Modelling and Design of Multi-Agent Systems*

David Kinny Michael Georgeff

Australian Artificial Intelligence Institute
171 Latrobe Street, Melbourne 3000, Australia
{*dnk,georgeff*}*@aaii.oz.au*

Abstract. Agent technologies are now being applied to the development of large-scale commercial and industrial software systems. Such systems are complex, involving hundreds, perhaps thousands of agents, and there is a pressing need for system modelling techniques that permit their complexity to be effectively managed, and principled methodologies to guide the process of system design. Without adequate techniques to support the design process, such systems will not be sufficiently reliable, maintainable or extensible, will be difficult to comprehend, and their elements will not be re-usable.

In this paper, we present techniques for modelling agents and multi-agent systems which adapt and extend existing Object-Oriented representation techniques, and a methodology which provides a clear conceptual framework to guide system design and specification. We have developed these techniques for systems of agents based upon a particular Belief-Desire-Intention architecture, but have sought to provide a framework for the description of agent systems that is sufficiently general to be applicable to other agent architectures, and which may be extended in various ways.

1 Introduction

The *agent* paradigm in AI is based upon the notion of reactive, autonomous, internally-motivated entities embedded in changing, uncertain worlds which they perceive and in which they act. Agent technologies are now being applied to the development of large-scale commercial and industrial software systems in areas such as air traffic control, airline resource management, spacecraft management, simulation systems, business process and financial dealings management, and communications network management.

Such systems are complex, involving hundreds, perhaps thousands of agents, and there is a pressing need for system modelling techniques that permit their complexity to be effectively managed, and principled methodologies to guide the process of system design. Without adequate techniques to support the design process, such systems will not be sufficiently reliable, maintainable or extensible, will be difficult to comprehend, and their elements will not be re-usable.

* This work was supported in part by the Cooperative Research Centre for Intelligent Decision Systems under the Australian Government's Cooperative Research Centres Program.

Foremost amongst the modelling techniques and methodologies that have been developed for the design and specification of conventional software systems are various Object-Oriented (OO) approaches [2, 16], based upon the central notion of *objects* which encapsulate state information as a collection of data values and provide *behaviours* via well-defined interfaces for operations upon that information. OO methodologies guide the key steps of object identification, design, and refinement, permitting abstraction via *object classes* and inheritance within *class hierarchies*.

In attempting to develop a methodology and models that provide adequate support for the process of agent system design, our approach, pragmatically motivated, has been to explore how existing OO modelling techniques can be extended to apply to BDI agent systems. OO techniques have achieved a considerable degree of maturity, and there is widespread acceptance of their advantages. A large community of software developers familiar with their use now exists. By building upon and adapting existing, well-understood techniques, we take advantage of their maturity and aim to develop models and a methodology that will be easily learnt and understood by those familiar with the OO paradigm. Others have reached similar conclusions about the need for familiar, intuitive modelling techniques [14].

In specifying agent systems we employ a set of models which operate at two levels of abstraction. Firstly, from the external viewpoint, a system is modelled as an inheritance hierarchy of agent classes, of which individual agents are instances. Agent classes are characterized by their purpose, their responsibilities, the services they perform, the information about the world they require and maintain, and their external interactions. Secondly, from the internal viewpoint, we employ a set of models which allow structure to be imposed upon the informational and motivational state of the agents and the control structures which determine their behaviour. In our case, these are beliefs, goals, and plans.

The design methodology is based on the identification of the key roles in an application and their relationships, which guides the elaboration of the agent class hierarchy. Analysis of the responsibilities of each agent class leads to the identification of the services provided and used by an agent, its external interactions, and the goals and events to which it must respond. This progressive decomposition produces a fine-grained model of agency, which is then recomposed to produce concrete agents that are inherently modular. Decisions about agent boundaries may be deferred to a late stage of the design process.

In this paper, we assume a reasonable level of familiarity with OO modelling techniques. We have contrasted the similarities and differences between OO and AO approaches elsewhere [12]. Here, we focus on the details of the modelling techniques we have developed. In Section 2 we outline our AO modelling technique. In Section 3 we present in detail the belief, goal and plan models that describe individual agents, and in Section 4 the agent model which describes the structure of a multi-agent system. Finally, in Section 5 we present the essentials of the methodology that guides the development and refinement of these models.

2 An Agent-Oriented Modelling Technique

Object-Oriented modelling techniques [2, 16] describe a system by identifying the key *object classes* in an application domain, and specifying their *behaviour* and their relationships with other classes. The essential details of a system design are captured by three different types of models.

1. An *Object Model* captures information about objects within the system, describing their data structure, relationships and the operations they support.
2. A *Dynamic Model* describes the states, transitions, events, actions, activities and interactions that characterize system behaviour.
3. A *Functional Model* describes the flow of data during system activity, both within and between system components.

The dynamic and functional models serve to guide the refinement of the object model; in particular, the refinement of the operations that an object will provide. A fully refined object model is a complete specification of an object based system. The object concept is applied uniformly at all levels of abstraction.

By contrast, in specifying an agent system, we have found that it is highly desirable to adopt a more specialized set of models which operate at two distinct levels of abstraction. Firstly, from the *external viewpoint*, the system is decomposed into agents, modelled as complex objects characterized by their purpose, their responsibilities, the services they perform, the information they require and maintain, and their external interactions. Secondly, from the *internal viewpoint*, the elements required by a particular agent architecture must be modelled for each agent. In our case, these are an agent's beliefs, goals, and plans.

The description of an agent system from the external viewpoint is captured in two models, which are largely independent of our BDI architecture.

1. An *Agent Model* describes the hierarchical relationship among different abstract and concrete agent classes, and identifies the agent instances which may exist within the system, their multiplicity, and when they come into existence.
2. An *Interaction Model* describes the responsibilities of an agent class, the services it provides, associated interactions, and control relationships between agent classes. This includes the syntax and semantics of messages used for inter-agent communication and communication between agents and other system components, such as user interfaces.

From the internal viewpoint, each agent class is specified by three models, specific to our BDI architecture, that describe its informational and motivational state and its potential behaviour.

1. A *Belief Model* describes the information about the environment and internal state that an agent of that class may hold, and the actions it may perform. The possible beliefs of an agent and their properties, such as whether or not they may change over time, are described by a *belief set*. In addition, one or more *belief states* – particular instances of the belief set – may be defined and used to specify an agent's *initial mental state*.

2. A *Goal Model* describes the goals that an agent may possibly adopt, and the events to which it can respond. It consists of a *goal set* which specifies the goal and event domain and one or more *goal states* – sets of ground goals – used to specify an agent's initial mental state.
3. A *Plan Model* describes the plans that an agent may possibly employ to achieve its goals or respond to events it perceives. It consists of a *plan set* which describes the properties and control structure of individual plans.

Implicit in this characterization are the execution properties of the architecture which determine how, exactly, events and goals give rise to intentions, and intentions lead to action and revision of beliefs and goals. These properties, described in detail elsewhere [10], are responsible for ensuring that beliefs, goals, and intentions evolve rationally. For example, the architecture ensures that events are responded to in a timely manner, beliefs are maintained consistently, and that plan selection and execution proceeds in a manner which reflects certain notions of rational commitment [11, 15]

Our agent system modelling technique employs object classes and instances to describe different kinds of entities within a multi-agent system at different levels of abstraction. Unlike the standard OO approach, the meanings of relationships such as association, inheritance and instantiation are quite distinct for these different kinds of entities. By partitioning different types of entities into separate models we maintain these important distinctions, and simplify the process of consistency checking, within and between models.

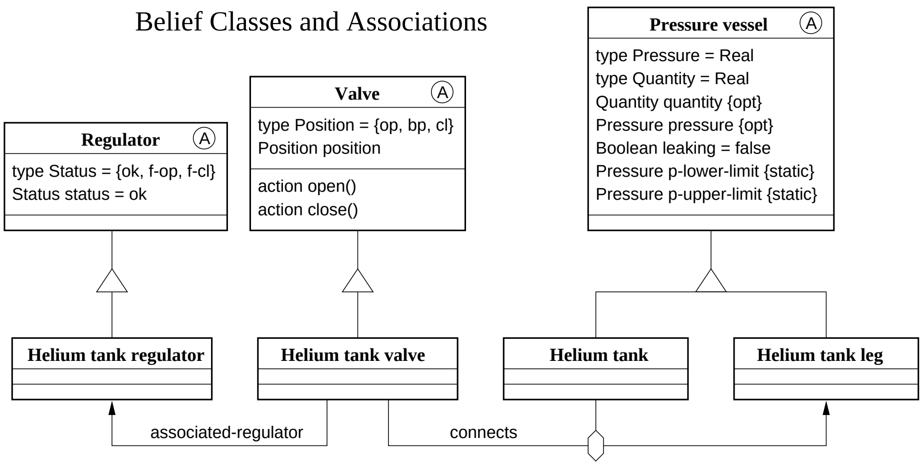
As a result of of our commitment to a particular BDI execution architecture, we can employ OO dynamic models, augmented with a notion of failure, as *directly executable* specifications which generate agent behaviour, i.e., as plans. This provides considerable advantages over the OO approach of programming object methods guided by the dynamic model. Moreover, plans are not required to be a *total* specification of behaviour; certain elements, such as successively trying different means to achieve a goal, are inherent in the the architecture.

The OO object and dynamic model representation techniques of [16], suitably extended and constrained, serves as a basis for our representations. Specifications of the models may be supplied by the agent designer in the form of diagrams developed with a specialized editing tool, or as text files. These serve as inputs to the compilation process that produces an executable system.

3 Modelling Individual Agents

3.1 The Belief Model

A belief model consists of a belief set and one or more belief states. The belief set is specified by a set of object diagrams which define the domain of the beliefs of an agent class. A belief state is a set of instance diagrams which define a particular instance of the belief set.



Derived Predicates and Functions

status(Regulator, Status)	Status status(Regulator)
position(Valve, Position)	Position position(Valve)
quantity(Pressure-vessel, Quantity)	
pressure(Pressure-vessel, Pressure)	
leaking(Pressure-vessel, Boolean)	Boolean leaking(Pressure-vessel))
p-lower-limit(Pressure-vessel, Pressure)	Pressure p-lower-limit(Pressure-vessel)
p-upper-limit(Pressure-vessel, Pressure)	Pressure p-upper-limit(Pressure-vessel)
associated-regulator(Helium-tank-valve, Helium-tank-regulator)	
connects(Helium-tank-valve, Helium-tank, Helium-tank-leg)	

Fig. 1. Belief Classes and Derived Predicates and Functions

Belief Sets A belief set is a set of predicates and functions whose arguments are terms over a set of built-in and user-defined type domains. These predicates, functions and domains are directly derived from the class definitions in the belief set diagrams, and the associations between them. Each class definition defines one or more domains, and each attribute, operation or association defines a predicate and/or function schema.

A belief set diagram is a directed, acyclic graph containing nodes denoting both *abstract* and *concrete* (instantiable) belief classes. Belief classes are represented by class icons, and abstract classes are distinguished by the presence of the adornment Ⓐ in the upper section of the icon. Edges in the graph denoting inheritance are distinguished by a triangle with a vertex pointing towards the superclass. Aggregation and other associations between belief classes are also permitted.

The classes defined in a belief set diagram correspond, in many cases, to real objects in the application domain, but, unlike an OO object model, the definitions do not define the behaviours of these objects. This is because they are not implementations of the objects, rather, they represent an agent's beliefs about those objects. Each belief class serves to define the type signatures of attributes of the object, functions that may be applied to the object, and other predicates that apply to the object, including *actions*, which have a special role in plans.

A belief set is formally defined as follows.

Definition 1
Let I be a set of identifiers, $\mathcal{D} = \{D_1, \ldots, D_n\}$ be a set of type domains, and Z be a set of belief property specifiers.

- An attribute A is a pair $\langle n, D_i \rangle$, i.e., a named type domain. The type of the attribute is denoted $\mathrm{dom}(A)$.
- A predicate (schema) over $\{I, \mathcal{D}, Z\}$, denoted $p(A_1, \ldots, A_k)$, is a tuple $\langle p, Y, A_1, \ldots, A_k \rangle$ s.t. $p \in I$, $Y \subset Z$, $\forall i = 1 \ldots k$, $\mathrm{dom}(A_i) \in \mathcal{D}$.
- A function (schema) over $\{I, \mathcal{D}, Z\}$, denoted $f(A_1, \ldots, A_{k-1}) \mapsto A_k$, is a tuple $\langle f, Y, A_1, \ldots, A_k \rangle$ s.t. $f \in I$, $Y \subset Z$, $\forall i = 1 \ldots k$, $\mathrm{dom}(A_i) \in \mathcal{D}$.
- A belief set is a tuple $\langle I, \mathcal{D}, Z, P, F \rangle$ where P is a set of predicate schema over $\{I, \mathcal{D}, Z\}$ and F is a set of function schema over $\{I, \mathcal{D}, Z\}$.

□

Attributes, which define binary predicates, are specified in the usual way. If an attribute's value can never be unknown, an accessor function is also generated. Other predicates and functions are defined by specializations of the operation notation. An object of the class upon which the operation is defined is implicitly the first argument of the derived predicate or function.

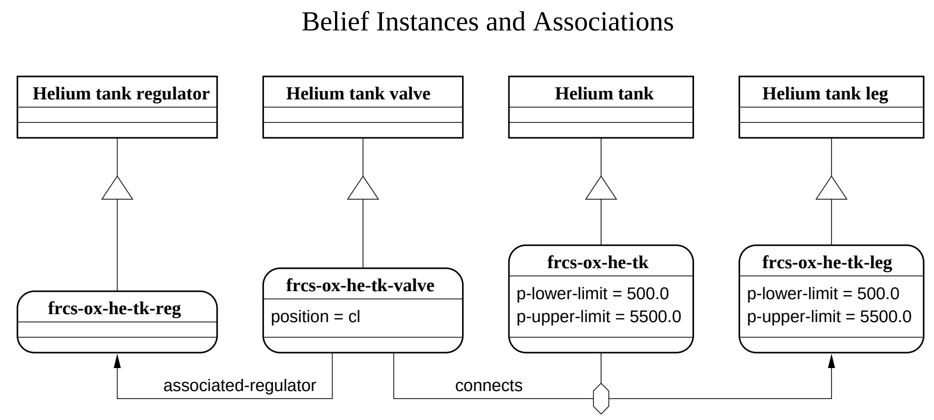
Predicates may also be defined by binary and higher order associations between classes. The multiplicity of these associations is indicated in the usual way. Figure 1 shows several associated belief classes from a NASA shuttle diagnosis application, and the predicates functions derived from them.

Some predicates and functions are not associated with any particular object, i.e. they do not have an implicit first argument. In this case, they can be specified as attributes and operations upon an anonymous (unnamed) class.

We extend the standard notation for attributes and operations by allowing an optional *property list*, which is used to specify certain properties of the derived predicates and functions, such as whether they are abstract, stored in the belief database or computed, whether they may change over time, and for predicates whether they have open- or closed-world negation semantics.

Belief Properties The predicates and functions that constitute an agent's belief set may be classified as *extensional*, *evaluable*, or *ephemeral*. Extensional predicates are stored as relations in the belief database. Extensional functions are automatically generated from extensional predicates that are functionally constrained, such as the binary predicates derived from attribute definitions and one-to-one associations.

Evaluable predicates and functions are computed when required by externally supplied functions in the underlying programming language. Actions are special evaluable predicates that have the effect of bringing about some change in the external environment.



Derived Predicate Instances

status(frcs-ox-he-tk-reg, ok)
position(frcs-ox-he-tk-valve, cl)
leaking(frcs-ox-he-tk, false)
p-lower-limit(frcs-ox-he-tk, 500.0)
p-upper-limit(frcs-ox-he-tk, 5000.0)
leaking(frcs-ox-he-tk-leg, false)
p-lower-limit(frcs-ox-he-tk-leg, 500.0)
p-upper-limit(frcs-ox-he-tk-leg, 5000.0)
associated-regulator(frcs-ox-he-tk-valve, frcs-ox-he-tk-reg)
connects(frcs-ox-he-tk-valve, frcs-ox-he-tk, frcs-ox-he-tk-leg)

Fig. 2. Belief Instances and Derived Predicates

Ephemeral predicates stand for properties or associations that are named but not remembered from moment to moment. Their major role is in the definition of abstract goals. Ephemeral functions are never evaluated.

Extensional and evaluable predicates and functions may be either *static* or *dynamic*. If static, they represent fixed relations and functions whose definitions do not depend on when they are evaluated. If dynamic, they may change over time. Ephemeral predicates are always dynamic. Extensional predicates may have either *open-* or *closed-world* negation semantics. Evaluable predicates always have closed-world semantics. Ephemeral predicates may not be negated.

These properties of predicates and functions are indicated by keywords in the property list associated with the attribute, operation, or association. By default predicates are dynamic, extensional, and have closed-world semantics, and functions are ephemeral. Properties may also be associated with classes and instances, and may represented either by property lists or by adornments on the class icon.

For example, in Figure 1, the attribute p-lower-limit of the class Pressure vessel has the property static, indicating that its value may not change, and the attribute pressure has the property optional, indicating that its value may be unknown. The class itself has an adornment indicating that it is abstract.

Belief States A belief state specifies a particular state of the agent's beliefs, and may be used to initialize its initial mental state. It consists of a set of instances of extensional predicates[2], which are specified by a set of belief diagrams containing object instances and associations between them. The predicate instances are derived from the instance diagrams in a manner analogous to the way in which the predicates are derived from the class diagrams, described above. Figure 2 shows a fragment of an instance diagram from the NASA RCS domain, and the corresponding predicate instances.

A belief state and belief model are formally defined as follows.

Definition 2
Let $\mathcal{B} = \langle I, \mathcal{D}, Z, P, F \rangle$ be a belief set.

- An instance of the predicate $p(A_1, \ldots, A_k) \in P$ is a tuple $p(a_1, \ldots, a_k) \in A_1 \times, \ldots, \times A_k$.
- A belief state S of $\mathcal{B}$ is a set of instances of predicates in P.
- A belief model is a pair $\langle \mathcal{B}, \mathcal{S} \rangle$, where $\mathcal{S}$ is a set of belief states of $\mathcal{B}$.

□

3.2 The Goal Model

Each agent class is associated with a particular *Goal Model*, consisting of a *goal set* and one or more *goal states*. The goal set specifies the domain of the goals of an agent of that class. Goal states are sets of ground instances of elements of the goal set which may be used to initialize an agent's initial mental state.

A goal set is, formally, a set of goal formula signatures. Each such formula consists of a modal goal operator applied to a predicate from the belief set. The modal goal operators are:

achieve (!) denoting a goal of achievement ("Make it that ϕ holds!"),
verify (?) denoting a goal of verification ("Is it that ϕ holds?"), and
test ($) denoting a goal of determination ("Determine if ϕ holds or not").

Goal formulae occur within activities in the bodies of plans and within their activation events (Section 3.3). In essence, such an activity is performed by finding the set of plans whose activation event matches the goal formula, and executing one or more of them to determine the success or failure of the activity. The different modalities determine how, exactly, such activities are performed. Depending on the modality, the type of the predicate, and the agent's initial beliefs about the predicate, different execution sequences may occur.

A goal of achievement succeeds (vacuously) if its predicate is believed to hold. Otherwise, if a matching plan executes successfully (they are tried sequentially according to a precedence ordering), the goal succeeds, else it fails. The postcondition of successful completion is that the predicate is believed to hold.

[2] Extensional functions are derived from predicates, hence not specified separately. The state of ephemeral and evaluable predicates and functions is opaque, and may not be initialized.

A goal of verification succeeds (vacuously) if its predicate is believed to hold, and fails (vacuously) if its predicate is believed not to hold. Only in the case that the predicate is unknown (hence, must be ephemeral or have open-world semantics) can plan execution result. If a matching plan executes successfully the goal succeeds, else it fails. The postcondition of successful completion is that the predicate is believed to hold.

A goal of determination succeeds (vacuously) if its predicate is believed to hold or not to hold. Only in the case that the predicate is unknown (hence, must be ephemeral or have open-world semantics) can plan execution result. If a matching plan executes successfully the goal succeeds, else it fails. The postcondition of successful completion is that the predicate is no longer unknown.

Definition 3
Let $B = \langle I, \mathcal{D}, Z, P, F \rangle$ be a belief set.

- A B-compatible goal set G is a set of tuples $\langle o, p, Y \rangle$ where $o \in \{!, ?, \$\}, p \in P$, and Y is a set of goal property specifiers.
- A B-compatible goal state T is a set of pairs $\langle o, q \rangle$ where $o \in \{!, ?, \$\}$ and q is a ground instance of some predicate $p \in P$.

□

The definition requires belief-goal compatibility; all of an agent's goals must be based upon predicates from its belief set. Not all combinations of these modal operators and predicates are sensible. For example, a goal of determination applied to an evaluable predicate will always succeed vacuously. Nonetheless, all possible combinations are permitted.

3.3 The Plan Model

A plan model consists of a set of plans, known as a *plan set*. Individual plans are specified as plan diagrams, which are denoted by a form of class icon. Plans may have attributes, but these may not be arbitrary, rather they are restricted to a set of predefined *reserved attributes*. A generic plan diagram appears in Figure 3. The lower section, known as the *plan graph*, is a state transition diagram, similar to an OO dynamic model. Unlike OO approaches, however, plans are not just descriptions of system behaviour developed during analysis. Rather, they are directly executable prescriptions of how an agent should behave to achieve a goal or respond to an event.

The elements of the plan graph are three types of node; *start states*, *end states* and *internal states*, and one type of directed edge; *transitions*. Start states are denoted by a small filled circle (●). End states may be *pass* or *fail* states, denoted respectively by a small target (◉) or a small no entry sign (⊘).

Internal states may be *passive* or *active*. Passive states have no substructure and are denoted by a small open circle (○). Active states have an associated *activity* and are denoted by instance icons. Activities may be subgoals, denoted by formulae from the agent's goal set, iteration constructs, including do and while loops, or in the case of a *graph state*, an embedded graph called a *subgraph*.

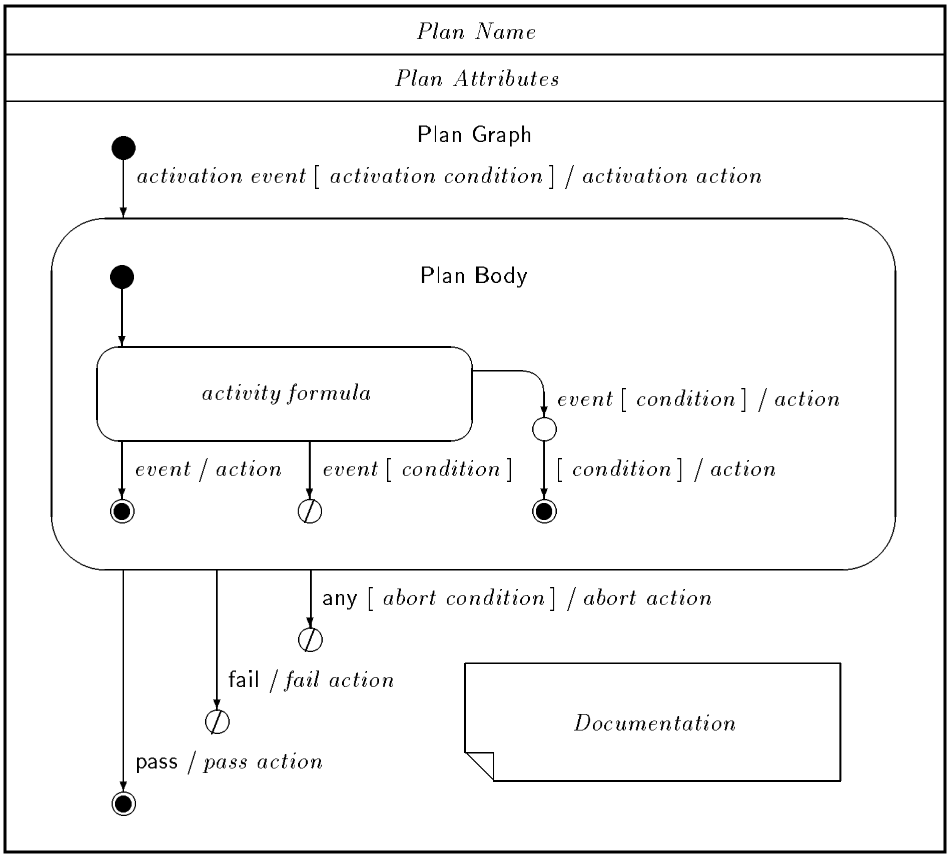


Fig. 3. Generic Plan Diagram

Events, conditions, and actions may be attached to transitions between states. In general, transitions from a state occur when the associated event occurs, provided that the associated condition is true. When the transition occurs any associated action is performed. Conditions are predicates from the agent's belief set. Actions include those defined in the belief set, and built-in actions. The latter include *assert* and *retract*, which update the belief state of the agent.

Failure Unlike conventional OO dynamic models, which are based upon Harel's statecharts [9], plan graphs have a semantics which incorporates a notion of failure. Failure within a graph can occur when an action upon a transition fails, when an explicit transition to a fail state occurs, or when the activity of an active state terminates in failure and no outgoing transition is enabled.

If the graph is the body of a graph state, then the activity of that state terminates in failure. If the graph is a plan graph, then the plan terminates in failure. If the plan has been activated to perform a subgoal activity in another plan, this may result in that activity terminating in failure, depending on the availability of alternative plans to perform the activity.

Plan Execution The initial transition of the plan graph is labelled with an *activation event* and *activation condition* which determine when and in what context the plan should be activated. The activation event may be a belief event which occurs when an agent's beliefs change or when certain external changes are sensed, leading to event-driven activation, or a goal event which occurs as a result of the execution of a subgoal activity in another plan, leading to goal-driven activation. An optional *activation action* permits an action to be taken when a plan is activated.

Each plan must be consistent with the belief and goal models of the agent class upon which it is defined; all predicates, goals and events occurring in the plan must be defined in the belief and goal models. The plan model may include plans that respond to different events or goals, plans that respond to the same event or goal in different, possibly intersecting contexts, and plans that respond to the same event and context. If multiple plans are applicable to a given event in a given context, they are activated in parallel if activation is event-driven, or sequentially until successful termination if activation is goal-driven.

Transitions from active states may be labelled with the events `pass` and `fail` which denote the success or failure of the activity associated with the state. Transitions from active states that are labelled with the event `any` may occur whenever their condition becomes true, allowing activities to be interrupted. A special case of this is the abort transition of a plan. Once the plan is activated, if at any time during the execution of its body the abort condition becomes true then it terminates in failure. The final transitions of the plan graph may be labelled with actions to be taken upon the success, failure or aborting of the plan.

Plan Properties Plans may have properties associated with them which are specified by reserved attributes. Some of these influence the way in which plan execution proceeds in response to a new goal or event.

- The *priority* property determines the order in which multiple concurrently active plans are executed.
- The *precedence* property determines the order in which plans that respond to a new goal are successively tried.
- The *noretry* property specifies that the goal should not be retried if this plan fails.

In the absence of any such properties, the set of plans applicable to a new goal is executed one by one in some arbitrary order until some plan succeeds. The precedence property allows the order in which this happens to be determined. The noretry property allows the system designer to ensure that if a certain plan is tried and fails, then no plans of lower precedence will be tried for that goal. Both these properties have important uses when modifying the behaviour of inherited plans; these are discussed in Section 4.2.

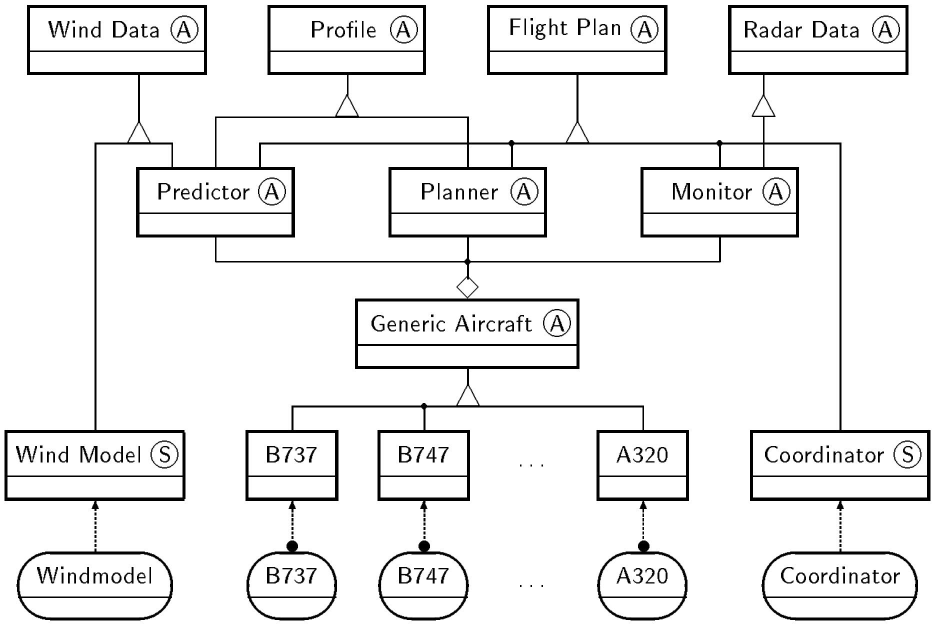


Fig. 4. ATM Agent Class and Instance Diagram

4 Modelling Multi-Agent Systems

Having described in detail the models that capture the state and behaviour of individual agents, we now proceed to describe how a multi-agent system is modelled. We begin by describing the agent model, which specifies the hierarchical relationship between different abstract and concrete agent classes, and identifies the agent instances which may exist within the system. In modelling agents by classes, the central problem is to give an appropriate semantics to relationships between classes such as inheritance, aggregation, and instantiation; we consider these issues in detail. Finally, we illustrate how the modelling technique may be employed to model layered architectures in two distinct ways.

4.1 The Agent Model

An Agent Model has two components; an *agent class model*, which defines abstract and concrete agent classes and captures the inheritance and aggregation relationships between them, and an *agent instance model*, which identifies agent instances and their properties. In systems containing only a small number of agent classes and instances they may combined into a single diagram. Figure 4 shows a simplified combined agent diagram from an Air Traffic Management application domain. Note that the attributes of agent classes do not appear in this diagram.

An agent class model is a directed, acyclic graph containing nodes denoting both *abstract* and *concrete* (instantiable) agent classes. Agent classes are represented by class icons, and abstract classes are distinguished by the presence of an adornment Ⓐ in the upper section of the icon. Edges in the graph denoting inheritance are distinguished by a triangle with a vertex pointing towards the superclass, and edges denoting aggregation by a diamond adjacent to the aggregate class. Other associations between agent classes are not allowed.

Agent classes may have attributes, but not operations. Attributes may not be arbitrary, rather they are restricted to a set of predefined *reserved attributes*. For example, each class must have associated belief, goal, and plan models, specified by the attributes beliefs, goals, and plans.

Multiple inheritance is permitted. Inheritance, as usual, denotes an *is-a* relationship, and aggregation a *has-a* relationship, but in the context of an agent model these relationships have a special semantics. Agents inherit and may refine the belief, goal, and plan models of their superclasses. Note that it is, for example, the set of plans which is refined, rather than the individual plans. Aggregation denotes the incorporation within an agent of subagents that do **not** have access to each other's beliefs, goals, and plans. Instantiation captures certain properties of agents, such as when they may come into existence and whether they may be cloned (multiply instantiated).

For example, in Figure 4, Monitor is an abstract agent which is a subagent of Generic Aircraft. Monitor is both a Radar Data agent and a Flight Plan agent. Predictor, another subagent of Generic Aircraft, is a Wind Data, aircraft Profile and Flight Plan agent. Monitor and Predictor do not share their beliefs about flight plans, Monitor has no beliefs about wind data or aircraft profiles, and Predictor has no beliefs about radar data. We consider the details of inheritance and aggregation further in the sections that follow.

Other attributes of an agent class include its belief-state-set and goal-state-set, which determine possible initial mental states. Particular elements of these sets may then be specified as the default initial mental state for the agent class, via the initial-belief-state and initial-goal-state attributes. For example, the belief model of the abstract aircraft Profile agent defines belief states corresponding to different aircraft types. A concrete aircraft agent such as B747 inherits these, but would only specify a particular instance, i.e., data values appropriate for a 747, in its belief state set and as its initial belief state.

4.2 Inheritance in the Agent Model

Inheritance is the fundamental relationship between agent classes which forms the basis of the structure of an agent class model. Each agent class is characterized by its belief, goal and plan models, which describe its internal state and behaviour. Inheritance allows one agent class to be defined as an extension or restriction of another; the belief, goal and plan models of the superclass may be extended and specialized in the subclass.

Inheritance of Beliefs An agent class inherits the belief models of its superclasses. As described in Section 3.1, a belief model consists of two components; a belief set, which describes the potential beliefs of an agent of that class (and their properties, such as whether or not they may change over time), and zero or more belief states – particular sets of beliefs – which may be used as possible values of the agent's initial mental state.

Formally, the elements of an agent class's belief set are a set of predicates and functions whose arguments are terms over a universe of predefined and user-defined type domains. These predicates and functions are directly derived from the belief class definitions of the agent class itself and combined with the belief models of its superclasses in the following manner:

- Types, predicates and functions directly derived from the belief classes and associations defined in the subclass belief diagrams are included in its belief model.
- Types defined in a superclass belief set are included provided no type with the same name is defined on the subclass, or in another superclass.
- Predicates and functions defined in a superclass belief set are included provided that all their attribute types are included.
- Predicate instances defined in a superclass belief state are included provided that the corresponding predicate definitions are included.

The net effect of this process is that definitions in a subclass may override those in a superclass, and incompatible definitions in different superclasses must be resolved by choosing in the subclass which of the inherited definitions to prefer. The latter can be done straightforwardly by defining a belief class on the subclass that inherits directly from the preferred definition, which can be referred to unambiguously by its *fully qualified name*, i.e., by including the superclass name as a prefix.

Within the set of predicates and functions derived from a particular set of belief diagrams, a reference to a particular predicate or function can always be uniquely resolved, given its name and the type of its first argument, because of the requirement that class, attribute, and association names in the belief model be unique. The method by which inherited belief models are combined preserves this desirable property.

This approach to the inheritance of beliefs allows an agent class to override and extend the belief models of its superclasses. An agent class may also specialize the elements of the belief models of its superclasses by redefining their properties, such as whether a function may change its value over time, whether a predicate has closed- or open-world semantics, etc.

Inheritance of Goals An agent class's goal model determines which goals and events may validly occur in the activation events of plans and as subgoals in activities in the bodies of plans. As described in Section 3.2, a goal model is defined by specifying for each modal goal operator the subset of the predicates in the belief set to which the operator may be applied. Events are handled in a similar manner. The goals and events thus defined in an agent class are combined with the goal models of its superclasses by inheriting those goals and events whose predicates were inherited into the subclass's belief model.

This approach to goal inheritance guarantees that belief-goal consistency is maintained. Just as was the case with beliefs, an agent class may also specialize goals and events inherited from its superclasses by redefining their properties, such as whether a goal should be retried on failure, etc.

Inheritance of Plans An agent class inherits the plan models of its superclasses. As described in Section 3.3, a plan model is a set of named plans, each of which has an activation event that specifies the event or goal to which it is relevant, and an activation condition that specifies the context in which it is valid. The plans associated with an agent class and its superclasses are combined as follows.

- All the plans defined in the subclass are included in its plan model.
- Plans in a superclass plan model are included provided they are compatible with the belief and goal models of the subclass, and no plan of the same name is defined on the subclass or in another superclass.

The net effect of this process is that the plan model of an agent class can affect the plans defined in its superclasses by:

- extending their coverage, by adding new plans to achieve existing goals in existing contexts, adding new plans to achieve existing goals in new contexts, or adding new plans to achieve new goals;
- excluding plans by redefining them in the subclass; and
- modifying the properties, such as priority or precedence, of inherited plans.

Note that the behaviour of inherited plans can also be affected indirectly by changes in the belief and goal models of the subclass. For example, a change to the properties of a predicate occurring in a goal may affect the way in which plans are executed in response to that goal (Section 3.3).

By default, inherited plans have lower precedence than those defined in a subclass, so that the order in which applicable plans are executed in response to a goal reflects the ordering of the agent inheritance hierarchy. This default precedence ordering may, however, be overridden, either by explicitly defined precedences, or by the mechanism of a *fully qualified goal*; a goal formula qualified by the name of an agent class. When searching for plans whose activation event matches such a goal, the search begins on the designated class.

When a plan is inherited from a superclass, its behaviour can be reused by a subclass plan, or specialized. For example, let A be an agent class that defines a plan P that responds to a goal ψ. We wish to define an agent class B that inherits from A, but restricts the context in which P may be activated by adding the conjunct ϕ to its activation condition. This may be done by defining a plan Q on B with the property noretry that responds to ψ and fails immediately if ϕ does not hold, else it posts the subgoal $\mathsf{A}::\psi$, activating P. As a result of the noretry property, P will never be directly activated by the goal ψ, even if Q fails.

4.3 Aggregation in the Agent Model

Aggregation is a secondary relationship between agent classes that allows their grouping together into a new class in a quite different way from (multiple) inheritance. The agents, called *subagents*, that form part of an aggregate are separate modules or name spaces; their belief, goal and plan models are quite independent. The aggregate class itself may also have belief, goal and plan models of its own. Subagents cannot directly affect each others beliefs, goals, and intentions; they interact by asynchronous message passing in exactly the same way as physically distinct agents[3].

The modelling technique we have developed allows a distinction to be drawn between two sorts of agent.

1. A *logical agent* is an encapsulation of state and behaviour, developed during the system design process, which may not even be independently actualizable.
2. A *physical agent* is the actualization of one or more logical agents, all of which share an identical extent in time, and coexist (in our implementation) within a single process.

Aggregation is a system structuring construct that allows the boundaries of physical agents in a multi-agent system to be set differently from those of individual logical agents. In effect, physical agents may themselves be simple multi-agent systems. For example, in Figure 4 each physical aircraft agent consists of a logically independent predictor, planner, and monitor subagent.

Just as in an object oriented system various processes exist, each of which contain many objects, so in an agent system processes (physical agents) may exist, each of which contains internally many (logical) agents. The question "How many agents are there in the system?" now has two different answers, depending on whether logical or physical agents are being considered. Decisions about the number and type of logical agents in a system may be decoupled from decisions about their physical realization, and deferred to a late stage of the system design process.

One consequence of the flexibility that these system structuring constructs offer is that complex, layered agent architectures may be viewed as systems of simpler, coupled individual agents. Moreover, this may be extended hierarchically, with individual layers themselves implemented as multi-agent systems. The ability to model such architectures within a uniform framework is one of the strengths of our approach.

4.4 Instantiation in the Agent Model

Instantiation, the third relationship that occurs in the agent model, is a relationship between an agent class and one or more instances of that class. It is used to capture certain properties of agents, such as when they may come into existence and whether they may be cloned (multiply instantiated).

An agent instance model is an instance diagram which defines both the *static agent set* – the set of agents that are instantiated at compile-time – and the *dynamic agent set* – the set of agents that may be instantiated at run-time. The former are distinguished by the adornment Ⓢ in the upper section of the icon.

[3] Communication between subagents may however be both faster and more reliable than between agents in separate processes.

Each agent instance is specified by an instance icon linked to a concrete agent class by an instantiation edge, represented as a dotted vector from instance to class. Static instances must be named, but the naming of dynamic instances may be deferred till their instantiation. A multiplicity notation at the instance end of the instantiation link may be used to indicate whether a dynamic class may be multiply instantiated.

The initial mental state of an agent instance may be specified by the initial-belief-state and initial-goal-state attributes, whose values are particular elements of the belief and goal state sets of the agent class. If not specified, the defaults are the values associated with the agent class. For dynamic agent instances these attributes may be overridden at the time the agent is created.

4.5 Implementing layered architectures

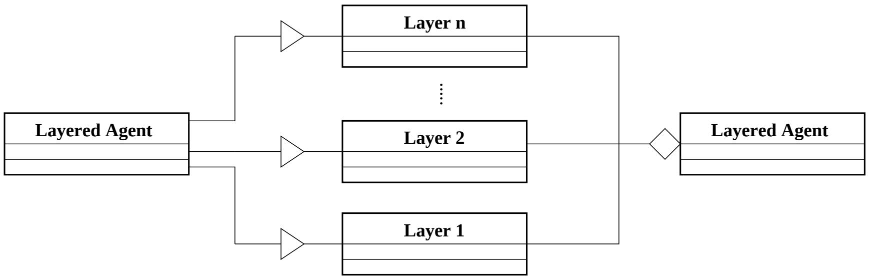


Fig. 5. Combining agent layers by inheritance and aggregation

The ability of our agent modelling technique and architecture to support physical agents that are themselves multi-agent systems highlights a particular property of the architecture; it's ability to straightforwardly implement more specialized architectures.

For example, consider a layered agent architecture, such as Interrap [13] (vertical) or the Touring Machine [5] (horizontal). Suppose the state and behaviour of each of these layers is specified as a separate logical agent class. There are then two straightforward ways to combine them in our framework, as illustrated in Figure 5.

Firstly, we can combine them by creating an agent class that inherits from each layer class. This approach leads to a tight coupling between the layers, in fact it requires that the beliefs, goals and plans of each layer class be disjoint (apart from those that serve to implement interlayer interfaces) since otherwise they would interfere with each other.

Secondly, we can employ aggregation to create an agent class that contains each layer as a subagent. With this approach, there are no constraints on the content of the layers, since each exists in a separate namespace. Layers are loosely coupled; they do not share state, and communication is by message passing, in exactly the same way as between physically distinct agents.

5 An Agent-Oriented Design Methodology

A methodology to support design and specification of agent systems should provide a clear conceptual framework that enables the complexity of the system to be managed by decomposition and abstraction. Our agent-oriented methodology advocates the decomposition of a system based on the key *roles* in an application. The identification of roles and their relationships guides the specification of an *agent class hierarchy*; agents are particular instances of these classes. Analysis of the *responsibilities* of each agent class leads to the identification of the *services* provided and used by an agent, and hence its external interactions. Consideration of issues such as the creation and duration of roles and their interactions determines *control relationships* between agent classes.

5.1 Developing the External Models

These details are captured in the Agent and Interaction models described in Section 2. The methodology for their elaboration and refinement can be expressed as four major steps.

1. Identify the roles of the application domain. There are several dimensions in which such an analysis can be undertaken; roles can be organizational or functional, they can be directly related to the application, or required by the system implementation. Identify the lifetime of each role. Elaborate an agent class hierarchy. The initial definition of agent classes should be quite abstract, not assuming any particular granularity of agency.
2. For each role, identify its associated responsibilities, and the services provided and used to fulfill those responsibilities. As well as services provided to/by other agents upon request, services may include interaction with the external environment or human users. For example, a responsibility may require an agent to monitor the environment, to notice when certain events occur, and to respond appropriately by performing actions, which may include notifying other agents or users. Conversely, a responsibility may induce a requirement that an agent be notified of conditions detected by other agents or users. Decompose agent classes to the service level.
3. For each service, identify the interactions associated with the provision of the service, the performatives (speech acts) required for those interactions, and their information content. Identify events and conditions to be noticed, actions to be performed, and other information requirements. Determine the control relationships between agents. At this point the internal modelling of each agent class can be performed.
4. Refine the agent hierarchy. Where there is commonality of information or services between agent classes, consider introducing a new agent class, which existing agent classes can specialize, to encapsulate what is common. Compose agent classes, via inheritance or aggregation, guided by commonality of lifetime, information and interfaces, and similarity of services. Introduce concrete agent classes, taking into account implementation dependent considerations of performance, communication costs and latencies, fault-tolerance

requirements, etc. Refine the control relationships. Finally, based upon considerations of lifetime and multiplicity, introduce agent instances.

Roles, responsibilities, and services are just descriptions of purposeful behaviours at different levels of abstraction; roles can be seen as sets of responsibilities, and responsibilities as sets of services. Services are those activities that it is not natural to decompose further, in terms of *the identity of the performer*. The roles initially identified serve as a starting point for the analysis, not an up-front decision about what agents will result from the process of analysis.

Once roles have been decomposed to the level of services and internal modelling performed, a fine-grained model of agency has been produced. When this is recomposed in accordance with the considerations mentioned above, the concrete agents which result may reflect groupings of services and responsibilities that differ from the original roles. The identification of agent boundaries is deferred until the information and procedures used to perform services have been elaborated. This results in concrete agents whose internal structure is inherently modular.

Simple service relationships and interactions between agents could be represented as associations within the agent model, but we have chosen to describe them in a separate model. Modelling of agent interactions is currently the subject of intensive research, and many modelling techniques, often quite complex, have been proposed and developed (see, for example, [1, 3, 4, 6, 7, 8, 17]). They address issues from information content and linguistic intent through to protocols for coordination and negotiation. We do not hold a strong view on the general suitability of particular techniques for modelling interactions, hence our methodology and modelling framework is designed to allow the selection of models appropriate to the application domain.

The interaction model also captures control relationships between agents, such as responsibilities for agent creation and deletion, delegation, and team formation. Modelling techniques for these relationships are the subject of ongoing research.

5.2 Developing the Internal Models

Our methodology for the development of these models begins from the services provided by an agent and the associated events and interactions. These define its purpose, and determine the top-level goals that the agent must be able to achieve. Analysis of the goals and their further breakdown into subgoals leads naturally to the identification of different means, i.e., plans, by which a goal can be achieved.

The appropriateness of a given plan, and the manner in which a plan is carried out, will in general depend upon the agent's beliefs about the state of the environment and possibly other information available to the agent, i.e., the agent's belief context. This may also include certain beliefs which represent working data. A context is represented in terms of various data entities and their relationships. Analysis of contexts results in the elaboration of the beliefs of an agent. To summarize, the methodology for internal modelling can be expressed as two steps.

1. Analyze the means of achieving the goals. For each goal, analyze the different contexts in which the goal has to be achieved. For each of these contexts, decompose each goal into activities, represented by subgoals, and actions. Analyze in what order and under what conditions these activities and actions need to be performed, how failure should be dealt with, and generate a plan to achieve the goal. Repeat the analysis for subgoals.
2. Build the beliefs of the system. Analyze the various contexts, and the conditions that control the execution of activities and actions, and decompose them into component beliefs. Analyze the input and output data requirements for each subgoal in a plan and make sure that this information is available either as beliefs or as outputs from prior subgoals in the plan.

These steps are iterated as the models which capture the results of analysis are progressively elaborated, revised, and refined. Refinement of the internal models feeds back to the external models; building the plans and beliefs of an agent class clarifies the information requirements of services, particularly with respect to monitoring and notification. Analyzing interaction scenarios, which can be derived from the plans, may lead to the redefinition of services.

Unlike object-oriented methodologies, the primary emphasis of our methodology is on roles, responsibilities, services, and goals. These are the key abstractions that allow us to manage complexity. We analyze the application domain in terms of what needs to be achieved, and in what context. The focus is on the end-point that is to be reached, rather than the types of behaviours that will lead to the end-point, which are the primary emphasis of OO methodologies.

Although this might seem a small paradigm shift, it is quite subtle and leads to a substantially different analysis. This is because goals, as compared to behaviours or plans, are more stable in any application domain. Correctly identifying goals leads to a more robust system design, where changes to behaviours can be accommodated as new ways of achieving the same goal. In other words, a goal-oriented analysis results in more stable, robust, and modular designs.

The context-sensitivity of plans provides modularity and compositionality; plans for new contexts may be added without changing existing plans for the same goal. This results in an extensible design that can cope with frequent changes and special cases, and permits incremental development and testing.

6 Conclusions

The primary contribution of this paper has been to provide the elements of a rigorous framework for modelling and specifying complex multi-agent systems. We have presented modelling techniques to describe the external and internal perspective of multi-agent systems, based on a BDI architecture, which build upon and adapt existing, well-understood object-oriented models. Our agent-oriented methodology, with its emphasis on roles, responsibilities, services, and goals, permits a fine-grained analysis that allows agent boundaries to be chosen flexibly and results in system designs that are robust, modular, and extensible.

We have given a semantics for inheritance, aggregation and instantiation relationships amongst agent classes and instances which provides powerful and flexible mechanisms for enforcing modularity of state and behaviour within agents, and for sharing them between agents. Related beliefs, goals, and plans may be encapsulated in separate classes which may then be grouped together, by aggregation or inheritance. The ability to take an agent class and refine it by the addition of further beliefs, goals, or plans provides a compositional framework for system design and encourages re-use. Encapsulation makes more tractable the task of analyzing interactions between plans, which is crucial to the process of design verification.

By building upon and adapting existing, well-understood techniques, we aim to take advantage of their maturity to develop models and a methodology which will be easily learnt and understood by those familiar with the OO paradigm. This is important if the design, implementation, and maintenance of multi-agent systems is to be carried out by software analysts and engineers rather than research scientists, and if they are to be successfully applied on a significant scale to commercial and industrial applications.

References

1. Mihai Barbuceanu and Mark S. Fox. COOL: A language for describing coordination in multi-agent systems. In *Proceedings of the International Conference on Multi-Agent Systems, ICMAS-95*, San Francisco, CA, 1995.
2. Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994.
3. Jennifer Chu-Carrol and Sandra Carberry. Generating information-sharing subdialogues in expert-user consultation. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI-95*, pages 1243–1250, Montréal, 1995.
4. Philip R. Cohen and Hector J. Levesque. Communicative actions for artificial agents. In *Proceedings of the International Conference on Multi-Agent Systems, ICMAS-95*, San Francisco, CA, 1995.
5. Innes A. Ferguson. Integrated control and coordinated behaviour: a case for agent models. In *Intelligent Agents: Proceedings of the ECAI-94 Workshop on Agent Theories, Architectures, and Languages. LNAI 890*, Amsterdam, 1995. Springer Verlag.
6. Tim Finin *et al.* Specification of the KQML agent communication language. Technical report, DARPA Knowledge Sharing Initiative, External Working Group, 1992.
7. Barbara J. Grosz and Candace L. Sidner. Plans for discourse. In P. R. Cohen, J. Morgan, and M. E. Pollack, editors, *Intentions in Communication.* MIT Press, Cambridge, MA, 1990.
8. Afsaneh Haddadi. *Reasoning About Interactions in Agent Systems: A Pragmatic Theory.* PhD thesis, University of Manchester Institute of Science and Technology, United Kingdom, 1995.
9. D. Harel and C. Kahana. On statecharts with overlapping. *ACM Transactions on Software Engineering and Methodology*, 1(4), 1992.
10. David Kinny. *The Distributed Multi-Agent Reasoning System Architecture and Language Specification.* Australian Artificial Intelligence Institute, Melbourne, Australia, 1993.

11. David Kinny and Michael Georgeff. Commitment and effectiveness of situated agents. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence, IJCAI-93*, pages 82–88, Sydney, 1991.
12. David Kinny, Michael Georgeff, and Anand Rao. A methodology and modelling technique for systems of BDI agents. In *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW '96. LNAI 1038*, Eindhoven, The Netherlands, 1996. Springer Verlag.
13. J. P. Müller, M. Pischel, and M. Thiel. Modelling reactive behaviour in vertically layered agent architectures. In *Intelligent Agents: Proceedings of the ECAI-94 Workshop on Agent Theories, Architectures, and Languages. LNAI 890*, Amsterdam, 1995. Springer Verlag.
14. J. Y. C. Pan and J. M. Tenenbaum. An intelligent agent framework for enterprise integration. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6), 1991.
15. Anand S. Rao and Michael P. Georgeff. An Abstract Architecture for Rational Agents. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning, KR '92*, pages 439–449, Boston, MA, 1992.
16. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliifs, NJ, 1991.
17. Candace L. Sidner. An artificial discourse language for collaborative negotiation. In *Proceedings of the Twelfth National Conference on Artificial Intelligence, AAAI-94*, pages 814–819, Seattle, WA, 1994.