Agent-orientated Modelling for Complex Systems

Mariusz Nowostawski
Martin Purvis
Stephen Cranefield
[mnowostawski,mpurvis,cranefield]@infoscience.otago.ac.nz

July 15, 2000

Abstract

Agent-oriented modelling and the development of agent-based software systems are topics of intense interest nowadays, with agent-based development frameworks now becoming available. Most of the existing agent frameworks are grounded in procedural and object-oriented environments and essentially offer limited extensions to those paradigms. This paper presents a model for agent-oriented software development that is fundamentally agent-based and can therefore be applied to the implementation of an agent-based framework that supports development in this paradigm. The presented model can be applied to agent-based software development at any level of abstraction, thereby supporting the development of advanced and complex agent systems that have been built up from simpler agents. The core elements of this model are decoupled from each other and both small and lightweight in terms of their semantics and number of primitive operations. The primary architectural components of agent-oriented development are identified, and an example architecture for an agent platform is presented.

Keywords: agent-based modelling, self-organisation, information systems, agent-oriented platforms,

1 Agent-Oriented Modelling and Design

The modelling of complex systems in terms of an interacting collection of autonomous agents is a natural practice: it is spontaneously adopted by

most children and has been generally used to provide explanations for complex natural phenomena since the beginnings of recorded history. However support for agent-based modelling has been slow to arrive to the software development community. Software development modelling frameworks, which have been constrained by the need to provide direct mappings from their high-level constructs into computational elements, have so far been oriented primarily towards procedural and object-oriented paradigms – not in terms of agents. In recent years, however, there has been a growing interest in the use of autonomous agents to construct complex software systems, even though there is only limited infrastructural agent-based framework support currently available [12]. In this paper we present a model for agent-oriented software development that goes beyond existing agent-based approaches and provides for a complete and consistent environment that flexibly supports the most general notions of agent modelling. The idea is to specify an agent model that supports all varieties of modern software systems, including even the implementation of agent frameworks that support this kind of development. The model can be applied to agent-based system development at any level of abstraction, leading to the support of both top-down and bottom-up development of complex systems:

- advanced and complex agent-oriented systems can be built from simpler agent-oriented systems, and
- the problem of how to design complex problem-solving agents can be attacked by hierarchically refining complex agents into collections of simpler agents, until a point is reached where the simple agents carry out tasks for which the solution (task execution) is understood.

The scheme of complex agents being composed of simpler agents identifies a recursion that ultimately ends up in a simple, primitive (kernel) agent. These primitive agents are specified to be lightweight and small both in their semantics and in the number of their primitive operations.

In this paper we present the design philosophy of the agent framework model and the fundamental components. The specification is language- and implementation-independent: all schemas are expressed in XML Schema language [24], and XML [23] is used for the representation of specification information when it is exchanged in the system (see the Appendix for details).

2 Agent Framework

2.1 Existing frameworks

It is recognised that the construction of complex information systems should not be based on large, monolithic structures, but should instead be based on a collection of decoupled and modular processing units. In this way the developers can more easily achieve effective flexibility, failure recovery, maintainability, etc. Such thinking has led to the development of two-tier, and more recently three-tier and multi-tier, client/server models. Taken further, one arrives at the notion of a distributed collection of objects, each of which may perform the role of a client or server as the occasion demands. Despite the generality of this latter idea, though, it still does not address all the difficulties and complexity issues that can arise in connection with the development of complex information systems. We believe that the better approach is to design and implement a large information system in terms of a distributed collection of agents, rather than objects.

In the context of object-oriented systems, an object offers the abstraction of a localised, encapsulated *state* (characterised by a set of attributes) and a publicly accessible set of *services*. In contrast to an object, a software agent has some additional features: besides a state and services (in this context sometimes referred to as "capabilities") it has *goals*, *autonomy*, and (usually) an ability to *cooperate* with other agents. The addition of these simple features raises the attractive possibility of software engineers performing high-level agent-oriented modelling and then being able to map them directly into complex, distributed information system implementations [12]. In order to support this kind of agent-oriented software engineering, there is a need for agent-oriented frameworks that fully support the agent-modelling paradigm at multiple levels of abstraction. Existing agent-oriented frameworks fall short of this goal – they only provide an inter-agent communication infrastructure based on message-passing, together with simple extensions to existing procedural and object-oriented frameworks for building coarse-grained components. Furthermore such systems do not support hierarchical, agent based refinement. In this paper we argue that agent-oriented techniques should be used not only for the representation and implementation of coarse-grained business components, but for all levels of abstraction.

Current agent frameworks, such as ZEUS [10], JADE [1, 2, 17], FIPA-OS [16], JATLite [13], Jackal [5], and others, are limited with respect to their supported communication patterns. They are fashioned around simple peer-to-peer message passing models. Message passing, useful though it is, is insufficient to support the full range of communication patterns, such as

content-based addressing and data streaming. Message passing is suitable for discrete communication, but it does not deal with communication in the general sense. Thus agent system builders are sometimes compelled to use an existing agent framework in conjunction with other middle-ware technologies, such as CORBA [8] or Java Enterprise Edition [19] in order to provide a larger spectrum of communication patterns.

A second limitation with current agent frameworks is concerned with the issue of control. The agent paradigm cleanly separates the encapsulated properties and behaviour of individual agents from the manner in which they are controlled or organised. This then accommodates a more general perspective concerning multi-agent behaviour, including the possibility of self-organisation [14]. Existing agent platforms, however, employ predefined control mechanisms that are sometimes "wired" into the agents. For the internals of an agent, which is usually not specified by current agent specifications and is thereby left up to the developer to implement, hybrid solutions are common. For the case of object-oriented frameworks, the developer is usually forced to use threads to build complex agent behaviour. If the internals of an agent were implemented by employing agents, recursively, there would be more encapsulation and flexibility available in terms of control.

2.2 A new agent framework

To counter some of the limitations of existing frameworks, we believe that the notion of the *agent* should be used uniformly to encapsulate local behaviour at all levels in agent-oriented systems. Thus to build a complex agent, one could decompose the various complex tasks of the agent into activities performed by more primitive agents. These internal agents could, in turn, be decomposed into yet more primitive agents until a level is reached such that an agent is simple enough to be responsible for a single concern or course of actions. Using this approach, agents are employed uniformly at all levels, and there need not be custom control solutions that span several levels of abstraction, nor need there be forced decisions to be made concerning whether the nature of an agent should include the notion of a thread-like control structure.

In order to encapsulate the local behaviour of the agent and allow flexibility with respect to multi-agent control, we designate some essential features of the internal agent architecture. The notions of *goal* and *role* are identified here. A *goal* is a declarative representation of a state that an agent attempts to bring about in the world. The behaviour of an agent is defined in terms of pro-active *tasks* which must be carried out in order to achieve an agent's specified goals. A *role* represents a specific type of behaviour that is suitable for certain types of goals. Agent control is consequently encapsulated in the

agents' roles and is ultimately goal-driven.

With respect to agent communication, existing frameworks only support message passing. There are, however, other significant communication patterns; we identify three major patterns, two in the area of discrete communication and one in the area of continuous communication, that cover the range of communication styles and which we wish to support. For discrete communication, our agent framework supports message passing and tuple-space communication. For continuous communication, it supports data streaming.

The following sections describe the features of our agent framework in greater detail.

3 Communication

Over the years the variety of proposed computer architectures has led to a number of different communication models for the passage of information between computing entities. Two fundamental communication models that have emerged are message passing and shared memory models. The agent paradigm, which we argue is a fundamental modelling approach, should be usable in connection with both of these communication models. However the trend in agent-based frameworks up to now has been to concentrate more or less exclusively on message passing [12, 21], which neglects other communication modes of interest, such as blackboard systems, anonymous coupling, and content-based addressing. These latter communication modes have found direct support in other communication framework models based on shared memory that have been proposed in specialized research areas outside the agent research community, such as tuple spaces in Linda [7]. The recognition of the modelling possibilities of tuple spaces and the fact that they can lead to better decoupling than solutions based on message passing is now reaching the wider software engineering community, as can be witnessed by the recent interest shown in practical implementations [4, 18, 25].

Note that both message passing and shared memory communication are generally powerful: each can be implemented on top of the other. Tuple spaces can be simulated by pure message passing solutions, via the use of facilitators and brokers; and message passing can be simulated via tuple spaces. Nevertheless we shall argue that tuple spaces is the more general and elegant mechanism in the context of agent-based systems. (Although *tuple spaces* is the original term, we will here refer to *tuple-space-based* communication simply as *space-based* communication, since the key notion here is that of a *space*, which in this context is a shared container for objects that can be accessed concurrently by multiple agents. Our references to space-

based communication will be to the general design pattern rather than to any specific implementation.)

In addition to discrete communication, it is also appropriate to consider how continuous streamed communication can be supported in connection with agent-based modelling. Data streaming cannot be generally supported by discrete techniques, since a given amount of data to be transferred can always exceed the capacity of a tuple space or message buffer. For the general case, though the data to be streamed can be theoretically unlimited, as in the case of visual data from an optical sensor.

3.1 Space-based communication

Space-based communication was put on a clear footing with the work of David Gelerntner [7] in connection with his work on the Linda system for distributed computing. We draw from his idea the notion of a *space*, which is a container, to and from which various active components can read, write, and remove objects. Different primitive operations are associated with specific implementations. The space-based model can be applied not only to agent-based systems but to any parallel and distributed processing system, and there is evidence that it is more flexible and expressive than message passing, while at the same time being competitive with message passing in terms of performance [3]. One of the significant advantages of space-based communication is that it can support both agent-to-agent message passing and agent communication by means of the environment, as in ant-based agent systems. Since all direct communication is between agents and a single, virtual entity, the space, it is possible to maintain the anonymous nature of communication, which in turn is advantageous for the support of self-organising agent systems.

We consider a space to have at least three basic primitive operations: *insert* (inserts an object with specified attributes into the space), *lookup* (locates an object via a search on its attributes) and *remove* (removes an object from the space). There exists additionally an event notification service that indicates that an object has been changed. The space however should be treated as a design pattern and can be implemented in a number of different ways.

3.2 Message passing

As mentioned at the beginning of this section, the current tendency is for agent-based frameworks to support only message passing. Although all the leading agent communication languages e.g. [6, 15] are designed around the

message passing model, we believe that it is more appropriate to use space-based communication rather than message passing as the core communication model because of the following argument. It is straightforward to implement message passing and explicit receiver-type addressing in terms of a space-based model. And it is similarly straightforward to support data-flow-based communication and content-based addressing in terms of the space-based model. On the other hand from the reverse direction, it is a non-trivial project to support content-based addressing and data-flow communication via message passing.

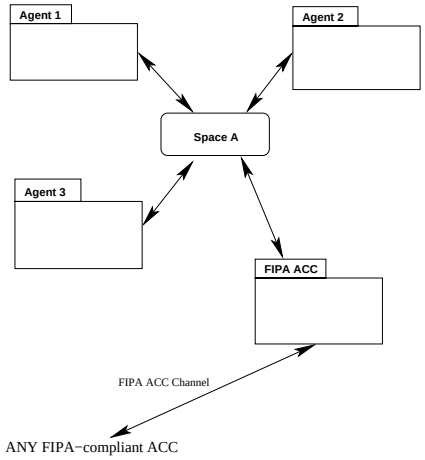

Figure 1: FIPA Communication integration.

For example, to provide message passing via a general purpose space-based model, some specialised agents must be provided which can take care of message delivery. An example of message passing according to the agent communication architecture specified by the Foundation for Intelligent Physical Agents (FIPA) [6] is shown in Figure 1. To support FIPA-based communication (Figure 1), it is simply necessary to install a specialised agent that implements the FIPA Agent Communication Channel (ACC), which will take care of transporting, marshalling, and encoding of messages according to the FIPA specification. FIPA does not specify any architectural restriction concerning the kind of communication that must take place within a given platform: the FIPA communication specification applies to communication from one platform to another by means of FIPA ACCs. Consequently it is easy to extend the communication shown in Figure 1 to include message broadcasting and multicasting.

3.3 Data streaming

Up to this point we have considered only discrete communication, and the space-based approach is sufficient to serve as a grounding for the full spectrum of discrete communication. There are, however, real applications where a continuous stream of information or data must be modelled, and it is artificial to be restricted only to discrete communication. Consequently it is useful for a generic agent framework to provide for a standard streaming mechanism. The agent framework should not make streaming provisions that are specific to a particular implementation of data streaming, so we restrict ourselves here simply to the high-level pattern of streaming and specify that a standard stream header be used. The stream header, which identifies exactly how the information is to be streamed, is inserted into the space, and this header can be shared with other active components (agents) that access the space. The actual streamed data (the body of the stream) is accessed via the medium specified in the stream header.

Thus the stream header contains all the information needed for connecting to, reading data from, and parsing the data from a given stream. It also specified how the information streamed is encoded or compressed and its format. For the actual performance of streaming that is carried out at a lower level, any of a number of possibilities may be employed, such as sequences via CORBA IIOP, ATM sockets, FTP protocols, etc.

3.4 Custom communication spaces

Customized communication spaces may be constructed for specific purposes involving more complex communication patterns, such as one based on constraints in the spatial or temporal domain. For example it would not be difficult to model ant-like agent algorithms [22] by using such a custom design communication space. To do so, one would need to specify specific rules concerning which agents can access a given space at what times. In the case of ant colony when some pheromone is dropped in the environment, it can only be smelled by other ants within a region local to the dropping (spatial constraint) and only for a certain time after the occurrence of the dropping (temporal constraint). It is straightforward to implement the temporal constraint as an information deletion agent that deletes information from the space according to some time-driven algorithm. Implementing the spatial constraint (assuming continuous values for spatial location) could be based on some spatial calculations that determine individual access rights: only agents located within a specified distance to the pheromone dropping would have access to the information about the pheromone.

4 Agent Control

A fundamental design principle of the agent architecture presented here is to separate those components that are implementation-independent from those that could vary with the implementation. This would facilitate the development of agent implementations customized for particular requirements, such as speed and/or size constraints. The upper-level agent functionality is self-contained and maintained in a fashion that is independent of any implementation. Its specification is expressed in XML Schema, and this facilitates the use of XML for the transport of agent definitions, tasks, goals, etc. (see the Appendix for details). The execution of the agent behaviour relies on a lower-level, core library that varies with the application domain and is currently left up to the implementer (although we intend to specify some core functionality in the future in order to support more platform interoperability).

In this architecture each agent has some basic features:

- a goal, which may be a composite set of more primitive goals
- a role, which specifies the agent's responsibilities
- a current task and a mechanism to execute that task
- input and output communication ports

An agent will execute its current task with the intention of achieving its goal. The actual mechanism for task execution is left unspecified and may encompass powerful techniques, such as the Procedural Reasoning System (PRS) [11, 20]. However, unlike the original agent system used in connection with PRS, object-oriented encapsulation of knowledge is assumed, which provides for an improvement in modularity. In general the architecture presented here employs object-oriented modelling for the representation of services and passive components and agent-based modelling for representing active components uniformly at each modelling level of the system. Since objects can be thought of as degenerate agents (agents with some inactive elements), the entire agent architecture can be thought of as represented in terms of agents. This is one of the significant advances that separates our approach from traditional agent architectures, such as PRS.

In the following subsections we provide a more detailed description of these architectural components.

4.1 Goal

A *goal* may be either a *primitive goal* or a composite entity that represents a combination of other goals. A *primitive goal* is represented as a particular state of internal and/or external variables that the agent attempts to achieve. The goal always takes a boolean value - its achievement is either TRUE or FALSE (*achieved* or *not achieved*). Thus goals here are similar to those in JAM [9], UMPRS and PRS-C [20, 11]. However, unlike JAM, the model presented here does not have the notion of "performing a goal", which has been introduced into JAM to enable mixing of declarative and procedural styles of task invocation, but which in our view only corrupts modularity and encapsulation.

Goals can represent both stable states and volatile states, consequently the actual check for goal achievement is performed on the primitive goals in connection with the native implementation. Note that a goal is not necessarily achieved by direct action on the part of the given agent: the particular state which is treated as a goal could also be achieved by a sudden change in the environment, by other entities pursuing their actions, by user intervention, etc. Moreover a goal, once achieved, can subsequently change back into the state of *not achieved*. In this case it is appropriate to speak of *maintaining* a goal. Thus the intention of an agent might not only be to achieve a goal but also to maintain it, as well.

For our system only primitive goals operate on internal and external states directly; complex goals are defined in terms of other goals thereby improving maintainability and readability of the design. Thus only primitive goals are concerned with the native implementation, whereas all domain-level goals are kept implementation independent.

4.2 Task

A *task* is a course of actions that can lead to achieving a goal. It consists of preconditions, a body, postconditions, and a context. The body of a task contains control information specifying how subgoals are achieved. If the body of a task contains (native) executable statements, this task is called *primitive*.

Both preconditions and postconditions operate on goals. A task is applicable if and only if all the goal-states specified in the precondition are either achieved or not-achieved according to the specification (i.e. they are TRUE). A task that has been executed successfully will result in the goal-states specified in the postcondition being set as achieved or not-achieved accordingly. If a completed task fails to establish or maintain its postcondition, it is a

failed task. In addition to pre- and postconditions, a task also has a context, which is a logical expression on goal-states. During task execution these constraints must be maintained (the context must be kept TRUE), otherwise the task must be abandoned. If the task is applicable and has a postcondition corresponding to the agent's goal, the task is *goal applicable*.

The state of the world and the state of the agent itself are referenced in the task precondition, postcondition, and context by means of goal-states and only goal-states. This feature is necessary for decoupling declarative aspect of the behaviour from its (native) procedural one. Inside the task body, the state of the world and the state of the agent (by means of internal and external variables) may be referenced and manipulated directly. This distinguishes the current system from PRS-based systems [11, 20, 9] and makes the architecture purely goal-oriented (which we consider to be a "cleaner" option than that of existing systems).

Tasks can be maintained in a task library, and appropriate ones can be selected for execution, given the current state of an agent, its goals, and its role (see next subsection).

4.3 Role

A *role* specifies an agent's responsibilities: it declares a set of goals which can be achieved and/or maintained. Note that a role does not provide for any behaviour, i.e. information concerning how a goal is to be achieved - it simply declares and encapsulates the agent's responsibilities (goals to be achieved). In a sense, it can be considered to be analogous to a thread type in the object-oriented paradigm.

When an agent has a known role, this information can be used in order to select a particular task from a task library that can be used by the agent in order to achieve its goals. The task library contains all the tasks that have been defined and implemented for a given domain. The actual mechanism that specifies the coupling between roles and tasks, i.e. between an agent's goals and its course of action, can take any of a number of forms and can, in fact, be performed dynamically at runtime. The selection of a particular task, the scheduling of its execution, and the manner in which it is executed is performed by a specialised entity (agent), which is analogous to the interpreter of PRS-like systems. Unlike JAM and PRS-like systems, however, our architecture does not specify any particular interpreter, but simply leaves the task management activity up to an agent that must perform these duties. This maintains design flexibility and ensures consistency with the recursive agent paradigm. As a consequence, reification of the "interpreter" agent will return an agent that implements particular roles (i.e. selects and

couples tasks appropriate for a given role) in connection with an attempt to achieve the given goals. In particular, the native "interpreter" agent could (but doesn't have to) use a PRS-like interpretation engine, or it could contain hard-coded reactive, behavioural rules.

For the case of more complex systems, there may not be tasks that are ready to be used in connection with an attempt to achieve some of the goals. In such a case it may be necessary to construct a plan that specifies which tasks should be executed, and in what order, by the agent. This capability can be modelled by the architecture as shown in Figure 2.

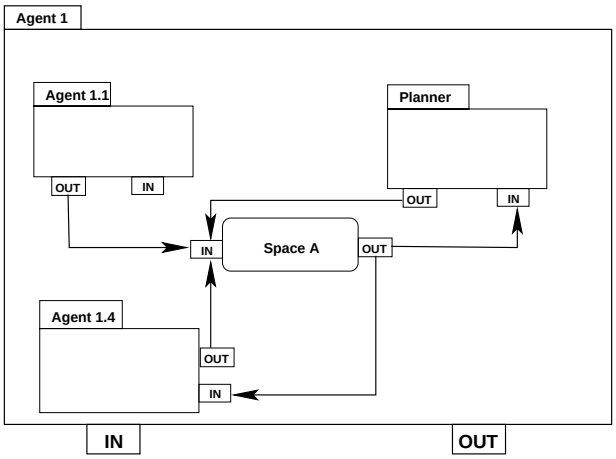

Figure 2: Internal Planner integration.

The Planner agent will continuously wait for new state information submitted to the space from other agents. This information can essentially constitute new goals for which new tasks must be made ready. As shown in Figure 2, the model is flexible, since the responsibilities of the Planner agent could be assumed by a new agent at any point in time.

4.4 Agent system

Now consider a system of agents that cooperate in order to carry out some activities. The basic building block is a *primitive agent*, which embodies a set of roles, i.e. instantiates a set of responsibilities. (It may be desirable, but not essential, to allow agents to acquire new roles and drop existing ones during runtime.) An essential feature of a primitive agent is that, no matter what its responsibilities are and no matter how many possible roles it has, the primitive agent can only have a single, top-level goal. This corresponds

to a single thread in the traditional object-oriented programming paradigm. We should consider the primitive agent, in fact, to be a replacement of the notion of a thread in the context of the agent architecture.

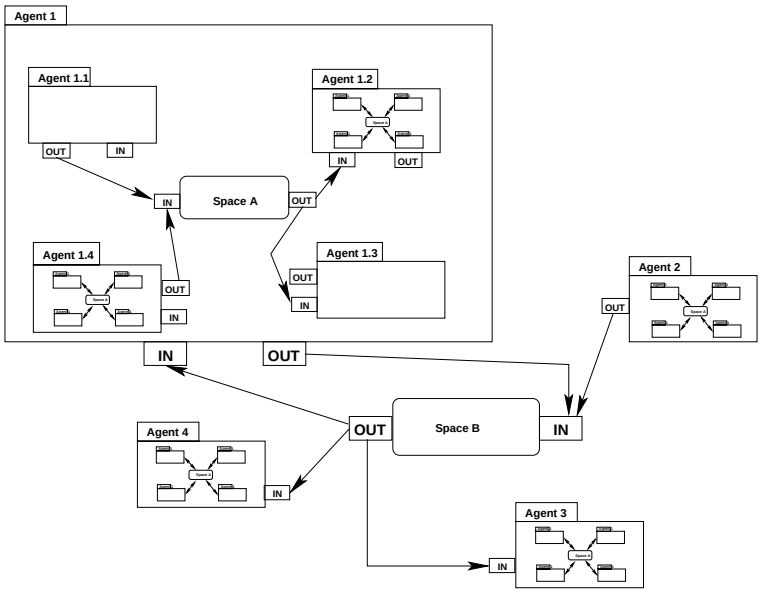

Figure 3: Agent System example.

A *composite agent* is an agent-system that is composed of other agents that make up the internal architecture (see Figure 3). As introduced in Section 3.1, all the communication between agents is performed by means of space-based communication. Thus a central item in every composite agent is a space, which is used to enable communication between different internal agents within the given agent. In the normal course of system design, useful behaviour is expected to be achieved by these agent systems, which are made up of a number of simple (but not necessarily primitive) agents that achieve/maintain goals, execute tasks, etc.

An agent system is characterised by its agent configuration, which specifies (a) a list of all the agents in the system, together with their states and current top-level goals and (b) the space state - the list of all objects currently held in the agent's space. In Figure 3, an example agent system is shown, indicating how complex agents can be specified in terms of a set of smaller agents.

Figure 4 shows a simplified UML-like class diagram of the essential internal agent architectural components. The diagram reflects our earlier discussion concerning the control structures. Note, in particular, the absence of the

space entity from this diagram. This is because the concept of a space is implemented as an object-oriented service and, in the context of this diagram, would fit in as a specialised agent (all agents are ultimately implemented as class objects). In the context of control, we emphasise that an agent encapsulates all necessary control mechanisms for running and controlling other agents. In particular an agent can set up an agent's goals to be achieved and/or maintained start/stop/restart/kill running agents.

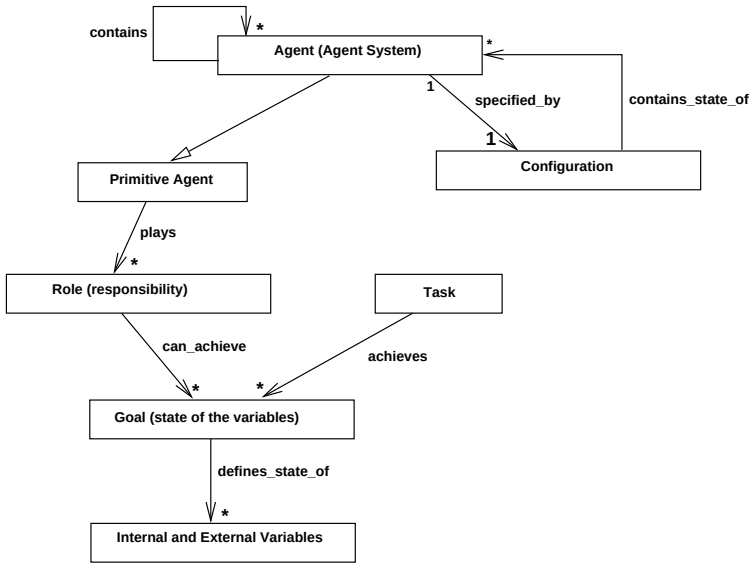

Figure 4: The Schema of Control Architecture.

5 Conclusions

We have described a general and flexible agent architecture that maintains the agent-modelling paradigm at multiple levels of refinement. The current developed prototype that is based on space-based processing must be extended and tested in the future with various types implementation modules ("interpreter" agents). In the next stage of the research, experiments will be performed on these prototypes with different architectural styles and self-organising agent patterns. We also intend to specify more fully how the adoption of this framework by some existing agent frameworks can lead to interoperability enhancements.

With respect to more general aspects of interoperability, we note that there are further issues that need to be addressed. For "static" systems, it

is relatively straightforward to identify all the basic system components and statically map all the roles and communication patterns (as is done in connection with systems following the FIPA specification). However it is more difficult to specify these matters for self-organising systems. In this latter case, two levels of interoperability must be achieved: low-level transport interoperability and higher-level domain and conceptual interoperability. Our approach represents a general approach to enhance interoperability possibilities for each of these levels. We believe it establishes a general framework on top of which further advances can be made.

Appendix

This appendix shows XML Schema defining the higher level concepts for the proposed agent architecture. We consider these schemas to represent a starting point for the formal specification for agent-oriented models.

```
     === goal.xsd ===

<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">

  <xsd:annotation>
    <xsd:documentation>
           Goal specification.
     </xsd:documentation>
   </xsd:annotation>

  <xsd:element name="Goal" type="GoalType"/>
  <xsd:element name="comment" type="xsd:string"/>

  <xsd:complexType name="GoalType">
     <xsd:attribute name="isPrimitive" type="xsd:boolean"/>
     <xsd:attribute name="name"   type="xsd:string"/>
     <xsd:element name="goals" type="GoalList"/>
     <xsd:element ref="comment" minOccurs="0"/>
  </xsd:complexType>

  <xsd:complexType name="GoalList">
       <xsd:element name="goal"   type="xsd:string"  minOccurs="0"/>
  </xsd:complexType>

</xsd:schema>
```

```
     === role.xsd ===

<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">

  <xsd:annotation>
    <xsd:documentation>
           Role specification.
     </xsd:documentation>
   </xsd:annotation>

  <xsd:element name="Role" type="RoleType"/>
  <xsd:element name="comment" type="xsd:string"/>

  <xsd:complexType name="RoleType">
     <xsd:attribute name="isPrimitive" type="xsd:boolean"/>
     <xsd:attribute name="name"   type="xsd:string"/>
     <xsd:element name="goals" type="GoalList"/>
     <xsd:element name="roles" type="RoleList"/>
     <xsd:element ref="comment" minOccurs="0"/>
  </xsd:complexType>

  <xsd:complexType name="GoalList">
       <xsd:element name="goal"   type="xsd:string"  minOccurs="0"/>
  </xsd:complexType>

  <xsd:complexType name="RoleList">
       <xsd:element name="role"   type="xsd:string"  minOccurs="0">
  </xsd:complexType>

</xsd:schema>



   === agent.xsd ===

<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">

  <xsd:annotation>
    <xsd:documentation>
           Single Agent  Specification.
```

```
      </xsd:documentation>
    </xsd:annotation>

  <xsd:element name="Agent" type="AgentType"/>
  <xsd:element name="comment" type="xsd:string"/>

  <xsd:complexType name="AgentType">
     <xsd:attribute name="isPrimitive" type="xsd:boolean"/>
     <xsd:attribute name="name" type="xsd:string"/>
     <xsd:element name="roles" type="RoleList"/>
     <xsd:element name="agents" type="AgentList"/>
     <xsd:element ref="comment" minOccurs="0"/>
  </xsd:complexType>

  <xsd:complexType name="RoleList">
       <xsd:element name="role"   type="xsd:string"  minOccurs="0">
  </xsd:complexType>

  <xsd:complexType name="AgentList">
       <xsd:element name="agent"   type="xsd:string"  minOccurs="0"/>
  </xsd:complexType>

</xsd:schema>



    === configuration.xsd ===

<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">

    <xsd:annotation>
      <xsd:documentation>
            Agent  System Configuration.
      </xsd:documentation>
    </xsd:annotation>

  <xsd:element name="Configuration" type="ConfigurationType"/>
  <xsd:element name="comment" type="xsd:string"/>

  <xsd:complexType name="ConfigurationType">
     <xsd:attribute name="name" type="xsd:string"/>

     <xsd:element name="space" type="SpaceConfig"/>
```

```
    <xsd:element name="agents" type="AgentList"/>

    <xsd:element ref="comment" minOccurs="0"/>
  </xsd:complexType>

  <xsd:complexType name="SpaceConfig">
       <xsd:element name="objectID"   type="xsd:string"  minOccurs="0">
  </xsd:complexType>

  <xsd:complexType name="AgentList">
       <xsd:element name="config"   type="AgentConfigurationType"
                 minOccurs="0"/>
  </xsd:complexType>

  <xsd:complexType name="AgentConfigurationType">
   <xsd:element name="agent"   type="xsd:string"  minOccurs="1"
                maxOccurs="1" />
   <xsd:element name="config"  type="ConfigurationType"
                minOccurs="1"
                maxOccurs="1" />
  </xsd:complexType>

</xsd:schema>
```

References

[1] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE - A FIPA-compliant agent framework. Project home page at `http://sharon.cselt.it/projects/jade`, 2000.

[2] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE Programmers Guide. Project home page at `http://sharon.cselt.it/projects/jade`, June 5 2000.

[3] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Reactive tuple spaces for mobile agent coordination. In *Proceedings of Mobile Agents 98*, 1998.

[4] IBM Corporation. TSpaces Programmer's Guide. Guide for Version 2.1.1, online at: `http://www.almaden.ibm.com/cs/TSpaces/`, 2000.

[5] R. Scott Cost, Tim Finin, Yannis Labrou, Xiaocheng Luan, Yun Peng, Ian Soboroff, James Mayfield, and Akram Boughannam. Jackal: A Java-Based Tool for Agent Development. *Working Notes of the Workshop on Tools for Developing Agents AAAI 98*, 1998. Available at: `http://jackal.cs.umbc.edu/aaai98.pdf`.

[6] FIPA. FIPA Spec 2 - 1999. Agent Communication Language. Draft, Version 0.1, web site at `http://www.fipa.org/spec/index.html`, 16 April 1999.

[7] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 1(2), January 1985.

[8] Object Management Group. CORBA/IIOP 2.3.1 Specification. Available at: `http://cgi.omg.org/cgi-bin/doc?formal/99-10-07`, October 1999.

[9] Marcus J. Huber. JAM Agents in a Nutshell. Email: marcush@home.com, home page: `http://members.home.net/marcush/IRS`, July 1999.

[10] Lyndon Lee Hyacinth Nwana, Divine Ndumu and Jaron Collis. ZEUS: A Tool-Kit for Building Distributed Multi-Agent Systems. *Applied Artifical Intelligence Journal*, 13(1):129–186, 1999.

[11] SRI International. Procedural Reasoning System User's Guide. A manual for version 1.96, online: `http://www.ai.sri.com/~prs`, March 17th 1999.

[12] Nicholas R. Jennings. Agent-oriented software engineering. *Artificial Intelligence*, 117(2), 2000.

[13] Heecheol Jeon, Charles Petrie, and Mark R. Cutkosky. JATLite: A Java Agent Infrastructure with Message Routing. Available at: `http://www-cdr.stanford.edu/ProcessLink/papers/jat/jat.html`.

[14] Kevin Kelly. *Out of Control: The New Biology of Machines, Social Systems and the Economic World*. Perseus Publishing, May 1995. ISBN: 0201483408.

[15] Yannis Labrou and Tim Finin. A proposal for a new KQML specification. Technical report, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD 21250, 3 September 1997. Available at: `http://www.cs.umbc.edu/~jklabrou/publications/tr9703.ps`.

[16] Mikko Laukkanen. Evaluation of FIPA-OS 1.03. Cellular System Develompent, Sonera Mobile Operator, Sonera Ltd.
Email: mikko.t.laukkanen@sonera.com, February 2000.

[17] Mikko Laukkanen. Evaluation of JADE 1.2. Cellular System Develompent, Sonera Mobile Operator, Sonera Ltd.
Email: mikko.t.laukkanen@sonera.com, February 2000.

[18] Sun Microsystems. JavaSpaces Specification - 1.0. Online at: `http://java.sun.com/products/javaspaces/`, 1999.

[19] Sun Microsystems. Designing Enterprise Applications with the Java2 platform, Enterprise Edition. Version 1.0 Final Release, March 22 2000.

[20] K. L. Myers. A procedural knowledge approach to task-level control. In *Proceedings of the Third International Conference on AI Planning Systems*, 1996.

[21] Hyacinth S Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):1–40, September 1996.

[22] H. Van Dyke Parunak. Go to the Ant: Engineering Principles from Natural agent Systems. (75), 1997.

[23] W3C. Extensible Markup Language (XML) 1.0. W3C Recommendation 10-February-1998, online: `http://www.w3c.org/xml`.

[24] W3C. XML Schema. W3C Working Draft, 7 April 2000, online: `http://www.w3c.org/xml`.

[25] Alan Wood and Ronaldo Menezes. Ligia: A Java based Linda-like runtime system with garbage collection of tuple spaces. Email:wood@minster.york.ac.uk.