A Multi-Plane State Machine Agent Model

Ladislau Bölöni and Dan C. Marinescu
Computer Sciences Department
Purdue University
West Lafayette, IN 47907

September 16, 1999

Contents

Abstract

This paper presents a framework for implementing collaborative network agents. Agents are assembled dynamically from components into a structure described by a multi-plane state machine model. This organization lends itself to an elegant implementations of remote control, collaboration, checkpointing and mobility, defining features of an agent system. It supports techniques, like agent surgery difficult to reproduce with other approaches.

The reference implementation for our model, the Bond agent system, is distributed under an open source license and can be downloaded from `http://bond.cs.purdue.edu`.

1 Introduction

The field of agents is witnessing the convergence of researchers from several fields. Some see agents as a natural extension of the object-oriented programming paradigm, [14, 15]. One of the most popular books on artificial intelligence reinterprets the whole field in terms of agents [2]. Contemporary work on the theory of behavior provides the foundations for theoretical models of agents [13, 15]. Interface agents are considered the next evolutionary step in the development of visual interfaces. The field of robotics is using agents to model the behavior of their artifacts. This diversity of views as well as the wide range of applications of agents leads to numerous programming paradigms, languages, communication methods, and design concepts in agent system implementation. So far no design methodology has emerged as a clear winner, applicable for the majority of situations.

When implementing an agent framework for developers with different backgrounds and applications, the goal should be to maximize the flexibility and provide a rich palette of options. We believe that the agent model and the design concepts presented in this paper ensures flexibility in the choice of programming paradigms, languages, communication models, and operating modes for the agent.

The agent model we propose reflects our philosophy regarding the critical aspects of an agent architecture namely, agent communication, remote instantiation and control, checkpointing and mobility. A unique feature of our model is the ability to modify agents at run time, what we call *agent surgery*. There are several examples of self-modifying programs, mutating viruses being one of them. Programs written in Lisp and Lisp-like languages often treat code as data, and the remote modification of code is possible, though seldom used. Still, to our best knowledge the Bond system is the only mainstream, Java-based agent toolkit which allows a radical, remote intervention in the structure of an agent. On the other hand, the mobility model in this design is weaker than in systems designed for mobility like Aglets [8] or Telescript [7].

Our preliminary work on the theoretical foundations of this model can be consulted elsewhere A reference implementation for this design is the agent framework of the Bond distributed agent system [3]. In this paper we attempt to keep to a minimum references to the implementation details of the Bond system, but these references are sometimes unavoidable, because they provide the proof of concept for our design decisions.

This paper is organized as follows. In the next section we position our approach in the context of various agent systems and contrast the agent execution model to reactive and non-interactive models. Then, in Section 3, we introduce a component-based architecture and present the structure of an agent in our model. In Section 4 we follow the lifecycle of an agent, and explain how this agent structure can be used to implement the trademarks of an agent, e.g. the pursue of the agenda, remote control, mobility, checkpointing, and surgery. In Section 5 we present several usage scenarios based upon our experience with the Bond system. Conclusions are drawn and further research goals are presented in Section 6.

2 Agent Systems and Models

A number of agent systems are developed in universities and research laboratories and few of them reach the commercial product stage. In this section we overview several agent systems and agent models that influenced our ideas and position our design and implementation in the context of the existing systems. We present the elements specific to our design model and the implementation of Bond in the context of other systems and justify the important design decisions.

Let us consider first the **relationship to theoretical models**. Currently the most elaborated and rigorous model of agency is currently the *Belief - Desire - Intention* model [13, 15]. Although BDI is a very powerful model, its complexity and reliance on the notation of mathematical logic, preclude its wide-spread use for implementing agent systems. We are currently lacking the tools to map easily this model to object-oriented languages like Java or C++. Most Java-based agent systems do not rely explicitly on a theoretical model for agents. There are some exceptions, for example the JACK agent system [10] is using an extension to the Java language to implement the BDI model.

Our system is based upon the AM_1^{mp} model. We believe this model to be well suited to an object-oriented implementation, though less powerful than BDI (we sometimes call it "poor man's BDI"). This model allows us to reason about agents while using an object-oriented programming style.

Another aspect of agents is the **communication method**. Most agent systems use a message-oriented style although recently agents based upon CORBA MASIF specification [11], e.g. Grasshopper [12] are emerging. The agents using message-oriented communication either rely on an agent communication language like KQML [9] or FIPA, or use free-format communication. Examples of agent toolkits using KQML are JATLite from Stanford [5] and the commercial product AgentBuilder [16]. Systems using free-format communication are the Aglets [8] or Voyager [17]. The Bond agent system uses KQML as the communication language, but the design principles are largely independent on the communication language.

Another important consideration in the design of an agent system is **mobility**. Frameworks like IBM's Aglets [8] or General Magic's Telescript [7] consider migration a defining property of an agent and a basic design principle is to allow the agent to migrate at any time. In Bond, migration is considered a rare event in the life of an agent and migration is possible only under certain circumstances. We deliberately choose to implement a *weak migration model* as discussed in Section 4.

Our design has unique features, direct consequences of our philosophy to represent the behavior of the agent as a *multi-plane state machine* as discussed in depth in Sections 3, and 4.

- Behavior embedded into a data structure. The overall behavior of the agent is described by the blueprint. The strategy associated with a state defines the behavior in that state.
- Explicit concurrency. The agent structure is based on several state machines running concurrently.
- Dynamic agent modification, agent surgery.
- Integration of heterogeneous behavior models. The design allows the creation of BDI agents conforming to the theoretical model, actor based models or ad-hoc behavior models. A variety of programming languages and styles can be used in the same agent.
- Possibility of automated programming. While the automated programming at the fine grain level is still unrealistic at the current level of knowledge, the structural components of the model are coarse grain enough and their semantics is simple enough that we can envision that they can be created automatically based on the needs and circumstances.

The design presented in this paper can be positioned among the mainstream, multipurpose agent toolkits with some unique features. While it may be weaker than specialized tools for specific tasks, it offers the advantage of flexibility.

A model for specifying behavior frequently used in embedded systems and UML [23] based systems is the statechart model [24] designed by David Harel. Our multiplane agent state machine can be seen as a different way of expressing the parallelism which in statecharts are expressed as *concurrent substates*. On the other hand we have chosen to use a simpler state machine than those used in state charts. For example, in statecharts transitions can have conditions and actions associated with them, while in our model, only states can generate actions and transitions are unconditional. The theoretical foundation behind this decision is that in our model the structural component (the multiplane state machine), the active components (the strategies) and the model of the world are clearly separated. If we would introduce a condition on a transition, that would clearly be a boolean function on the model, thus making the state machine dependent on the model. If an action would be associated with the transition that would either imply that actions can be generated outside strategies, or alternatively that there is a strategy which is not determined by the state vector. Both these semantics are expressed in our model by *inserting an intermediate node* between the source and destination, the strategy of these node than performing the desired action or evaluating the condition. Thus, the multiplane state machines in our model can be larger for the same task than the corresponding statechart, but they are easier to analyze and generate, because of the simpler semantics. On the other hand, real time systems are easier to specify in the statechart format. Another feature of statecharts, the possibility to define embedded substates is also missing in our system. It's functionality in most cases can be replaced by the state vector of the multiplane state machine. We are currently investigating the possible benefits of introducing substates in our model: although they improve the expressiveness of the system, they also introduce difficulties in implementing checkpointing, migration and agent surgery.

Although there are numerous papers dealing with the definition of "agents" we choose to clarify our position to justify design decisions presented later in the paper. We adhere to the definition of Stan Franklin and Art Graesser [1]: *an* **autonomous agent** *is a system situated within and part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.* The **agent execution model** assumes that the agent has an explicit goal. Agents receive external events and generate actions. However the actions of an agent are determined by the pursue of its goal. Of course, events may trigger immediate responses, but agents perform actions even without any external input. An agent terminates its execution when its goal is accomplished.

The other execution models are: the **non-interactive (batch) execution model**, and the **reactive application execution model**, see Figure 1. The first is characteristic for traditional numerical simulation. The goal is to generate output results given a set of input data. All the inputs are available at the start of the application. The execution does not involve any interaction between the user and the environment from start to finish. In the reactive execution model, applications provide immediate responses to user inputs or external events. The application does not have an explicit goal, never takes the initiative and its termination is also a response to an external event. In absence of external events, the application is in an *idle cycle*. Examples are interactive programs like word processors, operating system shells, traditional servers e.g. file servers or http servers, embedded systems.

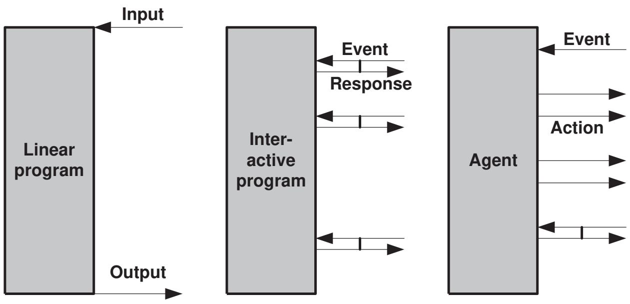

Figure 1: The execution model for non-interactive, reactive applications and agents. The first are characterized by a continuous progression from the input to the generation of output. Reactive applications are characterized by the well defined event-response pairs. The behavior of agents is characterized by the spontaneous generation of actions, although event-response pairs may also exist.

The execution model for agents is the more general and complex of the three models. Combinations of execution models are also possible, e.g. some modern servers like those used in airline ticket reservation or e-commerce perform a number of actions between the original request and the reply. These servers can be viewed as agents.

Since an agent can emulate every other execution model, some researchers are inclined to view every program as an agent. Although this approach may be sometimes useful, it does not lend itself to efficient implementations, the simpler the execution model the more optimal implementation is possible. For example we can optimize the response time of a stateless server far better than of an agent with a complex state.

The agent structure presented in this paper is designed to fulfill the requirements of the agent execution model presented above and to ensure optimal performance for situations when the agent must keep a complex state, interact with other agents and take actions on its own. Although this framework can be used to implement non-interactive programs or servers in the classical sense, these implementations may be suboptimal.

3 A Component-Based Architecture for Agents

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 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 2. 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 [6] and Python through the JPython interpreter [4]. 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 The lifecycle of the agent

The true test of any agent design framework is in implementation. How difficult is it to construct an agent? What is the overhead of the migration? How difficult is to remotely control an agent, or to create a federation of agents? These primary questions translate into how easy, and how efficient the basic operations related to the life cycle of an agent can be implemented and how to minimize the user's interactions during the agent's life time. In this section we present the implementation of the life cycle of the agents in our implementation of the design, the Bond system.

The main events in the life cycle of an agent are the *creation*, *starting*, *running* and *terminating.* Also agents can be *interrupted* and *restarted.* If the agent is restarted at a different location, we are talking about *migration.* Bond agents can be modified during runtime, an operation called *agent surgery.*

The `agent factory`, is an object involved in several operations during the life cycle of all agents coexisting at a Bond site. The agent factory is created on demand by the *autoprobe* object of a Bond resident. Once created, the agent factory responds to messages in the *agent control subprotocol* and creates new agents, performs agent surgery, assists in agent migration. The agent factory object accepts a structural description of an agent written in the `Blueprint` language. Bond agents can be created without the use of Blueprint, but our experience shows that the use of a it provides substantial advantages from the developers' point of view.

4.1 Creating and starting an agent

Bond agents are created by sending a `Create` message to the agent factory. This message may contain the blueprint of the agent or specifying the location of the blueprint. The agent factory assembles the agent using components available locally or remotely.

An agent is started by sending the `StartAgent` message to the agent. Upon receipt of this message: (a) the state vector of the multi-plane state machine becomes the initial state specified in the blueprint, (b) the current strategies are installed, (c) the execution thread is created, and (d) the *action scheduler* starts to execute actions according to the current strategies.

4.2 Running and external control

In the default running mode the active strategies of the agent perform actions. By default the *action scheduler*, the agent's main thread of control schedules various strategies to perform their actions. The default scheduler basically performs a round-robin scheduling, but it can be replaced by other schedulers, for multi-threaded or priority based scheduling.

There are some exceptions from this rule. The actions of event-handler strategies are triggered by the arrival of external messages, while the actions of GUI strategies are triggered in response to user events e.g. clicking on a button.

Any Bond object including an agent, may receive messages. In Bond we group semantically related messages into *subprotocols* [18]. Examples of subprotocols are monitoring, security, agent control, and so on. The *control subprotocol* of an agent consists of messages corresponding to external transitions in the multi-plane state machine. External as well as internal transitions are specified in the Blueprint script. Transitions indicating the success or failure of certain operations are internal.

An external program or an agent may trigger a transition of the multi-plane state agent by sending the message associated with the transition. If we want to inhibit the external control of an agent all the transitions should be declared internal. The control subprotocol is created dynamically and it disappears after the termination of the agent. When an agent undergoes surgery, the control subprotocol is modified dynamically.

4.3 Soft stop, checkpointing and restart

The execution of Bond agents can be stopped with the `SoftStop` message. This message instructs the action scheduler to stop the execution of the agents on the next *action boundary*. Thus a a soft stop is not instantanenous, and the time until it occurs depends upon the action scheduler (single threaded or multi-threaded) and on the granularity of the actions. At a soft stop of an agent the message handling is blocked, so the strategies triggered by messages or user input are blocked too.

In the stopped state, the agent can be checkpointed, by saving its model to persistent storage. The easiest way for doing this is by serializing the model, which implements the Java `Serializable` interface. However users might want to implement an incremental saving system, for example for transaction processing.

An agent in the softstopped state can be restarted by sending the `Restart` message. The state of the restarted agent is identical with the agent before the soft stop. However, for multi-threaded agents certain race conditions may be modified, and also certain messages or user actions may be lost, depending on the queueing properties of the underlying transport protocol or event system.

4.4 Migration

Bond agents implement the notion of *weak migration*, to migrate an agent a soft stopped state must be traveresed. The migration is triggered by sending a `Migrate` message to the agent factory at the site of the agent.

The sequence of the events in the migration process is: (a) the agent factory performs a soft stop on the agent and generates the blueprint of the agent, (b) the agent factory on the remote side is contacted and the blueprint of the agent is sent, (c)the agent factory on the new side reassembles the agent from the blueprint and transfers the model from the original site. In Bond this operation is called *realizing* an object. (d) the `relocate()` function is called on the model. By default this function call propagates further to other elements of the model. This function can be filled in by the user to adjust the objects to their new location. (e) the agent at the old site is disposed of. If the `:forwarder yes` parameter was specified, a forwarder object is installed which will forward messages sent to the old site to the new location of the agent. (f) the agent factory on the new site sends a success message to the originator of the migration message and restarts the agent on the new location.

The success of migration requires that the information in agent's model be moved to another site. Information like handles to open files are meaningful only locally. A set of rules must be observed to make the model mobile - for example, keeping all immovable information inside atomic actions. This implies that we should open and close a file inside a single action.

Our view is that migration should be a relatively rare event in the life of agents, so we did not take additional actions to enforce the mobility of the model, which might diminish the performance of the agent.

4.5 Agent surgery

We call *agent surgery* the dynamic modification of an agent. The behavior of a Bond agent is described as a data structure (multi-plane state machine). This data structure can be modified, and this modification changes the behavior of the agent.

The agent surgery is triggered by the `Modify` message sent by an external object to the agent factory controlling the agent. One of the parameters contains or points to a *surgical blueprint script* describing the modifications in the structure of the agent.

The sequence of actions in this process is:

(1)A *transition freeze* is installed on the agent. The agent continues to execute normally, but if a transition occurs the corresponding plane is frozen. The transition will be enqueued.

(2) The agent factory interprets the blueprint script and modifies the multi-plane state machine accordingly. There are some special cases to be considered: (a) If a whole plane is deleted, the plane is brought first to a soft stop - i.e. the last action completes. (b) If the current node in a plane is deleted, a *failure* message is sent to the current plane. If there is no failure transition from the current state, the new state will be a null state. This means that the plane is disabled and will no longer participate in the generation of actions.

(3) The transition freeze is lifted, the pending transitions performed, and the modified agent continues its existence.

4.6 Termination, zombie state and disposal

Agents are terminated when their agenda function is true. The agenda function is checked by the action scheduler after each action, so the last action of an agent is fully terminated. At this moment the agent performs a soft stop to carry out the executing actions on parallel planes and then the strategies are uninstalled. This uninstalls the message handlers for the message handling strategies. Uninstalling GUI strategies usually closes the windows associated with the program.

In this state the agent do not have an active thread anymore, the control subprotocol and the subprotocols implemented by strategies are not recognized any more. Nevertheless, the agent can still process the subprotocols implemented in the agent object like agent control and property access. This is important because allows remote objects to access the model of the agent, which contains important information. [1] We call this state of the agent the *zombie* state in analogy with the Unix concept of zombie processes (although the analogy is not quite perfect, zombie agents being still useful).

Agents in the zombie state can be revived by an external start message. This is meaningful only when the model was modified too, because otherwise the agenda will be immediately satisfied and the agent will terminate again.

Agents are disposed by sending the `unregister` message. At that time agents are unregistered from the local directory and garbage collected.

5 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.

5.1 Using planes to implement facets of behavior

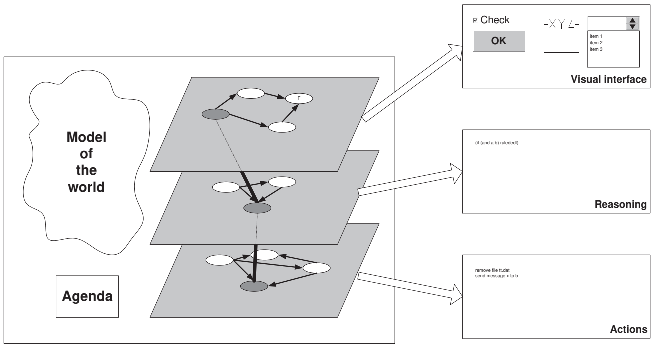

Figure 2: Different plans of the agent represent different facets of the behavior of the agent. In this case we have a plane controlling the visual interface, a plane for reasoning and a plane for performing actions.

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

[1] In some cases, the whole result of running the agent is to collect some information, to perform computations etc. This information is kept in the model, and the agenda is satisfied when the information is obtained. Of course one can choose to keep the agent alive until the information is transfered - but this is just a waste of processing power because of the idle threads.

- **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 2 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 [20] contains a communication plane for gathering data from online quote services, a Jess-based [6] reasoning plane for processing the information and a visual interface plane to present the information to the user.

5.2 Using messages to implement remote control

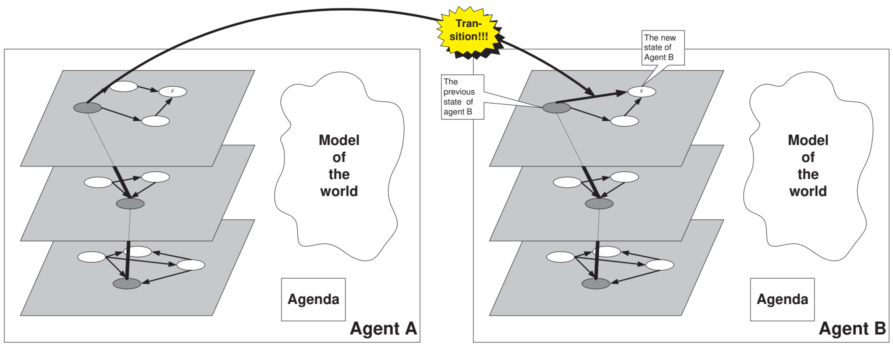

Figure 3: Example of remote control using messaging. The message sent by agent A triggers a transition on the first plane of agent B thus changing its behavior.

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 5.2 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 [19] where external transitions are used to synchronize the results between independent solver agents.

5.3 Cooperation through information sharing

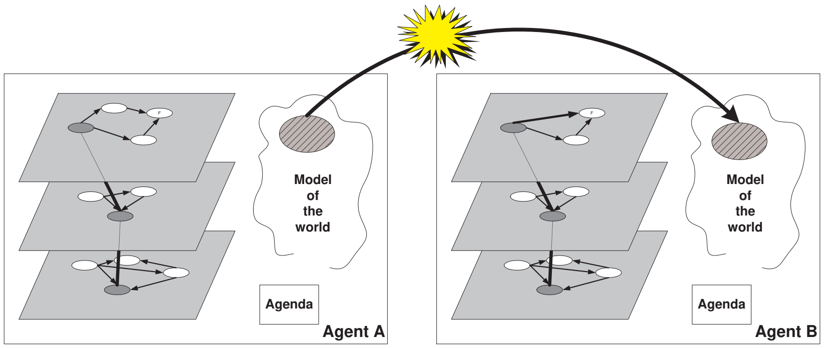

Figure 4: Example of inter-agent cooperation using knowledge sharing. Agent A pushes part of its model into the model of agent B.

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 [22]. Figure 5.3 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.

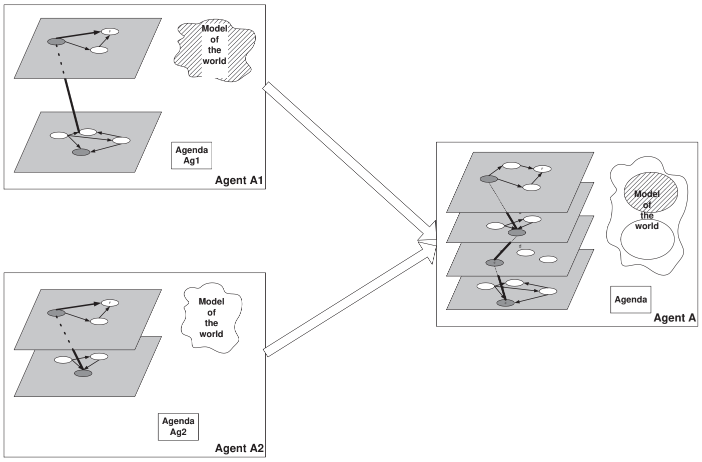

Figure 5: Joining agents: the new agent contains all the planes of the initial agents and a combination of their models. The agenda, in this case is the conjunction of the agendas of the original agents.

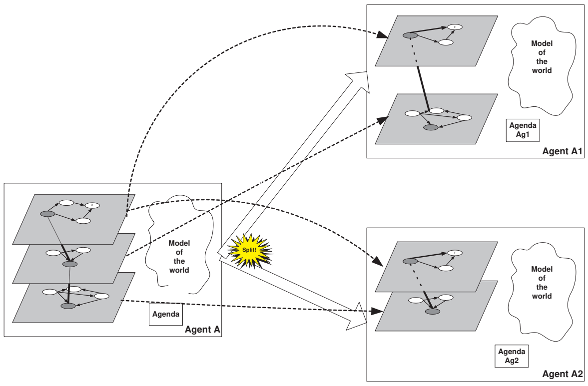

Figure 6: Splitting agents: the new agents contain a part of the planes of the original agent - in our case one plane is replicated in both new agents. The agents inherit the full model of the original agent while the conjunction of their agendas is the agenda of the original agent.

5.4 Joining and splitting agents

Two of the simplest surgical operations on agents are the joining and splitting. When **joining** two agents, see Figure 5, 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 6). 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 5.5. 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.

5.5 Trimming agents

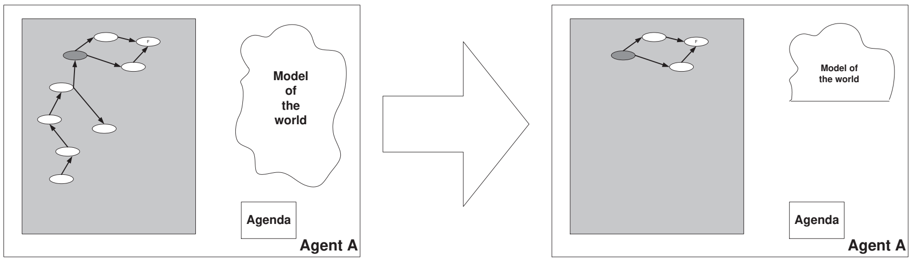

Figure 7: Trimming agents: the example presents the trimming of a single plane agent. The parts of the state machine which are not reachable from the current state are eliminated. Also the part of the model which can not be accessed by the remaining strategies is eliminated too.

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 7. 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 [21] 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.

6 Conclusions and future work

In this paper we presented a multi-plane state machine structure for designing collaborative network agents. The reference implementation for our model, the Bond agent system is in active development at the Bond distributed systems lab at the Computer Science department of Purdue University. The agent framework is distributed under an open source license (LGPL) and the second beta release of version 2.0 can be downloaded from `http://bond.cs.purdue.edu`.

We used the Bond agent framework system for a number of applications:

- multi-agent system for solving partial differential applications [19]
- agent-based system for negotiating the bandwidth for an MPEG video stream
- shared whiteboard collaborative application,
- real-time stock market watcher,
- news gatherer application for collecting news items from various internet sites

Though it does not provide a universal solution, our agent model allows an efficient and elegant design for a range of concrete applications. It supports techniques, like agent surgery difficult to reproduce with other approaches.

Several projects building upon our agent framework are underway: (a) an workflow management framework, (b) a resource management framework, (c) applications to data acquisition and analysis in structural biology, (d) applications to weather modeling.

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] S. Franklin and A. Graesser *Is it an agent, or just a program?* Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages, Springer Verlag 1996

[2] S. Russel, P. Norvig *Artificial Intelligence - A Modern Approach* Prentice Hall, 1995

[3] *A gentle introduction to Bond* `http://bond.cs.purdue.edu/guide/Intro.ps`

[4] *JPython homepage* `http://www.jpython.org`

[5] C. J. Petrie *Agent-Based Engineering, the Web, and Intelligence* IEEE Expert., December 1996.

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

[7] Tommy Thorn *Programming languages for mobile code* ACM Computing Surveys Vol. 29, No. 3 (Sept. 1997), Pages 213-239

[8] Danny Lange, Mitsuru Oshima "Programming and Deploying Java Mobile Agents with Aglets" Adison-Wesley 1998

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

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

[11] MASIF -The CORBA Mobile Agent Specification `http://www.omg.org/cgi-bin/doc?orbos/98-03-09`

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] *Voyager white papers* `http://www.objectspace.com/products/vgrWhitePapers.htm`

[18] L. Bölöni, R. Hao, K.K. Jun and D.C. Marinescu *Subprotocols: an object oriented solution for semantic understanding of messages in a distributed object system*

[19] 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.

[20] Stock Watcher application for Bond `http://bond.cs.purdue.edu/application/stockwatcher.html`

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

[22] 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, ParkRoyal Brisbane, Brisbane, Australia, December 14-16, 1998.

[23] G. Booch, J. Rumbaugh, I. Jacobson *The Unified Modelling Language User Guide* Addison Wesley, 1999

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