A Methodology and Modelling Technique for Systems of BDI Agents

January, 1996

Technical Note 58

By:

David Kinny
Michael Georgeff
Anand Rao
Australian Artificial Intelligence Institute
171 Latrobe Street
Melbourne 3000, Australia
{dnk,georgeff,anand}@aaii.oz.au

This paper appears in Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW’96, published by Springer as Lecture Notes in Artificial Intelligence 1038.
This work was supported in part by the Australian Research Council.
Abstract

The construction of large-scale embedded software systems demands the use of design methodologies and modelling techniques that support abstraction, inheritance, modularity, and other mechanisms for reducing complexity and preventing error. If multi-agent systems are to become widely accepted as a basis for large-scale applications, adequate agent-oriented methodologies and modelling techniques will be essential. This is not just to ensure that systems are reliable, maintainable, and conformant, but to allow their design, implementation, and maintenance to be carried out by software analysts and engineers rather than researchers. In this paper we describe an agent-oriented methodology and modelling technique for systems of agents based upon the Belief-Desire-Intention (BDI) paradigm. Our models extend existing Object-Oriented (OO) models. By building upon and adapting existing, well-understood techniques, we take advantage of their maturity to produce an approach that can be easily learnt and understood by those familiar with the OO paradigm.
1 Introduction

Managing complexity is perhaps the most challenging task facing designers of large-scale embedded software systems. It is now widely accepted that the construction of reliable, maintainable, and extensible systems that conform to their specifications requires, *inter alia*, the use of design methodologies and modelling techniques that support abstraction, structuring, inheritance, modularity, and other mechanisms for managing their inherent complexity.

Perhaps foremost amongst the methodologies that have been developed for the design, specification, and programming of conventional software systems are various Object-Oriented (OO) approaches, 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*.

OO methodologies provide a uniform paradigm which is useful across a range of system scales and implementation languages. They 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. The OO design and development environment is well supported by diagram editors and visualization tools. There is even, perhaps, a convergence of viewpoints amongst the major proponents of different variations on the OO theme.

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. It supports a flourishing research community which has made substantial progress in recent years in providing a theoretical and practical understanding of many aspects of agents and multi-agent systems. Currently, many flowers bloom — there is a multitude of viewpoints on what exactly constitutes an agent, on how they should be structured, and how collections of agents interacting with each other and the environment can be used to implement complex systems. Despite this healthy lack of consensus, the benefits of implementing agent systems are little disputed, and several agent architectures have progressed to being usable technologies.

As yet, however, there are not many examples of the successful application of agent system technologies on a significant scale. In part this is due to the absence of mature languages and software tools, but our experience suggests that the absence of methodologies that allow system complexity to be effectively managed is a greater obstacle. OO methodologies are not directly applicable to agent systems — agents are usually significantly more complex than typical objects, both in their internal structure and in the behaviours they exhibit. We believe that if multi-agent systems are to become widely accepted as a basis for large-scale commercial and industrial applications, adequate design methodologies and modelling techniques will be essential. This is not just to guarantee that such systems are sufficiently reliable, maintainable, and conformant, but to allow their design, implementation, and maintenance to be carried out by software analysts and engineers rather than AI researchers. Others have reached similar conclusions about the need for familiar, intuitive modelling techniques [Pan and Tenenbaum, 1991].

Our research program has centred upon the design, implementation and theoretical understanding of a particular Belief-Desire-Intention (BDI) agent architecture [Georgeff and Lansky, 1986, Rao and Georgeff, 1991] which has now achieved considerable maturity. Recently, we have been applying this technology as the basis of a number of medium to large-scale software systems. The application domains of these systems include air-traffic management [Ljungberg, 1992], air-combat simulation [Tidhar et al., 1995], and business process management.

Realizing that existing formalisms for describing and reasoning about agents do not provide adequate support for the *process* of agent design, we have been attempting to develop suitable
methodologies and models. Our approach, pragmatically motivated, has been to explore how existing OO modelling techniques can be extended to apply to BDI agent systems. By building upon and adapting existing, well-understood techniques, we take advantage of the maturity of the OO approach and aim to develop models and a methodology that will be easily learnt and understood by those familiar with the OO paradigm.

In this paper we present an overview of our agent-oriented methodology, focussing upon the models we have developed. Elsewhere [Kinny et al., 1995], we provide a more detailed description of the methodology and the process of multi-agent system design. In the sections that follow we will outline the models and methodology, present an application domain which illustrates their use, describe particular models in more detail, and compare our approach to other work in this area.

2 An Agent-Oriented 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. OO methodologies [Booch, 1994, Rumbaugh et al., 1991] advocate the decomposition of a system by identification of the key object classes in the application domain, focussing upon 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.

2.1 Agents from the External Viewpoint

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.
These details are captured in two models.

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.

These models are largely independent of our BDI architecture. 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, [Barbuceanu and Fox, 1995, Chu-Carrol and Carberry, 1995, Cohen and Levesque, 1995, Finin et al., 1992, Grosz and Sidner, 1990, Haddadi, 1995, Sidner, 1994]). 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.

2.2 Agents from the Internal Viewpoint

The BDI paradigm provides a “strong” notion of agency; agents are viewed as having certain mental attitudes, Beliefs, Desires and Intentions, which represent, respectively, their informational, motivational and deliberative states. In our BDI architecture an agent can be completely specified by the events that it can perceive, the actions it may perform, the beliefs it may hold, the goals it may adopt, and the plans that give rise to its intentions. These are captured, for each agent class, by the following models.

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.

   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 [Kinny, 1993], 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

---

1 We distinguish beliefs from the notion of knowledge, as defined, for example, in the literature on distributed computing, as beliefs are only required to provide information on the likely state of the environment. The distinction between desires and goals, while important from a philosophical perspective, is not significant in this context.
consistently, and that plan selection and execution proceeds in a manner which reflects certain notions of rational commitment [Kinny and Georgeff, 1991, Rao and Georgeff, 1992]

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 methodolo-

gies.

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.
3 An Air-Traffic Management System

In this section we describe informally the structure of the air-traffic management (ATM) system which we have developed. The system is responsible for assisting a human Flow Controller to determine the landing sequence of aircraft on multiple runways at a single airport so as to maintain safety and other constraints while minimizing delay and congestion.

Following the methodology in the previous section, we arrived at a system design consisting of three permanent agents with the roles of Coordinator, Sequencer and Windmodel, and a variable number of aircraft agents, each of which is associated with a particular flight and exists only during the time that the aircraft is under the control of the flow controller.

The Coordinator is responsible for the creation and deletion of Aircraft agents and the distribution to them of initial flight plans. The Sequencer is responsible for determining landing time assignments, which it does by interacting with aircraft agents and the flow controller. The Windmodel maintains a 4-dimensional model of past, present and future wind conditions in the controlled airspace.

An Aircraft agent consists of three active components; a Predictor, Monitor and Planner. Conceptually they were modelled as separate roles, but because of their common lifetimes and close interactions they were aggregated into a single concrete agent. They are generic, i.e., identical in all aircraft agents. Aircraft agents also contain a performance profile component specific to the aircraft type.

The Predictor is responsible for a number of services. Its primary service is to compute the expected time of arrival (TOA) at the waypoints specified in the flight plan, including the final landing point. Inputs to the computation are the flight plan, which may be modified during the flight, the aircraft performance profile, which is determined by the aircraft type, and wind conditions; to obtain the latter it uses the services of the Windmodel agent. Different wind conditions may apply at different stages of the flight, primarily due to altitude change. The TOA computation produces both estimated times and performance envelopes, which are the earliest and latest times that the aircraft could arrive at the waypoint while remaining within its permitted operating profile. The Predictor provides this information to the Monitor and Planner.

The Monitor receives 3-dimensional location information derived from radar data and compares actual TOAs with those predicted. If significant deviations occur, it analyzes the reason for the occurrence and notifies the Predictor and Planner. Deviations may be due to the aircraft not following the planned path, not flying at the assigned altitude, not holding the planned air-speed, or inaccuracies in the wind information. The first two of these are directly detectable from the radar data. To distinguish the latter two the behaviour of multiple aircraft must be compared, so Monitor requests Windmodel to perform this global analysis.

When the Sequencer assigns (or revises) the aircraft's landing time assignment (LTA), the Planner is requested to construct a set of plans that will allow the aircraft to land at that time. A plan here is the future trajectory, air-speed and altitude profile of the aircraft. Acceptable plans are highly constrained; trajectories are restricted, holding points are limited, air-speeds must be multiples of 10 knots; in general, the issuing of instructions to the pilot should be minimized. The Planner uses various different strategies, algorithmic and heuristic, to produce these plans; moreover, the choice of strategy depends on the stage of the flight. The Planner then sends the set of plans to the flow controller.

Once the flow controller has chosen which plan to adopt, the Planner responds to deviations detected by the Monitor by determining whether the aircraft can still meet its LTA. If so, the current plan is modified as required. If not, the Planner notifies the Sequencer, which computes a new LTA, and the cycle repeats.
4 The Modelling Technique

As mentioned previously, OO modelling techniques are based upon an object model which employs classes and instances uniformly at all levels of abstraction. The dynamic and functional models serve as a description of object behaviour which guides the refinement of the object model.

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 meaning of relationships such as association, inheritance and instantiation is 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 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, 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 or text files for input to the compilation process that produces an executable system.

In the following sections we present in more detail the features of the agent, belief, and plan models. A full description appears elsewhere [Kinny and Georgeff, 1995].

4.1 Agent Models

An Agent Model has two components.

1. An Agent Class Model - a set of class diagrams which define abstract and concrete agent classes and capture the inheritance and aggregation relationships between them.

2. An Agent Instance Model - a set of instance diagrams that identify agent instances.

In systems containing only a small number of agent classes and instances they may combined into a single diagram. Figure 1 shows a simplified combined agent diagram from the ATM application domain. Note that the attributes of agent classes do not appear in this diagram.

4.1.1 Agent Class Models

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 the adornment $A$ 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 user-named data items, rather they are restricted to a set of predefined reserved attributes. For example, each class may 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.

For example, in Figure 1, 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.

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.

Abstract agent classes, aggregation, and inheritance provide powerful mechanisms for enforcing modularity of beliefs, goals, and plans within agents, and for sharing them between agents. Related beliefs, goals, and plans may be encapsulated in separate abstract classes which may then, by aggregation or inheritance, be grouped together to form a concrete agent class. Decisions about agent boundaries may be deferred to a late stage of the design process. 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.
Figure 2: ATM Belief Classes and Derived Predicates and Functions

### 4.1.2 Agent Instance Models

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.

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.
4.2 Belief Models

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.

4.2.1 Belief Sets

Formally, a belief set is a set of typed predicates whose arguments are terms over a universe of predefined and user-defined function symbols. These predicates and functions are directly derived from the class and instance definitions in the belief set diagrams. The classes and instances defined therein correspond, in many cases, to real entities in the application domain, but, unlike an OO object model, the definitions do not define the behaviours of these entities. This is because they are not implementations of the entities, rather, they represent an agent’s beliefs about those entities.

A class in a belief set diagram serves to define the type signatures of attributes of an 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. Attributes, which define binary predicates, are specified in the usual way. If an attribute is never undefined, 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 an implicit first argument to the derived function or predicate.

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 2 shows two associated belief classes from the ATM domain, and the predicates and 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 of 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.

Properties may also be associated with classes, instances and associations, and may represented either by property lists or by adornments. For example, in Figure 2, the attribute flight has the property static, indicating that its value may not change, the attribute lta has the property optional, indicating that its value may be undefined, and the predicate on-path has the property eval, indicating that it is computed.

4.3 Plan Models

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. 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*.

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 agent’s belief state.

**4.3.1 Failure**

Unlike conventional OO dynamic models, which are based upon Harel’s statecharts [Harel and Kahana, 1992], 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.

4.3.2 Plan Execution

The initial transition of the plan graph is labelled with an 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. 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. An optional activation action permits an action to be taken when a plan is activated.

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.

5 Comparison and Conclusions

A number of agent-oriented systems based on BDI architectures have been developed, including PRS [Georgeff and Lansky, 1986], COSY [Burmeister and Sundermeyer, 1992], GRAPE* [Jennings, 1993], ARCHON [Jennings et al., 1995], INTERRAP [Muller et al., 1995], and dMARS [Kinny, 1993]. Likewise, agent-oriented languages, such as AGENT0 [Shoham, 1991, Shoham, 1993] and PLAca [Thomas, 1995], have been proposed as languages for programming agents.

However, there is very little work on how to analyze and design agent-oriented systems. Brazier et al. [Brazier et al., 1995] discuss a formal specification framework for multi-agent system design, called DESIRE (DESign and Specification of Interacting REasoning components). DESIRE emphasizes hierarchical task-based decomposition. It captures the task structure, sequences of subtasks, information exchange between tasks, knowledge decomposition, and role delegation. As with our agent model, DESIRE provides a compositional framework for modelling agents. However, the plan model discussed herein allows greater expressive power for describing task structure. In addition, our methodology emphasizes the importance of goals and services as opposed to tasks.

The object-oriented methodology proposed by Wirfs-Brock et al. [Wirfs-Brock et al., 1990] views a subsystem as a set of object classes which collaborate to fulfill a set of responsibilities. The notion of responsibility conveys a sense of the purpose of the objects in a subsystem and is characterized by the services provided by the subsystem. Thus, our notion of role is very similar to the notion of subsystem. The crucial difference is that, in an object-oriented paradigm, there is no programming construct that supports the realization of a subsystem whereas, in the agent-oriented paradigm, agents are used to realize particular instances of roles which then take on a separate identity and existence.
Within the object-oriented community, the notion of agents has also been used to capture user requirements. Dubois et al. [Dubois et al., 1993] employ temporal logic to specify the behaviours of agents, called agent level modelling, and the interactions between them, called society level modelling.

Burkhard [Burkhard, 1995] considers a number of design choices for agent-oriented languages and the consequences of these for programming open systems. Our methodology and agent modelling formalism provide one way of achieving many of the desired features for a class of such languages.

In summary, the primary contribution of this paper has been to provide the elements of a rigorous framework for analyzing and building complex, distributed, multi-agent systems. We have presented a methodology and modelling technique to describe the external and internal perspective of multi-agent systems based on a BDI architecture, and have illustrated the approach using an implemented air-traffic management system. 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. Our agent system modelling technique builds upon and adapts existing, well-understood object-oriented models.

References


