Engineering AgentSpeak(L): A Formal Computational Model

MARK D'INVERNO, *Cavendish School of Computer Science, University of Westminster, London, W1M 8JS.*
E-mail: dinverm@wmin.ac.uk

MICHAEL LUCK, *Department of Computer Science, University of Warwick, Coventry, CV4 7AL.*
E-mail: mikeluck@dcs.warwick.ac.uk

Abstract

Perhaps the most successful agent architectures, and certainly the best known, are those based on the Belief-Desire-Intention (BDI) framework. Despite the wealth of research that has accumulated on both formal and practical aspects of this framework, however, there remains a gap between the formal models and the implemented systems. In this paper, we build on earlier work by Rao aimed at narrowing this gap, by developing a strongly-typed, formal, yet computational model of the BDI-based AgentSpeak(L) language. AgentSpeak(L) is a programming language, based on the Procedural Reasoning System (PRS) and the Distributed Multi-Agent Reasoning System (dMARS), which determines the behaviour of the agents it implements. In developing the model, we add to Rao's work, identify some omissions, and progress beyond the description of a particular language by giving a formal specification of a general BDI architecture that can be used as the basis for providing further formal specifications of more sophisticated systems.

Keywords: Agent languages, intelligent agents, BDI architectures, Z, formal specification.

1 Introduction

Agent-based systems are computational systems comprising one or more agents, each of which is an independent entity that perceives and acts in an environment in order to achieve its goals. Typically, agents are autonomous in that they are not reliant on outside intervention, reactive in that they respond in a timely fashion to changes in their environment, and proactive in attempting to achieve their goals through the use of plans, for example. Other characteristics often associated with them include social ability, persistence, mobility, learning, etc., but there is no general consensus.

While many different and contrasting agent architectures have been proposed, perhaps the most successful are those based on the Belief-Desire-Intention (BDI) framework. In the BDI view, agents continually monitor their environments and act to change them, based on the three mental attitudes of *belief*, *desire* and *intention*, representing informational, motivational and decision-making capabilities. It is certainly true to say that, over a number of years, a wealth of research has accumulated on both the formal and theoretical aspects of BDI agents through the use and development of various logics, for example, on the one hand, and on the practical aspects through the development of implementations of BDI agents on the other.

As with many areas of agent-oriented systems, however, there is a gap between these formal models and implemented systems. For example, implementations have typically involved simplifying assumptions that have resulted in the loss of a strong theoretical foundation for them, while logics have had small relation to practical problems [22]. Though this fragmentation into theoretical and practical aspects has been noted, and several efforts made in attempting to address this fragmentation in related areas of agent-oriented systems (for example, [8, 14, 20, 29]), there remains much to be done in bringing together the two strands of work.

Some progress has been made with BDI agents, in particular, and Rao has attempted to unite theory and practice in two ways. First, he provided an abstract agent architecture that serves as an idealization of an implemented system and as a means for investigating theoretical properties [23]. A second effort also started with an implemented system and formalized its operational semantics in an agent language. This resulted in AgentSpeak(L), which can be viewed as an abstraction of the implemented Procedural Reasoning System (PRS) [13] and the Distributed Multi-Agent Reasoning System(dMARS) [7], and which allows agent programs to be written and interpreted [22].

In this paper, we build on this latter work by developing a strongly-typed, formal, yet computational model of the AgentSpeak(L) language.

Using the Z specification language [27], the paradigm formal method of software engineering, we provide a specification that can be checked for type-correctness [26, 24], can be animated (as, for example, has been done for d'Inverno and Hu's hypertext specification [5] using the techniques described by Hewitt et al. [16]) to provide a prototype system, and can be formally and systematically refined to produce a provably correct implementation [28]. In developing the specification, we add to the original work of Rao, and progress beyond the description of a particular language, by giving a formal specification of a general BDI architecture that can be used as the basis for providing such formal specifications of more sophisticated systems. The spirit of this is thus similar to that of Garlan's reusable formal frameworks in which an underlying abstract specification enables refinement to many possible instantiations [11]. Indeed, the notion of framework specifications that can be instantiated and refined to specify systems of a similar class has already been adopted in several areas [6, 9, 19]. In line with this, the specification in this paper has been checked for type-correctness, but has not been animated nor refined to an implementation (though these are obvious next steps in development).

The organisation of the paper reflects the organisation of the architecture as follows. The first two sections set the stage by providing an overview of the AgentSpeak(L) system and then a brief introduction to the Z specification language. Section 4 introduces and defines the types and primitives necessary for the specification of the system, including beliefs, goals, plans and intentions. In Section 5, we then proceed to the specification of an AgentSpeak(L) agent and its state, and in Section 6 we give details of the way in which agent plans are instantiated, and address the actual operation of AgentSpeak(L) agents and their action-reasoning cycle. The paper ends with a discussion of the contribution of this work in continuing Rao's original efforts to unify practical and theoretical aspects, and in its use as a base for the development of formal specifications of implemented BDI systems, such as dMARS.

2 The AgentSpeak(L) System

Rao's AgentSpeak(L) is an attempt to provide an operational semantics of a language that can be viewed as an abstraction of implemented BDI systems such as PRS and dMARS. It is a language that is based on PRS, and with an essentially similar operational semantics. According to Rao, those language constructs in PRS that are not included in AgentSpeak(L) simply make the task of programming more efficient.

The basic operation of agents in AgentSpeak(L) is based around their beliefs, desires and intentions. An agent has beliefs (about itself, others and the environment), desires (in terms of the states it wants to achieve in response) and intentions as adopted plans. In addition, agents also maintain a repository of available plans, known as the *plan library*. Agents respond to changes in their goals and beliefs, which result from perception, and which are packaged into data structures called *events*, representing either new beliefs or new goals. They respond to these changes by selecting plans from the plan repository for each change and then instantiating one of these plans as an intention. These intentions comprise actions and goals or plans to be achieved, with the latter possibly giving rise to the addition of new plans to that intention.

The operation of an AgentSpeak(L) agent can be summarised as follows.

- If there are more events to process, select one.
- Retrieve from the plan library all the plans *triggered* by this event that can be executed in the current circumstances.
- Select for execution one of the plans thus generated, and generate an *instance* of that plan, known as the *intended means*.
- Add the intended means to the appropriate intention. This will be a new intention in the case of an external event arising through changes to beliefs, and an existing intention in the case of an internal event resulting from attempting to satisfy a goal in that intention.
- Select an intention and consider the next step in its top plan. If this is an action, perform it, and if it is a goal, add a corresponding event to the set of events yet to be processed.
- If the top plan is complete, consider the next plan, and if the intention is empty, the intention has succeeded and can be removed from the set of intentions.

This description captures the essential operation of AgentSpeak(L) agents. More detail will be added to the description, however, when the operation cycle is elaborated further with the formal specification that follows. Before proceeding to that specification, we first provide some details of the notation we use.

3 Notation

3.1 The Z Specification Language

In the current work, we have adopted the Z specification language [27] for three major reasons. First, it is sufficiently expressive to allow a consistent, unified and structured account of a computer system and its associated operations. Second, we view our enterprise as that of building programs. Z schemas are particularly suitable in squaring the demands of formal modelling with the need for implementation by

allowing transition between specification and program. Thus our approach to formal specification is pragmatic — we need to be formal to be precise about the concepts we discuss, yet we want to remain directly connected to issues of implementation. Z provides just those qualities that are needed, and is increasingly being used for specifying frameworks and systems in AI (for example, [2, 14, 21]) and related areas (for example, [4, 6]). Third, Z is the paradigm formal method of software engineering with wide usage in both industry and academia, and supported by a large collection of tools.

3.2 Syntax

The formal specification language, Z, is based on set theory and first order predicate calculus. It extends the use of these languages by allowing an additional mathematical type known as the *schema type*. Z schemas have two parts: the upper declarative part, which declares variables and their types, and the lower predicate part, which relates and constrains those variables. The type of any schema can be considered as the Cartesian product of the types of each of its variables, without any notion of order, but constrained by the schema's predicates. Modularity is facilitated in Z by allowing schemas to be included within other schemas. We can select a state variable, var, of a schema, $schema$, by writing $schema.var$.

To introduce a type in Z, where we wish to abstract away from the actual content of elements of the type, we use the notion of a *given set*. We write $[NODE]$ to represent the set of all nodes. If we wish to state that a variable takes on some set of values or an ordered pair of values we write $x : \mathbb{P}\, NODE$ and $x : NODE \times NODE$, respectively.

A *relation* type expresses some relationship between two existing types, known as the *source* and *target* types. The type of a relation with source X and target Y is $\mathbb{P}(X \times Y)$. A relation is therefore a set of ordered pairs.

When no element from the source type can be related to two or more elements from the target type, the relation is a *function*. A *total* function ($\rightarrow$) is one for which every element in the source set is related, while a *partial* function ($\nrightarrow$) is one for which not every element in the source is related. A sequence (seq) is a special type of function where the domain is the contiguous set of numbers from 1 up to the number of elements in the sequence. For example, the first relation below defines a *function* between nodes, while the second defines a *sequence* of nodes.

$$Rel1 = \{(node1, node2), (node2, node3), (node3, node2), (node4, node4)\}$$

$$Rel2 = \{(2, node3), (3, node2), (1, node4)\}$$

In Z, a sequence is more usually written as $\langle node4, node3, node2\rangle$.

The *domain* (dom) of a relation or function comprises those elements in the source set that are related, and the *range* (ran) comprises those elements in the target set that are related. In the examples above, $\mathrm{dom}\, Rel1 = \{node1, node2, node3, node4\}$, $\mathrm{ran}\, Rel1 = \{node2, node3, node4\}$, $\mathrm{dom}\, Rel2 = \{1, 2, 3\}$ and $\mathrm{ran}\, Rel2 = \{node2, node3, node4\}$.

Sets of elements can be defined using set comprehension. For example, the following expression denotes the set of squares of natural numbers greater than 10: $\{x : \mathbb{N} \mid$

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
$\mu\, d \mid P$	Unique value ascription
let $a == x$	Local variable definition

Logic

$\neg\ p$	Logical negation
$p \land q$	Logical conjunction
$p \lor q$	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
$\varnothing$	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
$\bigcup A$	Generalized union
$\#A$	Size of a finite set
$\{d;\ e \ldots \mid p \bullet x\}$	Set Comprehension

Relations

$A \leftrightarrow B$	Relation
$\mathrm{dom}\, R$	Domain of a relation
$\mathrm{ran}\, R$	Range of a relation
$R^{\sim}$	Inverse of a relation
$A \lhd R$	Anti-domain restriction
$R \oplus S$	Relational overriding

Functions

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

Sequences

$\mathrm{seq}\, A$	Set of finite sequences
$\mathrm{seq}_1\, A$	Non-empty set of sequences
$\langle\rangle$	Empty sequence
$\langle x, y, \ldots\rangle$	Sequence
$s \frown t$	Sequence concatenation
$head\ s$	First element of sequence
$tail\ s$	All but first element

Schema notation

S d --- p	Vertical schema
d --- p	Axiomatic definition
S T d --- p	Schema inclusion
ΔS S S'	Operation schema
$z.a$	Component selection

Conventions

$a?$	Input to an operation
a	State component before operation
a'	State component after operation
S	State schema before operation
S'	State schema after operation
ΔS	Change of state ($S \land S'$)
ΞS	No change of state
$OP_1 \fatsemi OP_2$	Operation composition

FIG. 1. Summary of Z notation.

$x > 10 \bullet x * x\}$. The way to write down predicates in Z is non-standard. To state that, say, the square of every natural number greater than 10 is greater than 1000, we write $\forall n : \mathbb{N} \mid n > 10 \bullet n * n > 100$. To state that all natural numbers are greater than or equal to 0 we simply write $\forall n : \mathbb{N} \bullet n \geq 0$.

The expression, $\mu a : A \mid P$, selects the unique element from the type A, which satisfies the predicate P.

Further details of the Z notation will be introduced as required through the course of the paper, with some definitions being provided in Appendix A. A summary of the notation to be used is given in Figure 1. For a more complete treatment of the Z language, the interested reader is referred to one of the numerous texts, such as [1, 3, 27]. Details of the formal semantics of Z are given in [25]. We will not consider such issues further in this paper.

4 Types and Primitives

Underlying AgentSpeak(L) as in other BDI systems (e.g. [10, 15, 17]) are the primitives that represent the basic mental attitudes in such components as goals, beliefs, intentions, and so on. In this section we define the basic types required to build the formal model. Though we explain each part of the specification, we assume some familiarity with Z. We illustrate some of the types using examples based on Rao's original description [22].

The sets of all constants and variables are given sets, and are denoted as $[Const]$ and $[Var]$ respectively. Similarly, the set of all functor symbols is denoted as $[FSym]$. As far as this specification is concerned, the contents of these sets are unimportant, and we use them directly without further elaboration. A *term* is either a constant, a variable, or a functor symbol with a non-empty sequence of terms as a parameter.

$$\begin{aligned} Term ::= & \; const\langle\langle Const\rangle\rangle \\ | & \; var\langle\langle Var\rangle\rangle \\ | & \; functor\langle\langle FSym \times \mathrm{seq}_1 \; Term\rangle\rangle \end{aligned}$$

4.1 Beliefs

Beliefs in AgentSpeak(L) typically include facts concerning static properties of the application domain, as well as new facts that are generated during the operation of the system. They are essentially descriptions in first-order predicate calculus. In Rao's example traffic-world simulation, these would include, for example, `adjacent(X, Y)`, `location(robot, X)`, `location(car, X)`, etc.

We define beliefs in our specification by building constructs from the above primitives, and through the introduction of the set of all predicate symbols, denoted by the given set, $[PredSym]$. Then, a belief *atom* (equivalently a predicate) is a predicate symbol with a sequence of terms as its argument.

Atom

$$\begin{aligned} & head : PredSym \\ & terms : \mathrm{seq}\; Term \end{aligned}$$

A belief literal is either an atom or the negation of an atom.

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

For example, if the type, *Atom*, contained only the elements $P(X, Y)$ and $Q(Y)$, then the type, *Literal*, would be equal to the following set.

$$\{not\ P(X, Y), pos\ P(X, Y), not\ Q(Y), pos\ Q(Y)\}$$

Beliefs are then either *belief literals*, or conjunctions of two beliefs.

$$Belief ::= literal\langle\langle Literal\rangle\rangle \mid and\langle\langle Belief \times Belief\rangle\rangle$$

The set of ground belief atoms, called *base beliefs*, comprises those atoms containing no variables. Using the example above, the predicate, `adjacent(car,robot)`, is a base belief since there are no variables. In order to specify this, we first define the auxiliary function, *beliefvars*, that returns the set of variables in a belief. This requires the specification of two further auxiliary functions, *atomvars* and *termvars*, which return the set of variables of an atom and a term, respectively.

$$\begin{array}{l} termvars : Term \rightarrow (\mathbb{P}\ Var) \\ atomvars : Atom \rightarrow (\mathbb{P}\ Var) \\ beliefvars : Belief \rightarrow (\mathbb{P}\ Var) \\ \hline \forall\ a : Atom;\ c : Const;\ v : Var;\ f : FSym;\ t : \mathrm{seq}\ Term;\ m, n : Belief \bullet \\ \quad termvars\ (const\ c) = \varnothing \wedge \\ \quad termvars\ (var\ v) = \{v\} \wedge \\ \quad termvars\ (functor(f, t)) = \bigcup(\mathrm{ran}(map\ termvars\ t)) \wedge \\ \quad atomvars\ a = \bigcup(\mathrm{ran}(map\ termvars\ a.terms)) \wedge \\ \quad beliefvars\ (and\ (m, n)) = beliefvars\ m \cup beliefvars\ n \wedge \\ \quad beliefvars\ (literal\ (pos\ a)) = atomvars\ a \wedge \\ \quad beliefvars\ (literal\ (not\ a)) = atomvars\ a \end{array}$$

It is then a simple matter to define the set of ground beliefs.

$$BaseBelief == \{b : Belief \mid beliefvars\ b = \varnothing\}$$

4.2 Plans

Plans specify the way in which an agent accomplishes tasks, and are vital to any deliberative agent. In AgentSpeak(L) as in other BDI architectures, the task of generating plans is not addressed, and instead a library of plans is available to the agent for selection. Plans in AgentSpeak(L) contain either goals to be achieved or actions to be performed. In addition, plans have an *invocation condition*, which states when they are triggered by perceived changes to the environment, and a *context*, which states the beliefs that must hold for the plan to be selected. Before moving to the specification of plans, we must first define the goals and actions they comprise, and the preconditions that determine their applicability.

Motivational attitudes in the desire component of the BDI architecture are manifested through the *goals* in the system. In AgentSpeak(L), as in standard AI terminology, a goal is a system state that an agent wants to bring about. There are two

types of goals, both of which are represented as a predicate symbol and a sequence of terms (i.e. an atom) prefixed with an appropriate identifier. For an *achievement* goal, by which the agent wants to bring about a system state in which the atom is a true belief, an *achieve* identifier is used, while for a *test* goal, by which the agent wants to test if the atom is a true belief, a *query* identifier is used.

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

In order to achieve goals or accomplish tasks, an agent must perform actions. In this specification, the set of action symbols is a given set denoted by $[ActionSym]$, and the set of actions is specified in the same way as atoms, above.

Action

$$name : ActionSym$$
$$terms : \mathrm{seq}\, Term$$

The preconditions of a plan relate to changes in the goals and beliefs of an agent, which are updated in response to *triggering events*. These triggering events occur when there is a perceived change in the environment, or when a new goal is acquired. We define a *trigger* to be either a belief or a goal, and a triggering event to involve the addition or deletion of a trigger.

$$Trigger ::= belief\langle\langle Belief\rangle\rangle \mid goal\langle\langle Goal\rangle\rangle$$
$$TriggerSymbol ::= + \mid -$$
$$TriggerEvent == TriggerSymbol \times Trigger$$

Now we can specify an AgentSpeak(L) plan, which comprises three components.

- First, an *invocation condition* detailing the circumstances, in terms of beliefs or goals, that caused the plan to be triggered, is specified by a *triggering event*.
- Similarly, a *context* specifies the beliefs of the agent that must hold for the plan to be selected for execution.
- Finally, the part of the plan that specifies the sequence of *formulae* that the agent needs to perform, is known as the plan *body*. This determines what the agent must *do*. A formula is either an action to be executed, an achieve goal to be satisfied, or a test goal to be answered.

This is illustrated in Figure 2(a) in which there is an invocation condition and a context, and a sequence of formulae arranged in the order in which they must be addressed.

$$Invocation == TriggerEvent$$
$$Context == \mathbb{P}\, Belief$$
$$Formula ::= actionformula\langle\langle Action\rangle\rangle \mid goalformula\langle\langle Goal\rangle\rangle$$
$$Body == \mathrm{seq}\, Formula$$

Plan

$$inv : Invocation$$
$$context : Context$$
$$body : Body$$

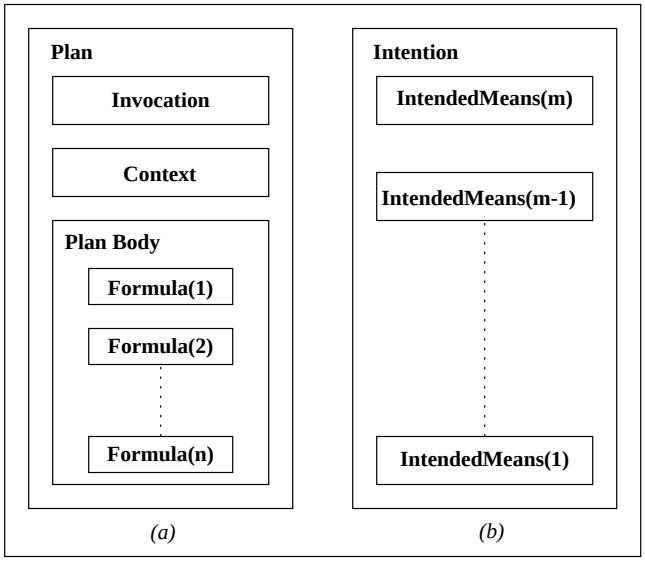

FIG. 2. The AgentSpeak(L) plan and intention

Consider a very simple example of a plan, given below, which is triggered whenever a compulsive robot car-thief finds itself next to a car. In order to select this plan, the robot must believe that the car is currently empty. The plan body, which details how to gain control of the car, consists of a sequence of formulae. In this case, the robot must first perform the primitive action to move to the car, then the primitive action to get in the car, and then achieve the subgoal to start the car. The actual achievement of this subgoal may require further plans.

$$inv = (+, \mathit{belief}\ \texttt{adjacent(Robot, Car)})$$
$$\mathit{context} = \{\mathit{literal}\ (\mathit{pos}\ \texttt{empty(Car)})\}$$
$$\mathit{body} = \langle \mathit{actionformula}\ \texttt{MoveTo(Car)}, \mathit{actionformula}\ \texttt{GetIn(Car)},$$
$$\mathit{goalformula}\ (\mathit{achieve}\ \texttt{Start(Car)})\rangle$$

A *plan instance* is a partially instantiated plan (in the sense that some of the plan's variables may have been bound to terms), and is referred to as the *intended means*. The intended means represents a *copy* of the original plan that now serves as a *mental attitude* directing behaviour as opposed to a *recipe* for behaviour. The distinction between plans as recipes and plans as mental attitudes is very important in the study of BDI agents and in this specification we distinguish between calling the former plans and the latter plan instances. In AgentSpeak(L), a plan instance has the same type as a plan (although in other BDI systems such as the Procedural Reasoning System [13], plan instances and plans do not have the same type [7]).

$$\mathit{PlanInstance} == \mathit{Plan}$$

4.3 Intentions

Intentions are plans for which a commitment to execute is made. Such commitment is needed to avoid infinite evaluation of alternatives to force action. More precisely, an *intention* is a stack of intended means, each of which makes a contribution to the achievement of the intended means at lower levels, culminating in the achievement of the original intended means at the bottom. A schematic of an intention is given in Figure 2(b), the numbers representing the order in which the intended means are generated.

The original intended means at the bottom of the stack causes a subgoal to be posted, and thus leads to the addition of a further intended means to carry out the tasks required to achieve this subgoal. The intended means at the top of the stack is the one currently executing and, once it has executed (and the subtask has been completed), it is removed. Then, the next formula in the next (second) intended means can be attempted (as the first formula has executed successfully), and the plan continues its execution, possibly requiring other, new intended means to be added to the stack, in turn.

Formally, we represent an intention as a non-empty sequence of plan instances, with the first element of the sequence representing the top of the stack.

$$Intention \;==\; \mathrm{seq}_1\; PlanInstance$$

Events provide the basis for agent operation in that they are *cues* for reasoning and action. In AgentSpeak(L), events result either from an external source, in which case they are just external triggering events unrelated to an intention, or from the execution of a current intention, in which case they are (subgoal) triggering events with an explicit connection to an intention. Thus, we define the *Event* type as follows, where *int* represents an *optional* intention. (The definition of *optional* and related concepts, *defined*, *undefined* and *the*, can be found in Appendix A.)

$$\begin{array}{|l} \textit{Event} \\ \hline trig : TriggerEvent \\ int : optional[Intention] \\ \hline \end{array}$$

As stated above, in an external event, the intention is not defined, while in an internal event, it is defined.

$$ExternalEvent == [Event \mid undefined\ int]$$
$$InternalEvent == [Event \mid defined\ int]$$

In order to construct an event, we specify the auxiliary function, *MakeEvent*, which creates an instance of type, *Event*, from its constituent parts.

$$\begin{array}{|l} MakeEvent : (TriggerEvent \times optional[Intention]) \to Event \\ \hline \forall\, t : TriggerEvent;\ i : optional[Intention] \bullet \\ \quad MakeEvent\ (t, i) = (\mu\, e : Event \mid e.trig = t \wedge e.int = i) \end{array}$$

5 AgentSpeak(L) Agents

Thus far, we have described and formally specified the mental components and data structures that are needed to describe AgentSpeak(L) agents and BDI agents in general. We have not, however, described the way in which these components are put together to define the agents themselves. In this section, we bring together this somewhat disparate group of components for exactly this purpose.

An agent in AgentSpeak(L) consists of the following four distinct items.

- The *plan library* is a repository that contains the available plans for the agent's use. These plans are pre-compiled plans, and do not require any actual planning on the part of the agent. In the schema below we define a variable, *capabilities*, as a set of actions. It is evaluated by taking every action from the body of each plan in the plan library (using the generic function, *mapset*, that is defined in Appendix A).
- The *event-selection function* selects an event to process from a set of events. Events are accumulated in this set as they are perceived by the agent.
- Similarly, the *applicable-plan-selection function* is responsible for selecting a plan from those plans in the plan library that are applicable in the current circumstances.
- Finally, the *intention-selection function* selects an intention to execute from the set of current intentions.

Each of these components is necessary for the operation of the agent, considered in the next section. Formally, we write this agent definition as the schema below. Note that we state that an agent must have a non-empty set of plans and that the selection functions are only defined for non-empty sets.

AgentSpeakAgent

$$
\begin{array}{l}
planlibrary : \mathbb{P}_1\ Plan \\
intselect : \mathbb{P}_1\ Intention \rightarrow Intention \\
planselect : \mathbb{P}_1\ Plan \rightarrow Plan \\
eventselect : \mathbb{P}_1\ Event \rightarrow Event \\
capabilities : \mathbb{P}_1\ Action \\
\hline
capabilities = \{p : planlibrary;\ a : Action \mid \\
\qquad a \in mapset\ actionformula^{\sim}\ (\mathrm{ran}\, p.body) \bullet a\}
\end{array}
$$

Given this description, it is possible to specify the state of an agent, which describes the agent at run-time. This is defined by

- the components that comprise the AgentSpeak(L) agent as specified in the *AgentSpeakAgent* schema above;
- the set of beliefs of the agent;
- its set of intentions;
- the set of events yet to be processed by the agent; and
- the set of actions the agent is to perform.

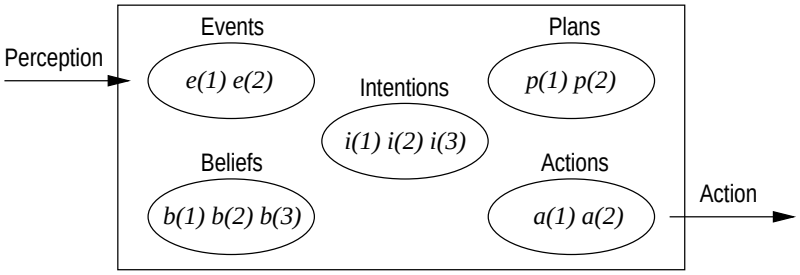

FIG. 3. The AgentSpeak(L) Agent State

Figure 3 provides an illustration of the agent's state, which is completely determined by its events, beliefs, intentions and actions as described. Since it is not clear what the goals of an agent are at any time, there are many possible ways of defining them. One possibility is that the goals of an agent can be determined by retrieving all the goals (both achieve and query goals) from each of its current intentions. This possibility is defined in the following schema specifying the agent's state, in which the redundant variable, *goals*, represents the goals from the body of every plan instance in the set of current intentions.

In addition, it is necessary to record which of the current intentions are active and which are suspended. The motivation for the variable, *status*, which records this information, is considered later. The first predicate in the schema states that every current intention has a status.

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

AgentSpeakAgentState

$$\begin{array}{l} AgentSpeakAgent \\ beliefs : \mathbb{P}_1\, Belief \\ intentions : \mathbb{P}\, Intention \\ events : \mathbb{P}\, Event \\ actions : \mathbb{P}\, Action \\ status : Intention \pfun Status \\ goals : \mathbb{P}\, Goal \\ \hline \mathrm{dom}\, status = intentions \\ goals = \{i : intentions;\ p : PlanInstance;\ g : Goal \mid \\ \qquad p \in (\mathrm{ran}\, i) \land g \in (mapset(goalformula^{\sim})(\mathrm{ran}\, p.body)) \bullet g\} \end{array}$$

Before the agent begins its operation, it is initially given a set of beliefs, or a knowledge base, but it has no events to process, no intentions and no actions to perform, as shown in Figure 4.

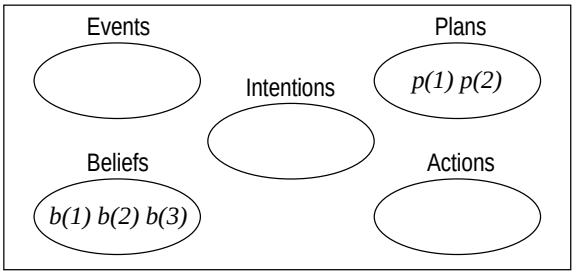

FIG. 4. The Initial AgentSpeak(L) State

InitAgentSpeakState

AgentSpeakAgentState

$actions = \varnothing$
$intentions = \varnothing$
$events = \varnothing$

6 The AgentSpeak(L) Agent Cycle of Operation

6.1 Agent Operation Overview

As stated earlier, the model of operation of AgentSpeak(L) agents is essentially similar to that of other BDI systems such as PRS and dMARS. There are two modes of operation involving either responding to an event or executing intentions. The former is illustrated in Figure 5 and is described below.

1. The agent selects an event (e).
2. The agent generates all the plans whose invocation conditions match this event. These plans are known as the *relevant* plans ($\{p(1), p(2), p(3)\}$).
3. From these relevant plans, the agent identifies those with pre-conditions that are currently satisfied with respect to the agent's beliefs. These plans are known as *applicable* plans ($\{p(2), p(3)\}$).
4. If several plans are both relevant and applicable, one is chosen ($p(2)$) and used to form a *plan instance* (known as the *intended means*). The agent's intentions are then updated in the following way: if the selected event is external, a new intention is generated; if the selected event is internal, the plan instance is added to the head of the intention that posted it.

The second aspect of the agent's operation cycle, which takes place after the intentions have been updated, is the execution of the intentions. One intention (referred to as the *selected intention*) is chosen from the set of intentions, the intended means at the top of this intention stack is identified as the *executing plan*, and the next formula in this plan is identified as the *executing formula*. Then, depending on the selected intention and the executing formula of the executing plan, there are three possible courses of action as follows.

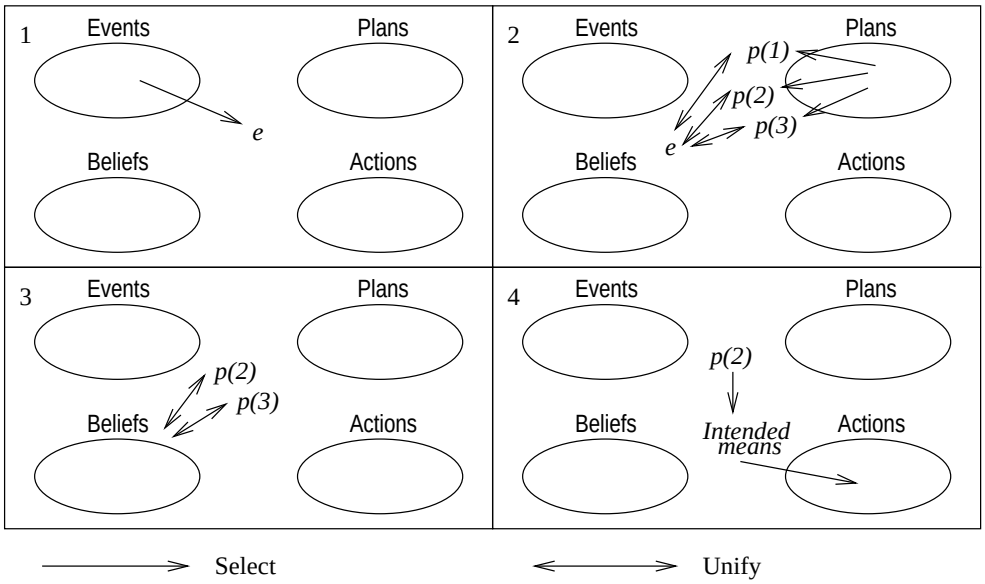

FIG. 5. Updating Intentions in response to a Perceived Event

1. If the executing formula is an achieve goal, a new goal event is generated and posted to the set of events awaiting processing. In this case the intention must be suspended until this goal is achieved (by another plan) so that it cannot be selected for execution again. (Though this is not made explicit in Rao's original description of AgentSpeak(L) [22], the restriction must be introduced to avoid suspended intentions incorrectly being selected by the intention-selection function.)
2. If the formula is a query goal then, if it can be unified with the set of beliefs, the *most general unifier* (*mgu*) of the query goal and the set of beliefs is applied to the rest of the executing plan. (Section 6.2 and Appendix B provide further details of unifiers.)
3. If the next formula in the current plan is an action, it is placed in a buffer for future performance.

In the last two of these cases, the executing formula is removed. If there is then no next formula in the executing plan, but a next plan in the selected intention, the *mgu* of the invocation condition of the *second plan* on the stack and the invocation of the *executing plan* is found, and this substitution is applied to the second plan in the stack. If there is no next formula and no next plan, then the intention has succeeded since it is empty, and can be removed from the set of intentions. (Again, note that this latter possibility is not addressed in Rao's original operational semantics [22], but is an important case that demands explicit consideration.)

With this description and the preceding specification of agents, their components and their state, it is possible to specify the AgentSpeak(L) agent operation formally.

In Z, an operation is represented as a change in state. An operation schema defines the pre-conditions that must be true for the operation to take place and also how the state is affected. Dashed state variables represent the state after the operation and

undashed state variables represent the state before the operation.

While there are no changes to the variables of *AgentSpeakAgent* in the AgentSpeak(L) agent cycle, the variables defining the *AgentSpeakAgentState* will change. The Δ convention states that some state variables change and the Ξ convention states that all the dashed variables are equal to their undashed counterparts (i.e. no state change).

$$\Delta AgentSpeakAgentState$$
$$AgentSpeakAgentState$$
$$AgentSpeakAgentState'$$
$$\Xi AgentSpeakAgent$$

Before proceeding, there still remains one operation to specify, which is the perception by an agent of a new triggering event. This is the *external* event that was mentioned earlier, arising from causes that are not part of the cycle outlined above. The set of events is updated appropriately in response, as specified in the schema below.

$$NewExternalEvent$$
$$trigger? : TriggerEvent$$
$$\Delta AgentSpeakAgentState$$
$$events' = events \cup \{(\mu\, e : Event \mid e.trig = trigger? \land undefined\ e.int)\}$$

6.2 Determining Relevant and Applicable Plans

We begin by considering how, starting with a plan as a *recipe* incorporating procedural knowledge, an agent can generate a plan as an *adopted mental attitude* to guide action.

In essence, an agent's behaviour is determined by its intentions, which comprise sequences of executing plans. These plans are selected for execution when they are deemed to be both *relevant* to the events in the set of events waiting to be processed, and *applicable* with respect to the current beliefs of the agent. The conditions of relevance and applicability are used to generate *bindings*, as we see below. A set of bindings that associates variables with terms is known as a *substitution*. (Full details of standard aspects of binding and unification are not included in the main part of this paper due to space constraints, though some further details are available in Appendix B.)

A plan is *relevant* with respect to an event if there exists a *most general unifier* to bind the invocation of the plan and the event so that they are equal. This is specified by the function, *genrelplans*, which is passed an event and a set of plans, and generates from that set the *relevant* plans, also recording the unifiers that make them relevant.

$$genrelplans : Event \to \mathbb{P}\, Plan \to \mathbb{P}(Plan \times Substitution)$$
$$\forall\, e : Event;\ lib : \mathbb{P}\, Plan \bullet$$
$$genrelplans\ e\ lib = \{p : lib;\ \sigma : Substitution \mid mgutriggerevents(e.trig, p.inv) = \sigma \bullet (p, \sigma)\}$$

A relevant plan is applicable if its *context* is a logical consequence of the beliefs of the agent. Thus, if we define a predicate, *LogicalConsequence*, to hold between a context condition (which is a set of beliefs) and the set of agent beliefs when the former is a *logical consequence* of the latter, we can define an *applicable plan*.

$$LogicalConsequence\_ : \mathbb{P}(\mathbb{P}\, Belief \times \mathbb{P}\, Belief)$$

In the original description of AgentSpeak(L), this notion of logical consequence is not explained. It is true to say that there are many possible ways of implementing it (such as resolution, for example), with some far more computationally expensive than others. Since the choice of the underlying mechanism may be constrained by the resources available, it is sensible to leave that question open.

Applicable plans are defined by the *genapplplans* function, which takes a set of pairs of (relevant) plans, their associated substitutions, σ, and a set of beliefs, and returns those plans that are applicable together with a further substitution, ψ, which, when composed with σ, binds the context of the relevant plan so that it is a logical consequence of the beliefs. Thus, in this case, $\sigma \ddagger \psi$ is referred to as the *applicable unifier*, and ψ is referred to as the *correct answer substitution*. A definition of substitution composition can be found in Appendix B.3.

$$AssRelSubs == \mathbb{P}(Plan \times Substitution)$$

$$genapplplans : AssRelSubs \to \mathbb{P}\, Belief \to \mathbb{P}\, AssRelSubs$$

$$\begin{aligned}
&\forall\, relsubs : AssRelSubs;\ bels : \mathbb{P}\, Belief \bullet \\
&\quad genapplplans\ relsubs\ bels = \\
&\qquad \{rel : Plan;\ \sigma, \psi : Substitution \mid (rel, \sigma) \in relsubs \land \\
&\qquad (\forall\, b : rel.context \bullet LogicalConsequence(\{ASBelief\ (\sigma \ddagger \psi)\ b\}, bels)) \\
&\qquad\qquad \bullet\ (rel, \sigma \ddagger \psi)\}
\end{aligned}$$

6.3 Agent Operations

We now consider the actual agent operation, and specify the agent selecting an event, represented by the locally defined variable, *selectedevent*, generating all the plans relevant to this event, *relevantplans*, and then generating all the applicable plans, *applicableplans*, as illustrated earlier in Figure 5. Determining the relevant plans amounts to an application of the *genrelplans* function (specified above) to the selected event and the plan library. Similarly, determining the applicable plans involves applying the corresponding function to the relevant plans and beliefs. Then, the applicable plan, *selectedplan*, and the applicable unifier, *applicableunifier*, are found by applying the plan-selection function. The intended means, *intendedmeans*, is generated by instantiating the plan using the applicable unifier.

The intentions of the agent can then be updated. If the event is external, then a new intention consisting of the intended means is generated and set to *active*, whereas if the event is internal then the intended means is pushed onto the intention that generated the internal event, as described earlier. Since execution of this updated intention can now resume, its status is set to *active*. The expression, $\{oldintention\} \lhd status$, removes the old intention from the domain of the status function, and the pair,

$\{(newintention, active)\}$, then sets the updated intention to be active. (Note that in Z, **let** defines a local variable, while *the* is defined to be a function that extracts the element from a defined member of type *optional*, introduced earlier. Its definition is given in Appendix A.)

ProcessEvent

$\Delta AgentSpeakAgentState$

$events \neq \varnothing$
let $selectedevent == eventselect\ events \bullet$
 let $relevantplans == genrelplans\ selectedevent\ planlibrary \bullet$
  let $applicableplans == genapplplans\ relevantplans\ beliefs \bullet$
   let $selectedplan == planselect\ (\mathrm{dom}\ applicableplans) \bullet$
    let $applicableunifier == applicableplans\ selectedplan \bullet$
     let $intendedmeans == ASPlan\ applicableunifier\ selectedplan \bullet$
    $selectedevent \in ExternalEvent \Rightarrow$
     $(intentions' = intentions \cup \{\langle intendedmeans \rangle\} \land$
     $status' = status \cup \{(\langle intendedmeans \rangle, active)\}) \land$
    $selectedevent \in InternalEvent \Rightarrow$
     (**let** $oldintention == (the\ selectedevent.int) \bullet$
      let $newintention == \langle intendedmeans \rangle \frown oldintention \bullet$
      $(intentions' = (intentions \setminus \{oldintention\}) \cup \{newintention\} \land$
      $status' = (\{oldintention\} \lhd status) \cup \{(newintention, active)\}))$

Once the agent has updated its intentions according to the intended means in response to the processing of an event, it moves to the second part of its operation cycle, concerned with the execution of its intentions. We refer to this part of the agent cycle as *intention-execution*. The schema below includes extra, redundant, state variables in order to make the specification of an agent's operation more readable, and to make a stronger connection between the formal specification and any subsequent implementation. We defer discussion of these variables until the next paragraph when we state how they relate to intention-execution.

AgentIntExecutionState

$AgentSpeakAgentState$

$selectedintention : Intention$
$executingplan : Plan$
$executingformula : Formula$

First, an active intention is selected for execution. In the schema below, the variable, $selectedintention'$, is the selected intention, $executingplan'$ represents the first plan in this intention and $executingformula'$ the first formula in this plan. Note that the precondition states explicitly that the event queue may be either empty or non-empty for intentions to be executed, in contrast to Rao's original description, which only allows execution of intentions if this set is non-empty. The third predicate asserts that the selected intention must be active.

SelectIntention

$\Delta AgentIntExecutionState$
$\Xi AgentSpeakAgentState$

$events = \varnothing \lor events \neq \varnothing$
$selectedintention' = intselect\ intentions$
$status\ selectedintention' = active$
$executingplan' = head\ selectedintention$
$executingformula' = head\ executingplan.body$

Next, we specify the three possibilities that arise from the different cases in the formula at the top of the intention. These vary depending on whether the formula is an achieve goal, a query goal or an action. Each of these cases is detailed below.

First, if the formula is an achieve goal, it is assumed that it cannot immediately be achieved, and a goal event is created. This event is added to the set of current events to be processed by the agent. In turn, it alerts the agent to finding a plan to achieve the goal so that the execution of the current intention stack can continue.

PostAchieveGoal

$\Delta AgentIntExecutionState$

$executingformula \in \mathrm{ran}\, goalformula$
$(goalformula^{\sim}\ executingformula) \in \mathrm{ran}\, achieve$
let $achievegoal == goalformula^{\sim}\ executingformula \bullet$
$\quad events' = events \cup$
$\quad\quad \{MakeEvent\ ((+, goal\ achievegoal), \{selectedintention\})\}$

In this case, the status of the intention that has just executed must be set to *suspended*.

SuspendIntention

$\Delta AgentIntExecutionState$

$status' = status \oplus \{(selectedintention, suspended)\}$

Second, if the formula is a query goal that can be unified with the set of current beliefs, then the resulting *mgu* is applied to the rest of the executing plan. In the following schema, *querygoal* represents the query goal, *mgu* is the most general unifier of the goal and the beliefs, and the *executingplan* variable is the result of applying *mgu* to *executingplan*.

AchieveQueryGoal

$\Delta AgentIntExecutionState$

$executingformula \in \mathrm{ran}\, goalformula$
$goalformula^{\sim}\ executingformula \in \mathrm{ran}\, query$
let $querygoal == goalformula^{\sim}\ executingformula \bullet$
$\quad$ **let** $mgu == mguquery(querygoal, beliefs) \bullet$
$\quad\quad executingplan' = ASPlan\ mgu\ executingplan$

Third, if the formula is an action, it is added to the set of actions to be performed.

PostAction

$$\Delta AgentIntExecutionState$$

$$executingformula \in \operatorname{ran} actionformula$$
$$actions' = actions \cup \{actionformula^{\sim} executingformula\}$$

The original description of AgentSpeak(L) does not explicitly address what happens when an action is performed (whether it is successful or fails) or when a query goal cannot be unified with the beliefs. This is a serious omission for the system. However, in the previous two cases of achieving a query goal and adding an action to the set of those to be performed, the executing formula is explicitly removed from the body of the executing plan.

RemoveFormula

$$\Delta AgentIntExecutionState$$

$$executingplan'.body = tail\ (executingplan.body)$$
$$selectedintention' = \langle executingplan' \rangle \frown tail\ selectedintention$$
$$intentions' = (intentions \setminus \{selectedintention\}) \cup \{selectedintention'\}$$

If, after the formula is removed, there are no more formulae in the current plan (that is, $\#executingplan.body = \langle\rangle$), but there are still more plans in the intention stack ($\#selectedintention > 1$), then the current plan is finished and the next plan is ready to be executed. However, since additional binding constraints may have been introduced as a result of the execution of the current plan to achieve the triggering goal of the next plan, any new bindings must be carried through into the next plan. In order to achieve this, the *mgu* of the goal that triggered the executing plan (which must have come from the second plan in the intention) and the invocation condition of the current executing plan is constructed, and the binding is then applied to the next plan in the stack.

In the schema below, three local variables are defined. First, *nextplan*, represents the second plan in the executing intention. Second, the formula at the head of the body of *nextplan* is represented by *triggeringgoal*, which must have led to the instantiation of the plan just achieved. Third, the most general unifier of *triggeringgoal* and the invocation condition of the executing plan is denoted by *mgu*. The new executing plan is then generated by applying *mgu* to *nextplan*, and the intentions are updated in the same way as defined in the last two predicates of the *RemoveFormula* schema. Finally, the *subgoal* event that generated the plan just achieved is isolated and removed from the set of events. (Again, this operational necessity is not considered in the original description of AgentSpeak(L).)

AchievePlanOnly

$\Delta AgentIntExecutionState$

$executingplan.body = \langle\rangle$
$\#selectedintention > 1$
let $nextplan == selectedintention\ 2 \bullet$
let $triggeringgoal == goalformula^{\sim}(head\ nextplan.body) \bullet$
let $mgu == mgugoal(executingplan.inv, triggeringgoal) \bullet$
$executingplan' = ASPlan\ mgu\ nextplan$
$selectedintention' = \langle executingplan' \rangle \frown (tail\ selectedintention)$
$intentions' =$
$(intentions \setminus \{selectedintention\}) \cup \{selectedintention'\}$
$events' = events \setminus \{(\mu\, e : InternalEvent \mid$
$e \in events \land the\ e.int = tail\ selectedintention)\}$

After this occurs, the top formula (i.e. the triggering goal) must be removed, since it has now been achieved, as specified by *RemoveFormula*. This is specified at the end of this section.

If, after achieving a formula, the plan has completed and there are no further plans in the executing intention, then the intention has been achieved and can be removed from the agent's set of intentions. Similarly, the *external* event that originally generated this intention is also removed. In attempting to formalise this operation, a problem in Rao's (and hence AgentSpeak(L)'s) representation of an event is highlighted. An external event *cannot*, in general, be identified by the completing intention because the intention component of such an event is *undefined*. Since variables of the invocation condition of the achieved plan may have been bound by previous plans in the stack, it is not possible simply to compare the trigger of the event and the invocation condition of the plan. In general, therefore, it is not possible to identify the associated external event of a completed intention. It can only be identified if there is only one external event with a trigger that can be unified with the invocation condition of the completing plan. Though we make this assumption in this specification, the way to specify the more general case would be to associate each event with a unique identifier that is recorded as part of the plan instance it generates.

AchievePlanAndIntention

$\Delta AgentIntExecutionState$

$executingplan.body = \langle\rangle$
$\#selectedintention = 1$
$intentions' = intentions \setminus \{selectedintention\}$
$events' = events \setminus \{(\mu\, e : ExternalEvent \mid (e \in events) \land$
$(\exists\, s : Substitution \bullet unifitrigevents(s, (executingplan.inv, e.trig)))\}$

For completeness, we must also specify an agent that has not achieved a plan after removing a formula.

NotAchievePlan

$\Xi AgentIntExecutionState$

$executingplan.body \neq \langle\rangle$

Finally, we can specify some procedural aspects of AgentSpeak(L) operation by showing how the schemas defined here are put together.

Before giving the complete picture, we must first specify what happens when either an action is added to the set of those to be performed or a query goal is achieved. There are three alternative possibilities that may arise: first, the plan alone has been achieved, in which case the first formula from the next plan must be removed (as it generated the completed plan); second, the plan and the corresponding intention have both been achieved; and third, the plan has not been achieved as there are more formulae. The total operation, CHECKPLAN, below, which has a precondition of *true*, is defined as the disjunction of these schemas. It uses operation schema composition, denoted by $\fatsemi$.

$$CHECKPLAN == (AchievePlanOnly \fatsemi RemoveFormula) \lor AchievePlanAndIntention \lor NotAchievePlan$$

With this, we can also specify the operation of an agent executing a formula. When executing a formula, the possible outcomes are either that an achieve goal is added to the set of events and the intention is suspended; that an action is added to those to be performed and the action formula is removed; or that a query goal has been achieved and the goal formula is removed. In both of the latter two cases, since the formula is removed, the plan must be checked as described previously.

$$ExecuteFormula == (PostAchieveGoal \fatsemi SuspendIntention) \lor ((PostAction \lor AchieveQueryGoal) \fatsemi RemoveFormula \fatsemi CHECKPLAN)$$

7 Conclusions

In providing a formal Z specification of AgentSpeak(L), we are attempting to address several distinct concerns. The first of these is shared with Rao's original work, as stated in the introduction, to bring together the disparate strands of theory and practice, and to bridge the gap between formal specifications and implemented systems. While we do not claim that the specification presented above is directly executable, it can be said to be implementable, in that it is animatable and makes full implementation much closer by providing a clean and explicit representation of the state and operations on state that must underlie any implementation. Indeed, animating the specification is made simpler by the existence of tools for doing so, such as [16]. By using the standard Z specification language, we also tackle the problem from a software engineering perspective and make the specification accessible and amenable to implementation by, for example, identifying data structures required for operation. Moreover, this reformalization has revealed a number of errors and omissions in the original formulation, including some relating to the specification of aspects of binding and plans, an omission of one possible case in the agent operation cycle, and the incorrect assertion that an intention stack can only be executed if the event queue is non-empty, which is inappropriate.

This identification of errors is due to three things: the process of re-specification itself; the fact that, in Z, preconditions for operations are given explicitly, thus enabling us to identify missing cases in the original description; and, primarily, the property of Z being strongly-typed so that specifications can be checked for syntax and type correctness [26]. An obvious next step would be to move forward with regard to verification, and to develop proofs relating to the specification, possibly through the use of proof tools such as Z/EVES [24]. Though this has not yet been done, further development of this work will aim to do this as well as to animate and implement the specification.

Second, the specification provides a base for further specification of more sophisticated systems such as, for example, dMARS. The choice of AgentSpeak(L) as a place to start is deliberate because of its status as a simplified version of more sophisticated systems, so that we can use it as an abstraction that can be instantiated and complicated further, both to provide specifications of implemented systems, and to examine aspects of the model in more detail.

Our third concern is to map this architecture onto the formal agent framework previously constructed using Z [18] in order to engender closer links between alternative agent models. This specification is the first step along this path. Related to all of these points is the final concern of developing a library of agent architectures and agent components that are not tied to particular implementation platforms. One of the key overall project goals in pursuing work on formal specification of agent systems is to support the principled development of practical systems. We envisage a collection of specifications for both existing systems and distinct agent components that may be selected as appropriate for the problem at hand, and used as the basis of implementation. The specification of AgentSpeak(L) in this paper should be seen as one part of this much larger ambition.

Acknowledgements

Many thanks to Anand Rao, David Kinny and Michael Georgeff who provided many illuminations and insights in discussions with the first author during development of the specification contained in this paper. Thanks also to the University of Westminster and the Australian Artificial Intelligence Institute for supporting and hosting the first author during the development of this work. In addition, we would like to thank the anonymous referees who made many valuable comments. The specification contained in this document has been checked for type-correctness using the fuzz package [26].

References

[1] J. Bowen. *Formal Specification and Documentation using Z: A Case Study Approach.* International Thomson Computer Press, 1996.

[2] I. D. Craig. *The Formal Specification of Advanced AI Architectures.* Ellis Horwood, 1991.

[3] A. Diller. *Z: An introduction to formal methods.* John Wiley, second edition, 1994.

[4] M. d'Inverno and J. Crowcroft. Design, specification and implementation of an interactive conferencing system. In *Proceedings of IEEE Infocom, Miami, USA. Published IEEE*, 1991.

[5] M. d'Inverno and M. Hu. A Z specification of the soft-link hypertext model. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation, Lecture Notes in Computer Science, 1212*, pages 297–316, Heidelberg, 1997. Springer-Verlag.

[6] M. d'Inverno, G. R. Justo, and P. Howells. A formal framework for specifying design methodologies. *Software Process: Improvement and Practice*, 2(3):181–195, September, 1996.

[7] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages (ATAL'97)*. Springer Verlag, To Appear 1998.

[8] M. d'Inverno and M. Luck. Formalising the contract net as a goal directed system. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, LNAI 1038*, pages 72–85. Springer-Verlag, 1996.

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

[10] K. Fischer, J. P. Müller, and M. Pischel. A pragmatic BDI architecture. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II: Proceedings of the Second Workshop on Agent Theories Architectures and Languages, LNAI 1037*, pages 203–218. Springer-Verlag, 1996.

[11] D. Garlan. The role of formal reusable frameworks. *ACM SIGSOFT: Software Engineering Notes*, 15(4):42–44, 1990.

[12] M. R. Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, 1987.

[13] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 677–682. AAAI Press / MIT Press, 1987.

[14] R. Goodwin. A formal specification of agent properties. *Journal of Logic and Computation*, 5(6):763–781, 1995.

[15] A. Haddadi and K. Sundermeyer. Belief-desire-intention architectures. In G. M. P. O'Hare and N. R. Jennings, editors, *Foundations of Distributed Artificial Intelligence*, pages 231–246. John Wiley and Sons, 1996.

[16] 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, editors, *ZUM'97: The Z Formal Specification Notation, Lecture Notes in Computer Science, 1212*, pages 37–51, Heidelberg, 1997. Springer-Verlag.

[17] N. R. Jennings. Specification and implementation of a belief desire joint-intention architecture for collaborative problem solving. *Journal of Intelligent and Cooperative Information Systems*, 2(3):289–318, 1993.

[18] M. Luck and M. d'Inverno. A formal framework for agency and autonomy. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 254–260. AAAI Press / MIT Press, 1995.

[19] M. Luck and M. d'Inverno. Engagement and cooperation in motivated agent modelling. In *Distributed Artificial Intelligence Architecture and Modelling: Proceedings of the First Australian Workshop on Distributed Artificial Intelligence, Lecture Notes in Artificial Intelligence, 1087*, pages 70–84. Springer Verlag, 1996.

[20] M. Luck, N. Griffiths, and M. d'Inverno. From agent theory to agent construction: A case study. In *Intelligent Agents III: Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages, Lecture Notes in Artificial Intelligence, 1193*, pages 49–63. Springer-Verlag, 1997.

[21] 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.

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

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

[24] M. Saaltink. The Z/EVES system. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation, Lecture Notes in Computer Science, 1212*, pages 72–85, Heidelberg, 1997. Springer-Verlag.

[25] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

[26] J. M. Spivey. *The fUZZ Manual*. Computing Science Consultancy, 2nd edition, 1992.

[27] J. M. Spivey. *The Z Notation*. Prentice Hall, Hemel Hempstead, 2nd edition, 1992.

[28] J. Woodcock and J. Davies. *Using Z: Specificiation, Refinement and Proof*. Prentice Hall, 1996.

[29] M. J. Wooldridge and N. R. Jennings. Formalizing the cooperative problem solving process. In *Proceedings of the Thirteenth International Workshop on Distributed Artificial Intelligence*, 1994.

Appendices

A The Z Notation

A.1 Z Definitions

Some further details are provided here of the extensively used operators, ran and dom. Additional laws explaining how these operators may be applied can be found in [27].

$$[X, Y]$$

$$\mathrm{dom} : (X \leftrightarrow Y) \to \mathbb{P}\, X$$
$$\mathrm{ran} : (X \leftrightarrow Y) \to \mathbb{P}\, Y$$

$$\forall R : X \leftrightarrow Y \bullet$$
$$\quad \mathrm{dom}\, R = \{x : X;\ y : Y \mid (x, y) \in R \bullet x\} \land$$
$$\quad \mathrm{ran}\, R = \{x : X;\ y : Y \mid (x, y) \in R \bullet y\}$$

A.2 Auxiliary Z Definitions

The following generic Z definitions are used in the specification contained in this document, and are included here for completeness.

First we define two functions, *map* and *mapset*, which take another function as an argument and apply it to every element in a list or set respectively,

$$[X, Y]$$

$$map : (X \to Y) \to (\mathrm{seq}\, X) \to (\mathrm{seq}\, Y)$$
$$mapset : (X \to Y) \to (\mathbb{P}\, X) \to (\mathbb{P}\, Y)$$

$$\forall f : X \to Y;\ x : X;\ xs, ys : \mathrm{seq}\, X \bullet$$
$$\quad map\ f\ \langle\rangle = \langle\rangle \land$$
$$\quad map\ f\ \langle x\rangle = \langle f\ x\rangle \land$$
$$\quad map\ f\ (xs \frown ys) = map\ f\ xs \frown map\ f\ ys$$
$$\forall f : X \to Y;\ xs : \mathbb{P}\, X \bullet$$
$$\quad mapset\ f\ xs = \{x : xs \bullet f\ x\}$$

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

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

$$[X]$$

$$defined\_, undefined\_ : \mathbb{P}(optional[X])$$
$$the : optional[X] \pfun X$$

$$\forall xs : optional[X] \bullet defined\ xs \Leftrightarrow \#\ xs\ =\ 1 \land$$
$$undefined\ xs \Leftrightarrow \#\ xs\ =\ 0$$
$$\forall xs : optional[X] \mid defined\ xs \bullet$$
$$the\ xs = (\mu\, x : X \mid x \in xs)$$

B Binding

B.1 Substitutions

Here, we provide further details of the standard definitions of binding and unification, described in Z.

The standard definition of a substitution is a mapping from variables to terms such that no variable contained in any of the terms is in the domain of the mapping [12]. This is represented as a partial function between variables and terms since, in general, only some variables will be mapped to a term. (Remember that *termvars* is a function that returns the variables of a term, as defined earlier, at the end of Section 4.1.)

$$Substitution == \{f : Var \pfun Term \mid (\mathrm{dom}\, f) \cap (\bigcup(mapset\ termvars\ (\mathrm{ran}\, f))) = \varnothing\}$$

B.2 Applying Substitutions

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

$$ASVar : Substitution \to Var \to Term$$

$$\forall \psi : Substitution;\ v : Var \bullet ASVar\ \psi\ v = (\{x : Var \bullet (x, var\ x)\} \oplus \psi)\ v$$

We can then define what it means for a substitution to be applied to a term.

$$ASTerm : Substitution \to Term \to Term$$

$$\forall t : Term;\ f : FunSym;\ ts : \mathrm{seq}\ Term;\ \psi : Substitution \mid t = functor\ (f, ts) \bullet$$
$$t \in \mathrm{ran}\, const \Rightarrow ASTerm\ \psi\ t = t \land$$
$$t \in \mathrm{ran}\, var \Rightarrow ASTerm\ \psi\ t = ASVar\ \psi\ (var^{\sim} t) \land$$
$$t \in \mathrm{ran}\, functor \Rightarrow ASTerm\ \psi\ t =$$
$$(\mu\, new : Term \mid first\ (functor^{\sim} new) = f \land$$
$$second\ (functor^{\sim} new) = map\ (ASTerm\ \psi)\ ts\ )$$

The function for applying a substitution to an atom is simply a matter of applying the binding to each term in the term list. Applying a function to an action, given by *ASAction*, is defined analogously.

$$ASAtom : Substitution \to Atom \to Atom$$

$$\forall a, b : Atom;\ s : Substitution$$
$$\bullet\ ASAtom\ s\ a = b \Leftrightarrow b.head = a.head \land$$
$$b.terms = map(ASTerm\ s)a.terms$$

$$ASAction : Substitution \rightarrow Action \rightarrow Action$$

$$\forall a, b : Action;\ s : Substitution \bullet ASAction\ s\ a = b \Leftrightarrow b.name = a.name \wedge b.terms = map(ASTerm\ s)a.terms$$

The functions for applying substitutions to beliefs, triggering events, formulae and plans can then be constructed similarly and are defined below.

$$ASBelief : Substitution \rightarrow Belief \rightarrow Belief$$

$$\begin{aligned}\forall b, c : Belief;\ l : Literal;\ a : Atom;\ s : Substitution \bullet\\ ASBelief\ s\ (literal\ (pos\ a)) = literal\ (pos\ (ASAtom\ s\ a)) \wedge\\ ASBelief\ s\ (literal\ (not\ a)) = literal\ (not\ (ASAtom\ s\ a)) \wedge\\ ASBelief\ s\ (and(b, c)) = and((ASBelief\ s\ b), (ASBelief\ s\ c))\end{aligned}$$

$$ASEvent : Substitution \rightarrow TriggerEvent \rightarrow TriggerEvent$$

$$\begin{aligned}\forall b : Belief;\ a : Atom;\ s : Substitution \bullet\\ ASEvent\ s\ (+, belief\ b) = (+, belief\ (ASBelief\ s\ b)) \wedge\\ ASEvent\ s\ (-, belief\ b) = (-, belief\ (ASBelief\ s\ b)) \wedge\\ ASEvent\ s\ (+, goal\ (achieve\ a)) = (+, goal\ (achieve\ (ASAtom\ s\ a))) \wedge\\ ASEvent\ s\ (-, goal\ (achieve\ a)) = (-, goal\ (achieve\ (ASAtom\ s\ a))) \wedge\\ ASEvent\ s\ (+, goal\ (query\ a)) = (+, goal\ (query\ (ASAtom\ s\ a))) \wedge\\ ASEvent\ s\ (-, goal\ (query\ a)) = (-, goal\ (query\ (ASAtom\ s\ a)))\end{aligned}$$

$$ASFormula : Substitution \rightarrow Formula \rightarrow Formula$$

$$\begin{aligned}\forall s : Substitution;\ act : Action;\ atom : Atom \bullet\\ ASFormula\ s\ (actionformula\ act) = actionformula\ (ASAction\ s\ act) \wedge\\ ASFormula\ s\ (goalformula\ (achieve\ atom)) =\\ \qquad goalformula\ (achieve\ (ASAtom\ s\ atom)) \wedge\\ ASFormula\ s\ (goalformula\ (query\ atom)) =\\ \qquad goalformula\ (query\ (ASAtom\ s\ atom))\end{aligned}$$

$$ASPlan : Substitution \rightarrow Plan \rightarrow Plan$$

$$\begin{aligned}\forall s : Substitution;\ old : Plan \bullet\\ ASPlan\ s\ old = (\mu\ new : Plan \mid\\ \quad new.inv = ASEvent\ s\ (old.inv) \wedge\\ \quad new.context = mapset\ (ASBelief\ s)\ (old.context) \wedge\\ \quad new.body = map\ (ASFormula\ s)\ (old.body))\end{aligned}$$

B.3 Composition of Substitutions

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

$$\_ \ddagger \_ : (Substitution \times Substitution) \rightarrow Substitution$$

$$\begin{aligned}
&\forall \tau, \sigma : Substitution \mid \\
&\quad (\mathrm{dom}\,\sigma) \cap ((\mathrm{dom}\,\tau) \cup \bigcup(mapset\ termvars\ (\mathrm{ran}\,\tau))) = \varnothing \bullet \\
&\quad\quad \tau \ddagger \sigma = (\tau \cup \{x : Var;\ t : Term \mid (x,t) \in \sigma \bullet (x, ASTerm\ \tau\ t)\})
\end{aligned}$$

B.4 Unification

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

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

$$\begin{aligned}
&\forall t_1, t_2 : Term;\ s : Substitution \bullet \\
&\quad unifiterms(s, (t_1, t_2)) \Leftrightarrow (ASTerm\ s\ t_1 = ASTerm\ s\ t_2)
\end{aligned}$$

A similar predicate can be defined for unifying two trigger events.

$$unifitrigevents\_ : \mathbb{P}(Substitution \times (TriggerEvent \times TriggerEvent))$$

$$\begin{aligned}
&\forall e_1, e_2 : TriggerEvent;\ s : Substitution \bullet \\
&\quad unifitrigevents(s, (e_1, e_2)) \Leftrightarrow (ASEvent\ s\ e_1 = ASEvent\ s\ e_2)
\end{aligned}$$

Other predicates for unifying expressions in AgentSpeak(L) can be defined similarly and are not given here.

A substitution is *more general* than another substitution if there exists a third substitution which, when composed with the first, gives the second.

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

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

The most general unifier of two expressions is a substitution that unifies the expressions such that there is no other unifier that is more general. These are defined as *partial* functions since any two expressions may not have a unifier at all. In the specification given in this paper, it is necessary to specify such a function for two events, a goal with a set of beliefs and a triggering event with a goal. Due to space constraints, only the definition of the first function is provided, while for the others we simply specify the signature since they are defined analogously.

$$mgutriggerevents : (TriggerEvent \times TriggerEvent) \nrightarrow Substitution$$

$$\begin{aligned}
&\forall t_1, t_2 : TriggerEvent;\ \sigma : Substitution \bullet mgutriggerevents\ (t_1, t_2) = \sigma \Leftrightarrow \\
&\quad (unifitrigevents(\sigma, (t_1, t_2)) \wedge \\
&\quad\quad \neg\, (\exists \omega : Substitution \bullet (unifitrigevents(\omega, (t_1, t_2)) \wedge (\omega\ mg\ \sigma))))
\end{aligned}$$

$$\begin{aligned}
&mguquery : (Goal \times \mathbb{P}\, Belief) \nrightarrow Substitution \\
&mgugoal : (TriggerEvent \times Goal) \nrightarrow Substitution
\end{aligned}$$

Received 8 August 1997