A Component Agent Model - from Theory to Implementation

Ladislau Bölöni and Dan C. Marinescu
Computer Sciences Department, Purdue University
West Lafayette, In, 47907, USA
(boloni, dcm@cs.purdue.edu)

November 1, 1999

Abstract

This paper presents a theoretical model for software agents with an emphasis of immediate applicability. While clearly influenced by the BDI model, this model describes the behavior of agents in terms familiar to programmers using an object oriented / component based style. While weaker in requirements that BDI, the model can still be used to reason about agents and to model their interactions. Then we present a practical implementation of the model and discuss the benefits of a solid theoretical foundation for agent development.

1 Introduction

In the days when the software development time for a product is sometimes measured in weeks, it may not be fashionable to argue in favor of theoretical model for software development. The reason we are inclined to do so is our belief that ad-hoc solutions ultimately lead to chaotic software design, large number of bugs, and unpredictable behavior.

The Belief-Desire-Intention (BDI) model [5, 7] is the most rigorous theoretical model for agents available now. However, software tools to easily map the theoretical concepts of BDI into working agents are largely unavailable. The significant effort needed by developers to adopt a programming style based upon logical statements, when currently the de-facto programming languages for agent development are object-oriented languages like Java or C++ is one of the reasons for this state of affairs. Still a number of agent systems like JACK [3] are based upon the BDI model while providing a familiar programming style based on extensions to the Java language. Most agent systems however are designed as extensions of the object-oriented model without having a theoretical model for the notion of agents, e.g. the AgentBuilder [8].

This paper introduces a constructive model for software agents, and a practical implementation of it. A non-exhaustive list of requirements for the model are:

- The model should be complete enough to allow automated reasoning about agents. The reasoning ability should *degrade gracefully* with the accuracy of implementation.
- The concepts of the model should map easily into the object-oriented / component-based programming style.
- The model should be naturally extensible to handle multi-agent systems.
- The model should avoid to formalize concepts that do not lend themselves to reasonably easy to use and efficient implementations. These concepts should be treated as "black boxes", leaving the programmer implementing the agent the freedom to come up with an ad-hoc solution. At the same time, these black boxes may be formalized later, as the software engineering methods evolve.

This last goal leads to a model weaker than BDI - we are considering it sometimes as "poor man's BDI". It is less ambitious then the BDI model, and ignores some of the domains covered by BDI, like planning or the advanced models of time. Existing models like BDI or statecharts [17] definitely influenced us in designing our model. We used this model as the theoretical underpinning of the agent framework in the Bond distributed object system. Our experience shows that developers coming from an object-oriented school can accept relatively easily the requirements of the model, and the final result are agents easier to analyze and represent than those created without theoretical foundation.

2 An object model for agents

There are two views of the world we may consider, (a) the *view of an external observer* with a perfect knowledge of the world, who does not need take any action to acquire its knowledge and (b)the *view of an agent*, with an incomplete and imperfect knowledge who needs to take actions to improve its knowledge about the world and these very actions change the state of the world.

When we define the goal of the agent, we are thinking in terms of the first view, because we want to modify the true state of the world. However, the actions taken by the agent can be defined only in terms of the second view, the only one available to the agent. Sometimes we have to define pairs of concepts, like the behavior function of the agent (as objectively seen from outside) and the strategy function of the agent (as it was intended from inside).

2.1 The world of agents

We assume a *world* composed of a set of objects $W = w_1, w_2, ...$ and a *time axis* t. Every object w_i has a set of properties represented as functions of time $p_{ij}(t)$. The set of properties of all objects at every given time specifies the *state of the world*. The state of the world evolves in time as events unfold. An *event* α is a change of the properties of a set of objects in a time interval $[t_{start}, t_{end}]$. We call an event an *action* if we can clearly identify the cause of the event. For example we can clearly identify the events caused by an agent, as being the actions of the agent. However changes in the properties of the world occur without us being able to identify a cause for them, and these will be treated as events. For the purpose of this paper all events and actions are considered instantaneous ($t_{start} = t_{end}$).

An event can be represented as the set of changes it induces in the world

$$\alpha_k = \Delta_k p_{i_1 j_1}, \Delta_k p_{i_2 j_2}, ...$$

Here $\Delta_k p_{xy} = p'_{xy} - p_{xy}$ where p_{xy} is the value of property y of the object W_x before the event while p'_{xy} is the value of the same property after the event.

From the point of view of the outside world an agent is an object $A \in W$ which generates discrete actions α in time. We model an agent as a function on the world (usually called *agent behavior function*)

$$ABF : (W * \times A*) \rightarrow \alpha*$$

The agent can generate only a limited set of actions $C(A)$, called the *capabilities of the agent* defined as

$$C(A) = \{\alpha \mid \ \exists W \subset W* \quad ABF(W, A) = \alpha\}$$

Definition 1 *We call a* **desired state of the world** *a subsets of the possible states of the world when the boolean function $D : W \rightarrow \{true, false\}$ has the value true.*

The desired state of the world is practically defined on the set of the properties of the objects composing the given world. We call a *controlled world* a world where all events are actions of the agent. We call a *weak non-controlled world* one where there are events which are not actions of the agent. To define a *strong non-controlled world* we introduce the concept of an intended action. The agent generates an intended action α. However the changes in the world correspond to a different event α'. We call a world strong non-controlled if $\alpha \neq \alpha'$. [1] The agent behavior function partitions the world of agents into equivalence classes. Informally, two agents are equivalent if their behavior is identical and they are indistinguishable to the world.

Definition 2 *Two agents are* **equivalent** *if they have the same agent behavior function.*

Definition 3 *We call* **abstract agent building problem** *finding an agent behavior function $ABF(W)$ such that from any state W it will reach another state of the world W_{goal} such that $D(W_{goal}) = true$ in finite time.*

[1] BDI has a similar concept.

2.2 The basic agent model AM_0

In the previous section we defined facts assuming a perfect knowledge of the world. In this section our definitions refer only to the information available to the agent and assume a strongly nondeterministic world.

We assume that the knowledge of the agent about the world is captured in a *model of the world* $M = m_1, m_2 \ldots$. Each element of the model is an approximation of a corresponding object in the world $m_{w_i} \approx w_i$, and their properties are approximations of the properties of the corresponding object. An approximation of time can also be member of the model.

Definition 4 *We call* **knowledge representation** *a function $K : (W*) \to M$ which maps objects of the model world into objects of the world.*

From the agent's point of view, we cannot use the desired state function, because it is defined on the properties of the world, not accessible to the agent. The **agenda of the agent** is a boolean function applied to the model of the world:

$$Ag : M* \to \{true, false\}$$

The agenda is an approximation of the desired state function: $Ag(M) \approx D(W)$.

Similarly, in a constructive model of the agent we cannot use the agent behavior function, instead we define the *strategy* of an agent A as a function which maps the model of the world into a set of intended actions

$$S : (M* \times Ag*) \to \alpha*$$

The agent behavior function is practically a distorted version of the strategy after the mapping from intended actions to real events.

Definition 5 *The AM_0 model of an agent is the triplet (Ag, S, M) consisting of an agenda Ag, a strategy S and a model of the world M.*

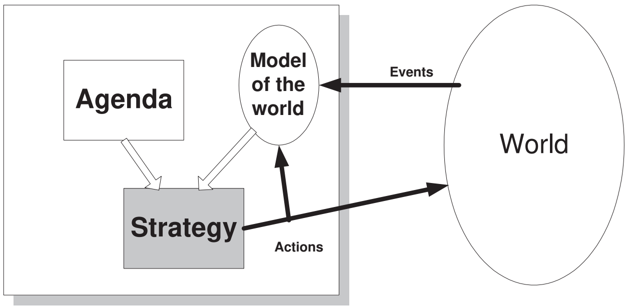

Figure 1: The AM_0 model of agents. The agent consists of the strategy, agenda and model of the world. Events are changing the world and the model. If the originator of an event is the strategy, we call it an action.

Definition 6 *We call* **concrete agent building** *the following problem: given an original model M_0 and an agenda Ag find a strategy function $S(Ag, M)$ such that the agenda will be satisfied in a finite time.*

2.2.1 On the difficulty of agent building problem. The skeptical assumption

The main difference between the abstract and the concrete agent building problem is that the concrete ABP needs to perform actions to update its model. This is true, even in the case of a controlled world. In practice, agents are spending most of their time gathering information about the world. This feature is masked in the case of the abstract ABP.

However, there is a deeper problem, namely that we have a *skeptical* view of the knowledge of our agents. There is a long philosophical tradition in exploring the validity of our knowledge. In ancient Greek the word *skepticos* was for an "inquirer", someone who was unsatisfied and still looking for the truth. In a modern philosophical acceptance, a skeptic is somebody who has raised doubts about any knowledge beyond the one acquired through direct experience. We cannot make claims about the entire world but for the more restricted world where agents are operating, this is exactly the case.

While philosophy may not give as a solution for agents, it certainly gives us a clue about the difficulty of the problem. Since the idea of skepticism appeared first in the Upanisads (c.1000- c.600 BC) and in the Western philosophy in the Platonic school in the 3rd century BC, almost every major philosopher had to take a stand for or against this view. Without exploring further in this problem, our stand is that the position of Kant (written to refute the skepticism of Hume) is the most practicable for someone interested to actually implement an agent: start the agent with a collection of irrefutable, *a priori* knowledge, while maintaining a level of doubt on all the rest of the knowledge acquired e.g. by attaching confidence levels according to the method of knowledge acquisition.

One of the particularly dangerous problems is a consequence of the fact that the agenda of the agent is defined only based on the model. Thus, the agenda can be satisfied easily, modifying only the model, without even touching the external world. We call this phenomena *self-deception*.

We present pragmatic assumptions about agents, necessary for a reasonable implementation. The correctness of these assumptions is a precondition for the reasoning we are going to perform on our agents. These assumptions are:

- The *a priori* knowledge (the initial model) of the agent is correct.
- The agent takes a good faith effort to synchronize its model with the status of the world.
- The methods of exploration available to the agent together with the original model M_0 allow a sufficiently good approximation of the state of the world such that the agenda is satisfied only in the desired state of the world.

Still, the previous discussion is of little help to someone who wants to implement an agent in practice. Our agent model AM_0 include a single large strategy function $S(Ag, M)$ which encapsulates the whole difficulty of the agent building problem. Our intent is to **create a model for a constructive solution for the agent building problem**. In the following section we will decompose the AM_0 model into more detailed models. These models are equivalent with AM_0 and we argue that they can be implemented easily in an object-oriented programming style.

2.3 Decomposition in the plane. Expressing "change"

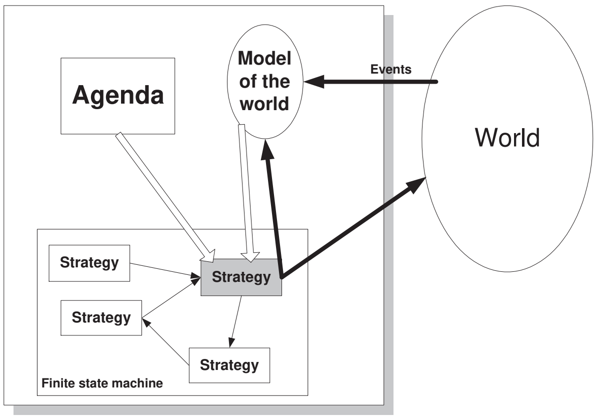

Figure 2: The AM_1 agent model. The monolithic strategy of the AM_0 model is replaced with a group of strategies, out of which only one is active in any given moment. The change of the strategy is determined by the current state of the finite state machine F.

From the implementation point of view the AM_0 model is a programmer's nightmare, because of the large monolithic strategy function S. This function is responsible for handling all the events and generating all the actions during the lifetime of an agent. We introduce the notion of *n-th order strategy functions* to allow a natural way of decomposing a strategy.

Definition 7 *The complete set of the model variables define the* **complete state** *of the agent. We define the* **n-th order state** *of the agents the set of the first n model variables $m_1, m_2..m_n$. We assume that these variables take their values from a discrete and finite set.*

Being in a particular n-th order state X characterized by the values $m_1^x, m_2^x...m_n^x$. -

Definition 8 *An* **n-th order strategy** *bound to an n-th order state $X = m_1, m_2..m_n$ is defined as:*

$$S_x^{(n)}(\{m_{n+1}, m_{n+2}, \ldots\}, Ag) = S^{(0)}(\{m_1^x, m_2^x, ...m_n^x, m_{n+1}, m_{n+2} \ldots\}, Ag)$$

An n-th order strategy is a *projection* of the 0-th order strategy for a fixed value of the first strategies or, conversely the 0-th order strategy is composed of a set of n-th order strategies for every possible value of the n-th order state. Because our goal is to construct the agent out of components, we take the second view.

Definition 9 *The AM_1* **model** *of an agent is the quadruplet $(Ag, S^{(1)}, M, F)$. Ag is the agenda, $S^{(1)}$ is a set of first order strategies $S^{(1)} = \{S_1^{(1)}, ...S_n^{()1}\}$. The model of the world M can be written as $M = m_0 \bigcup m_1, m_2...$ where $m_0 \in \{m_{0_1}, m_{0_2} \ldots m_{0_n}\}$. For any state where the first model variable has the value m_{0_i} the strategy of the agent is determined by the first order strategy $S_i^{(1)}$. The finite state machine $F^{(1)}$ determines the events that change the first model variable m_0.*

In the AM_1 model the first order state m_0 describes the current strategy of the agent. The existence of the finite state machine explicitly describes the possible transitions between the first order states. From a practical point of view we hope that the finite state machine would be relatively sparse, meaning that only a small subset of the possible transitions are legal, capturing the fact that a logical succession of strategies are applied towards achieving the goal of the agent.

Definition 10 *Two agents are* **equivalent** *if having the same model of the world generate the same action.*

AM_0 and AM_1 define the same equivalence classes of agents.

Property 1 *AM_0 and AM_1 define the same equivalence classes of agents.*

Definition 11 *We define recursively the* **n-th order agent model** *AM_n based upon the definition of AM_{n-1}, by introducing for at least one of the (n-1)-th order strategies of AM_{n-1} a set of k n-th order strategies, (where k is the number of possible values of m_n) and a set of finite state machines which describe the transitions in the model variable m_n for any given valid value of the model variables $m_0 \ldots m_{n-1}$.*

Property 2 *Every agent model AM_n for $\forall n \in N^+$ defines the same equivalence classes of agents as AM_0.*

This property raises two questions: if all models are equivalent to AM_0 what are the reasons to introduce more complicated models? The other question is that the AM_n models built can be trivial cases, and did not tell us if the decomposition of strategies really works.

The answer to the first question is that higher order models capture the natural engineering tendency to assemble solutions from smaller components. Also, some of the results in the next sections are based exactly on this decomposition: actually, the stronger this decomposition is, the stronger the results. The second question is more subtle: one can certainly imagine problems which cannot be decomposed in a meaningful way (for example, none of the model variables has discrete values). What we can say is that most problems in practice are well suited to be decomposed into sub-problems. These issues will be explored in more detail in the following section.

2.4 Decomposition in space - Expressing simultaneity

The AM_n model presented in the previous chapter decomposes the unique strategy of the model AM_0 into a number of strategies active one at a time. In this section we propose a method for further decomposition - we are going to decompose the current strategy of the AM_n model into a number of strategies active concurrently.

Definition 12 *We call an m-plane* **scheduling function** *a function*

$$\sigma : (M* \times Ag* \times t) \rightarrow \{0, 1...m-1\}$$

The **scheduling interval of** *σ is n time value t_sched such that*

$$\forall n \in \{0, 1...m-1\}, \forall M, \forall Ag, \forall t, \quad \exists \Delta t < t_{sched} \text{ such that } \sigma(M, Ag, t + \Delta t) = n.$$

This definition basically expresses our notion of a valid scheduling - there is a time interval in which every plane will be scheduled.

Definition 13 *We call a* **multiplane strategy** S^{mp} *with the planes* $S^0, S^1 \dots S^m - 1$ *with the associated scheduling function* σ *a function*

$$S(M, Ag) = \begin{cases} S^0(M, Ag) & \textit{if } \sigma(M, Ag) = 0 \\ S^1(M, Ag) & \textit{if } \sigma(M, Ag) = 1 \\ \dots & \\ S^{m-1}(M, Ag) & \textit{if } \sigma(M, Ag) = m - 1 \end{cases}$$

A multi-plane strategy expresses the idea that a strategy function can perform actions which are dealing with different parts of the world. The $AM_1^m p$ agent model uses multi-plane strategies to express its behavior.

Definition 14 *The* AM_1^{mp} *agent model is a quintuplet* $(Ag, S^{(1)mp}, M, F^{mp}, \sigma)$. *Ag is the agenda,* $S^{(1)mp}$ *is a set of first order multi-plane strategies. The model M can be written as* $M = m_0^0, m_0^1 \dots m_0^{m-1} \dots m_1, m_2 \dots$ *where* $\forall i \in \{0 \dots m-1\} \;\; m_0^i \in \{0 \dots n\}$. *The current behavior of the agent is determined by a multi-plane strategy composed of* $S^{m_0^0}, S^{m_0^1} \dots S^{m_0^{m-1}}$. *The multi-plane state machine* F^{mp} *determines the events that change the model variables* $m_0^0 \dots m_0^{m-1}$.

A graphical representation of the AM_1^{mp} model is presented in Figure 3.

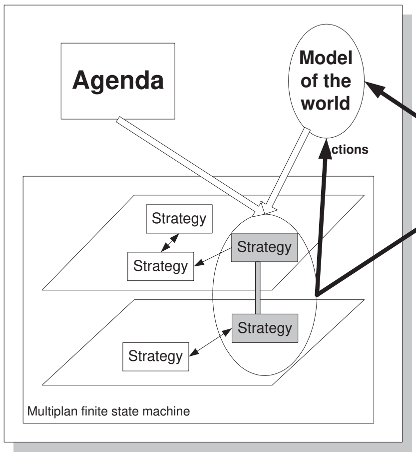

Figure 3: The structure of the AM_1^{mp} agent model. The behavior of the agent is defined by a number of strategies acting simultaneously. The change in this strategy is expressed through a multi-plane state machine.

3 Reference implementation

We created a reference implementation for the agent design presented in the previous section in the framework of the Bond system. Bond is a Java based distributed object system and agent framework, with an emphasis on flexibility and performance. It is composed of a core level, containing the object model and message oriented middleware, a service layer containing distributed services like directory and persistent storage services and the agent framework, providing the basic tools for creating autonomous network agents together with a database of commonly used strategies which allow developers to assemble agents with no or minimal amount of programming.

Now we present a component-based architecture for software agents. In this architecture an agent consists of a group of active objects linked together by a data structure, rather than a large monolithic code. The behavior of the agent is determined by the active and passive objects and the data structure. The active objects usually consist of compiled code, thus can be executed with little additional overhead. The data structure can be modified with ease allowing for flexible behavior.

The structure of the agent is presented in Figure 4. The four major components of an agent are: the model, the agenda, the state machines, and strategies.

- The **model of the world** is a container object which contains the information the agent has about its environment. There is no restriction of the format of this information: it can be a knowledge base or ontology composed of logical facts and predicates, a pre-trained neural network, a collection of meta-objects or different forms of handles of external objects (file handles, sockets, etc), or typically, a heterogeneous collection of all these. The model also stores the information the agent has about itself: one example being plans or intentions (if the agent conforms to the BDI model).

- The **agenda object** defines the goal of the agent. The agenda implements a boolean function and a distance function on the model. The boolean function shows if the agent accomplished its goal or not. The agenda acts as a termination condition for the agents, except for agents with a *continuous agenda* where their goal is to maintain the agenda as being satisfied. The distance function may be used by the strategies to choose their actions.

- The **multi-plane state machine** of the agent is a data structure composed of a number of state machines arranged in *planes*. The state of each state machine is defined by the active node. The state of the agent is defined by a *vector of states*. An agent changes its state by performing *transitions*. In turn transitions are triggered by internal or external *events*. External events are messages sent by other agents or programs. The set of external messages that trigger transitions of the agent's state machine defines the *control subprotocol* of the agent.

- Each node of the multi-plane state machine has associated a **strategy object**. Strategy objects generate *actions* based upon the model and the agenda of the agent. Strategies do not reveal internal state information - their behavior is determined exclusively by the model and the agenda. The strategies must store their state in the model. Actions are considered atomic from the agent's point of view, external and/or internal events interrupt the agent only between actions. Each action is defined exclusively by the agenda of the agent and the current model. A strategy can terminate by triggering a transition or by generating an internal event. After the completion of the transition the agent moves into a new state where a different strategy defines its behavior.

The implementation we propose for the agent execution model is based on *strategies*. Informally, a strategy is a function which takes as parameters the model of the world and the agenda of the agent and returns actions. From the implementation point of view, a strategy is a Java object with a function called `action()` that performs the actions needed at the given instance.

The strategies are activated: (a) in response to external events and (b) as the flow of control requires while pursuing the agent's agenda. *Messages*from remote applications, and *user interface events* like pressed keys, mouse-clicks are examples of external events. The strategies are activated by the event handling mechanism - the Java event system for GUI events, or the messaging thread for messages in case of external events, or by an *action scheduler* .

The state of the agent is defined by a *vector of states*, which implies that the behavior or the agent is determined by a *vector of strategies*.

This structure allows us to assign different strategies for handling different types of events - for example a strategy from one plane handles the messages, while the other plane is handling the user interface events. One of the planes may provide reasoning or planning functions, one the execution, another one carry out housekeeping operations. The strategies in these planes are activated by the action scheduler.

The multi-plane structure provides the means to express concurrent agent activities. The actual nature of the parallelism is determined by the scheduling mechanism used by the action scheduler. In case of a round-robin activation mechanism the actions belonging to different strategies are interleaved without overlapping while multi-threaded execution allows for truly concurrent actions. Other possible activation schemes are priority-based and preemptive.

The agents are described by their *active components* (the strategies) and the *structural components* - the multi-plane state machine.

A strategy should be compatible with the agent implementation language, Java in case of Bond. There are two requirements a software component should meet to be a valid strategy: it should allow its state to be linked to the model and it should break its behavior into actions. JavaBeans, ActiveX objects, C++ libraries or functions in interpreted languages can all be valid strategies. In the Bond system, besides Java-written strategies, we are currently supporting strategies written in Jess [1] and Python through the JPython interpreter. Any other language can be used through the Java native interface.

The structural component of an agent, the multi-plane state machine, can be constructed as a program, but a more flexible approach is a textual description, interpreted by an *agent factory*. In the Bond system we are using the `blueprint` language to describe the structure of an agent. Other possibilities, like a visual programming interface are also possible.

4 Usage scenarios

In this section we present five scenarios for Bond agents. The goal is to show how the architecture we propose leads to elegant implementations of practical applications. The scenarios are based on the experience we have accumulated using the Bond system to implement various agents. Although we try to be as general as possible, this section necessarily contains more references to our specific implementation (the Bond system) than other sections of this paper.

4.1 Using planes to implement facets of behavior

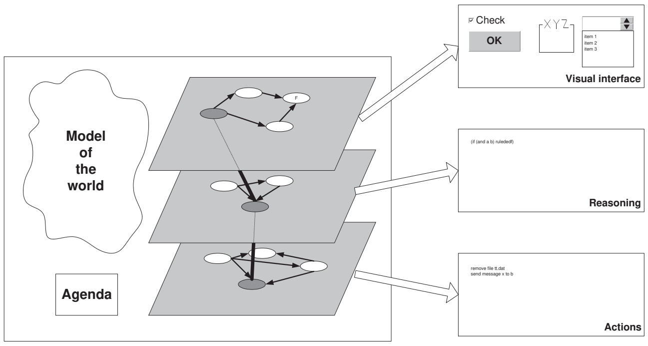

Figure 4: Different plans of the agent represent different facets of the behavior of the agent. In this case we have a plane controlling the visual interface, a plane for reasoning and a plane for performing actions.

The behavior of an agent is often *multifaceted*, it consists of several loosely coupled aspects. A full-featured agent may exhibit several facets:

- **Reasoning.** Agents with reasoning abilities usually run in a cycle, generating new true statements based upon current knowledge.
- **Visual interface.** Most agents present a visual interface and interact to the user by: (a) presenting a part of the knowledge of the agent (i.e. a part of the model) in a visual format, and (b) collecting user interface events from the user.
- **Reactive behavior.** Agents react to external events e.g. messages, user interface events, signals, etc.
- **Active behavior.** Agents perform actions in pursuit of their agenda even without external events.

In most cases, a separation of these facets is possible, and the relative independence of the facets justifies their separate treatment. For example the various steps taken by an agent to pursue its goal are changes in its active behavior, but these changes may not necessarily lead to a change in its reactive behavior, the look of the user interface, or the reasoning process of the agent.

We argue that the multi-plane state machine structure provides an elegant way to express the multifaceted behavior of an agent, every plane expresses a facet of the behavior of the agent. There are no restrictions on the nature and behavior of planes, so the agent designer can create the structure most suitable to the problem at hand. However, the independence of facets is relative, significant interdependence existing between them. In the multi-plane state machine structure, the interdependence amongst planes is captured by the fact that (a) all planes share a common model (knowledge) and (b) transitions triggered by one plane are applied to the whole structure, providing a signaling mechanism amongst planes.

Figure 4 represents an example of an agent that presents a visual interface to the user, performs a background reasoning, and takes actions. These three facets of the behavior are reflected into an agent with three planes, each one dedicated to a facet.

All Bond agents use the multi-plane structure to implement the facets of the agent behavior. For example our stock-market watcher agent contains a communication plane for gathering data from online quote services, a Jess-based [1] reasoning plane for processing the information and a visual interface plane to present the information to the user.

4.2 Implementing a BDI agent

An agent conforming to the BDI (belief-desire-intention) theoretical model is organizing the behavior into three groups of statements, usually expressed in the notation of first order logic.

The BDI model can be easily implemented in the multiplane framework in the following way. The beliefs, desires and intentions are stored separately in the model - possibly in separate namespaces. The agenda of the agent will be a boolean function which is true if the statements in the desire group are simultaneously satisfied.

The BDI agent consists of at least two planes. The first of them, the *planning* plane contains strategies which generate intentions based on the current desires and beliefs of the agent. These strategies can be implemented the more conveniently using logical programming languages like Prolog or Clips. The Jess [1] implementation of for which the Bond system offers direct support is a convenient choice for this plane. The *execution* plane contains strategies which generate actions based on the current intentions of the agent. These strategies can be implemented in a language with a better support for imperative execution. The structure of an agent implementing the BDI model is presented in Figure 4.2.

This approach also allows to add additional planes for specifying aspects of agents like the visual interface which are cumbersome to express in terms of logical statements.

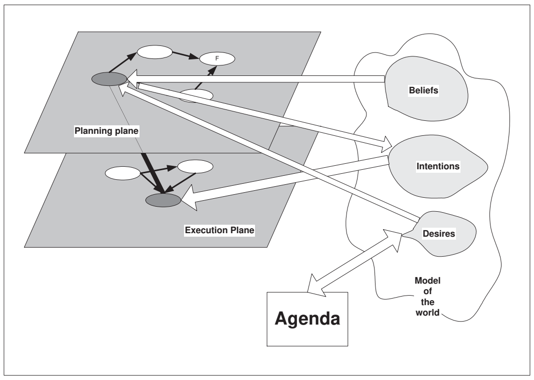

Figure 5: Example of the implementation of a BDI agent in the multiplane framework. The agent has two planes implementing the planning and the execution behavior of the agent.

4.3 Using messages to implement remote control

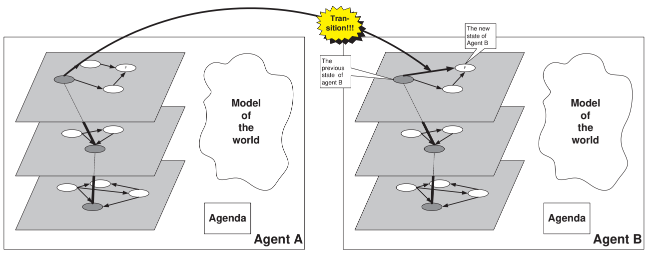

Figure 6: Example of remote control using messaging. The message sent by agent A triggers a transition on the first plane of agent B thus changing its behavior.

A significant part of the inter-agent communication can be described as *control*: the behavior of the *controlled* agent is changed as a result of an action of a *controller* agent.

In terms of our structure, the behavior of the agent is described by the state vector, and it can be changed by transitions, which alter one or more states in the state vector. One way to trigger transitions is by an external message.

In our implementation of the model, transitions can be triggered by KQML messages. Messages are grouped in *subprotocols* which is determined by the multi-plane state machine of the agent, which in its turn is described by the blueprint. For consistency reasons not all transitions in the multi-plane state machine can be triggered remotely, because the semantics of certain transitions, for example `success` depends upon the current strategy.

Figure 4.3 present an example of use. Agent A wants to change the behavior of agent B, by changing the strategy on the first plane. Thus it sends a message labeled with the given transition. The transition will be performed on all the planes of agent B, if there is a transition with the specific label. In our case, only on the first plane it is a transition with the label as specified, thus the only state from the state vector which is changed is the state of the first plane.

The use of messages as information control is used in most multi-agent implementations using Bond. One example is the network of agents for solving partial differential equations [9] where external transitions are used to synchronize the results between independent solver agents.

4.4 Cooperation through information sharing

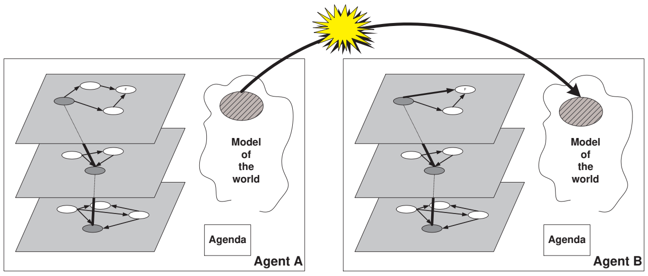

Figure 7: Example of inter-agent cooperation using knowledge sharing. Agent A pushes part of its model into the model of agent B.

Another important aspect of agents is that they can cooperate towards achieving certain goals. We argue that every agent cooperation can be described in terms of knowledge sharing. In our structure this means that a part of the model of an agent is copied to the model of another agent.

There are basically two ways of sharing knowledge, depending upon the agent initiating of the process:

- **push mode** an agent copies part of its model to the model of the other agent.
- **pull mode** an agent copies a part of the model of the remote agent into its own model.

Security and semantic concerns related to the information sharing models must be addressed. We have to determine what part of the model will be shared, the identities of the agents, the confidence level in the shared knowledge and so on.

Our implementation, the Bond system contains support for information sharing at the communication layer level, and contains various mechanisms enforcing security for inter-agent cooperation [15]. Figure 4.4 presents an example of cooperation through knowledge sharing using the push mode. Agent A is pushing a part of its model to the model of agent B.

Although the knowledge sharing model applies to any possible cooperation model, it may not be the best way of dealing with some cooperation patterns. For example, a service negotiation would be implemented as a successive series of very small information interchanges. Obviously, this case is better handled if we view the negotiation as a message interchange and apply the results of recent research in this area . In our model we may create several negotiation planes such that multiple negotiations may be conducted simultaneously by the agent.

Cooperation through knowledge sharing is normally used when an agent starts a new agent and passes part of its knowledge to the newly created agent. One example is the PDE solver agent system mentioned before where partial results of computations are communicated in this way.

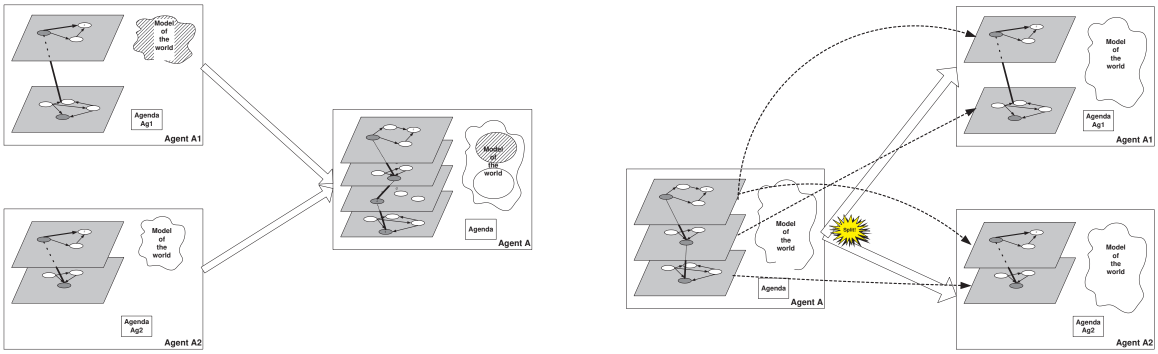

Figure 8: Splitting agents: the new agents contain a part of the planes of the original agent - in our case one plane is replicated in both new agents. The agents inherit the full model of the original agent while the conjunction of their agendas is the agenda of the original agent. Joining agents: the new agent contains all the planes of the initial agents and a combination of their models. The agenda, in this case is the conjunction of the agendas of the original agents.

4.5 Joining and splitting agents

Two of the simplest surgical operations on agents are the joining and splitting. When **joining** two agents, see Figure 8, the multi-plane state machine of the new agent contains all the planes of the two agents and the model of the resulting agent is created by merging the models of the two agents. The safest way is to separate the two models (for example through use of namespaces), but a more elaborate merging algorithm may be considered. As our design does not specify the knowledge representation method, the best approach should be determined from case to case. The agenda of the new agent is a logical function (usually an "and" or an "or") on the agendas of the individual agents. It is tempting to consider the joining of agents as a boolean operation on agents, and maybe to envision an algebra of agents. While the subject definitely justifies further investigation, the design presented in this paper do not qualify for such an algebra. The more difficult problem is handling the "not" operator, which applied to the agenda would render the current multi-plane state machine useless.

In case of agent **splitting** we obtain two agents, the union of their planes gives us the planes of the original agent (Figure 8). The splitting need not be disjoint, some planes (e.g an error handling or housekeeping) may be replicated in both agents. Both agents inherit the full model of the original agent, but the models may be reduced using the techniques presented in section 4.6. The agendas of the new agents are derived from the agenda of the original agent. The conjunction or disjunction of the two agendas gives the agenda of the original agent.

There are several cases when joining or splitting agents are useful: (a) Joining control agents from several sources, to provide a unified control, (b) Joining agents to reduce the memory footprint by eliminating replicated planes, (c) Joining agents to speed up communication, (d) Migrating only part of an agent, (e) Splitting to apply different priorities to parts of the agent.

Joining and splitting of agents is used by our implementation of agents implementing workflow computing.

4.6 Trimming agents

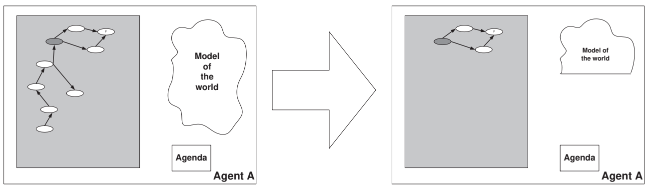

Figure 9: Trimming agents: the example presents the trimming of a single plane agent. The parts of the state machine which are not reachable from the current state are eliminated. Also the part of the model which can not be accessed by the remaining strategies is eliminated too.

The state machines describing the planes of an agent may contain states and transitions unreachable from the current state. These states may represent execution branches not chosen for the current run, or states already traversed and not to be entered again. The semantics of the agent does not allow some states to be entered again, e.g. the initialization code of an agent is entered only once.

If the implementation allows us to identify the set of model variables which can be accessed by strategies associated with states, we can further identify parts of the model, which can not be accessed by the strategies reachable from the current state. The Bond system uses *namespaces* to perform a mapping of the model variables to strategy variables, thus we can identify the namespaces which are not accessed by the strategies reachable from the current state vector.

If the agenda of the agent can be expressed as a logical function on model variables and this is usually the case, we can simplify the agenda function for any given state, by eliminating the "or" branches that cannot be satisfied from the current state of the agent.

All these considerations allow us to perform the "trimming" of agents, for any given state to replace the agent with a different, smaller agent as shown in Figure 9. In this example we used an agent with a single plane, but the process is identical for multi-plane agents. Both the multi-plane state machine, the model and the agenda can be simplified.

While stopping an agent to "trim" it is not justified for every situation there are several cases when we consider it to be useful:

- **Before migration.** As migration is sometimes proposed in order to "bring the code to the data" when the data is larger than the code, trimming the agent allows us to reduce the amount of code and internal data transferred. Furthermore, as for migration the agent has to be stopped anyhow, the penalty for the trimming time is usually compensated by the fact that less data and code needs to be transferred over the network.

- **Before checkpointing.** Similar to the migration case, the agent has to be stopped for checkpointing, and trimming can reduce the amount of data to be saved.

- **At runtime.** Trimming can be performed even on running agents, for example by an external thread or by a strategy of the agent. Doing this can be justified by the need to reduce the memory footprint of the agent. A careful examination is needed to find out whether the eliminated parts are actually becoming free memory. For compiled languages like C or C++ it is probably impossible to free the code memory, but even Java was not being able to garbage collect classes as of version 1.1. Freeing data implies different approaches whether the language has a garbage collector or not.

Our design implements just the framework and mechanisms for agent trimming. Determining the parts which can be trimmed is a problem in itself and various techniques can be used. Trimming the multi-plane state machine can be done by reachability analysis. Trimming the model depends very much on the specific implementation of the strategies. The Sethi-Ullman algorithm for reusing temporary variables from the theory of compiler construction [14] may be used. Trimming the agenda can use techniques from the theory of lazy evaluation of boolean expressions.

Generally this method can be considered as an extension of various techniques already in use in compiler and programming language theory to a different level of granularity, strategies instead of individual instructions. While the technique remains the same, the different granularity produces a shift in the cost/benefit analysis: there is not enough benefit in freeing an individual instruction from the memory, but it may be worthwhile for a strategy consisting of a large block of code.

The default migration implementation in the Bond system is using trimming to reduce the amount of data transferred in the migration process.

5 Conclusions

In this paper we proposed a theoretical model for building network agents and presented our implementation of the model in the Bond system.

Our experiences show that the model proposed here can be implemented with ease by programmers with object oriented experience, as the model do not propose a radical departure from the original programming style. Still, the close conformity with the theoretical model requires a certain level of self-discipline from the programmers, and from the project leader.

Once implemented the theoretical model gives significant advantages to the system. Migration and checkpointing of agents is achieved at no extra cost, as a side-effect of the conformance to the theoretical model. Exciting new possibilities, like agent surgery, trimming and joining and splitting of agents are opened.

We continue to work both on the theoretical and the practical aspects of the system, [10], [16]. Our more theoretical research covers the reasoning on agents and the integration of results obtained on the BDI model in our object-oriented approach. On the practical side, our focus is on the run-time modification and custom assembly of agents. A number of applications of the Bond Agent Framework [11], [12], [13] prove its versatility. The Bond system is released under an open source license and can be downloaded from `http://bond.cs.purdue.edu`.

Acknowledgments
The work reported in this paper was partially supported by a grant from the National Science Foundation, MCB-9527131, by the Scalable I/O Initiative, and by a grant from the Intel Corporation.

References

[1] E. Friedman-Hill. *Jess, The Java Expert System Shell.* Distributed Computing Systems, Sandia National Laboratories, SAND98-8206, 1999.

[2] T. Finin, et al. *Specification of the KQML Agent-Communication Language*, DARPA Knowledge Sharing Initiative draft, June 1993

[3] *JACK Intelligent Agents* `http://www.agent-software.com.au/jack.html`

[4] M. Breugst, I. Busse, S. Covaci, T. Magedanz *Grasshopper - A Mobile Agent Platform for IN Based Service Environments* IEEE Intelligent Networks Workshop, Bordeaux, France, May 1998.

[5] Rao, A. S. and Georgeff, M. P. *Modeling rational agents within a BDI-architecture* Proceedings of Knowledge Representation and Reasoning (KR&R-91), pages 473-484.

[6] M. Wooldridge, N. R. Jennings and D. Kinny *A Methodology for Agent-Oriented Analysis and Design* Proceedings of the Agents'99, pp. 69, Seattle, USA, May 1999.

[7] D. Kinny, M. Georgeff and A. Rao *A methodology and modelling technique for systems of BDI agents* in W. Van de Velde and J. W. Perram, editors, "Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (LNAI Volume 1038) pp. 56. Springer-Verlag, Berlin, Germany 1996.

[8] *AgentBuilder: An Integrated Toolkit for Constructing Intelligent Software Agents* `http://www.agentbuilder.com/Documentation/ WhitePaper/index.html`

[9] P. Tsompanopoulou, L. Bölöni and D.C. Marinescu *The Design of Software agents for a Network of PDE Solvers* Agents For Problem Solving Applications Workshop, Agents '99, Seattle, USA May 1999.

[10] L. Bölöni and D. C. Marinescu. *A Multi-Plane State Machine Agent Model* Technical Report CSD TR 99-027, Purdue University, September 1999.

[11] K. Palacz and D. C. Marinescu, *An Agent-Based Workflow Management System* Computer Sciences Department, Purdue University, CSD-TR#99-032, October 1999.

[12] K.K. Jun, L. Bölöni, D. K.Y. Yau, and D. C. Marinescu, *Intelligent QoS Support for an Adaptive Video Service* Computer Sciences Department, Purdue University, CSD-TR#99-033, October 1999.

[13] K.K. Jun, L. Bölöni, K. Palacz and D. C. Marinescu, *Agent-Based Resource Discovery* Computer Sciences Department, Purdue University, CSD-TR#99-034, October 1999.

[14] R. Sethi and J. D. Ullman. *The generation of optimal code for arithmetic expressions* Journal of the ACM, 17(4):715-728, October 1970.

[15] R. Hao, Bölöni, K. Jun, and D.C. Marinescu. *Bond System Security And Access Control Model.* IASTED International Conference on Parallel and Distributed Computer and Networks, ACTA Press, pp.520-524, 1998.

[16] L. Bölöni and D.C. Marinescu *An Object-Oriented Framework for Building Collaborative Network Agents* in *Intelligent Systems and Interfaces*, (A. Kandel, K. Hoffmann, D. Mlynek, and N.H. Teodorescu, eds). Kluewer Publising House, (1999), (in press).

[17] D. Harel, E. Gery *Executable Object Modelling with Statecharts* Proc. 18th Int. Conf. Soft. Eng., IEEE Press, March 1996, pp 246-257.