

Design of a Concurrent Agent-Oriented Language

October, 1994

Technical Note ??

By:

Devindra Weerasooriya
Dept. of Computer Science
University of Melbourne
Parkville 3052, Australia
email: dev@cs.mu.oz.au

Anand S. Rao
Level 6, 171 La Trobe Street
Melbourne 3000, Australia
email: anand@aaii.oz.au

Kotagiri Ramamohanarao
Dept. of Computer Science
University of Melbourne
Parkville 3052, Australia
email: rao@cs.mu.oz.au

This research was supported by the Cooperative Research Centre for Intelligent Decision Systems under the Australian Government's Cooperative Research Centres Program. This paper also appears in the *Intelligent Agents: Theories, Architectures, and Languages. Lecture Notes in Artificial Intelligence LNAI 890*, ed. Wooldridge, M. and Jennings, N. R., Springer Verlag, Amsterdam, Netherlands, 1995.

Abstract

This paper describes the concurrent agent-oriented language AgentSpeak. AgentSpeak can model distributed autonomous agents, situated in dynamic environments, that are reactive as well as proactive towards the environment. Agents are organized into agent families offering certain services to other agents. Services are realized through the execution of an associated Plan. Each agent will also be associated with a Database. Some of the services, and a portion of the database could be public; i.e. available outside the agent. Therefore it follows that the remainder of the database, the remaining services and all of the plans will be private to the agent-family. The language supports and extends concurrent object-oriented language features such as synchronous and asynchronous messages and has well developed group communication primitives.

1 Introduction

Both the Distributed Artificial Intelligence (DAI) community and the Object-Based Concurrent Programming (OBCP) community have been investigating the concept of designing active, autonomous, concurrently running entities that solve problems cooperatively [7]. While the former is primarily concerned with representational and reasoning capabilities of such autonomous entities (called agents), the latter is primarily concerned with developing language constructs that facilitate the implementation of such entities. As a result of this dichotomy a number of DAI systems have been built with little or no common language constructs; conversely, OBCP languages are found wanting in sophisticated agent models and behavioral specifications that are inherent in DAI systems.

The notion of an agent has been extensively studied within the planning and DAI communities. An agent is defined as follows [17]: An agent is an entity, situated in a changing environment that continuously receives perceptual input and, based on its internal mental state exhibits reactive or proactive behavior that potentially changes the environment in which it is situated. The complexity of the internal mental state differentiates many of the agent-oriented systems.

As noted by Wooldridge and Jennings [23] a particular type of agents, called Belief-Desire-Intention (BDI) agents, has been studied extensively [16; 17; 18; 1; 22; 15]. Logical specifications of BDI agents using multi-modal logics with both linear-time and branching-time characterisations have been studied [16]. The Procedural Reasoning System (PRS) [8] is a system that is loosely based on the logical specifications of BDI agents. Formal verification of BDI agents based on model-checking techniques have also been investigated [18]. Furthermore, PRS has been applied to a number of important practical problems, such as air traffic control [12], spacecraft systems handling [11], telecommunications management [11], and air-combat modelling [14].

Research into the specification, design, and verification of agents, (in particular BDI agents) has matured to such an extent that one can now abstract the key concepts behind agents. The primary characteristics of agents are:

- the presence of a complex internal mental state that includes beliefs, desires (or goals), plans, and intentions;
- proactive or goal-directed behavior, in addition to reactive behavior;
- communication through structured messages or speech acts;
- ability to be distributed across a wide-area network;
- capable of acting and reacting to changes in the environment in real-time¹;
- concurrent execution of plans within an agent and between agents; and
- reflective or meta-level reasoning capabilities.

Although, one can build DAI systems based on an agent-oriented BDI architecture that embodies some or all of the above concepts there are many advantages in creating language constructs that capture these features. Some of these advantages are as follows:

¹Real-time here refers to real-time human decision-making, which is typically of the order of a few seconds to a couple of minutes

- It allows software engineers involved in language and compiler design to find efficient implementations of language constructs without getting distracted by foundational and conceptual issues related to representation and reasoning – the primary focus of DAI researchers.
- It will allow programmers to program in a higher-level language – in terms of the goals to be achieved and the intentions that an agent is committed to – without needing to understand any particular DAI system or the foundational issues related to agents.
- It is likely to facilitate more widespread use of agent-oriented concepts, as more conventional imperative, rule-based and object-oriented languages get extended with agent-oriented language constructs.

Researchers in object-oriented programming have been extending the original notion of objects by incorporating one or more of the features that we have associated with agents. As a result, one has a proliferation of various extensions to objects that make them active, concurrent, distributed, reflective, persistent, and real-time. However, there is no single object-oriented language that encapsulates all the above mentioned features². The primary aim of our work is to abstract the essential aspects of agents and design language constructs for such abstractions. These language constructs will embody the essential aspects of agents (as well as various extensions to objects currently being attempted) within a unified framework. It will have a solid semantic and theoretical basis in the logical formalization of agents. This paper presents an initial attempt in this direction by describing the design of a concurrent agent-oriented language called AgentSpeak. AgentSpeak has sophisticated agent modeling capabilities as found in PRS and appropriate language constructs, influenced by work in OBCP [9; 10; 3]. ;

2 AgentSpeak: Language Constructs

2.1 Agent Family

An *agent family* is analogous to a class in object-oriented languages. An instance of an agent family is called an *agent*. An agent family contains a *public* area and a *private* area. A part of the database of relations and the services offered by the agent family are public and can be accessed by other agents. The plans that are the means of providing service and the remainder of the database are treated as private to the agent family. In addition, the agent may perform certain services that are private, i.e., satisfying its own desires.

```
agent-family <agent-family-name> '{'
  public:
    database '{<relation-name>{;<relation-name>}}}'
    services '{<service-name>{;<service-name>}}}'
  private:
    database '{<relation-name>{;<relation-name>}}}'
    services '{<service-name>{;<service-name>}}}'
    plans '{<plan-name>{;<plan-name>}}}'
'}
```

²If such a language were created then whether it is called an agent-oriented language or an object-oriented language is of no significance.

Since the plans are not public, other agents in the environment can only request that certain services (i.e., end) be performed, but cannot specify the plans (i.e., means) by which these services are performed. This property is called the *means-end transparency* of agents.

Agent instances, also referred to as agents, are instances of the agent family, i.e., are generated from the agent-family template. A generic function called `create-agent-instance` creates an instance of an agent family. The agent is initialized with a database of relation instances. It is evident that the agent will inherit public/privacy classifications of the family to which it belongs.

```
<agent-instance> := create-agent-instance(<agent-family-name>,
                                         <relation-instance>,<relation-instance>,...)
```

On the creation of an agent instance a unique system generated handle will be created to access the agent instance anywhere in the network. This unique agent handle will have the following form:

```
<agent-qual>.<agent-family-name>.<machine-host>.<creation-time>
```

This implies that no two agents of the same family, created at the same location will have the same name. The qualifier is a tag that distinguishes the handle as one relating to agents. Note that this does not in any way imply that the agents must remain forever in the machine where they were created. All subsequent migration, general-addressing and message passing will be handled by name-server-agents or binder-agents. A similar feature of accessing named processes across anywhere in the network is available in Agent PProcess Interaction Language (APRIL) [13].

2.2 Database Relations

A database definition consists of a set of relations. A relation consists of a relation name followed by a list of fields and their types. For the purposes of this paper we take these types to be the standard types (i.e., integer, real, char, and enumerated type). User defined types will not be considered. Also, if there are not too many relations they can be defined as part of the agent-family definition.

```
relation <relation-name> (<type> <field> {;<type> <field>})
```

A relation instance is a particular instantiation of the above relation. A relation instance consists of the relation name and values of the respective fields. The generic function `create-relation-instance` is used to create a relation instance.

```
<relation-instance> ::=
    create-relation-instance(<relation-name> ::
                            field:<field-value>{, field:<field-value>})
```

The `create-relation-instance` function will associate with each relation instance a unique handle, that will serve to identify each relation instance within the agent-space which is generically distributed. The handle will be system generated with the following format.

```
<relation-qual>.<relation-name>.<machine-host>.<creation-time>
```

The latter could also serve as an index of the particular tuple.

Field level comparisons will be permissible within the system; the notation `<relation-name>.<field>` will be used to denote the value of a particular field.

Relation instances correspond to the beliefs of the Procedural Reasoning System and belief formulas of the formal logic [17]. However, unlike the logical system AgentSpeak does not allow nesting of beliefs.

2.3 Services

An agent is deemed to exist for the purpose of accomplishing its own desires and offer certain service to other agents. Services belonging to the former category are private and those belonging to the latter category are public. A service consists of a service name and a list of service statements. Each service statement is one of three types:

- a service to achieve a certain relational tuple;
- a service to query the existence of particular relational instances;
- a message telling the agent of a particular relational instance;

Services can be defined either globally or within the scope of an agent-family. The scope of the definition suggesting whether the latter is public or private.

```
<service-statement> ::= service <service-name>
                        (<service-type><relation-instance>)
```

```
<service-type> ::= achieve | query | told
```

A service could also be pre-instantiated in expectation of a deferred invocation using the `create-service-instance` generic function.

```
<service-instance> := create-service-instance(<service-name>,
                                             <service-type>, <relation-instance>)
```

As in the case of agents and relations this creation will result in the generation of a unique handle having the following format.

```
<service-qual>.<service-name>.<machine-host>.<creation-time>
```

Private service instances correspond to internal goals of an agent in PRS and the goal formulas of the formal logic [17]. Public service instances correspond to messages from other agents requesting the testing or achievement of certain formulas in PRS. Unlike the formal logic, AgentSpeak does not allow arbitrary nestings of belief and goal operators.

2.4 Plans

Plans are the means of performing services. A plan is identified by its plan name. It specifies the service name and the abstract situation in which a plan might be applicable. If the plan is applicable the goal statements are performed. After the successful performance of all the goal statements the situation that is to be asserted is specified abstractly.

```
plan <plan-name> '{'
  invoke on <service-statement>
  with context <abs-situation>
  perform
    <goal-statement>{; <goal-statement>}
  finally assert <abs-situation>{'}
```

An abstract situation is a conjunction or disjunction of relations. A goal statement can be any of the following types:

- an assignment statement;
- a while statement;
- an if-then-else statement;
- a non-deterministic or statement;
- a service statement; or
- a speech-act statement.

An abstract situation, denoted as an **abs-situation** is nothing but either a conjunction or a disjunction of relational instances.

```

<abs-situation> ::= <relation-instance>{ or <abs-situation>}
                  { and <abs-situation>}

<goal-statement> ::= <assign-stat> | <while-stat> |
                    <if-then-else-stat> |
                    <non-deterministic-or-stat> |
                    <service-statement> |
                    <speech-act-stat>
<assign-stat>    ::= <variable> := <value>
<block>          ::= <goal-statement> {; <goal-statement>}
<while-stat>     ::= while <abs-situation> do <block>
<if-then-else-stat> ::= if <abs-situation>
                        then <block>
                        else <block>
<non-deterministic-or-stat> ::= non-deterministically do '{'
                                <block> {or <block>}'

```

The semantics of assignment statements, while statements, if-then-else statements and non-deterministic or statements are taken to be understood. Now we elaborate on the speech act statements.

2.5 Speech Acts

Agents communicate with each other by sending messages. By default, message passing in AgentSpeak is asynchronous. In other words, an agent can send a message whenever it likes; irrespective of the state of the receiving agent. The messages are placed in the receivers mail-box. Synchronous and mixed mode messages are also supported.

In addition to the agent-to-agent message passing as described above, the language also supports agent-to-agent-family message passing [2]. The latter will allow agents to request services on an agent-family basis, without having to specify a particular agent. In this case messages are routed through **message-spaces**, each one of which is linked to a particular agent-family. Message spaces will be scanned by the respective agent processes.

Although communication through message-spaces would be the preferred mode of implementation; either broadcasting or unicasting can also be implemented. In the case of broadcasting it will be necessary for a receive function in the receiving-agent to filter the messages that are due to them. Given the need for an agent to possess a comprehensive list

of network addresses, unicasting would not be a preferred solution on account of the absence of location transparency.

We consider three different categories of speech act statements: inform, request while waiting for a reply, and request without waiting for a reply.

```
<speech-act-stat> ::= (<inform-act-stat> |  
                      <request-with-wait-stat> |  
                      <request-act-stat>)
```

We consider each one of these categories in detail below. Although, we have provided only a small set of basic speech acts, more complex speech acts (similar to that defined by KQML [5]) can be easily defined using this basic set.

2.5.1 Inform

Suppose an agent is sending a message with a certain priority to another agent. If the message carries informational content, but no obligation on the part of the receiving agent to do anything we have an *inform* speech act. This type of speech act captures asynchronous communication between agents.

In the case of agent-to-agent-family communication the message is sent to an agent family. If the message is to be received by all agents that are instances of the agent family we use the *inform-all-in-family* speech-act and if the message is to be received by any one agent in the agent family we use the *inform-one-in-family* speech-act.

More formally, the syntax of the inform speech acts are as follows:

```
<inform-act-stat> ::=  
  inform(<agent-instance>, <service-instance>, <priority>) |  
  inform-all-in-family(<agent-family-name>,  
                       <service-instance>, <priority>) |  
  inform-one-in-family(<agent-family-name>,  
                      <service-instance>, <priority>)
```

In the above definition *priority* is taken to be an integer number. In the case of an inform speech act the only service type allowed is *told* as it is the only service-type that does not oblige the receiving agent to act on the information.

2.5.2 Request with Wait

Consider an agent sending a request with a certain priority to another agent and waiting for a reply to be placed in a receiving template. While the sending agent is waiting it is not performing any other activity and thus models synchronous communication between agents.

In the case of agent-to-agent-family communication the request is sent to an agent family. Depending on whether the request is for all agents in the family, or any one agent in the family we have two different speech acts.

More formally, the syntax of the request speech acts with the sending agent waiting for a reply are as follows:

```
<request-with-wait-stat> ::=  
  request-with-wait(<agent-inst>, <service-inst>,  
                  <priority>, <receiver-relation-inst>) |  
  request-with-wait-all-in-family(  
    <agent-family-name>, <service-inst>, <priority>,
```

```

    <receiver-relation-inst>) |
request-with-wait-one-in-family(
    <agent-family-name>, <service-inst>, <priority>,
    <receiver-relation-inst>)

```

The service types allowed in the above cases are **achieve** and **query**. The `<receiver-relation-inst>` is a `<relation-inst>` that is sent to the receiver.

2.5.3 Request with No Wait

Suppose that an agent sends a request to another agent and the reply to the request is not needed by the sending agent immediately. In this situation it will be unnecessary for the sending agent to wait for a reply. In turn when the receiving agent has processed the request it will send a reply that will be placed in a specific template provided by the sending agent. When the sending agent needs the reply it can query the database for the template and get the result. With this mode of communication the receiving template will have to be instantiated with the message, as the sender will need to know this at the time of reply processing.

In the case of agent-to-agent-family communication the request is sent to an agent family. Once again, depending on whether the request is for all agents in the family or any one agent in the family we have two different speech acts:

More formally, the syntax of the request speech acts with the sending agent not waiting for a reply are as follows:

```

<request-act-stat> ::=
request(<agent-inst>, <service-inst>,
    <priority>, <receiver-relation-inst>) |
request-all-in-family(<agent-family-name>, <service-inst>,
    <priority>, <receiver-relation-inst>) |
request-one-in-family(<agent-family-name>, <service-inst>,
    <priority>, <receiver-relation-inst>)

```

The service types allowed in the above cases are once again **achieve** and **query**. The `<receiver-relation-inst>` is a `<relation-inst>` that is sent to the receiver.

2.6 Messages

When a speech act is sent by an agent to another agent or agent family it is given a unique message identifier and the identity of the agent which sent the message. The message syntax is as follows:

```

<message> ::= message(<message-id>, <sending-agent-instance>,
    <speech-act-stat>)

```

This message is then received by the agent or agent family at the other end where the message is decoded. All speech acts are processed by the receiving agent and appropriate actions are taken.

3 Operational Semantics

AgentSpeak requires that all agent families be defined at compile time. The main program creates instances of agent families by instantiating the public and private databases. Optionally, it can instantiate the agent with a public service instance which is executed as soon as the agent is created. At creation time each agent is associated with a private *mail-box*.

An agent at any point in time can be in any of the following three states: *active* – when it is executing a plan instance; *idle* – when there is no plan instance; or *waiting* – when it is waiting for a message from the external environment or waiting for a certain relation instance to become true.

When a speech act is received by an agent it is placed in the mail box. If the agent is in an *idle* state the agent becomes *active* and responds to the speech act by first selecting plans whose invocation service statement matches the service instance of the speech act.

Such plans are called *relevant plans*. The abstract situation of the context of such relevant plans is then matched with the current database relations of the agent. All relevant plans which have such a match are called *applicable plans*. The invocation bindings and the context bindings obtained during the process of finding relevant plans, and applicable plans, respectively, are used to create *plan instances*. By default ³, the system will select one of these plan instances and start performing the goal statements. Such a selected plan instance is called an *intention*. At any particular instance, there can be many intentions active. Each intention is an independent thread in itself. Thus the agent, as a whole, is multi-threaded.

Unlike object-oriented systems the plan of an agent need not be performed sequentially from the first goal statement to the last goal statement. Any service statement in the performance of the plan results in a service instance which is sent to the agent's mail box. At this stage the agent can process both external service instances or internal service instances. If there is no external service instance of a priority higher than the internal service instance, the internal service instance is handled by selecting the applicable plans for that service instance. This process goes on till all the goal statements of the original plan are performed or the plan fails at some stage.

When an agent performs any of the speech acts which requires waiting for a reply from another agent the plan instance and all its parents are suspended resulting in the agent moving to the *waiting* state. If there are other speech acts to be performed in the agent's mail-box, the agent becomes active and a speech act with the highest priority is chosen for processing and is processed. When the reply for a waiting plan instance is received the currently executing plan instance is suspended and the reply is processed. This allows the agent to process the highest priority service and at the same time not remain idle while it is waiting for a reply. The semantics of processing speech acts when an agent is in each of the three states is given in Table 1.

4 Example

Having described the syntax and the operational semantics of AgentSpeak, we consider an example in a Computer Integrated Manufacturing (CIM) application [4]. An extract of the AgentSpeak-code for the CIM example is presented in this section.

A small part of the overall production process is to make bolts. A robot picks up a rivet from a stock of rivets and holds it in the lathe. The lathe produces a thread on the rivet to convert it into a bolt. The bolt is then placed into a box of finished bolts.

³One mechanism to override this default is for the user to write meta-level plans for selecting the plan instances.

<i>The starting state of R</i>	<i>Highest priority (P1) speech act (S1) in mail box</i>
Idle	Process speech act S1.
Active with speech act S2 with priority P2	if $P1 \leq P2$ then continue with speech act S2.
Active with speech act S2 with priority P2	if $P1 > P2$ then suspend S1 and activate S2.
Waiting	Process speech act S1 pending reply.

Table 1: Processing Speech Acts in the Mail Box

The robot making the bolts from rivets, called the **bolt-robot**, may be asked by the **door-robot** (frame to door fixer) to deliver the bolts. When this happens the bolt-robot suspends making bolts and delivers the stock of finished bolts to the door-robot. While the bolt-robot R2B is delivering the bolts, the door-robot is fixing the power window unit on the door frame.

Modeling this scenario requires a balance between reactive and proactive behavior. It also requires asynchronous message passing and synchronization of actions. The robots can be modeled as autonomous agents running the language AgentSpeak.

One can define three agent families; **bolt-robots** for making bolts, **lathes** for threading the rivets, and **door-robots** for making doors from the frames.

First consider the description of the **bolt-robots** agent family. The **position** of the robots, the **rivet-box** and **bolt-box** are public. These robots offer two services to the other agents, namely **make-bolts** and **deliver-bolts**. The private database consists of other relations, such as **holding** and **power-status**. The plans that are required to provide the services include **rivets-to-bolts**, **manhattan-move**, **move-straight** etc.

```
agent-family bolt-robots {
  public:
    database {position; rivet-box; bolt-box}
    services {make-bolts; deliver-bolts}
  private:
    database {holding; power-status; self-status}
    services {move; grasp-rivet; choose-lathe; mount-rivet;
              ungrasp-bolt}
    plans {rivets-to-bolts; manhattan-move; move-straight; pick;
           drop; lower-arm; grasp; raise-arm; deliver}
}
```

The **position** of a robot has two fields – its x-coordinate position, **X-pos** and its y-coordinate position, **Y-pos**. The **rivet-box** consists of the **Quantity** field which contains the number of rivets in the box, the **X-pos** and **Y-pos** fields denoting the rivet box’s position. The definition of **bolt-box** is similar. The relation **holding** consists of two fields, a boolean **Value** which is **true** or **false** and the **Object** being held. The relation **power-status** consists of the field **Secs-left** that indicates the number of seconds of power left. We use the standard convention that variables start with an upper-case letter and constants start with a lower-case letter.

The services **make-bolts** and **deliver-bolts** achieve a state of the environment where the relation **bolt-box** has changed. The service **make-bolts** changes the quantity of **bolt-box**. The service **deliver-bolts** changes the position of **bolt-box**.

```

relation position (int X-pos; int Y-pos)
relation rivet-box (int Quantity; int X-pos; int Y-pos)
relation bolt-box (int Quantity; int X-pos; int Y-pos)
relation holding (boolean Value; [bolt,rivet] Object)
relation made-thread (boolean Value; identity Lathe-identity)
relation agent-status (identity Agent-instance;
                      [free, busy] Status)
relation power-status (int Secs-left)

service make-bolts (achieve bolt-box)
service deliver-bolts (achieve bolt-box)

```

Consider a plan `rivets-to-bolts` that achieves the goal of making a bolt from a rivet. It is invoked when there is a service request to `make-bolts` by achieving a state of the environment where the quantity of the bolts in the `bolt-box` is increased. Even though the plan is written with the relation name `bolt-box` and its fields at the time of invocation when a plan instance is created the values provided in the service instance will be substituted for these fields.

The context consists of a conjunction of relational instances to make sure that the robot is not holding either a bolt or a rivet. Other constraints, such as, there is at least one rivet in the rivet-box and there is sufficient power for the robot, are checked before the goal statements are to be performed.

The outer while loop ensures that the robot is making bolts until the rivet box is empty. The next statement requires the robot to move by achieving a state where its position is the same as the position of the rivet-box. The move is a private service of the agent and is not available to other agents. There may be many plans for moving and the successful performance of any of one of these plans would be sufficient for the robot to execute the next step of grasping a rivet.

Next the bolt robot sends a message to all agent instances of the lathe agent family, asking for their status. It waits for their replies and then chooses a particular lathe agent that is free. It then requests the position of that lathe agent and moves to that lathe agent. The rivet is mounted and the lathe agent is requested to make a thread on the rivet. Once the job is done the robot moves to where the bolt box is located and then drops it in the bolt box. The loop continues with the robot moving to the rivet box. This will continue until there are no more rivets. The plan for making bolts called `rivet-to-bolts` is given below.

```

plan rivets-to-bolts {
  invoke on make-bolts(achieve,
                      bolt-box(Quantity, X-pos, Y-pos))
  with context
    ((rivet-box.Quantity > 0) and
     (power-status.Secs-left > 300) and
     holding(false, bolt) and
     holding(false, rivet))
  perform
    while (rivet-box.quantity > 0) do {
      move(achieve,
          position(rivet-box.X-pos, rivet-box.Y-pos));
      grasp-rivet(achieve, holding(true, rivet));
    }
}

```

```

status-request :=
    create-service-instance(get-lathe-status,
                            query, self-status(Status));
request-with-wait-all-in-family(lathe, status-request,
                                1, agent-status(Agent, Status));
choose-lathe(query, agent-status(Lathe-inst, free));
position-request :=
    create-service-instance(get-position,
                            query, position(X-pos, Y-pos));
request-with-wait(Lathe-inst, position-request,
                  1, position(Lathe-x-pos, Lathe-y-pos));
move(achieve, position(Lathe-x-pos, Lathe-y-pos));
mount-rivet(achieve, mounted(rivet, Lathe-inst));
thread-make-request :=
    create-service-instance(make-thread,
                            achieve, made-thread(true, Lathe-inst));
request-with-wait(Lathe-inst, thread-make-request,
                  2, made-thread);
move(achieve, position(bolt-box.x-pos, bolt-box.y-pos));
ungrasp-bolt(achieve, holding(false, bolt)}
}
}

```

In the above example the relations `self-status`, `position`, `made-thread` as well as the services `give-lathe-status`, `give-lathe-position` and `make-thread` are public for the lathe-agent. In all cases of message passing note that the agent and service instances will need to instantiate with their respective handles. The mechanics of the latter have not been explored in this paper.

While a bolt robot is performing the above plan, it is possible for the packaging robot to send a higher priority message to deliver bolts. If this happens, by default, the bolt robot will suspend its current intention (i.e., plan instance) of making bolts and instead adopt a plan to deliver bolts. Once this is done it will resume the plan of making bolts. This default can be overridden by the user writing a meta-level plan. In the above case, such a meta-level plan may allow the bolt robot to complete the current iteration of the loop (i.e., finish the bolt which it has started) and then attend to the request.

5 Comparisons and Conclusions

As discussed earlier AgentSpeak abstracts some of the useful concepts of modelling agents that have been used to build multi-agent systems in DAI. In addition, it has language constructs that enable a programmer to program in higher-levels of abstractions than normally found in conventional procedural languages and more recent concurrent object-oriented languages.

The paper draws its inspiration from several sources within the OBCP and DAI areas. The agent structure has been substantially influenced by the object structure of C++ [20] and the message passing mechanics have derived much from those of ABCL [24]. However, the notion of services and plans that can achieve these services (which make AgentSpeak pro-active in its behaviour) is absent in both C++ and ABCL; and the primitives of group

communication are absent in ABCL. In fact, The group communication mechanisms were influenced by the tuple-spaces of Linda [2]. However, constructs that enable a proactive modeling of a problem within AgentSpeak, are not found in Linda.

The operational semantics of AgentSpeak is similar to the agent-oriented systems, Procedural Reasoning System (PRS) [8] and its more recent cousin the Distributed Multi-Agent Reasoning System (dMARS). However, unlike AgentSpeak, both of these are agent-oriented architectures and provide a graphical language for writing plans.

Agent-oriented languages such as AGENT0 [19] and PLACA (PLAnning Communicating Agents) [21] are similar in spirit to AgentSpeak. They differ in the definition of the mental state – AgentSpeak considers a mental state to consist of beliefs (or relations), goals (or services), plans, and intentions; AGENT0 considers a mental state to be a list of capabilities, beliefs, and intentions; PLACA has a list of plans, in addition to the components of AGENT0. AGENT0 and PLACA view agent-oriented programming (AOP) as a specialization of object-oriented programming (OOP). However, we view AOP as being an enhancement of OOP. As a result, we have notions such as agent families and agent instances, which are enhancements of the notions of classes and objects.

Our aim of designing an agent-oriented language that is high-level, easy to use, and can be formally specified, executed, and verified, is similar to the goals behind the design of Concurrent Metatem [6]. Both AgentSpeak and Concurrent Metatem have powerful communication primitives; the mental state of AgentSpeak is more complex than that of Concurrent Metatem. Agent behaviours in Concurrent Metatem are specified as temporal logic specifications that are directly executed. Plans in AgentSpeak are based on the plans of PRS that can be translated into a variant of dynamic logic specifications [17; 15]. Hence, the execution of AgentSpeak plans can be viewed as execution of dynamic logic specifications. However, we do not envisage using a theorem prover to execute the plans. The formal relation between an agent program written in AgentSpeak and the execution of corresponding dynamic logic expressions can be shown using techniques similar to that adopted elsewhere [15].

AgentSpeak is still in its design phase. Although, we have provided a foundation and a direction for the language, there are several issues that have to be resolved prior to an implementation. The inheritance mechanism for agents, explicit mechanics of handling multiple-intentions, plan abandonment or interruption, security and integrity of belief-sets within a concurrent execution environment, meta-level programming, reflective capability and mechanics of multi-agent actions being probably the more important ones among them.

The language constructs of AgentSpeak can be implemented as extensions of object-oriented languages, such as C++, or more conventional languages, such as Lisp and Prolog. An interesting possibility is to implement AgentSpeak using the process interaction capabilities of April [13].

Experience has shown that building distributed real-time applications where significant human decision-making is involved, using an agent-oriented architecture, significantly reduces development time [12; 11; 14]. Furthermore, modifications to the system can be made at a fraction of the cost and time compared to building the applications using conventional languages. The success of AgentSpeak will be determined to a large extent on how it can achieve similar reductions in development and maintenance costs, but allow programmers with little or no AI knowledge to achieve such results.

Acknowledgments

The authors would like to thank Mike Georgeff for valuable comments on earlier drafts of this paper and the the anonymous reviewers for their useful suggestions. This research was

supported by the Cooperative Research Center for Intelligent Decision Systems under the Australian Government's Cooperative Research Centers Program. It has also been supported by the Australian Research Council and the Department of Computer Science at the University of Melbourne.

References

- [1] M. E. Bratman, D. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349–355, 1988.
- [2] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4), November 1989.
- [3] S. A. Dobson. *An approach to Scalable Parallel Programming*. PhD thesis, Dept. of Computer Science, University of York, 1993.
- [4] E. Dubois, P. Du Bois, and M. Petit. O-o requirements analysis: An agent perspective. In *Lecture notes in Computer Science - 707*, pages 458–481, 1993.
- [5] T. Finin, J. Weber, G. Wiederhold, M. Genesereth, R. Fritzson, D. McKay, J. McGuire, R. Pelavin, S. Shapiro, and C. Beck. Specification of the kqml agent-communication language: Draft. Technical report, The DARPA Knowledge Sharing Initiative, External Interfaces Working Group, Baltimore, USA, 1993.
- [6] Michael Fisher. Representing and executing agent-based systems. In *Pre-proceedings of the workshop on Agent Theories, Architectures and Languages. Also appears as Lecture Notes in Computer Science (this Volume)*, Amsterdam, Netherlands, 1994. Springer Verlag.
- [7] L. Gasser and J. P. Briot. Object-based concurrent programming and distributed artificial intelligence. *Distributed Artificial Intelligence Theory and Practice*, 1992.
- [8] M. P. Georgeff and A. L. Lansky. Procedural knowledge. In *Proceedings of the IEEE Special Issue on Knowledge Representation*, volume 74, pages 1383–1398, 1986.
- [9] A. Goscinski. *Distributed Operating Systems - The Logical Design*. Addison Wesley, 1991.
- [10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.
- [11] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6), 1992.
- [12] N. Karppinen, A. Lucas, M. Ljungberg, and P. Repusseau. Artificial Intelligence in Air Traffic Flow Management. Technical Report 16, Australian Artificial Intelligence Institute, Carlton, Australia, 1991.
- [13] F. G. McCabe and Keith L. Clark. April – agent process interaction language. In *Pre-proceedings of the workshop on Agent Theories, Architectures and Languages. Also appears as Lecture Notes in Computer Science (this Volume)*, Amsterdam, Netherlands, 1994. Springer Verlag.

- [14] A. Rao, D. Morley, M. Selvestrel, and G. Murray. Representation, selection, and execution of team tactics in air combat modelling. In *Proceedings of the Australian Joint Conference on Artificial Intelligence, AI'92*, 1992.
- [15] A. S. Rao. Means-end plan recognition: Towards a theory of reactive recognition. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning (KRR-94)*, Bonn, Germany, 1994.
- [16] A. S. Rao and M. P. Georgeff. Asymmetry thesis and side-effect problems in linear time and branching time intention logics. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, Sydney, Australia, 1991.
- [17] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In C. Rich, W. Swartout, and B. Nebel, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.
- [18] A. S. Rao and M. P. Georgeff. A model-theoretic approach to the verification of situated reasoning systems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, Chamberey, France, 1993.
- [19] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.
- [20] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 1993.
- [21] S. R. Thomas. The placa agent programming language. In *Pre-proceedings of the workshop on Agent Theories, Architectures and Languages. Also appears as Lecture Notes in Computer Science (this Volume)*, Amsterdam, Netherlands, 1994. Springer Verlag.
- [22] M. Wooldridge. This is myworld: The logic of an agent-oriented testbed for dai. In *Pre-proceedings of the workshop on Agent Theories, Architectures and Languages. Also appears as Lecture Notes in Computer Science (this Volume)*, Amsterdam, Netherlands, 1994. Springer Verlag.
- [23] M. Wooldridge and N. R. Jennings. Agent theories, architectures, and languages: A survey. In *Pre-proceedings of the workshop on Agent Theories, Architectures and Languages. Also appears as Lecture Notes in Computer Science (this Volume)*, Amsterdam, Netherlands, 1994. Springer Verlag.
- [24] A. Yonezawa and M. Tokoro. Modelling and programming in an object-oriented concurrent language abcl/1. In *Object-oriented Concurrent Programming*, pages 55–89. The MIT press, 1987.