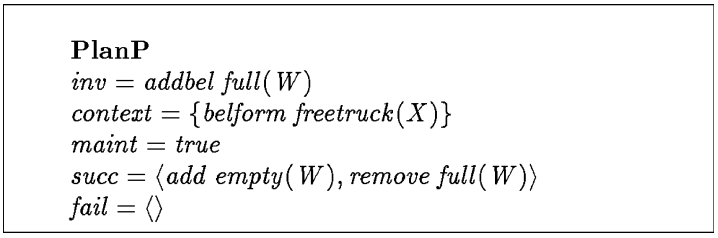


Figure 5. A dMARS plan for rubbish collection (PlanP).

its belief base to record the fact that the bin is now empty. Part of a possible body for such a plan to empty the bin is illustrated in Figure 6. First, it includes steps to find the location of the bin, which is bound to variable Y, and the location of the truck, which is bound to variable Z. (Remember that variable W is now *bound* to *binB* in order to establish the truth of the invocation condition), and X is bound to *truckT*.

The next stage of the plan is to test whether the bin and the truck are currently at the same location. If not, the right branch is traversed and an achieve goal to get them to the same location is encountered. At this point, a subgoal is posted to the event queue and, when processed, requires another plan (say *PlanQ*) to achieve it. Once *PlanQ* has succeeded, the execution of *PlanP* can continue. (In fact, this subgoaling may continue with *PlanQ* requiring further plans, but we defer discussion of this until later in the paper.)

At this point (on the left branch), the agent can attempt to perform the external action of loading the bin onto the truck before trying to achieve the goal of getting the truck to the *dump* which may, in turn, require further planning. Lastly, the bin is emptied and the plan succeeds. (Note that although the plan might seem incomplete because the bin has not been returned to its original location, this was not part of the aim of the plan.)

It can be seen from the example above that when a plan is adopted by an agent for execution more information besides just the plan (i.e., trigger, context, body, maintenance conditions and internal actions) needs to be represented. We also need to include, amongst other things, any bindings that have been used during the execution of the plan, the current state reached in the plan, and the branches that are available to an agent in attempting to move to a new state from the current state. This extra information reflects the important distinction between plans as recipes for action (which are contained in the plan library) on the one hand, and plans the agent has adopted and set about executing, (which are components of the *mental state* of the agent) on the other. We distinguish plans as recipes from plans as components of mental state directing action, by referring to the latter as *plan instances*.

To provide a full explanation of plan instances, we require definitions of how bindings, substitutions and unification takes place in dMARS. The reader unfamiliar with such issues is therefore referred to Appendix A, which considers these aspects at length. Similarly, it is also necessary to define functions on the tree structures of the plan body, details of which can be found in Appendix B. These important aspects are

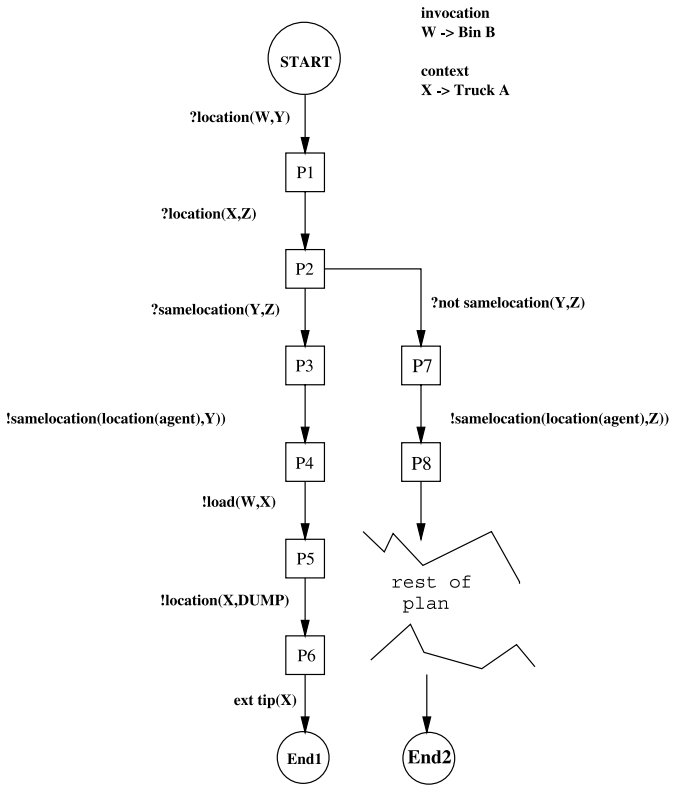


Figure 6. A dMARS plan body for PlanP.

not discussed here in order to avoid impeding the flow of the exposition, but both are required to specify how dMARS plans are instantiated as intentions, and we consider this next.

6. Plan instances

The basic execution mechanism for dMARS agents, described in Section 2, involves an agent matching the invocation condition and context of each plan against the chosen event in the event queue and the current set of beliefs, respectively. It then generates a set of candidate, matching plans, selects one, and makes a *plan instance* for it. This plan instance represents the status of the plan through its course of execution, and 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 that can be traversed from this state;

- (if determined) the branch it is attempting to traverse;
- an identifier to uniquely identify the plan instance to the agent owner from the set [*PlanInstId*] of all such identifiers;
- and finally, the *status* of the plan (either "active", indicating that the plan is currently executing, or "inactive", indicating that the plan has temporarily been suspended).

When a branch cannot be traversed (for example, 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 the branch to attempt next, but if it is undefined, no such choice has been made.

A plan instance is thus formally specified in the following schema. The predicates ensure that the plan instance is well defined, and in turn assert the following:

- that the current state is one of the plan's states;
- that if the current state is a leaf node then the set of possible next branches available is empty;
- that the set of possible branches currently available from the current state is a subset of all the branches leading from the current state as defined in the original plan (from which the plan instance is derived);
- and that the selected branch is always one of the possible branches.

The schema uses the auxiliary function, *NextBranches*, to identify the set of possible next branches from a given state in a plan, and *AllStates*, to give all the states of a plan. Full definitions of these functions can be found in Appendix B.

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

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{PlanInst}} \\
\hline
plan : Plan \\
env : Sub \\
state : State \\
choices : \mathbb{P}\, Branch \\
branch : optional[Branch] \\
status : Status \\
id : PlanInstId \\
\hline
state \in AllStates\ plan \\
state \in \mathrm{dom}\, End \Rightarrow choices = \{\} \\
choices \subseteq NextBranches\ plan\ state \\
branch \subseteq choices \\
\hline
\end{array}$$

For example, suppose that *PlanP*, introduced earlier, has been selected as a plan instance, and instantiated with the variable *W* bound to *binB* in order to make the plan invocation condition true, and the variable *X* bound to *truckT* in order to make

$$
\begin{aligned}
&plan = PlanP\\
&env = \{(X, binA), (W, truckT), (Y, box4), (Z, box5)\}\\
&state = P2\\
&choices = \{\{?samelocation(Y, Z)\}, \{?not\ samelocation(Y, Z)\}\}\\
&branch = \{\}\\
&status = active
\end{aligned}
$$

Figure 7. The plan instance for *PlanP*.

the context true. Further suppose that the agent has executed this plan so that it has just established that the location of the bin is *box*4 and that the location of the truck is *box*5. The variables Y and Z are therefore bound to the constants *box*4 and *box*5, respectively. The current state reached by the agent is therefore $P2$ of Figure 6. Now, if the agent has not committed to either of the possible branches, then the plan instance at this point would be represented as described in Figure 7.

At this point, the agent needs to determine whether *box*4 and *box*5 are the same. Clearly they are not and the agent would follow the course as determined by the right hand branch. The next branch then states that the agent must now get to the truck by making its location the same as that of the variable Z which is bound to the location of the truck. At this point this subgoal is posted as an internal event (which will be discussed and formally introduced later in the paper), and the status of the plan is set to suspended.

6.1. *Subtypes of plan instance*

Now, when a plan is first selected as a plan instance, the current state is set to the first state in the plan, and its status is set to *active*. 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). We can therefore define three important types of plan instances as follows:

$$
\begin{aligned}
&Initial == \{p : PlanInst \mid p.state = Start\ p.plan \wedge p.status = active\}\\
&Succeeded == \{p : PlanInst \mid p.state \in (\mathrm{dom}\ End)\}\\
&Failed == \{p : PlanInst \mid p.state \notin (\mathrm{dom}\ End) \wedge p.choices = \{\}\}
\end{aligned}
$$

7. Intentions and events

7.1. *Intentions*

Intentions have been discussed in the literature at great length over the years, from Bratman's work on the BDI model of practical reasoning [3] through to Cohen and

Levesque's [7] notion of intention being defined in terms of the commitment an agent has to make when it is faced with a number of alternative, desirable courses of action. In these views, intention is seen as the key to resource-bounded intelligent action since, in general, an agent cannot hope to achieve all of its goals and must decide which goals to fix on. The goals to which an agent is committed become its intentions, directing (in general) its perception, action and reasoning.

There are many formal models that explore desirable properties of agent intentions including, for example, that of never intending to achieve something that is already true or that can never be true [54]. Despite all this, an intention in dMARS, and in other BDI-inspired architectures, is operationalised as 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. It is the only plan that may have an active status.

To illustrate, we continue with our example, and assume that *PlanP* is instantiated as a plan instance in response to the new external event of discovering a full bin. However, during the course of executing the plan, subgoals that require further planning must be achieved. For example, *PlanQ* might be selected to enable the agent to get to the same location as the truck. Once *PlanQ* is accomplished successfully, the agent can resume execution of *PlanP*. However, *PlanQ* may itself require a further plan, *PlanR*, to achieve a subgoal within *PlanQ*. In this case, the resulting intention (call it *IntentionI*) would be written as follows:

$$IntentionI = \langle PlanR, PlanQ, PlanP \rangle$$

The general structure of intentions is shown in Figure 8 where the plan instance generated initially in response to the external event is the plan instance (*PlanInstance*(1)) at the bottom of the stack. Whilst the status of the plan instances from 1 to $m-1$ must be *suspended*, the plan instance at the top of the stack may be active (if it is currently executing) *or* suspended (if it has posted a subgoal that has not yet been processed) as will be discussed later. Formally, an intention is simply a sequence of plan instances.

$$Intention == \mathrm{seq}_1\ PlanInst$$
$$\forall\, i : Intention;\ p : PlanInst \bullet$$
$$(p \in (\mathrm{ran}(tail(i)))) \Rightarrow p.status = suspended$$

7.2. *Events*

Events are perceived and subsequently processed by agents in order to establish which plans should be selected as intentions. An event essentially comprises the triggering event itself (that activates plans) and, optionally, a plan instance identifier

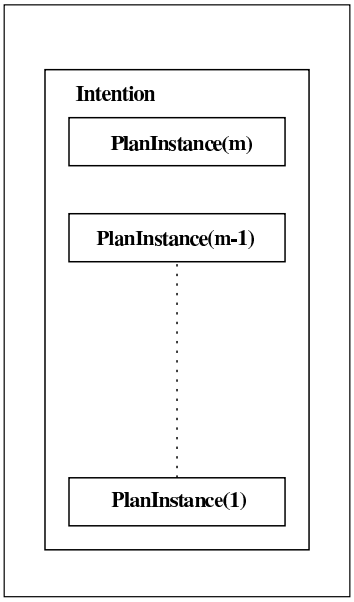


Figure 8. The dMARS intention structure.

that identifies the event-generating plan, an environment, and a set of plan instances that may already have failed and may not be retried. (Note that attempting a new plan is not the same as backtracking. Whereas backtracking restores system state, reposting a goal after plan failure does not – it takes off from where the last plan failed.) Recall that a plan fails when all branches leading from a non-end state (an intermediate or start state) have been attempted for traversal, but have failed and have therefore been removed. In this case an end state of the plan cannot be reached and so it cannot succeed.

Event

$$trig : Trigger$$
$$id : optional[PlanInstId]$$
$$env : optional[Sub]$$
$$failures : \mathbb{P}\, PlanInst$$

The simplest type of event is an external event such as adding a new belief in response to perception. Such events are not associated with an existing plan instance when they first enter the buffer (though of course they will become associated later, once a plan instance is generated for them).

ExternalEvent

$$Event$$

$$trig \notin (\text{ran } goalevent)$$
$$env = \{\}$$
$$failures = \{\}$$

By contrast, a subgoal event is an internal event that occurs when the branch of an executing plan at the top of an intention is an achieve goal that cannot be achieved immediately. In this case, the environment variable of the event is defined, containing a subset of the bindings from the environment of the executing plan that posted the goal. More specifically, the environment of the event is restricted to bindings only for those variables that are contained in the trigger of this subgoal event. This is because it is only these bindings that are relevant when the event is, in turn, processed to generate new applicable and relevant plans.

Any plan instances that fail to achieve the subgoal are added to the *failures* variable to make sure they are never retried.

(In order to proceed with the full specification, we must first define auxiliary functions to return the set of variables contained in a goal and a situation formula.)

$$
\begin{array}{l}
sitvars : SitForm \rightarrow (\mathbb{P}\ Var) \\
goalvars : Goal \rightarrow (\mathbb{P}\ Var) \\
\hline
\forall\, b_1, b_2 : BelForm;\ s_1, s_2 : SitForm \bullet \\
\quad sitvars\ true = \{\} \land \\
\quad sitvars\ false = \{\} \land \\
\quad sitvars\ (belform\ b_1) = belformulavars\ b_1 \land \\
\quad sitvars\ (or(s_1, s_2)) = sitvars\ s_1 \cup sitvars\ s_2 \land \\
\quad sitvars\ (and(s_1, s_2)) = sitvars\ s_1 \cup sitvars\ s_2 \land \\
\quad goalvars\ (achieve\ b_1) = belformulavars\ b_1 \land \\
\quad goalvars\ (query\ s_1) = sitvars\ s_1
\end{array}
$$

In the following schema the final predicate states that the variables contained within the environment are all contained in the trigger event. This requires the inverse ($^\sim$) of the function constructor (*goalevent*) that creates the trigger type from the goal type (i.e., $goalevent^\sim$) to be applied to the trigger. The function *goalvars* can then be applied to the resulting goal to obtain the set of variables contained within it.

$$
\begin{array}{l}
\underline{\quad GoalEvent \quad} \\
Event \\
\hline
trig \in (\mathrm{ran}\ goalevent) \\
defined\ env \\
\mathrm{dom}(the\ env) \subseteq goalvars\ (goalevent^\sim trig) \\
\hline
\end{array}
$$

8. 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, respectively, 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* [42].

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 either causes the generation of a new intention, or adds a plan instance 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.

8.1. The dMARS agent state

As in other BDI architectures such as AgentSpeak(L) [42], 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. The belief domain specifies the set of belief formulae representing all possible beliefs of the agent. Similarly, the goal domain is a set of temporal formulae that includes all the belief formulae in the belief domain, prefixed with query and achieve, as well as other temporal formulae consisting of predicate symbols not contained in the belief domain. Every goal of a plan must be contained in the goal domain. The designer of an agent may also specify the basic capabilities of an agent in terms of the external actions it can perform. In this case any external action contained in a plan of the plan library must be contained within these capabilities.

$$
\begin{array}{|l}
\textit{dMARSAgent} \\
\hline
\textit{library} : \mathbb{P}\, \textit{Plan} \\
\textit{intentionselect} : \mathbb{P}_1\, \textit{Intention} \rightarrow \textit{Intention} \\
\textit{planselect} : \mathbb{P}_1\, \textit{Plan} \rightarrow \textit{Plan} \\
\textit{eventselect} : \mathrm{seq}_1\, \textit{Event} \rightarrow \textit{Event} \\
\textit{substitutionselect} : \mathbb{P}_1\, \textit{Sub} \rightarrow \textit{Sub} \\
\textit{selectbranch} : \textit{PlanInst} \nrightarrow \textit{Branch} \\
\textit{beliefdomain} : \mathbb{P}\, \textit{BelForm} \\
\textit{goaldomain} : \mathbb{P}\, \textit{Goal} \\
\textit{expertise} : \mathbb{P}\, \textit{ExtAction} \\
\hline
\forall\, p : \textit{library};\ a : \textit{ExtAction} \bullet \\
\quad a \in (\textit{mapset}(\textit{ext}^{\sim})(\textit{AllBranches}\ p)) \Rightarrow a \in \textit{expertise} \\
\forall\, b : \textit{beliefdomain} \bullet \\
\quad \{\textit{query}\ (\textit{belform}\ b), \textit{achieve}\ b\} \subseteq \textit{goaldomain} \\
\forall\, p : \textit{library};\ g : \textit{Goal} \bullet \\
\quad g \in (\textit{mapset}(\textit{goal}^{\sim})(\textit{AllBranches}\ p)) \Rightarrow g \in \textit{goaldomain} \\
\hline
\end{array}
$$

Typically the event-selection function takes the head of the event queue which means that the dMARS agent will process events on a first-come first-served basis.

$$\textit{eventselect} = \textit{head}$$

In specifying the state of the agent, we indicate which aspects may change over time. These components are the agent's beliefs (which are ground belief formulae), intentions (sequences of plan instances), and events yet to be processed (represented as a sequence). Clearly, any belief of the agent must be attainable by applying some substitution to a belief formula within its belief domain.

$$
\begin{array}{|l}
\textit{dMARSAgentState} \\
\hline
\textit{dMARSAgent} \\
\textit{beliefs} : \mathbb{P}\, \textit{Belief} \\
\textit{intentions} : \mathbb{P}\, \textit{Intention} \\
\textit{events} : \mathrm{seq}\, \textit{Event} \\
\hline
\forall\, \textit{bel} : \textit{beliefs} \bullet \\
\quad (\exists\, s : \textit{Sub};\ \textit{bf} : \textit{beliefdomain} \bullet \\
\qquad \textit{ASBelForm}\ s\ \textit{bf} = \textit{bel}) \\
\forall\, \textit{pi} : \textit{PlanInst} \bullet \\
\quad (\textit{pi} \in (\bigcup \{i : \textit{intentions} \bullet (\mathrm{ran}\, i)\}) \Rightarrow \\
\qquad (\textit{pi}.\textit{plan} \in \textit{library})) \\
\hline
\end{array}
$$

An operation only affects the state of the dMARS agent rather than the agent itself. The Ξ symbol states that the state variables contained within the *dMARSAgentState* schema do not change.

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

Initially, an agent is provided with an event queue, and sets of beliefs and intentions that "pump prime" its subsequent intention generation and action. This initial state is specified as follows:

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

Finally, we must specify that agents can perceive external events that are placed at the end of the event buffer.

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

8.2. 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 the way an agent ascertains those plans in its plan library that are considered as ways of reacting to the new event. (A description of the mgu can be found in Appendix A.)

Relevant plans are specified in the function *relplans*, 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 mgu for p. The signatures of the functions defining mgus are given in Appendix B. If the event is a subgoal event, and therefore contains a substitution environment, it must be applied to the invocation conditions before the relevant plans are generated.

$$relplans : Event \rightarrow \mathbb{P}\, Plan \rightarrow \mathbb{P}(Plan \times Sub)$$

$$\begin{aligned}
&\forall\, e : Event;\ lib : \mathbb{P}\, Plan \bullet \\
&\quad undefined\ e.env \Rightarrow relplans\ e\ lib = \\
&\quad \{p : lib;\ \sigma : Sub \mid mguevents\ (e.trig, p.inv) = \sigma \bullet (p, \sigma)\} \wedge \\
&\quad defined\ e.env \Rightarrow relplans\ e\ lib = \{p : lib;\ \sigma : Sub \mid \\
&\quad mguevents\ (ASTrigger\ (the\ e.env)\ e.trig, p.inv) = \sigma \bullet (p, \sigma)\}
\end{aligned}$$

Let us return to our example of *PlanP* and consider how it might be selected as a relevant plan. Suppose an external event, provided as input to the agent, states that *binB* is full. (Remember that an external event contains no information other than the trigger, which is typically the addition or removal of a belief.) Now there may be many plans whose invocation can be unified with the predicate *full*(*binB*) in the same way as *PlanP*. The function *relplans* below returns all such plans together with the minimal substitution which matches the event trigger and the invocation. In the case of *PlanP* that substitution is simply $\{(W, binB)\}$. There are, of course, other substitutions that could be applied such as $\{(W, binB), (X, truckU)\}$, but the first is *more general* since it is a subset of the second. Indeed we can see that it is a mgu. (Consult Appendix A for a more general description of unification.)

Now, a relevant plan is applicable if its context is a *logical consequence* of the beliefs of the agent. That is the agent will only attempt to execute a relevant plan if its context currently holds with respect to its beliefs.

Thus, we can define a predicate, *dMarsLogCons*, to hold between a situation formula and the belief base of an agent if the situation formula is a logical consequence of the belief base. (Note that dMARS is resource-bounded and does not perform full-fledged logical inference. There are many potential ways of implementing different forms of inference for this purpose, some of which are more or less resource-intensive than others. In the simplest case, this might amount simply to seeing whether the situation formula is contained in a belief database.)

$$dMarsLogCons\_ : \mathbb{P}(SitForm \times \mathbb{P}\, BelForm)$$

Using this logical consequence relation, we can 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, *applplans*, 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. These updated substitutions contain the bindings required to make that plan both relevant and applicable. (The definition of substitution composition can be found in Appendix A.)

$$
\begin{array}{|l}
applplans : \mathbb{P}(Plan \times Sub) \rightarrow (\mathbb{P}\, BelForm) \rightarrow \mathbb{P}(Plan \times Sub) \\
\hline
\forall\, relsubs : \mathbb{P}(Plan \times Sub);\ bels : \mathbb{P}\, BelForm \bullet \\
\quad applplans\ relsubs\ bels = \{rel : Plan;\ \sigma, \psi : Sub \mid \\
\quad\quad (rel, \sigma) \in relsubs \wedge \\
\quad\quad dMarsLogCons(ASSit\ (\sigma \ddagger \psi)\ (the\ rel.context), bels) \\
\quad\quad\quad \bullet\ (rel, \sigma \ddagger \psi)\}
\end{array}
$$

Returning to our example, the function *applplans* takes a set of (relevant) plan-substitution pairs $\{\ldots, (PlanP, \{(W, binB)\}), \ldots\}$ and, for each plan and substitution, attempts to find another substitution which, when composed with the existing substitution, can be applied to the context of the plan to make it a logical consequence of the beliefs of the agent.

Let us now assume that the agent has correct beliefs about the world as represented in Figure 3. The beliefs will therefore include the predicate *belform free*$(truckT)$. Composing the applicable substitution $\{(W, binB)$ with the substitution $\{(X, truckT)\}$ results in the following substitution.

$$\{(W, binB), (X, truckT)\}$$

If this is applied to the context of *PlanP* (*belform free*(X)), the result is *belform free*$(TruckT)$, which is a logical consequence of the agents' beliefs. (In fact the belief is actually contained in the agents belief base.) Clearly no other substitution is going to contain a set of bindings which is a subset that can make the plan both applicable and relevant. (This is what is meant by the mgu, which is defined in Appendix A.) However, there may well be other plans which are both relevant and applicable and the agent according to a particular environment, must then choose between them. It can do this non-deterministically or by using some form of meta-reasoning [45].

8.3. 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 then relevant plans and, in turn, applicable plans are

determined. An applicable plan is selected and used to generate a plan instance using the function *CreatePlanInstance* defined here. It simply takes a plan and a substitution, and creates a plan instance in its initial state.

$$CreatePlanInst : Plan \rightarrow Sub \rightarrow PlanInst$$

$$\forall\, p : Plan;\ s : Sub \bullet CreatePlanInst\ p\ s = (\mu\, inst : Initial \mid inst.plan = p \land inst.env = s)$$

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. As the status of any newly created plan instance is active, the status of the intention is updated automatically. Note that a failed plan instance cannot be re-selected for an internal event. Also, if the event is external then it must be updated to include the identifier of the new plan instance to which it is associated.

In the schema below, note that as *plans* is defined as the set of all applicable plans and associated unifiers, applying the function to the selected plan returns its unifier. In the final predicate, the triggering intention is removed from the set of intentions, the new plan instance is pushed onto it and replaced into the set of agent intentions.

NewPlanInst

$$\Delta dMARSAgentState$$

$$events \neq \langle\rangle$$
$$\mathsf{Let}\ event == eventselect\ events \bullet$$
$$\mathsf{Let}\ plans == applplans\ (relplans\ event\ library)\ beliefs \bullet$$
$$\mathsf{Let}\ selectedplan == planselect\ (\mathrm{dom}\ plans) \bullet$$
$$\mathsf{Let}\ unifier == plans\ selectedplan \bullet$$
$$\mathsf{Let}\ instance == CreatePlanInst\ selectedplan\ unifier \bullet$$
$$event \in ExternalEvent \Rightarrow (\mathsf{Let}\ new == MakeEvent(event.trig, \{instance.id\}, \{\}, \{\}) \bullet intentions' = intentions \cup \{\langle instance \rangle\} \land events' = (events \rhd \{event\}) \cup \{(events^{\sim} event, new)\}) \land$$
$$event \in GoalEvent \Rightarrow instance \notin (event.failures) \land (\mathsf{Let}\ trigint == (\mu\, i : intentions \mid (head\ i).id = (the\ event.id)) \bullet intentions' = intentions \setminus \{trigint\} \cup \{\langle instance \rangle \frown trigint\})$$

The schema above uses the following definition for *MakeEvent* which simple creates an element of type *Event* from its constituent parts.

$$
\begin{array}{|l}
MakeEvent : (Trigger \times optional[PlanInstId] \times optional[Sub] \\
\qquad \times \mathbb{P}\, PlanInst) \rightarrow Event \\
\hline
\forall\, t : Trigger;\ f : \mathbb{P}\, PlanInst;\ s : optional[Sub]; \\
\qquad id : optional[PlanInstId] \bullet \\
\quad MakeEvent\,(t, id, s, f) = \\
\qquad (\mu\, e : Event \mid \\
\qquad\quad e.trig = t \wedge e.env = s \wedge e.failures = f \wedge e.id = id)
\end{array}
$$

8.4. *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*. Essentially, at any time, the agent has a set of intentions (each of which is a sequence of plan instances) that it is currently executing. It must first select an intention to execute (either non-deterministically or using some form of meta-level reasoning) with the qualification that the uppermost plan instance of this intention has an *active* status. The agent then locates the current state in the plan body of the plan instance, and determines the set of available branches that leave that state. From these branches, one is selected, which is either an external action (that is performed), a query goal (that is unified with the agent's beliefs) or an achieve goal (that causes an internal (subgoal) event to be posted and the status of the plan to be set to suspended).

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.

$$
\begin{array}{|l}
\hline
AgentIntExecutionOperationState \\
\hline
dMARSAgentState \\
intention : optional[Intention] \\
plan : optional[PlanInst] \\
branch : optional[Branch] \\
\hline
\end{array}
$$

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

$$
\begin{array}{|l}
\hline
SelectIntention \\
\hline
\Delta AgentIntExecutionOperationState \\
\hline
events = \langle\rangle \\
the\ intention' = intentionselect\ intentions \\
the\ plan' = head(the\ intention') \\
(the\ plan').status = active \\
(the\ branch') = selectbranch\ (the\ plan') \\
\hline
\end{array}
$$

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. First, however, we introduce the auxiliary function, *AchieveBranch*, which takes a plan instance and moves it on to the next state as determined by the *branch* variable of the plan instance.

$$
\begin{array}{|l}
AchieveBranch : PlanInst \pfun PlanInst \\
\hline
\forall p : PlanInst \bullet defined\ p.branch \Rightarrow AchieveBranch\ p = \\
\quad (\mu new : PlanInst \mid \\
\quad new.plan = p.plan \land \\
\quad new.state = NextState\ p.plan\ p.state\ (the\ p.branch) \land \\
\quad new.status = p.status)
\end{array}
$$

$$
\begin{array}{|l}
\hline
BranchSucceed \\
\Delta AgentIntExecutionOperationState \\
\hline
the\ plan' = AchieveBranch\ (the\ plan) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
BranchFail \\
\Delta AgentIntExecutionOperationState \\
\hline
(the\ plan').choices = (the\ plan).choices \setminus branch \\
\hline
\end{array}
$$

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

8.4.1. 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.

$$
\begin{array}{|l}
executeaction : PlanInst \pfun Sub
\end{array}
$$

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

$$
\begin{array}{|l}
\hline
BranchExtActionSucceed \\
\Delta AgentIntExecutionOperationState \\
\hline
the\ branch \in \mathrm{ran}\ ext \\
the\ plan \in \mathrm{dom}\ executeaction \\
(the\ plan').env = (the\ plan).env \ddagger executeaction\ (the\ plan) \\
\hline
\end{array}
$$

The branch is then traversed to reach the next state, specified by the schema *BranchSucceed* defined 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:

$$BranchExtActionSucceed \fatsemi BranchSuceed$$

An unsuccessful branch fails and there is no state change.

BranchExtActionFail

$$\Xi AgentIntExecutionOperationState$$

$$the\ branch \in \text{ran}\ ext$$
$$the\ plan \notin (\text{dom}\ executeaction)$$

After this occurs the branch must be removed.

$$BranchExtActionFail \fatsemi BranchFail$$

8.4.2. 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.

$$performintaction : (\mathbb{P}\ Belief) \rightarrow IntAction \rightarrow (\mathbb{P}\ Belief)$$

$$\forall b : Belief;\ i : IntAction;\ bs : \mathbb{P}\ Belief \bullet$$
$$i = add\ b \Rightarrow performintaction\ bs\ i = bs \cup \{b\} \wedge$$
$$i = remove\ b \Rightarrow performintaction\ bs\ i = bs \setminus \{b\}$$

BranchIntAction

$$\Delta AgentIntExecutionOperationState$$

$$(the\ branch) \in (\text{ran}\ int)$$
$$\text{Let}\ action == (int^{\sim}(the\ branch)) \bullet$$
$$beliefs' = performintaction\ beliefs\ action \wedge$$
$$action \in (\text{ran}\ add) \wedge beliefs' \neq beliefs \Rightarrow$$
$$events' = events \frown$$
$$\langle MakeEvent(addbel\ (add^{\sim}action), \{\}, \{\}, \{\})\rangle \wedge$$
$$action \in (\text{ran}\ remove) \wedge beliefs' \neq beliefs \Rightarrow$$
$$events' = events \frown$$
$$\langle MakeEvent(removebel\ (remove^{\sim}action), \{\}, \{\}, \{\})\rangle$$

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.

8.4.3. 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 mgus 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 situation formula contained in the querygoal follows from the set of beliefs, according to the substitution which is maximal.

BranchQueryGoalSucc

$\Delta AgentIntExecutionOperationState$

$the\ branch \in \text{ran}\ goal$
$goal^{\sim}(the\ branch) \in \text{ran}\ query$
Let $qgoal == (goal^{\sim}(the\ branch));\ env == (the\ plan).env \bullet$
$\quad \exists s : Sub \bullet unifiquery\ (s, (ASGoal\ env\ qgoal, beliefs)) \wedge$
$\quad$ (Let $sub == mguquery\ (ASGoal\ env\ qgoal, beliefs) \bullet$
$\quad\quad (the\ plan').env = env \ddagger sub$)

Where no such unification is possible, the branch fails.

BranchQueryGoalFail

$\Delta AgentIntExecutionOperationState$

$the\ branch \in \text{ran}\ goal$
$goal^{\sim}(the\ branch) \in \text{ran}\ query$
Let $qgoal == (goal^{\sim}(the\ branch));$
$\quad env == (the\ plan).env \bullet$
$\neg\ (\exists s : Sub \bullet unifiquery\ (s, (ASGoal\ env\ qgoal, beliefs)))$

8.4.4. Achieve goals. Finally, with an achieve goal, g, 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. The identifier of the new internal event is set to the current executing plan.

BranchAchieveGoal

$\Delta AgentIntExecutionOperationState$

$(the\ branch) \in \text{ran}\ goal$
$goal^{\sim}(the\ branch) \in \text{ran}\ achieve$
$(the\ plan').status = suspended$
Let $g == goal^{\sim}(the\ branch);\ env == (the\ plan).env \bullet$
$events' = events^{\frown}$
$\langle MakeEvent\ ((goalevent\ g), \{(the\ plan).id\}, \{env\}, \{\})\rangle$

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

8.5. *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, $(the\ plan).env$, is applied to the success conditions, $(the\ plan).plan.succ$, to give a sequence of ground internal actions, *groundacts*. 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 C.

AchievePlan

$\Delta AgentIntExecutionOperationState$

$the\ plan \in Succeeded$
Let $succacts == (the\ plan).plan.succ;$
$env == (the\ plan).env \bullet$
  Let $groundacts == map\ (ASIntAction\ env)\ succacts \bullet$
  $beliefs' = fold\ performintaction\ beliefs\ groundacts$

Two further cases arise if a plan succeeds. If there are more plans in the intention, the current substitution environment, $(the\ plan).env$, is updated to include the appropriate bindings from both the achieved plan, *plan*, and the environment of the next plan in the stack, *next.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. (The term $(the\ intention)\ 2$ returns the second plan instance within the intention.) *TEVars*, returns the set of variables of a trigger event. Also the internal event which generated the completed plan is removed.

AchievePlanOnly

AchievePlan

$\#(the\ intention) > 1$
Let $next == (the\ intention)\ 2 \bullet$
 Let $newenv == ((TEVars\ (the\ plan).plan.inv) \lhd next.env) \ddagger$
  $((the\ plan).env \oplus next.env) \bullet$
  $the\ intention = tail\ (the\ intention) \land$
  $(the\ plan').env = newenv \land$
  $(the\ plan').status = active$
  $\text{ran}\ events' = \text{ran}\ events \setminus$
$\{(\mu\, e : GoalEvent \mid e \in \text{ran}\ events \land the\ e.id = (the\ plan).id)\}$

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

AchievePlanAndIntention

$$
\begin{array}{|l}
\hline
\mathit{AchievePlan} \\
\hline
\#(\mathit{the\ intention}) = 1 \\
\mathit{intentions}' = \mathit{intentions} \setminus \mathit{intention} \\
\text{ran}\ \mathit{events}' = \text{ran}\ \mathit{events} \setminus \{(\mu\, e : \mathit{ExternalEvent} \mid \\
\qquad e \in \text{ran}\ \mathit{events} \wedge \mathit{the}\ e.\mathit{id} = (\mathit{the\ plan}).\mathit{id})\} \\
\hline
\end{array}
$$

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, *groundacts'*, 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, g, 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.

FailPlan

$$
\begin{array}{|l}
\hline
\Delta \mathit{AgentIntExecutionOperationState} \\
\hline
\mathit{the\ plan} \in \mathit{Failed} \\
\mathsf{Let}\ g == (\mu\, e : \mathit{Event} \mid \mathit{the}\ e.\mathit{id} = (\mathit{the\ plan}).\mathit{id}); \\
\quad \mathit{env} == (\mathit{the\ plan}).\mathit{env} \bullet \\
\quad \mathsf{Let}\ \mathit{failacts} == (\mathit{the\ plan}).\mathit{plan.fail} \bullet \\
\quad \mathsf{Let}\ \mathit{groundacts} == \mathit{map}\ (\mathit{ASIntAction}\ \mathit{env})\ \mathit{failacts} \bullet \\
\quad \mathit{beliefs}' = \mathit{fold\ performintaction\ beliefs\ groundacts} \wedge \\
\qquad \text{ran}\ \mathit{events}' = (\text{ran}\ \mathit{events} \setminus \{g\}) \cup \\
\qquad \{\mathit{MakeEvent}(g.\mathit{trig}, g.\mathit{id}, g.\mathit{env}, (g.\mathit{failures} \cup \mathit{plan}))\} \\
\qquad \mathit{the\ intention}' = \mathit{tail}(\mathit{the\ intention}) \\
\hline
\end{array}
$$

9. Evaluation and comparison

9.1. *Variations on dMARS*

In the preceding sections, we have described the basic dMARS architecture and semantics. However, many aspects of this basic framework may be adjusted without fundamentally affecting the architecture itself. In particular, the formalisation of dMARS given here allows us to identify those aspects that may be easily modified, and facilitates the modification.

For example beliefs could equally well be represented as database tables. This does not affect the general architecture or reactive planning capability and the

specification would change very little. In this case it would simply be a matter of redefining the belief data structure and, then redefining related elements such as logical consequence.

Goals could be equally well (or better) be represented as any temporal formula. Thus, a goal could be any temporal formula in any temporal language, and the invocation condition on plans could equally be any temporal formula. If the latter logically implies the former, then the plan is applicable. This is a very clean extension of the dMARS invocation mechanism. Again, it would be a simple matter to change the specification to accommodate this with almost no change to the rest of the architecture specification.

We can demonstrate how other data structures can be accommodated within the same basic architectural framework more specifically by outlining another very real benefit of this specification; it allows a very easy comparison to be made with other languages that have been similarly specified [38]. The BDI model as described here, and as instantiated in dMARS, is the inspiration for a host of related models and systems and we specifically look at comparisons with AgentSpeak(L) [13] and 3APL [12].

The abstract programming language, 3*APL* [27, 28], with a well-defined formal semantics in terms of a transition system, has been based on the BDI model. 3*APL* 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 3*APL* 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.

By comparing systems using the abstract specification provides for a more uniform and unifying perspective. We show how this can be achieved readily with these languages by briefly comparing definitions and schemas from the three languages.

We begin by considering data structures. AgentSpeak(L), 3APL and dMARS both have their atom, action, term and belief primitives defined in almost exactly the same way. The only difference is that 3APL allows for a slightly richer representation of belief allowing, for example, implication.

$$
\begin{aligned}
3APLBelief ::= \; & pos\langle\langle Atom\rangle\rangle \\
| \; & not\langle\langle Atom\rangle\rangle \\
| \; & and\langle\langle 3APLBelief \times 3APLBelief\rangle\rangle \\
| \; & imply\langle\langle 3APLBelief \times 3APLBelief\rangle\rangle
\end{aligned}
$$

dMARS and AgentSpeak(L) both use triggering events in order for subgoals of plans to be placed in a queue for further planning, and uses *intentions* as sequences of plans to be executed. In 3APL, events are unnecessary since goals are themselves modified in the process of planning and acting, and instead of using intentions, 3APL simply attempts to execute its current goals.

The definition of a 3APL Goal is therefore slightly more sophisticated allowing, for example, sequential composition and choice. An AgentSpeak(L) goal on the other hand is simply either a query or an achieve atom.

$$ASGoal ::= achieve\langle\langle Atom\rangle\rangle \mid query\langle\langle Atom\rangle\rangle$$

$$\begin{aligned} 3APLGoal ::= {} & action\langle\langle Action\rangle\rangle \\ \mid {} & query\langle\langle Belief\rangle\rangle \\ \mid {} & achieve\langle\langle Atom\rangle\rangle \\ \mid {} & seqcomp\langle\langle Goal \times Goal\rangle\rangle \\ \mid {} & choice\langle\langle Goal \times Goal\rangle\rangle \\ \mid {} & goalvar\langle\langle GoalVariable\rangle\rangle \end{aligned}$$

AgentSpeak(L) plans are much more simple than dMARS plans and simply consist if a Trigger event, a pre-condition which is a set of beliefs and a body which is a sequence of goals and actions.

$$ASFormula ::= action\langle\langle Action\rangle\rangle \mid goal\langle\langle ASGoal\rangle\rangle$$

$$\begin{aligned} ASInvocation &== TriggerEvent \\ ASContext &== \mathbb{P}\, Belief \\ ASFormula &::= actionformula\langle\langle Action\rangle\rangle \mid goalformula\langle\langle Goal\rangle\rangle \\ ASBody &== \mathrm{seq}\, ASFormula \end{aligned}$$

ASPlan

$inv : ASInvocation$
$context : ASContext$
$body : ASBody$

A 3APL agent uses plans called *practical reasoning rules* not only to plan in the more conventional sense, but also to *reflect* on its goals. Reflection allows an agent to re-consider one of its plans in a situation in which the plan will fail with respect to the goal it is trying to achieve or has already failed, or where a more optimal strategy can be pursued. In 3APL practical reasoning rules are divided into four classes: rules, which are used not only to respond to the current situation but also to create new goals; plan-rules, which are used to find plans for achievement goals; failure-rules, which are used to replan when plans fail; and optimisation-rules, which can replace less effective plans with more optimal plans. A practical reasoning rule consists of an (optional) head, which is a goal, an (optional) body which is a goal, a guard which is a belief and a type to define its purpose.

$$3APLPRType ::= reactive \mid failure \mid plan \mid optimisation$$

$$
\begin{array}{|l}
\hline
3APLPRrule \\
head, body : optional[3APLGoal] \\
guard : 3APLBelief \\
type : PRType \\
\hline
head = \varnothing \Leftrightarrow type = reactive \land \\
thehead \in (\text{ran } achieve) \land body \neq \varnothing \Leftrightarrow type = plan \\
\hline
\end{array}
$$

The agents themselves are defined by their expertise and rulebase (planbase) in 3APL, dMARS and AgentSpeak(L). However, the state of agents at run-time differ. The state of AgentSpeak(L) and dMARS agents includes beliefs, executing intentions, events to be processed, and actions to be performed, while the state of 3APL agents simply contains beliefs and goals. At the agent state level we therefore need fewer state variables to specify 3APL agents. We do not provide details of the formalisation where as it should be reasonably clear to the reader how these would all compare.

It is also a straightforward matter to make comparison between the operation of different systems. In AgentSpeak(L) and dMARS, there are two parts to the operation of agents, processing events and executing intentions. Processing an event involves selecting a plan triggered by the event and adding it as an intention to an intention stack. Executing intentions involves selecting an intention, locating its topmost plan, and performing the plan's next component. In 3APL, the agent either applies its rules to its current goals, which involves manipulating it's goals to which the rules can apply, or executes them which amounts to executing either a basic action or query goal at the front of a goal.

10. Conclusions

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 [15]. 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., [16]).

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 dMARS architecture in different languages and

environments is therefore a realistic possibility. This kind of transition is demonstrated, for example, through the provision of a simple agent simulation environment [37], and more recently through the development of a sophisticated Jini-based development environment [1], both based on an extensive agent framework [14] represented in Z. While both of these implementations address the specific issues involved in instantiating a broad encompassing framework, the work in this paper focuses on the specification of a mature, implemented and deployed system, in the reverse direction.

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 [55], and also for Rao's AgentSpeak(L) [42], 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 [44]. In future work, we hope to investigate such axiomatisations.

Equally importantly, the provision of a formal specification of a system such as dMARS (for which there does not exist even a complete *informal* specification), using a standard language, allows an easy and simple comparison of it with alternative architectures and systems specified in the same way. For example, related work on reformalising AgentSpeak(L) [13] and 3*APL* [12] enables exactly such comparisons to be made to identify points of agreement and difference [29]. In addition, the accumulation of these efforts results in an accessible resource in the specification of techniques for development of agent systems. Not only might this not otherwise be available, but it is unlikely to be so in a form relevant both to agent architects and developers. In other words, we have provided what amount to architectural building blocks that can be used in the development of agent languages and architectures in general, to combat *wheel re-invention*.

Some features of various BDI systems still remain to be included in specifications such as this. For example, some BDI plan representation languages support parallelism, *wait* goals, continuous processes, programmatic variables, etc. The original PRS supported internal actions that went beyond assertions or retractions from the agent database as does dMARS. These are not considered in the model provided here, but are clearly avenues for further development of this work.

The BDI model as manifested by PRS and dMARS remains central to agent-based systems, just as it has done over the course of the last 15 years or so. Indeed, the claim still holds. Only very recently, Georgeff argued that BDI embodies the key requirements for systems to function effectively in the uncertain and dynamic environments of the emerging information age [20]. If this is so, then a clear and precise specification and understanding of the manifestation of this model and its relation to systems development is required. Through the description and specification of dMARS contained in this paper, we have provided exactly that.

Appendix A. Substitutions and unification

In this appendix, we define binding and unification as used in dMARS, which is critical to providing a comprehensive model of the system's operation. *Binding* refers to the way a variable may be associated with a term. For example, the variable X may be associated with the constant 4, and the variable Y with the term $f(U, V)$. A *substitution* is a set of such bindings from variables to terms. For example, the substitution containing the two bindings above would be $\{(X,4),(Y,f(U,V))\}$. Formally, a substitution is a partial function between variables and terms. In general, substitutions are partial functions, since only some variables will be mapped to terms.

$$Sub == Var\ Term$$

However, any term in the range of the substitution (defined by the auxiliary function *subrangevars* below) cannot contain any of the variables in the domain of the relation.

$$\begin{array}{|l} subrangevars : Sub \to (\mathbb{P}\ Var) \\ \hline \forall\, s : Sub \bullet subrangevars\ s = \bigcup(mapset\ termvars\ (\mathrm{ran}\ s)) \end{array}$$

Therefore, the intersection of the set of variables in the domain with the set of variables contained in the terms of the range must be empty.

$$\forall\, s : Sub \bullet \mathrm{dom}\ s \cap subrangevars\ s = \{\}$$

A.1. Substitution application

Next, definitions are provided to specify the application of substitutions to various dMARS expressions. The function *ASVar* 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.

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

Similarly, we can then define what it means for a substitution to be applied to a term. If it is a constant, it is unchanged; a variable requires that the function defined above is applied; and a function requires recursive application of the substitution to each of the terms within it.

$$
\begin{array}{|l}
ASTerm : Sub \rightarrow Term \rightarrow Term \\
\hline
\forall\, t : Term;\ f : FunSym;\ ts : \mathrm{seq}\, Term;\ \psi : Sub \mid t = fun\ (f, ts) \bullet \\
\quad t \in \mathrm{ran}\, const \Rightarrow ASTerm\ \psi\ t = t \wedge \\
\quad t \in \mathrm{ran}\, var \Rightarrow ASTerm\ \psi\ t = ASVar\ \psi\ (var^{\sim} t) \wedge \\
\quad t \in \mathrm{ran}\, fun \Rightarrow ASTerm\ \psi\ t = \\
\qquad (\mu\, new : Term \mid first\ (fun^{\sim} new) = f \wedge \\
\qquad\quad second\ (fun^{\sim} new) = map\ (ASTerm\ \psi)\ ts\ a)
\end{array}
$$

In exactly the way we can define the application of a substitution to an internal action, a situation formula, a plan, a goal, a belief formula and a trigger event, as given by *ASAtom*, *ASIntAction*, *ASSit*, *ASGoal*, *ASBelForm* and *ASTrigger*, respectively.

$$
\begin{array}{|l}
ASAtom : Sub \rightarrow Atom \rightarrow Atom \\
ASIntAction : Sub \rightarrow IntAction \rightarrow IntAction \\
ASSit : Sub \rightarrow SitForm \rightarrow SitForm \\
ASGoal : Sub \rightarrow Goal \rightarrow Goal \\
ASBelForm : Sub \rightarrow BelForm \rightarrow BelForm \\
ASTrigger : Sub \rightarrow Trigger \rightarrow Trigger \\
\hline
\forall\, a, b : Atom;\ s : Sub \bullet \\
\quad ASAtom\ s\ a = b \Leftrightarrow b.head = a.head \wedge \\
\quad b.terms = map(ASTerm\ s) a.terms \wedge \\
\quad (ASBelForm\ s\ (pos\ a)) = pos(ASAtom\ s\ a) \wedge \\
\quad (ASBelForm\ s\ (nota)) = not(ASAtom\ s\ a) \\
\forall\, f, g : SitForm;\ l, m : BelForm;\ a : Atom;\ s : Sub \bullet \\
\quad ASSit\ s\ (belform\ l) = belform\ (ASBelForm\ s\ l) \wedge \\
\quad ASSit\ s\ (and(f, g)) = and\ ((ASSit\ s\ f), (ASSit\ s\ g)) \wedge \\
\quad ASSit\ s\ (or(f, g)) = or\ ((ASSit\ s\ f), (ASSit\ s\ g))
\end{array}
$$

A.2. Substitution composition

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 $\tau \ddagger \sigma = \{u/A, v/F(A, B, C), x/A, y/B, z/C\}$.

The auxiliary function *allvars* returns the set of *all* variables contained in a substitution

$$
\begin{array}{|l}
allvars : Sub \rightarrow (\mathbb{P}\, Var) \\
\hline
\forall\, s : Sub \bullet allvars\ s = \mathrm{dom}\, s \cup subrangevars\ s
\end{array}
$$

$$\_ \ddagger \_ : Sub \times Sub \to Sub$$

$$\forall \tau, \sigma : Sub \mid (allvars \tau) \cap (\mathrm{dom}\, \sigma) = \{\} \bullet$$
$$\tau \ddagger \sigma = (\tau \cup \{x : Var;\ t : Term \mid (x, t) \in \sigma \bullet (x, ASTerm\ \tau\ t)\}$$

A.3. Unification

Unification is concerned with how substitutions can be applied to two expressions in order that they can be equated in some way. Clearly, there may be many substitutions that make two expressions equal and it is therefore important to select the substitution that contains the fewest number of bindings. This substitution is called the *most general unifier* (*mgu*).

A substitution is a *unifier* for two expressions if the substitution, applied to both of them, makes them equal.them equal.

$$unifiterms\ \_ : \mathbb{P}(Sub \times (Term \times Term))$$

$$\forall t_1, t_2 : Term;\ s : Sub \bullet$$
$$unifiterms\ (s, (t_1, t_2)) \Leftrightarrow (ASTerm\ s\ t_1 = ASTerm\ s\ t_2)$$

The definition is analogous for unifying trigger events.

$$unifitriggers\ \_ : \mathbb{P}(Sub \times (Trigger \times Trigger))$$

$$\forall e, f : Trigger;\ s : Sub \bullet$$
$$unifitriggers\ (s, (e, f)) \Leftrightarrow (ASTrigger\ s\ e = ASTrigger\ s\ f)$$

A substitution is *more general* than another substitution if there exists a third substitution which, when composed with the first, gives the second. (For example, the unifier $\{x/A, y/B, z/C\}$ is more general than the unifier $\{u/A, v/F(A, B, C), x/A, y/B, z/C\}$; the more general a unifier the lesser number of bindings it contains.)

$$\_\ mg\ \_ : \mathbb{P}(Sub \times Sub)$$

$$\forall \psi, \sigma : Sub \bullet \sigma\ mg\ \psi \Leftrightarrow (\exists \omega : Sub \bullet (\sigma \ddagger \omega) = \psi)$$

The *mgu* of two expressions is a substitution which unifies the expressions such that there is no other unifier that is more general. This is a partial function, since any two expressions may not have a unifier at all. However, if there does exist an *mgu* then it is unique.

$$mguterms : (Term \times Term) \nrightarrow Sub$$

$$\forall t_1, t_2 : Term;\ \sigma : Sub \bullet mguterms\ (t_1, t_2) = \sigma \Leftrightarrow unifiterms\ (\sigma, (t_1, t_2)) \wedge \neg\ (\exists \omega : Sub \bullet (unifiterms\ (\omega, (t_1, t_2)) \wedge (\omega mg\ \sigma)))$$

Here we define the signatures for the most general unifier of two trigger events.

$$mguevents : (Trigger \times Trigger) \nrightarrow Sub$$

$$\forall t_1, t_2 : Trigger;\ \sigma : Sub \bullet mguevents\ (t_1, t_2) = \sigma \Leftrightarrow unifitriggers\ (\sigma, (t_1, t_2)) \wedge \neg\ (\exists \omega : Sub \bullet (unifitriggers\ (\omega, (t_1, t_2)) \wedge (\omega mg\ \sigma)))$$

Appendix B. Auxiliary functions for dMARS plans

Five auxiliary definitions are required in order to specify the operation of a dMARS agent. These are functions that are applied to the tree structure that comprises the plan's body.

1. *Start* determines the start state of a plan.
2. *AllStates* gives all the states of a plan.
3. *AllBranches* specifies all the branches of a plan.
4. *NextBranches* identifies the set of possible next branches from a given state in a plan.
5. *NextState* gives the next state in a plan when applied to the current state and the branch traversed.

Recall that the basic representation of a tree a pair. The first element of the pair is a state and the second is a set of pairs. Each of these pairs contains a branch and a further tree.

$$(state, \{(branch_1, tree_1), (branch_2, tree_2) \ldots (branch_n, tree_n)\}$$

Therefore, for a non-primitive plan, the first of the pair represents the start node. The start state of a primitive plan is thus simply the single state that comprises its body.

$$Start : Plan \rightarrow State$$

$$\forall p : Plan \bullet p \in PrimitivePlan \Rightarrow Start\ p = End^{\sim} p.body \wedge p \notin PrimitivePlan \Rightarrow Start\ p = first(Fork^{\sim} p.body)$$

The set of all the states within the body of a plan is defined by recursing through the tree structure. In the base case, the leaf node is returned. The *AllStates* function is specified in terms of a further auxiliary function, *AllBodyStates*, defined on the *body* of a plan.

$$
\begin{array}{|l}
AllStates : Plan \rightarrow (\mathbb{P}\, State) \\
AllBodyStates : Body \rightarrow \mathbb{P}\, State \\
\hline
\forall\, b : Body \bullet \\
\quad b \in (\mathrm{ran}\, End) \Rightarrow AllBodyStates\ b = \{End^{\sim} b\} \wedge \\
\quad b \notin (\mathrm{ran}\, End) \Rightarrow AllBodyStates\ body = \\
\quad \{first(Fork^{\sim} b)\} \cup \\
\quad \bigcup(mapset\ AllBodyStates\ (mapset\ second\ (second(Fork^{\sim} b)))) \\
\forall\, p : Plan \bullet AllStates\ p = AllBodyStates\ (p.body)
\end{array}
$$

The branches of a tree can be defined using an analogous function although here, the application of the function in the base case returns the empty set (since an end state clearly has no branches).

$$
\begin{array}{|l}
AllBranches : Plan \rightarrow (\mathbb{P}\, Branch) \\
AllBodyBranches : Body \rightarrow \mathbb{P}\, Branch \\
\hline
\forall\, b : Body \bullet \\
\quad b \in (\mathrm{ran}\, End) \Rightarrow AllBodyBranches\ b = \{\} \wedge \\
\quad b \notin (\mathrm{ran}\, End) \Rightarrow AllBodyBranches\ b = \\
\quad (\mathrm{dom}(second(Fork^{\sim} b))) \cup \\
\quad \bigcup(mapset\ AllBodyBranches\ (\mathrm{ran}(second(Fork^{\sim} b)))) \\
\forall\, p : Plan \bullet AllBranches\ p = AllBodyBranches\ (p.body)
\end{array}
$$

Suppose there is some state in the body of a plan $s : State$, and some non-empty set of branches from this state to other trees, $next : \mathbb{P}_1(Branch \times Body)$ so that $Fork(s, next)$ is a subtree of the plan's entire body. The branches leading from state s are given simply by the domain of *next*. Assuming the existence of the subtree relation (_ *subtree* _) we define the function *NextBranches* as follows:

$$
\begin{array}{|l}
\_\ subtree\ \_ : \mathbb{P}(Body \times Body)
\end{array}
$$

$$
\begin{array}{|l}
NextBranches : Plan \nrightarrow State \nrightarrow (\mathbb{P}\, Branch) \\
\hline
\forall\, p : Plan;\ s : State;\ next : \mathbb{P}(Branch \times Body) \mid \\
\quad s \in AllStates\ p \wedge Fork(s, next)\ subtree\ p.body \bullet \\
\quad\quad NextBranches\ p\ s = \mathrm{dom}\ next
\end{array}
$$

Similarly, it is possible to define a function that takes some branch from a node (*path*) and returns the next node (which may be a leaf or fork node). It is achieved by

taking the set of pairs of branches and trees (*next*) and applying it to the chosen branch (*path*).

$$
\begin{array}{|l}
NextState : Plan \pfun State \pfun Branch \pfun State \\
\hline
\forall\, p : Plan;\ s : State;\ path : Branch;\ next : \mathbb{P}_1(Branch \times Body) \mid \\
\quad s \in (AllStates\ p) \land (s \notin (\mathrm{dom}\ End)) \land \\
\quad Fork(s, next)\ subtree\ p.body \land path \in (\mathrm{dom}\ next) \bullet \\
\qquad next(path) \in (\mathrm{ran}\ End) \Rightarrow \\
\qquad\quad NextState\ p\ s\ path = End^{\sim}(next(path)) \land \\
\qquad (next(path)) \in (\mathrm{ran}\ Fork) \Rightarrow \\
\qquad\quad NextState\ p\ s\ path = first\ (Fork^{\sim}(next(path)))
\end{array}
$$

Appendix C. Auxiliary generic Z definitions

In this appendix, we define generic auxiliary functions used in the main body of the paper.

The function, *fold*, takes a function, an initial value and a sequence, and first applies the function to the first element in the sequence and the initial value. Then it applies the function to the second element in the sequence and the result of the first function application and so on. For example, $fold\ f\ x \langle a, b \rangle = f(f\ x\ a)b$. The double line in the schema definition states that we are declaring *generic* functions, which means that, in the case of the *fold* function say, we can replace X and Y with any types we chose.

$$
\begin{array}{|l}
[X, Y] \\
\hline
fold : (X \to Y \to X) \to X \to (\mathrm{seq}\ Y) \to X \\
\hline
\forall\, f : (X \to Y \to X);\ x : X;\ y : Y;\ ys : \mathrm{seq}\ Y \bullet \\
\qquad fold\ f\ x\ \langle\rangle = x \land \\
\qquad fold\ f\ x\ (\langle y \rangle \frown ys) = fold\ f\ (f\ x\ y)\ ys
\end{array}
$$

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.

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

In specifying dMARS, 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*, which consists of the empty set and singleton sets containing elements of *T*. The predicates, *defined* and *undefined* test whether an element of *optional*[*T*] is defined (i.e., contains an element of type *T*) or not (i.e., is the empty set), and the function, *the*, extracts the element from a defined member of *optional*[*T*].

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

$$\begin{array}{|l} [X] \\ \hline defined\_, undefined\_ : \mathbb{P}(optional[X]) \\ the : optional[X] \pfun X \\ \hline \forall xs : optional[X] \bullet \\ \quad defined\ xs \Leftrightarrow \# xs = 1 \land \\ \quad undefined\ xs \Leftrightarrow \# xs = 0 \\ \forall xs : optional[X] \mid defined\ xs \bullet \\ \quad the\ xs = (\mu x : X \mid x \in xs) \\ \hline \end{array}$$

Acknowledgments

Many thanks to 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 [48]. Thanks also to Sorabain de Lioncourt who identified an error in the original specification.

References

1. R. Ashri and M. Luck, "Paradigma: Agent implementation through Jini," in A. M. Tjoa, R. R. Wagner, and A. Al-Zobaidie, (eds.), *Eleventh International Workshop on Databases and Expert System Application*, IEEE Computer Society, 2000, pp. 453–457.
2. J. P. Bowen, *Formal Specification and Documentation using Z: A Case Study Approach*, International Thomson Computer Press, 1996.
3. M. E. Bratman, D. J. Israel, and M. E. Pollack, "Plans and resource-bounded practical reasoning," *Computational Intelligence*, vol. 4, pp. 349–355, 1988.
4. B. Burmeister and K. Sundermeyer, "Cooperative problem solving guided by intentions and perception," in E. Werner and Y. Demazeau, (eds.), *Decentralized AI 3 – Proc. Third European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-91)*, Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, pp. 77–92, 1992.
5. P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas, "JACK intelligent agents – components for intelligent agents in Java," *AgentLink News*, 1999.

6. B. Chellas, *Modal Logic: An Introduction*, Cambridge University Press: Cambridge, England, 1980.
7. P. R. Cohen and H. J. Levesque, "Intention is choice with commitment," *Artif. Intell.*, vol. 42, pp. 213–261, 1990.
8. B. P. Collins, J. E. Nicholls, and I. H. Sørensen, "Introducing formal methods: The CICS experience with Z," in B. Neumann, et al., (eds.), *Mathematical Structures for Software Engineering*, Oxford University Press, 1991.
9. I. D. Craig, *The Formal Specification of Advanced AI Architectures*, Ellis Horwood: Chichester, 1991.
10. D. Craigen, S. L. Gerhart, and T. J. Ralston, "An international survey of industrial applications of formal methods," Technical Report NIST GCR 93/626-V1 & 2, Atomic Energy Control Board of Canada, US National Institute of Standards and Technology and US Naval Research Laboratories, 1993.
11. M. d'Inverno, M. Fisher, A. Lomuscio, M. Luck, M. de Rijke, M. Ryan, and M. Wooldridge, "Formalisms for multi-agent systems," *Knowl. Eng. Rev.*, vol. 12, no. 3, pp. 315–321, 1997.
12. M. d'Inverno, K. Hindriks, and M. Luck, A formal architecture for the 3APL agent programming language, in *First International Conference of B and Z Users*, Springer Verlag 1878, 2000, pp. 168–187.
13. M. d'Inverno and M. Luck, "Engineering AgentSpeak(L): A formal computational model," *Logic Comput.*, vol. 8, no. 3, pp. 233–260, 1998.
14. M. d'Inverno and M. Luck, *Understanding Agent Systems*, Springer, 2001.
15. M. d'Inverno and M. Luck, Practical and theoretical innovations in multi-agent systems research, *Knowl. Eng. Rev.*, vol. 17, no. 3, pp. 295–301, 2003.
16. M. d'Inverno, M. Priestley, and M. Luck, A formal framework for hypertext systems, *IEE Proc. – Software Eng. J.*, vol. 144, no. 3, pp. 175–184, 1997.
17. E. A. Emerson and J. Y. Halpern, "Sometimes and 'not never' revisited: On branching time versus linear time temporal logic," *J. ACM*, vol. 33, no. 1, pp. 151–178, 1986.
18. M. P. Georgeff, "Planning," *Ann. Rev. Comput. Sci.*, vol. 2, pp. 359–400, 1987.
19. M. P. Georgeff and A. L. Lansky, "Reactive reasoning and planning," in *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, Seattle, WA, 1987, pp. 677–682.
20. M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge, "The belief-desire-intention model of agency," in *Intelligent Agents V, LNAI 1555*, Springer, 1999, pp. 1–10.
21. M. P. Georgeff and A. S. Rao, "A profile of the Australian AI Institute," *IEEE Expert*, vol. 11, no. 6, pp. 89–92, 1996.
22. R. Goodwin, "A formal specification of agent properties," *J. Logic Comput.*, vol. 5, no. 6, 1995.
23. A. Haddadi, "Belief-desire-intention agent architectures," in G. M. P. O'Hare and N. R. Jennings, (eds.), *Foundations of Distributed Artificial Intelligence*, Wiley, 1996, pp. 169–185.
24. I. J. Hayes, "VDM and Z: A comparative case study," *Form. Aspect Comput.*, vol. 4, no. 1, pp. 76–99, 1996.
25. I. J. Hayes, (ed.), *Specification Case Studies*, (2nd edn.). Prentice Hall: Hemel Hempstead, 1993.
26. M. A. Hewitt, C. M. O'Halloran, and C. T. Sennet, "Experiences with PiZA, an animator for Z," in J. P. Bowen, M. G. Hinchey, and D. Till, (eds.), in *ZUM'97: 10th International Conference of Z Users, Lecture Notes in Computer Science*, Springer-Verlag: Heidelberg, 1997, pp. 37–51.
27. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer, "Formal Semantics for an Abstract Agent Programming Language," in *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories. Architectures and Languages, LNAI 1365*, Springer, 1998, pp. 215–229.
28. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer, "Control structures of rule-based agent languages," in *Intelligent Agents V. LNAI 1555*, Springer, 1999, pp. 381–396.
29. K. Hindriks, M. d'Inverno, and M. Luck, "Architecture for agent programming languages," in *Proceedings of the 14th European Conference on Artificial Intelligence*, 2000, pp. 363–367.
30. C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666–677, 1978.
31. M. J. Huber, "JAM: A BDI-theoretic mobile agent architecture," in *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, Seattle, WA, 1999, pp. 236–243.
32. F. Ingrand, M. Georgeff, and A. Rao, "An architecture for real-time reasoning and system control," *IEEE Expert*, vol. 7, no. 6, pp. 34–44, 1992.
33. N. R. Jennings, "Specification and implementation of a belief desire joint-intention architecture for collaborative problem solving," *J. Intell. Coop. Inform. Sys.*, vol. 2, no. 3, pp. 289–318, 1993.

34. C. B. Jones, *Systematic Software Development using VDM*, (2nd edn.). Prentice Hall, 1990.
35. K. Lano, "*The B Language and Method: A Guide to Practical Formal Development*," Springer-Verlag, 1996.
36. J. Lee, M. Huber, E. Durfee, and P. Kenny, "UM-PRS: An implementation of the procedural reasoning system for multirobot applications," in *CIRFSS94, Conference on Intelligent Robotics in Field, Factory, Service and Space*, MIT Press, 1994, pp. 842–849.
37. M. Luck and M. d'Inverno, From agent theory to agent construction: A case study, in *Intelligent Agents III. ATAL'96*, Springer-Verlag, 1997, pp. 215–230.
38. M. Luck and M. d'Inverno, "Unifying agent systems", *Ann. Math. Artif. Intell.*, vol. 37, no. 1–2, pp. 131–167, 2002.
39. R. Milner, *Communication and Concurrency*, Prentice Hall, 1989.
40. B. G. Milnes, "A specification of the Soar architecture in Z," Technical Report CMU-CS-92-169, School of Computer Science, Carnegie Mellon University, 1992.
41. K. L. Myers, "A procedural knowledge approach to task-level control," in B. Drabble, (ed.), *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, AAAI Press, 1996, pp. 158–165.
42. A. S. Rao, "AgentSpeak(L): BDI agents speak out in a logical computable language." in W. Van de Velde and J. W. Perram, (eds.), *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (LNAI Volume 1038)*, Springer-Verlag: Heidelberg, Germany, 1996, pp. 42–55.
43. 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)*, San Francisco, CA, June 1995, pp. 312–319.
44. A. S. Rao and M. P. Georgeff, "Modeling rational agents within a BDI-architecture," in R. Fikes and E. Sandewall, (eds.), *Proceedings of Knowledge Representation and Reasoning (KR&R-91)*, Morgan Kaufmann Publishers: San Mateo, CA, April 1991, pp. 473–484.
45. A. S. Rao and M. P. Georgeff, "An abstract architecture for rational agents," in C. Rich, W. Swartout, and B. Nebel, (eds.), *Proceedings of Knowledge Representation and Reasoning (KR&R-92)*, 1992, pp. 439–449.
46. M. Saaltink, "The Z/EVES system." in J. P. Bowen, M. G. Hinchey, and D. Till, (eds.), *ZUM'97: 10th International Conference of Z Users. Lecture Notes in Computer Science*, Springer-Verlag: Heidelberg, 1997, pp. 72–85.
47. J. M. Spivey, *Understanding Z: A Specification Language and its Formal Semantics*, Cambridge University Press: Cambridge, 1988.
48. J. M. Spivey, *The Fuzz Manual*, (2nd edn.). Computing Science Consultancy, 2 Willow Close, Garsington, Oxford OX9 9AN, UK, 1992.
49. M. Spivey, *The Z Notation*, (2nd edn.). Prentice Hall International: Hemel Hempstead, England, 1992.
50. M. Weber, "Combining Statecharts and Z for the design of safety-critical control systems," in M.-C. Gaudel and J. C. P. Woodcock, (eds.), *FME'96: Industrial Benefit and Advances in Formal Methods*, vol. 1051 of *Lecture Notes in Computer Science*, Formal Methods Europe, Springer-Verlag, 1996, pp. 307–326.
51. C. D. Wezeman, "Using Z for network modelling: An industrial experience report," *Comput. Stand. Inter.*, vol. 17, no. 5–6, pp. 631–638, 1995.
52. K. R. Wood, "A practical approach to software engineering using Z and the refinement calculus," *ACM Software Eng. Notes*, vol. 18, no. 5, pp. 79–88, 1995.
53. J. Woodcock and J. Davies, *Using Z: Specificiation. Refinement and Proof*, Prentice Hall: Hemel Hempstead, 1996.
54. Michael Woodlridge, *Reasoning about Rational Agents*, MIT Press, 2000.
55. M. Wooldridge, "This is MyWorld: the logic of an agent-oriented testbed for DAI," in M. Wooldridge and N. R. Jennings, (eds.), *Intelligent Agents: Theories, Architectures and Languages (LNAI Volume 890)*, Springer-Verlag: Heidelberg, Germany, 1995, pp. 160–178.
56. M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice," *Knowl. Eng. Rev.*, vol. 10, no. 2, pp. 115–152, 1995.