Architecture for Agent Programming Languages

Koen Hindriks[1] and **Mark d'Inverno**[2] and **Michael Luck**[3]

Abstract.

As the field of agent-based systems continues to expand rapidly, one of the most significant problems lies in being able to compare and evaluate the relative benefits and disadvantages of different systems. In part, this is due to the various different ways in which these systems are presented. One solution is to develop a set of architectural building blocks that can be used as a basis for further construction (to avoid re-inventing wheels), and to ensure a strong and effective, yet simple and accessible, means of presentation that allows for comparison and analysis of agent systems. In this paper, we address this issue in providing just such an architectural framework by using the 3APL agent programming language as a starting point for identification and specification of more general individual agent components. This provides three additional benefits: it moves the work further down the road of implementation, contributes to a growing library of agent techniques and features, and allows a detailed comparison of different agent-based systems specified in similar ways.

1 Introduction

Among the most significant of the problems that face the dynamic and rapidly-expanding field of agent-based systems is the drawing together of disparate strands of work ostensibly aimed at addressing the same issues. Indeed, the plethora of different agent theories, languages and architectures that have been proposed and developed in recent years highlights this particular problem. The "so what" reaction is one that is undeserved by many of the efforts that receive it, but is in part understandable. We argue that there are two key related reasons for this: first, it can be extremely difficult to compare and evaluate the relative benefits and disadvantages of different agent-based systems as a result of the different approaches taken to realise them; second, the focus on theories, architectures and languages has obscured a need to consider the fundamental building blocks with which they are built.

In attempting to avoid the pitfalls associated with this continual development of yet more agent-based languages and architectures with inadequate justification and relation to the broader field, we have been working on a more uniform perspective to enable a stronger inter-relation and comparison of different systems and approaches. For example, work on specifying dMARS [2] and AgentSpeak(L) [3] in a consistent fashion, and on comparing 3APL, AgentSpeak(L) and AGENT-0 [6] has attempted to address these concerns and, in so doing, has helped to clarify the agent-oriented approach and more general properties of agents. Equally, it has enabled a consideration of the needs for a set of building blocks for agent architectures by specifying schemas and actions for the updates of beliefs, goals (decision making), planning and reflection. In this paper we use the formal specification of 3APL as well as components from AgentSpeak(L) and dMARS specifications to compare and highlight the major distinctions between them.

The difficulty in comparing different agent-based approaches in part derives from the fact that there is no clear or formal presentation of the language or architecture in question. Even when systems are presented by means of formal semantics, the differences in the styles and formalisms used do not readily facilitate such comparison. In this paper, by contrast, we do not aim to introduce a new system or means for its description, but instead to use the particular case of the agent programming language 3APL [5] (pronounced "*triple-a-p-l*") and its architecture [6] to provide a way of understanding and specifying systems in a more general and more accessible way, and to provide a route to system development. This is achieved through the use of the standard well-known and commonly-used formal specification language, Z [8], which has also been used to specify several other agent properties, languages and architectures (eg. [2, 3, 4]). As a consequence, we get a uniform presentation of *both* the 3APL language *and* its architecture in a clear and concise way, which enables it to be more easily related to, and compared with, other systems. We believe that our work moves a step closer to a unified account of different agent languages and architectures. (Note that we aim for a unified *account* rather than unified languages or systems themselves.)

The contribution of this work is thus threefold. First, we present an outline operational specification of 3APL that can be used as the basis of a subsequent implementation, so that the transition from what might be called theory to practice is facilitated. Second, we allow an easy and simple comparison of 3APL and its *competitor* systems such as AgentSpeak(L) [7] and dMARS, as we demonstrate throughout the paper. The comparison considers the data structures required by each language, the requirements for defining an agent before and during run-time, and the basic operation of agents programmed in a given language. Third, we provide an accessible resource in the specification of techniques for the development of agent systems that might not otherwise be available in a form relevant both to agent architects and developers. In particular, we provide an intermediate level between an agent programming language and a complete architecture, and provide for a library of building blocks for constructing agent architectures.

The 3APL Programming Language 3APL is used to focus the comparison, by presenting its formal specification, and then showing points of divergence with other key systems. It supports the design and construction of intelligent agents for the development of complex systems through a set of intuitive concepts like beliefs, goals

[1] Department of Computer Science, Universiteit Utrecht, P.O. Box 80.089; 3508 TB Utrecht, The Netherlands, email: koenh@cs.uu.nl

[2] Cavendish School of Computer Science, University of Westminster, 115 New Cavendish Street, London W1M 8JS, UK, email: dinverm@wmin.ac.uk

[3] Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK, email: mikeluck@dcs.warwick.ac.uk

and plans, which can be used to describe and understand the computational system in a natural way. 3APL supports this style of programming by means of an expressive set of primitives to program agents, consisting of such sets of beliefs, goals and practical reasoning rules. Beliefs represent the issues the agent must deal with, while goals allow the agent both to focus on what it must achieve and to represent the way in which it can achieve it. In 3APL, goals are thus used to represent achievement goals and as *plans*. The practical reasoning rules provide the agent with planning capabilities to find an appropriate plan to achieve a goal, capabilities to create new goals, and capabilities to use the rules to revise a plan.

Originally, the operational semantics of 3APL was specified by means of Plotkin-style transition semantics [5]. Its re-specification, however, moves closer to a good implementation, because of the available Z tools for type-checking, animation, and so on. The resulting computational model includes data structures, operations and architecture, thereby isolating the data-types for an efficient implementation. Second, it provides an alternative perspective, highlighting different aspects of the language and architecture that are not manifested in a similar way in the transition style semantics.

In the specification that follows, we assume some familiarity with Z. Note that many details of 3APL are not included here due to space constraints, but a more complete specification of 3APL is available elsewhere [1] (as for the other systems compared [2, 3]).

2 Beliefs, Actions and Goals

Beliefs and goals are the basic components on which all of 3APL, dMARS and AgentSpeak(L) are based. In 3APL, beliefs are formulae from a first order language that is defined in the usual way, and first order terms are defined by means of given sets of first order variables and function symbols. Since the 3APL programming language distinguishes between first order variables and variables that range over goals, we can define a partition of the set of variables *Var*, and use *FOVar* to denote the set of first order variables and *Gvar* to denote the set of goal variables (such that $FOVar \cap GVar = \varnothing$, $FOVar \cup GVar = Var$). With the set of all function symbols denoted as $[FuncSym]$, a first order *term* is then either a first order variable or a function symbol with a (possibly empty) sequence of terms as a parameter.

$$FOTerm ::= var\langle\langle FOVar\rangle\rangle \mid functor\langle\langle FuncSym \times \operatorname{seq} FOTerm\rangle\rangle$$

2.1 Beliefs and Actions

Beliefs can now be defined by building types from the above primitives; a belief *atom* is a predicate symbol (the set of all such symbols denoted by $[PredSym]$) with a (possibly empty) sequence of terms as its argument. Beliefs are then either an atom, its negation, the conjunction or implication of two beliefs, true or false.

Atom

$$head : PredSym;\ terms : \operatorname{seq} FOTerm$$

$$Belief ::= pos\langle\langle Atom\rangle\rangle \mid not\langle\langle Atom\rangle\rangle \mid and\langle\langle Belief \times Belief\rangle\rangle \mid imply\langle\langle Belief \times Belief\rangle\rangle \mid \mathbf{false} \mid \mathbf{true}$$

The definition of terms is identical for 3APL, AgentSpeak(L) and dMARS. However, AgentSpeak(L) is the most limited in that it only allows the conjunction of beliefs, whilst dMARS allows for the disjunction in addition to the features of 3APL. It is a very simple matter to compare data structures in Z and to build a library of the possible different representations of beliefs that might be required when designing an agent language.

Agents accomplish tasks by performing actions, represented by action symbols and specified in the same way as atoms, as they are in AgentSpeak(L) and dMARS.

Action

$$name : ActionSym;\ terms : \operatorname{seq} FOTerm$$

2.2 Goals

In 3APL, goals are used to represent *both* the goals *and* the plans to achieve these goals of the agent. They are program-like structures that are built from basic constructs, such as actions, and regular imperative programming constructs, such as sequential composition and nondeterministic choice. Goals can be characterised as *goals-to-do*, mental attitudes corresponding to plans of action to achieve a state of affairs, or *goals-to-be*, corresponding to the state of affairs desired by an agent. For example, an agent may have adopted the *goal-to-do* of finishing a paper, and then sending it to the ECAI programme chair. This might be done in pursuit of the agent's *goal-to-be* of desiring the paper's acceptance.

Contexts Before formally describing goals, we introduce the notion of *contexts* (distinct from the notion of context that refers to plan preconditions in terms of beliefs in such systems as AgentSpeak(L)), which are goals with an extra feature called 'holes' that act as placeholders within the structure of goals. The role of contexts is to enable an elegant presentation of the architecture of 3APL, rather than in the 3APL language itself. More precisely, a *context* is either a basic action, a query goal, an achieve goal, the sequential composition of two contexts, the nondeterministic choice of two contexts, a goal variable, or "$\Box$", which represents a place within a context that might contain another context. In the definition below, we use the set of goal variables, *GVar*, to allow a process called *goal revision* to take place.

$$Context ::= bac\langle\langle Action\rangle\rangle \mid query\langle\langle Belief\rangle\rangle \mid achieve\langle\langle Atom\rangle\rangle \mid gvar\langle\langle GVar\rangle\rangle \mid comp\langle\langle Context \times Context\rangle\rangle \mid choice\langle\langle Context \times Context\rangle\rangle \mid \Box$$

The $\Box$, which denotes the placeholder or *hole* within a context, is distinct from a goal variable. Although both are placeholders, $\Box$ is a facility used for *specifying* 3APL, whereas goal variables are part of 3APL itself.

We can now define a goal as a context without any occurrences of $\Box$. The formal definition below uses an auxiliary function *squarecount* to count the occurrences of $\Box$ in a context.

$$Goal == \{g : Context \mid squarecount\ g = 0\}$$

The data structures used in both AgentSpeak(L) and dMARS are very different. In AgentSpeak(L) a goal is either an achieve atom, a query atom (as opposed to a query belief in 3APL) or an action.

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

In AgentSpeak(L), therefore, goals do not contain procedural knowledge as they do in 3APL by virtue of being the fundamental data structure essentially describing all possibilities for action, including an agent's *course* of action which, in AgentSpeak(L) and

dMARS, is distinct and represented in *plans*. In AgentSpeak(L), these plans comprise a *sequence* of basic actions, query goals and achieve goals (known as the *body* of the plan).

$$ASBody == \operatorname{seq} ASGoal$$

In dMARS, the body of a plan is a tree where the branches are goals (whether they be internal or external actions, query or achieve goals). For a plan to be successful, a path from the root to any leaf node must be found.

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

The data structure used clearly has ramifications for the operation of agents in the different systems. Both AgentSpeak(L) and dMARS require the use of *events* to trigger the placement of plan subgoals in a queue for further planning. The new plan generated for a subgoal is then added onto an existing stack of plans (or *intention*) for execution. However, in 3APL, events are unnecessary since goals are themselves modified in the process of planning and acting; instead of using intentions, 3APL simply attempts to execute its current goals.

2.3 Practical Reasoning Rules

The components described above come together in 3APL in *practical reasoning rules*, which are used both for traditional planning and for the less common *reflection* on goals. This latter aspect allows plans to be re-considered if they will fail with respect to the goal they are trying to achieve, if they have already failed, or it is possible to pursue a more optimal strategy. There are four kinds of such rules [6]: reactive rules to respond to the current situation and to create new goals; plan-rules to find plans to achieve goals; failure-rules to replan on failure; and optimisation-rules to replace less effective plans with more optimal ones.

$$PRType ::= reactive \mid failure \mid plan \mid optimisation$$

In more detail, 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. Informally, a rule with head g, body p and guard b, states that if an agent tries to achieve goal g and in a situation b, then it might consider *replacing* g by a plan p as a means to achieve it. If it is a *plan-rule*, g is of the form *achieve* s where s is a simple formula, and the rule states that plan p may offer a way to achieve g. If it is a *failure-rule* and g fails, it states that p may replace g. A *reactive-rule* has an empty head (and can be applied whenever the guard is true).

PRrule

$$head, body : opt[Goal];\ guard : Belief;\ type : PRType$$

$$head = \varnothing \Leftrightarrow type = reactive \land$$
$$the\ head \in (\operatorname{ran} achieve) \land body \neq \varnothing \Leftrightarrow type = plan$$

Guards serve two purposes: they specify situations in which rules might be considered, and they enable some parameters to be retrieved from the agent's beliefs.

This is quite similar to the definition of plans in AgentSpeak(L), which is defined by a triggering event (the addition or removal of a belief or goal), a set of pre-conditions and the body containing the procedural knowledge of the agent as a sequence of goals and actions (as described above). Thus both AgentSpeak(L) plans and 3APL practical reasoning rules contain a pre-condition (defined as a belief), a trigger and a body. However, the AgentSpeak(L) trigger is an *event*, whereas in 3APL it is an optional goal. Moreover, practical reasoning rules in 3APL can be used to deal with reactive behaviour, goal creation, plan failure and plan optimisation in addition to planning, partly through the inclusion of *goal variables* (as we see below).

3 Agents

In this section we show how agents are constructed based on these basic components. While focusing on 3APL, we contrast in particular with AgentSpeak(L); dMARS is similar to AgentSpeak(L). Indeed, from this point onwards, the divergence between systems is more pronounced, and dMARS is largely omitted from the discussion due to space constraints.

Agents and Mental State In 3APL, agents are characterised in terms of their beliefs, goals, practical reasoning rules and expertise; beliefs and goals are dynamically updated while rules and expertise are fixed and do not change. A 3APL agent can thus be defined as an entity with static *expertise* and *rulebase*, i.e. a set of practical reasoning rules.

3APLAgent

$$expertise : \mathbb{P}\, Action;\ rulebase : \mathbb{P}\, PRrule$$

This is equivalent to AgentSpeak(L) agents, which have a set of capabilities and a plan library. In 3APL, an agent's beliefs are recorded in its *beliefbase* and its goals in its *goalbase*, together making up the *mental state* of an agent that is updated during execution.

3APLAgentState

$$3APLAgent;\ belbase : \mathbb{P}\, Belief;\ gbase : \mathbb{P}\, Goal$$

In contrast, the state of AgentSpeak(L) agents includes beliefs, executing intentions, events to be processed, and actions to be performed.

ASAgentState

$$ASAgent;\ beliefs : \mathbb{P}\, Belief;\ intentions : \mathbb{P}\, Intention$$
$$events : \mathbb{P}\, Event;\ actions : Action$$

All of the systems we have mentioned in this paper have a similar initial state in that all state variables not part of the agent are set to some value (whether empty or defined by the user). In addition, only the mental state of agents may change during their operation, but not information contained in the Agent definition such as the expertise or the rulebase in 3APL.

Agent Operation Two semantic notions are used to parameterise the operation of a 3APL agent. First, the semantics of basic actions is defined by a global function, *execute*, which specifies that an action is an *update operator* on the beliefs of the agent. Since *execute* is a global function, any two agents capable of performing an action are guaranteed to do the same thing when executing it. This is

particularly important to prevent confusion when specifying and programming agents.

$$execute : Action \times \mathbb{P}\, Belief \pfun \mathbb{P}\, Belief$$

Second, a logical consequence relation *LCon*, determines the inferences an agent can derive from its beliefs, and is also global. It ensures that all agents draw conclusions from their beliefs in the same way, guaranteeing a "minimal amount of global consistency".

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

In relation to the semantic notions of beliefs and expertise, 3APL and AgentSpeak(L) are similar. The way in which a set of beliefs follows as a logical consequence of another set of beliefs is equivalent since in both systems they are not specified. In terms of performance of actions in the environment, however, AgentSpeak(L) does not specify any impact on the state of the agent. 3APL is more general and uses a global function definition that determines how an agent's beliefs change in response to performing an action (i.e. an update semantics on the agent's state is provided).

The particular constructs introduced so far, while derived from 3APL, can be viewed as a set of more general architectural building blocks that can be used to contribute to the linking and unifying of work on agent programs and architectures. The value of this lies in reducing the overhead of re-inventing the wheel, which occurs so frequently in both theory and practice. The next two sections specifically address the operation of 3APL.

4 The Application of Practical Reasoning Rules

Practical reasoning rules can be used to plan, revise, and create goals. The application of a rule r to a goal g results in the replacement of a subgoal g', which matches the head of rule r, by the body of rule r. If the body of the rule is empty, however, the subgoal is simply dropped. This yields a substitution that is applied to the entire resulting goal. When the head is empty, only the guard needs to be derivable from the beliefs of the agent, and a new goal (the body) is added to the goalbase of the agent.

In order to explain rule application, we introduce the notion of *front contexts*, which are contexts (see above) with precisely *one* occurrence of $\Box$ *at the front* of the context. Informally, an element *at the front* of a context means that an agent could choose to perform this element first, so that if a $\Box$ at the front of a context was replaced by a goal, then that goal *could be performed first*, before the remainder of the overall goal.

Essentially, the task of rule application is to find a front context fc such that if the subgoal g' is *inserted* for $\Box$ (at the front of fc), the resulting goal is identical to g. Applying r then amounts to *updating fc* with the body of the rule. There is a crucial difference here between inserting and updating. Inserting a goal in a front context simply means substituting the goal for the $\Box$ in the front context; while updating a front context with a goal means replacing $\Box$ with that goal *and also committing* to the choices made (pursuing a subgoal in a choice goal means committing to the branch in which the subgoal appears in the choice goal). To formalise this, we can define two similar functions to *insert* a goal into the square of a front context and to *update* a front context with a goal. we present the latter here as *UG* but, due to space constraints, we only give the signature of the former; details can be found in [1].

$$Insert : (Goal \times FrontContext) \to Goal$$

$$UG : (opt[Goal] \times FrontContext) \to opt[Goal]$$

$$\begin{array}{l}
\forall\, g : opt[Goal];\ fc : FrontContext;\ g' : Goal \bullet UG\,(g, \Box) = g \\
non - empty\ UG\,(g, fc) \Rightarrow \\
\quad (UG\,(g, comp(fc, g')) = \{comp(the\,(UG\,(g, fc)), g')\} \\
\quad \land\ UG\,(g, choice(fc, g')) = UG\,(g, fc)\ \land \\
\quad UG\,(g, choice(g', fc)) = UG\,(g, fc))\ \land \\
empty\ UG\,(g, fc) \Rightarrow \\
\quad (UG\,(g, comp(fc, g')) = \{g'\}\ \land \\
\quad UG\,(g, choice(fc, g')) = \varnothing\ \land \\
\quad UG\,(g, choice(g', fc)) = \varnothing)
\end{array}$$

Note that if the front context is of the form $choice\ (\Box, g)$ for some goal g, and a goal g' is inserted for $\Box$, *UG* yields $UG(g', choice\ (\Box, g)) = \{g'\}$. This reflects the fact that if an agent updates a choice goal, it requires the agent to commit to one of the two subgoals. The square is used to indicate which selection is made. In contrast, the *Insert* function simply substitutes a goal for a square, and we have $Insert(g', choice\ (\Box, g)) = \{choice(g', g)\}$.

A rule is *applicable* if the head unifies with a (sub)goal of the agent *and* the guard of the rule follows from the agent's beliefs. Formally, a rule, r, is applicable withe respect to a goal, g and set of beliefs, bb, if and only if:

$$\begin{array}{l}
(\exists\, \theta, \gamma : Substitution;\ subg : Goal;\ fc : FrontContext \bullet \\
Insert(subg, fc) = g\ \land\ mgu((the\ r.head), subg) = \theta\ \land \\
(\mathrm{dom}\,\gamma) \subseteq (beliefvars\ r.guard)\ \land \\
LCon(bb, \{SubsBelief(\theta \ddagger \gamma) r.guard\})))
\end{array}$$

The definition below uses the auxiliary functions, *beliefvars* which returns the set of first order variables contained in a belief, and *SubsBelief*, which applies a substitution to a belief. If the rule has no head, it is applicable simply if the guard follows from the belief-base.

Thus, if the head of the rule is not empty, applying the rule amounts to replacing a subgoal by the body of the rule. (If it is empty, the body is simply added to the goalbase but we do not specify that here). Care must be taken here to avoid interference of variables occurring in rules and those variables occurring in goals (cf. [5]). For this reason, all variables in the rule applied are renamed to variables not occurring in the target goal, using the function $RuleRename(r, V)$ not defined here, which renames the variables in rule r so that no variable from the set V occurs in the renamed rule. The auxiliary functions are *SubsGoal*, which is analogous to the function *SubsBelief* and *goalvars* which returns the set of first order variables in a goal.

ApplyRule

$$\begin{array}{l}
\Delta AgentState \\
g? : Goal;\ r? : PRrule;\ rr : PRrule \\
\hline
rr = RuleRename(r?, goalvars\ \{g?\}) \\
r?.type \neq reactive \Rightarrow \\
\quad (\exists\, fc : FrontContext;\ subg : Goal \bullet Insert(subg, fc) = g?\ \land \\
\quad\ (\exists\, \theta, \gamma : Substitution \mid (\mathrm{dom}\,\gamma) \subseteq (beliefvars\ rr.guard) \bullet \\
\quad\quad mgu(the\ rr.head, subg) = \theta\ \land \\
\quad\quad LCon(belbase, \{SubsBelief\ (\theta \ddagger \gamma) rr.guard\})\ \land \\
\quad\quad\ belbase' = belbase\ \land \\
\quad\quad\ gbase' = gbase \setminus \{g?\} \cup \\
\quad\quad\quad SubsGoal\ (\theta \ddagger \gamma)\ \{(Insert\ (the(rr.body), fc))\}))
\end{array}$$

5 The Execution of Goals

Execution of goals is specified through the computation steps an agent can perform on a goal, which correspond to the simple actions of either a basic action or a query on beliefs. Recall that the semantics of basic actions is given by a global function *execute* and the semantics of beliefs is specified by the *LCon* relation. Now, an agent is only allowed to execute a basic action or query that occurs at the front of a goal, i.e. it is one of the first things the agent should consider doing. The notion of front context is useful to find an action or query that the agent might execute. If there is a front context in which a basic action or query can be inserted for $\Box$, and which results in a goal of the agent, the agent might consider executing that basic action or query. After executing the goal, the goal needs to be updated, and this updating is the same as updating the front context by removing $\Box$.

The execution of a basic action amounts to changing the beliefbase of the agent in accordance with the function *execute*. The condition $(a, belbase) \in (\mathrm{dom}\, execute)$ expresses that the basic action a is enabled, and thus can be executed.

ExecuteBasicAction

$$\Delta AgentState$$
$$g? : Goal$$

$$(\exists\, fc : FrontContext;\ a : Action \mid a \in expertise \land Insert((bac\ a), fc) = g? \land (a, belbase) \in (\mathrm{dom}\, execute) \bullet belbase' = execute(a, belbase) \land gbase' = (gbase \setminus \{g?\}) \cup UG\ (\{\}, fc))$$

Queries are goals to check if some condition follows from the beliefbase of the agent. Any free variables in the condition of the query can be used to retrieve data from the beliefbase. The values retrieved are recorded in a substitution θ. A query can only be executed if it is a consequence of the beliefbase (otherwise, nothing happens).

ExecuteQueryGoal

$$\Delta AgentState$$
$$g? : Goal$$

$$(\exists\, fc : FrontContext;\ b : Belief \bullet Insert(query\ b, fc) = g? \land (\exists\, \theta : Substitution \bullet LCon\ (belbase, \{SubsBelief\ \theta\ b\}) \land belbase' = belbase \land gbase' = (gbase \setminus \{g?\}) \cup (SubsGoal\ \theta\ (UG\ (\{\}, fc)))))$$

Comparison In AgentSpeak(L) there are two aspects to its operation: 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 comprises selecting an intention, locating its topmost plan, and performing the plan's next component (either an action, a query goal or an achieve goal that posts a new event). If the intention finishes executing at this point (in the case of actions or queries), it can be removed. In 3APL, by contrast, the agent either applies rules, which requiress replacing the front context of a current goal with the body of the plan, or executing goals, in turn amounting to executing either a basic action or query goal at the front of a goal.

The control structure of AgentSpeak(L) thus involves the processing of all events by selecting the best plan, updating intentions as necessary, and then executing intentions. The control structure based on 3APL's classification of rules allows for much richer descriptions of agent operation. Indeed, many alternative control structures can be provided in this way for a 3APL agent. In essence, 3APL substantially simplifies AgentSpeak(L) whilst retaining all its functionality.

Lastly, we note that the concept of a *context*, used in the specification of 3APL to represent the operation of an agent, does not have a clear equivalent in the AgentSpeak(L) specification, in which execution and planning are specified simply by using several schemas and some simple functions to manipulate lists.

6 Conclusions

The criticism levelled against much recent agent research is that the particular contribution in relation to similar work and the broader field is unclear. This is due to an inability to identify the links between systems and compare them easily, and can lead to a profusion and proliferation of yet more agent theories, architectures and languages. The solution to this problem is to develop a set of architectural building blocks that can be used as a basis for further construction, and to ensure a strong and effective means of presentation that allows the differences and similarities to be easily identified and consequently compared and analysed.

By providing a Z specification of 3APL we are able to address exactly this difficulty in comparing it with other systems similarly specified in Z, such as AgentSpeak(L) and dMARS. By specifying both the language and the architecture in one unified framework, we are able to present a specification that reduces the complexity of the semantics of language and architecture considerably. One of the advantages is that we do not have to build two different systems, but only one. Our work illustrates that formal specification both enables the key building blocks for agent architectures to be identified, and allows for a comparison of the benefits and weaknesses of different agent frameworks, and of the expressive power of agent languages. Further work aims to perform exactly this function on the basis of the architectural framework presented here.

REFERENCES

[1] M. d'Inverno, K. Hindriks, and M. Luck, 'A formal architecture for the 3APL programming language', in *Proceedings of the first International Conference of B and Z Users*. Springer, (to appear 2000).

[2] 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, LNAI 1365*, pp. 155–176. Springer, (1998).

[3] M. d'Inverno and M. Luck, 'Engineering AgentSpeak(L): A formal computational model', *Journal of Logic and Computation*, **8**(3), 233–260, (1998).

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

[5] 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*, pp. 215–229. Springer, (1998).

[6] 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).

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

[8] J. M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall, Hemel Hempstead, 2nd edn., 1992.