1999

A Framework for Building Collaborative Network Agents

Ladislau Bölöni

Dan C. Marinescu

Report Number:
99-001

A FRAMEWORK FOR BUILDING COLLABORATIVE NETWORK AGENTS

Landislau Boloni
Dan C. Marinescu

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR #99-001
January 1999

A framework for building collaborative network agents

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

January 20, 1999

Abstract

Agents are programs which autonomously pursue their own agenda. Agents in distributed systems are expected to be remotely controllable, and should be able to cooperate to accomplish their tasks.

This paper presents the agent framework of the Bond distributed object system. Bond agents have the possibility to be controlled remotely and to cooperate with each other.

The emphasis in the Bond agent framework is on the ability of quick and dynamic creation of new agents from a library of code. The task of an application programmer is limited to specify the agenda, the finite state machine of the agent, and the strategies associated with each state. Bond agents can be specified using an *agent definition language* called `blueprint`. Agents are assembled during runtime by a factory object on the request of a beneficiary object. Bond also provides a large database of ready-made strategies, so in typical cases a Bond agent can be assembled without programming.

1 Introduction

The term intelligent or autonomous agent is a hot topic in computer science, with various, sometimes conflicting definitions. Stan Franklin and Art Graesser in their overview paper [1] after discussing a number of alternate formulations are reaching the following definition: *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.*

Other authors, like Leonard N. Foner [2] are considering the anthropomorphism of an agent an important factor. Our understanding of an agent is largely equivalent with Franklin and Graesser's, and we are not considering the features of user interface as a defining element of the agent. Some of the Bond agents have immediate interaction only with other agents, while other agents have a graphic user interface.

There is a close interdependence between distributed object systems and agents. In a broad sense every object can be considered as an agent. There is, however a consensus among researchers that an agent requires an independent execution thread, and the capacity to initiate actions on its own. Objects like servers, although they have their own execution thread do not qualify as agents, because they react only to external requests.

The Bond agent framework provides support for the external control of the start of the agent, the acquisition of information (soft state) about the world and utilizes a generic framework for appending strategies. The strategies currently used range from simple table lookup (for the monitoring agent) to relatively complicated search engines (for the metaprogram scheduling agent).

2 Distributed object systems supporting agents

In this section we overview several distributed object systems which provide support for agents. The Infospheres and Bond projects are academic research projects at Caltech and Purdue, while the IBM Aglets and the Objectspace Voyager are commercial systems. There are a large number of other projects which provide support for agents.

2.1 Caltech's Infospheres

The Infospheres infrastructure is a research architecture for compositional systems, which are systems built from interacting components.

The Infospheres infrastructure is implemented in Java. It contains a message oriented network layer (info.net), and the agent implementation called *djinns* in Infosphere. Djinns implement persistence, remote invocation and a service model based on service requests / service replies. Djinns are running as separate threads inside a *master djinn* which provides remote activation capabilities.

A related project is the Übernet infrastructure which provides a very flexible communication infrastructure allowing users to assemble custom protocol stacks.

2.2 IBM Aglets

IBM Aglets are a framework for programming mobile agents. The authors are considering that mobile agents are a single uniform paradigm for distributed object computing, encompassing synchrony and asynchrony, message-passing and object passing, and stationary and mobile objects. The analogy used is that mobile agents are for distributed object computing what the elementary particles in physics.

The Aglets Framework is based on the Java programming language. As a distributed object system it provides a global naming scheme for agents, a message passing scheme that supports both asynchronous and synchronous peer-to-peer communication between agents.

What differentiates the Aglets framework from other distributed object systems is the migration support. A *network agent class loader* allows the agent's Java byte code and state information to travel across the network, and an *execution context* provides agents with a uniform environment independent of the actual computer system on which they are executing. The package also defines the Agent Transfer Protocol (ATP) user to transfer agents over the network. ATP is an application-level standard protocol aimed at the Internet and using Universal Resource Locators (URL) for agent resource location, ATP offers a uniform and platform independent protocol for transferring agents between networked computers.

2.3 Objectspace Voyager

Objectspace Voyager is a Java Object Request Broker (ORB) designed to quickly and easily develop distributed systems. Besides the traditional features of an ORB it also contains support for mobile agents. The communication between object is done using *Voyager proxies* which are the equivalent of the CORBA or RMI *stubs* or the Bond *shadows*.

Voyager also contains a component model. A Voyager component extends the object with special interfaces (IIdentity, IMobility, ILifecycle and IProperty) that add value to the object.

Persistence is defined in Voyager as the ability of an object to live beyond an application's duration. Persistence is achieved by writing the object in a database. Objects written in the database are *autoloaded* and made active when a message is delivered to them.

Voyager also provides an agent framework. The main feature of a Voyager agent is its mobility. The mobility of the agent relies on the fact that the Voyager proxies are transportable, so if an agent uses only proxies to access objects, its state can be migrated and the execution resumed at the remote location.

2.4 The Bond distributed object system

The Bond distributed object system (`http://bond.cs.purdue.edu`) is built on a message oriented structure, using KQML as messaging language. Bond objects [3] are network objects in the sense that they can communicate with each other, can be instantiated and run remotely.

KQML, Knowledge Querying and Manipulation Language, is a product of the Knowledge Sharing Effort supported by DARPA, NSF, and AFOSR, for organization and coordination of autonomous agents, [4, 5]. Intended as an inter-agent communication language by its designers, KQML is used in Bond as an inter-object communication language. In Bond all objects can receive and send messages.

Although every Bond object fully understands the syntax of KQML, their understanding is limited to messages related to their functionality. Instead of the language/ontology pair as specified in the original design of KQML we introduced a different approach based on *subprotocols.* This extension is fully compatible with the KQML standards. Subprotocols are small, closed subsets of KQML commands. In programming languages terminology we can think of them as small, specialized languages. The attribute *closed* in this definition means that commands in a subprotocol do not reference commands outside the subprotocol, and the reply or acknowledgment is always a member of the same subprotocol with the question.

Subprotocols generally contain the messages needed to perform a specific task. Examples of generic Bond subprotocols are *property access* subprotocol, *agent control* subprotocol or *security* subprotocol. An alternative formulation would be that subprotocols introduce a *structure in the semantic space of the messages.* To create a fully functional distributed system, a typical object should implement a number of subprotocols. We call a *message pattern* the totality of messages a distributed object system should use in order to accomplish a certain task.

Two objects can communicate using messages which are members of the subprotocols implemented by both objects. Every Bond object implements at least the *property access subprotocol* which allows to remotely interrogate and set the properties of an object. The subprotocols implemented by an object is also a property of the object. If two objects want to communicate without having any previous knowledge about the other, the first thing to do is to interrogate the `SubprotocolsImplemented` property. After this, they can communicate using the intersection of the subprotocols implemented by both of them. A Bond object can implement a subprotocol in three ways: static implementation, acquiring subprotocols by probes and generating and learning new subprotocols.

Subprotocols are a key element in the implementation of Bond agents. For a more detailed description of Bond subprotocols, we refer the reader to [8].

3 The anatomy of Bond agents

The structure of the Bond agents presented in Figure 1 implements the theoretical model AM_1 as presented in [9].

The components of a Bond agent are:

- The **model of the world** is a container object which contains the information the agent has about its environment. This information is stored in the form of dynamic properties of the model object. 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

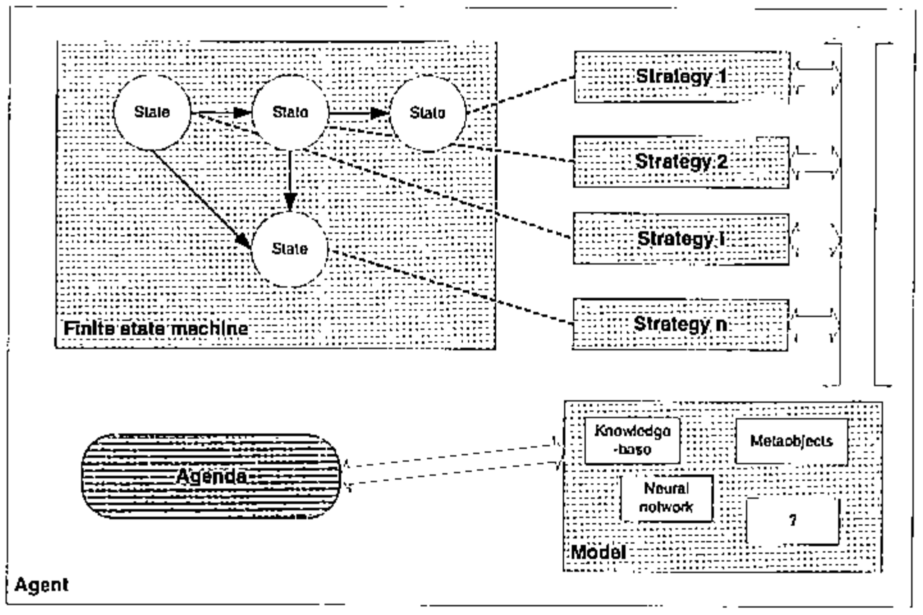

Figure 1: The anatomy of a Bond agent

neural network, a collection of meta-objects or different forms of handles of external objects (file handles, sockets, etc).

- The **agenda** of the agent, which defines the goal of the agent. The agenda is in itself an object, which implements a boolean function on the model 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 the agents marked as having 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 **finite state machine** of the agent. The current state is a model variable named `STATE-1`. Each state has an assigned strategy which defines the behavior of the agent in that state. An agent can change its state by performing *transitions*. Transitions are triggered by internal or external *events*. External events are messages sent by other agents or programs. The set of external messages which trigger transitions in the finite state machine of the agent defines the *control subprotocol* of the agent.

- Each state on an agent has a **strategy** defining the behavior of the agent in that state. Each strategy performs actions in an infinite cycle until the agenda is accomplished or the state is changed. Actions are considered atomic from the agent's point of view, external 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 by generating an internal event. After the transition the agent moves in a new state where a different strategy defines the behavior.

One of the interesting properties of the Bond agent framework is that all components are objects, which allow the Bond agents to be assembled dynamically and even modified during runtime. Another important feature is that the behavior of the agent is uniquely determined by the model (the model also contains the state which defines the current strategy). The model being in essence a data object can be saved, transferred over the network allowing for checkpointing and migration of Bond agents. The atomicity of the actions gives us natural checkpointing points.

3.1 Specifying an agent with the blueprint language

The Bond agent framework can be programmed at two levels. At the expert level, the developer can define its own new strategies and agendas by programming them directly in Java. At the *blueprint* level, the user can create new agents using the blueprint language of the Bond agent framework. With the blueprint a user can:

-specify the finite state machine of the new agent

-assign strategies from a strategy database to the states

-assign the agenda of the agent from the strategy database

-assemble new strategies following an *aspect oriented programming* approach using the strategy composition objects.

-assemble new agendas by applying boolean composition on predefined agendas

-create the control subprotocol of the agent

-initialize the variables of the model.

However a blueprint description can not define new strategies or agendas which can not described as boolean conditions on the model.

Some of the advantages of using blueprint over programming the agent in Java are:

- Blueprint programs are small text mode descriptions that can be transfered across the network, stored in a database or even embedded in KQML messages.
- Blueprint programs are sufficiently simple that they can be generated or modified by an agent. The possibility of agents building new custom agents opens up exciting new possibilities.
- There is a possibility to modify an agent at runtime by interpreting a new blueprint script.

3.1.1 The `blueprint` language

`blueprint` is a descriptive language for a Bond agent. There is two different way of using `blueprint`:

- *compiling:* the `blueprint2Java` compiler transforms a `blueprint` description into the static Java implementation of the agent.
- *interpreting:* the AgentFactory calls an interpreter and creates the agent at runtime.

Interpreted agents pay a penalty in their startup time, but at the runtime they behave exactly like the compiled agent. The startup penalty is usually small (under a second) so as a general rule the advantages of dynamic generation outweigh the disadvantage of the interpretation time.

```
blueprintprogram ::=
  ``begin'' ``blueprint'' ``agent'' IDENTIFIER ``;''
  [ Imports ]
  [ AddingStates ]
  [ AddExternalTransitions ]
  [ AddInternalTransitions ]
  [ SetAgenda ]
  [ InitializeModel ]
  ``end'' ``blueprint'' ``.''

Imports ::= (Import)*

Import ::= ``import'' JAVA_PATH ``;''

AddingStates ::= ( AddState )*

AddState ::=
   ``add'' ``state'' IDENTIFIER
   ``with'' ``strategy'' Strategy  [InitializeModel] ``;''

AddExternalTransitions ::=
   ``external'' ``transitions'' ``{``
   (AddTransition)+ ``}''

AddInternalTransitions ::=
   ``internal'' ``transitions'' ``{``
   (AddTransition)+ ``}''

AddTransition ::=
   ``from'' IDENTIFIER ``to'' IDENTIFIER ``on'' IDENTIFIER ``;''

SetAgenda ::=
   ``set'' ``agenda'' ``to'' Agenda ``;''

Strategy ::=
   IDENTIFIER |
   IDENTIFIER ``.'' IDENTIFIER |
   IDENTIFIER ``.'' IDENTIFIER ``::'' IDENTIFIER |
   ``composed'' ``{`` ( Strategy [InitializeModel] )* ``}'' |
   ``parallel'' ``{`` ( Strategy [InitializeModel] )* ``}''

Agenda ::=
   IDENTIFIER |
   IDENTIFIER ``.'' IDENTIFIER |
   ``(`` Agenda ``)'' |
   Agenda ``or'' Agenda |
   Agenda ``and'' Agenda

InitializeModel ::=
   ``model'' ``{`` ( SetModelVariable )+ ``}''

SetModelVariable ::= IDENTIFIER ``='' VALUE ``;''
```

4 The life cycle of an agent

A bondAgent can be created in the following ways:

- **Statically:** an agent derived from the *bondAgent* framework can create its components (finite state machine, states, strategies etc.) in its constructor.
- **Dynamically:** the agent is created by a factory object `bondAgentFactory` using a *blueprint*. The factory object generates the components of the agent either by creating them, either by loading them from persistent storage.

In the following, we present the lifecycle of the agent.

4.1 Creating an agent

A Bond agent can be either *instantiated*, if it is a statically created agent, or *assembled* from library components using a *blueprint*. If the agent is instantiated as a new thread in the current Bond executable it can be created as any other Java object, as in the following example:

```
bondAgent ba = new bondExampleAgent();
```

However, Bond agents are normally created using the `bondAgentFactory` object. This object allows us to remotely instantiate agents:

```
bAF.say("(achieve :content create-agent
                  :agent bondExampleAgent
                  :subprotocol AgentControl)")
```

In this case the variable `bAF` can be either a `bondAgentFactory` object or a shadow of it. This gives us the first advantage of the use of the factory, because agents can be instantiated remotely in a transparent way.

Another advantage is that the agents may be assembled dynamically from a blueprint, a text mode description of the agent. In this case the request can be:

```
bAF.say("(achieve :content assemble-agent
                  :blueprint http://bond.cs.purdue.edu/blueprints/Example.bpt
                  :subprotocol AgentControl)")
```

where we are actually instructing the factory to download the blueprint from the specified URL, and assemble the agent according to the specifications in the blueprint from the library components available locally. Alternatively, we can embed the blueprint in the message.

Whichever way of creating the agent we are using, the agent factory will reply with a message communicating the bondID and address of the new agent.

```
(tell :content agent-created
      :bondID the-new-bondID :address ector.cs.purdue.edu:2001
      :subprotocol AgentControl)
```

The usual procedure is that the object that requested the creation of the new agent (the *beneficiary* of the agent) creates a shadow of the agent `shAgent`, used for further interaction with the agent. The agent factory also creates a shadow of the agent's beneficiary, thus establishing a two-way relationship between the agent and the beneficiary. After creation the agent factory will no longer be used during the lifetime of the agent. This creation process is summarized in Figure 2

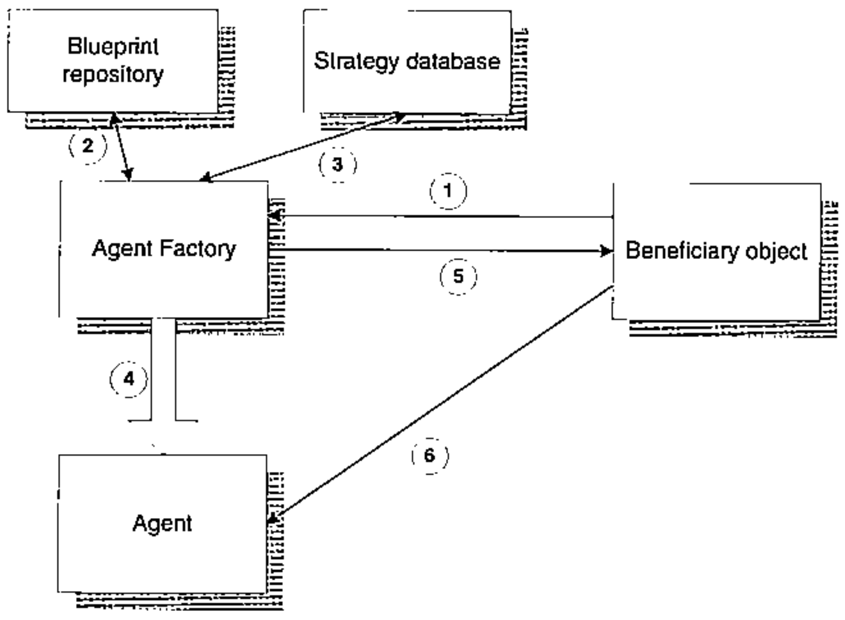

Figure 2: Creating an agent remotely using an agent factory. (1) The beneficiary object sends a create-agent message to the agent factory (2) The blueprint is fetched by the agent factory from a repository or extracted from the message (3) The strategies are loaded from the strategy database (4) The agent is created (5) The id of the agent is communicated back to the beneficiary, and (6) The beneficiary object controls the new agent

4.2 Initialization

After finishing the creation of the agent, the agent becomes an active bondObject. This means that it can receive and send messages, but does not yet have a thread on its own. The reason why agents are not started immediately after creation is because their agenda may be incomplete without some input. For example the agenda of an agent which performs the supervised execution of a legacy application is incomplete without the path of the program to be run. In its initialization phase the beneficiary gives information about the world to the agent, by writing it in the agent's model. Typically this information is related to the agenda of the agent, but a beneficiary can also provide some useful information about the current state of the world to the newborn agent.

For example, for the executor agent mentioned before we can set the command to be executed by:

```
shAgent.say("achieve :content setModel :name Commandline :value matlab"+
            ":subprotocol AgentControl)", this);
```

Of course the assumption is that the name `Commandline` has a meaning for the agent. This topic will be explored in more detail when we present strategies and namespaces.

4.3 Starting

After the initialization is finished the program can start the agent by sending a start message:

```
shAgent.say("achieve :content start-agent :subprotocol AgentControl)", this);
```

or, if the agent way instantiated locally we can simply use:

```
agent.start();
```

The agent creates its internal thread, initializes its state to the default state specified in its constructor or in the blueprint and starts to pursue its agenda according to the strategy associated with the current state.

4.4 Running

The normal way of operation of the agent is being into one of its states and performing actions according to the strategy in order to accomplish its agenda. Normally the agent performs an infinite loop by periodically asking the strategy about the next action to be executed. If there is no strategy associated with the current state, the agent waits.

Agents move from one state to the other by means of *transitions*. Transitions are triggered by events which reach the agent. The events may be *internal* if they are generated by a strategy or *external* if they are messages sent by external objects. Internal events are the `success` or `failure` events generated by strategies. External events are KQML messages send by remote or local objects to the agent. The set of KQML messages which trigger transitions in the agent form the *control subprotocol* of the agent. If the agent is assembled dynamically from a blueprint the control subprotocol is also generated dynamically.

4.5 Termination

An agent terminates when its agenda is accomplished (i.e. when the agenda object's `satisfiedBy(model)` function returns true). Agents which have an agenda marked with a continuous goal never terminate, unless interrupted from the outside.

If the agent has no beneficiary, the agent immediately exits. If the agent has a beneficiary, it sends a termination message, terminates it thread, but the objects associated with the agent, e.g, the model are not destroyed. The reason for this is that after the agenda is satisfied the beneficiary can read out values from the model - which may represent results of a computation, the very reason why the agent was started. Also the beneficiary may change values in the model and restart the agent, without the need to create it again. (Restarting the agent without changing the model does not make sense because the agenda uses only its model as input, and the agenda will be immediately satisfied upon restart).

The agent can be killed by the beneficiary by sending a `kill-agent` message

```
shAgent.say("achieve :content kill-agent :subprotocol AgentControl),this);
```

The same message can also be used to terminate agents with a continuous agenda.

5 Control and autonomous operation of the Bond agents

The behavior of an agent is completely determined by its internal state information. The state information of a Bond agent is contained entirely in the model of the agent. The external world influences the behavior of the agent only by its reflection into the model. There is however one model variable which requires a special handling, the current state of the finite state machine of the agent, because this variable determines the current strategy used by the agent.

To control the behavior of an agent, we need to set the state of the finite state machine. The state of the agent can not be changed arbitrarily. An internal or external object can change the state of the agent by triggering transitions in the finite state machine of the agent. The transitions are labeled by events, whose generation trigger the specific transition.

In this section we present the methods for controlling an agent from the inside, by the current strategy or from the exterior by other agents. We also present in more detail the structure of the strategies, and the possibilities offered by them to the agent programmer.

We consider Bond agents as being both *controllable* in the sense that their current strategy can be changed by a controlling authority sending external events, and *autonomous* because the strategies take actions only based on the agents agenda and the current knowledge as reflected in the model.

5.1 Internal control of the agent

The agent can be controlled internally by transitions generated by the strategies. A strategy generates *internal events*. There are two reserved internal events, `success` and `failure`, but the strategy can use any other events. These events are reserved to facilitate the assembly of agents from reusable strategies. Generating an internal event can be done in a strategy by the instruction like:

```
fsm.transition("success");
```

This should be the last statement within the action function of the agent, because the transition will change the state and the current strategy of the agent.

A strategy does not know about the structure of the finite state machine it is embedded into. Generating an internal event will trigger a transition which terminates the strategy by changing the state. The label of the internal event indicates the way in which the strategy was terminated. However, the new state and the associated strategy depends exclusively on the finite state machine of the agent.

For example, an internal event labeled `failure` usually takes the agent into a state which corresponds to an error. However, depending on the blueprint of the agent, this state may have a strategy which attempts to recover the error, or a different strategy which simply exits.

5.2 External control of the agent

An agent can be controlled externally using the messages in the `AgentControl` subprotocol, defined by the Bond system and allows the external object to create, destroy and interrogate the state of an agent. Besides this, every agent defines its own *control subprotocol*, the set of external messages that trigger transitions in a specific agent. This subprotocol is specific for every agent and it is determined by a finite state machine.

The control subprotocol of an agent is in the `Subprotocol` property of the finite state machine. A remote object can *learn* the subprotocol of the agent by reading this variable using the Property Access subprotocol.

5.3 Security aspects of agents

The distributed execution and remote control of the agents pose security threats. The Bond agent framework is integrated with the rest of the Bond security system, and allows the programmer to set the type of security he considers necessary for the current application.

The maximal security is obtained by setting the `beneficiary_only` variable of the agent. When set this switch forces the agent to accept only messages from its creator or replies to its own messages. Although this approach seems very restrictive it allows the creation of hierarchical agent systems as required by a large number of practical applications.

If a fine-grain security approach is needed, Bond agents, like any other Bond object can have a number of *security probes* attached. The Bond library contains security probes for a variety of security paradigms like firewall type security, password based access, ticket based access and authentication based access. A user may define its own security probe. Also, different access protocols may be set up for various types of messages [6].

5.4 The implementation of strategies in Bond

In every state of the internal finite state machine which has a non-null strategy associated to it, the agent performs actions according to the strategy. We call this mode of operation *autonomous*.

A strategy was defined as a function on the model and the agenda, and the agenda itself is a function on the model. This definition is a practical necessity because the agent has no direct access to the objects of the world. All the actions of the agents are based upon the model rather than the real world. However, the real goals of the agent do not refer to the model, but to the real world. This means that the strategy has to solve two simultaneous problems:

(a) bring the model to a status where the agenda is satisfied

(b) ensure that the model is reflecting the reality as closely as possible.

We conclude that the actions of the agent are: (a) directed to change the status of the world, but based on the status of the model and (b) update the model.

A Bond strategy is an object derived from `bondStrategy` which implements the function `nextAction()`. Whenever a transition occurs in the finite state machine of the agent normally a new strategy is installed. The `install()` function of the strategy is a good place to verify the state of the model, create the new variables as needed. If a strategy is going to access model variables frequently, it is a good idea to create some local pointers to the model variables, eliminating the cost of the name-based access.

The functionality of the strategies are embedded in the `nextAction()` function called by the agent in an infinite loop. The time spent by the agent in a particular call of `nextAction` should be limited. An action is *atomic* in the sense that it can not be interrupted by a transition.

The actions performed in the `nextAction()` function should be uniquely determined by its parameters: the model and the agenda. This allows the agent to be interrupted, saved/restored and migrated.

5.4.1 Strategies with and without a state

An important classification criteria for a strategy if it has a state or not.

- **Stateless strategies** where every instance of the strategy is equivalent. For example a search-based scheduling strategy is stateless. This means that the agent factory can create a new strategy for each agent based only on the class code.

- **Strategies with state** where different instances of the strategy object represent different strategies depending on their state. The agent factory should load the specific instance of the strategy from a persistent storage. An example is a neural-network based strategy, where an already trained neural-network is saved and reused repeatedly for the specific task it was trained for.

5.4.2 Strategy composition

The behavior of the agent in each state is determined by the strategy associated with the state. Usually the behavior of a complex agent is multifaceted in any given state. For example the agent performs an action, collects information about the environment and does some housekeeping operations. Although it is unrealistic to expect that we can build a strategy database comprehensive enough to deal with every function the agent designer wants to perform, usually at least some aspects of the agent's behavior are standard enough that they can be implemented using strategies from the database. These approach is made possible by the *strategy composition* mechanism in the Bond agent framework.

The strategy composer implements an interleaving mechanism, which allows a new strategy to be composed directly from a number of existing strategies. The strategies can be composed in a *round-robin* approach, when the strategies are allowed to take actions each after another, or in *parallel* when the actions of different strategies are executed concurrently in separate threads. The strategy composition mechanism is implemented by the `bondCompositeStrategy` object and is supported by the Blueprint language. There is a possibility to add or remove strategies during runtime, but this approach is not supported by Blueprint.

The strategy composition is a form of aspect oriented programming [7], where the composer object performs a similar function with the weaver in AspectJ.

Practically every well-written strategy can be used in a composition. However a special care should be taken for the possible interactions between strategies through common model variables. This problem is even more difficult for the parallel composition where race conditions may occur. Preferably, composed strategies would use disjunct namespaces, but this is not always possible.

Another problem is referring to the transitions from composed strategies. The default composer (`bondCompositeStrategy`) implements the following rules.

- *first failure* - the first failure transition from one of the strategies will trigger a failure transition for the composed strategy.
- *last success* - the last success transition triggers a success transition for the composite strategy. Previous success transitions only deactivate the given strategy (after a success transition, the action function of the strategy will not be called any more).
- all other transitions generated from any of the member strategies are translated as transitions for the composite strategy.

One of the immediate applications for strategy composition is to add timeouts to existing strategies by composing them with the `Util.WaitAndFail` strategy from the strategy database, which waits a period of time specified in the model variable `TimeOut` and then performs a failure transition. The following Blueprint sequence is creating a strategy which performs a computation which, if not terminated, will be interrupted after 30 seconds.

```
add state BoundedCalc with strategy
  composed {
     MyStrategy.Calc;
     Util.WaitAndFail { TimeOut = 30};
  };
```

5.5 The model of the agent

We define the model of the agent as the collection of the information the agent has about the world. This leads us to the difficult problem of *knowledge representation*. The model in Bond agents is represented with a `bondModel` object. As any Bond object, the model can contain an indefinite number of $(item, value)$ pairs as dynamic properties. The Bond agent framework does not impose by default a structure on the model. In essence, every strategy is free to write and read anything in the model, allowing the agent developer to use its own knowledge representation. At creation time, every agent starts with an empty model. During initialization the beneficiary can initialize the model with information useful for the agent to know before started. Further on, every strategy reads information from the model, and writes back new knowledge.

Many agents need to handle some standard situations like running remote programs, synchronizing with other agents, performing data transfers, performing operations on trigger conditions etc. Also, certain *aspects* of the strategies may be standard and reusable, like security checks, logging and checkpointing. The Bond framework provides a strategy database which allow developers to concentrate only on those aspects of their agent which is indeed particular.

However, the main requirement seemless cooperation between strategies is the fact that they should understand each others knowledge representation in the model. The solutions we are exploring are basically naming conventions applied to the model variables.

5.5.1 Namespaces

A namespace is a subspace of the names available for use in the model. The variables in a namespace have a common prefix. Strategies from a strategy group use a common namespace. For example, the model variable `commandline` being in the namespace `program` is recorded with the name `program.commandline` in the model. The `getModel()` and `setModel()` functions of strategies hide the namespace from the strategy. If the current namespace of the strategy group is `program` than `getModel(''commandline'')` is translated in `model.get(''program.commandline'')`. Namespaces avoid name conflicts between identical strategies applied to different objects from the environment, and also present a basis for evaluating predicates on the agent.

6 Case study: remote execution agent

In this section we present the development process for a Bond agent using the Blueprint language and the strategy database. The example we have chosen, a legacy application wrapper is a relatively simple case, with a large number of practical applications. A wrapper is an agent capable to start and control the execution of a legacy application and at the same time to communicate with other agents towards a common goal. The strategies and even the blueprint presented here can be reused in any of the cases where an agent should execute an external program.

6.1 From specification to the blueprint

We start from a informal specification of the agent. We want to build an agent which executes a legacy application. The agent should continuously supervise the execution, collect and make available the standard output in real time. The termination of the legacy application should be captured real-time, and the error messages (if any) should be captured and made available.

We have additional requirements related to pre and post-processing.

- Prepare the inputs of the legacy application by fetching the input files from remote sites, and the output files may need to be transfered to various places on the network (data staging).
- On termination, delete the temporary files created by the program and the input and output files which are not needed any more (garbage collection).

- If the execution of the program failed, try to execute the program on a different machine (error recovery).

For the first step, we ignore the additional requirements and concentrate only on the basic tasks of the program. It is obvious that we need a number of states which correspond to the different stages of execution. For a programmer who prefers to visualize programming concepts, it may be the best approach to draw the finite state machine. Our proposal is presented in Figure 3. Although the programmer should not stick to our format of drawing, it is a good idea to make a distinction between the external and internal transitions, because this allows us to immediately identify the control subprotocol of the agent.

The next step of creating the agent is to identify the strategies we need to perform the operations required in each status. The first step is to determine if there are strategies available in the strategy database. If this is a generic, well known problem, like in our case the execution of an external application, there is a good chance that there are already strategies which are doing it. In our case the strategies grouped in `bondExecStrategyGroup` are performing this. The documentation of the strategy group shows that we have three member strategies: `Starting`, `Supervising` and `ProgramTermination`. We will attach the strategies to the corresponding states. The result is presented in 3. For the time being we don't have a strategy to handle errors, so we attach a null strategy there.

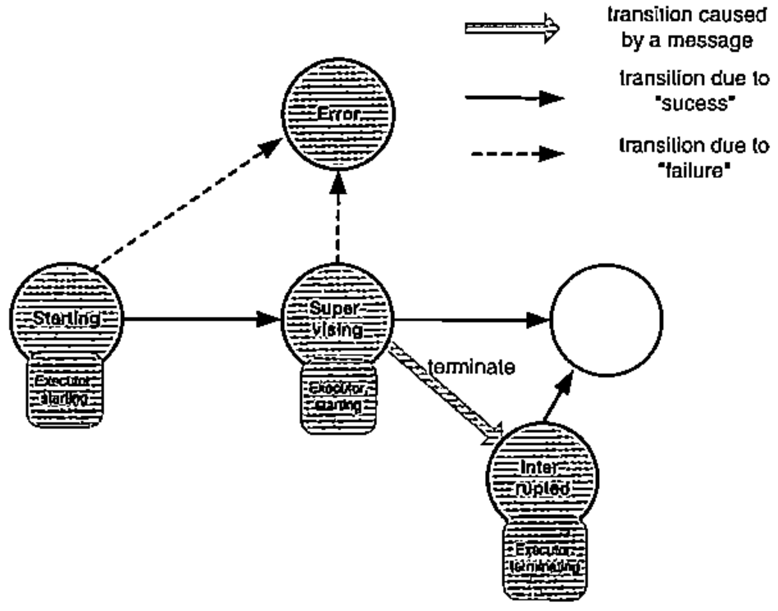

Figure 3: The finite state machine of the executor agent according to the initial specification

After assigning strategies the next step is to decide the namespaces the strategies are going to operate in. The rules are:

- The name of the namespaces should be chosen in an intuitive way.
- Interrelated strategies which describe different phases of an operation should go into the same namespace.
- Strategies which perform the same operation on different objects should go into different name spaces.

For trivial cases, we can use `default` as the namespace.

Now we have enough information to build the blueprint of the executor agent, which is presented in Figure 4.

```
begin blueprint
  agent Executor;
   add state Starting with strategy Executor.starting;
   add state Supervising  with strategy Executor.supervising;
   add state Error;
   add state Interrupted with strategy Executor.terminating;

   external transitions {
     from Supervising to Interrupted on terminate;
   }

   internal transitions {
     from Starting to Supervising on success;
     from Supervising to Terminated on success;
     from Starting to Error on failure;
     from Supervising to Error on failure;
   }

   set starting state to Starting;
   set agenda to Executor.finishedProgram;
end blueprint;
```

Figure 4: The blueprint of the executor agent

6.2 Extending an agent

The agent presented above is able to provide a solution for the basic requirements we specified. Now we show how we can extend the blueprint to handle the data staging problem.

We assume that the application we are building requires its input files to be accessible locally. Our goal is to bring the input files to the local machine before the program is started, the so called *data staging problem*.

We assume that we know the location of the remote files. The strategies in the `bondDataStagingStrategyGroup` can be used exactly for this. We are introducing a new state which handles the datastaging. The new finite state machine is shown in Figure 5.

We can create the new agent by modifying the blueprint. The difference is only four lines, and involves adding the new state, specifying its transitions, and setting the data staging as the new starting state. Figure 6 shows the new blueprint.

7 Conclusion and future work

In this paper we introduced an object oriented framework for building collaborative network agents. We view the agent as a composite object consisting of several other objects including a finite state machine, a model of the world, strategies associated with every state and an agenda.

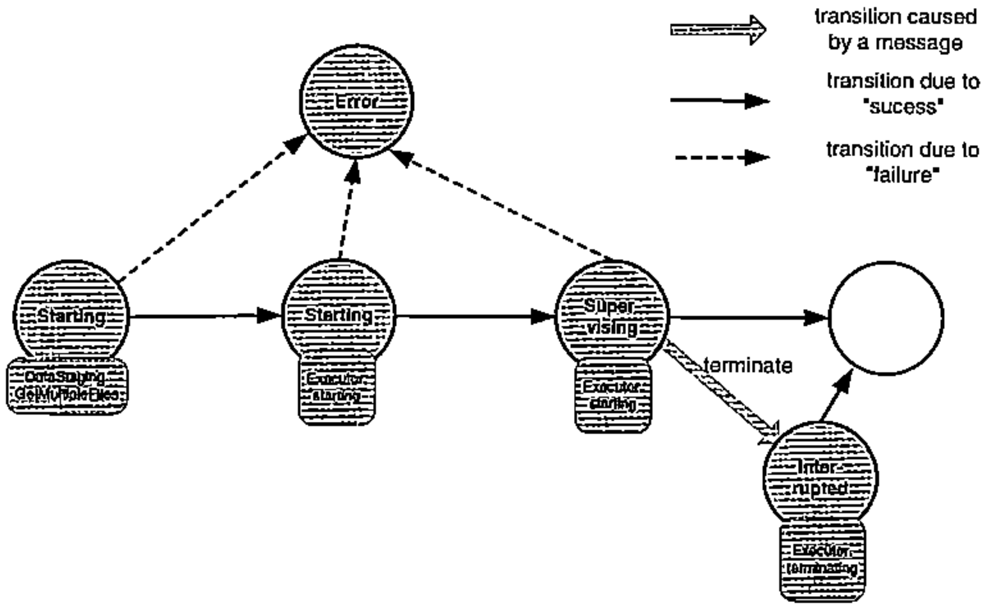
transition caused by a message
transition due to "sucess"
transition due to "failure"
Error
Starting
DataStaging GetMultipleFiles
Starting
Executor starting
Super-vising
Executor starting
terminate
Inter-rupted
Executor terminating

Figure 5: The finite state machine of the executor agent

```
begin blueprint
  agent Executor;
   add state BringingFiles with strategy DataStaging.GetMultipleFiles;
   add state Starting with strategy Executor.starting;
   add state Supervising  with strategy Executor.supervising;
   add state Error;
   add state Interrupted with strategy Executor.terminating;

   external transitions {
     from Supervising to Interrupted on terminate;
   }

   internal transitions {
     from Starting to Supervising on success;
     from Supervising to Terminated on success;
     from Starting to Error on failure;
     from Supervising to Error on failure;
     from BringingFiles to Error on failure;
     from BringingFiles to Starting on success;
   }

   set starting state to BringingFiles;
   set agenda to Executor.finishedProgram;
end blueprint;
```

Figure 6: The blueprint of the executor agent

We introduced an agent definition language called blueprint and describe mechanisms to create dynamically agents using an agent factory supplied with each Bond resident. An agent can be remotely controlled by its creator and possibly by other objects.

The Bond Agent Framework (BAF) allows a seamless integration of a reasoning system into an agent. For example an agent required to select one of several alternatives may by initially designed to make a random choice and when all other aspects of its design are satisfactory, the random choice strategy may be replaced by a strategy using an inference engine. This can be accomplished by modifying a single line of its description, or dynamically by sending a modified blueprint to the agent factory.

BAF also supports agent mobility. To migrate an agent from one site to another it is sufficient to send the blueprint and the model to the agent factory at the new site.

We also envision automatic assembly of agents by other agents from pre-existing components. Our framework encourages software reuse, various components of an agent including blueprints and strategies can be acquired from repositories.

We conclude the paper with an example of a wrapper agent capable to start-up a legacy application, and support its execution and communicate with other agents. A more sophisticated application a network of PDE solvers is described in [10].

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] L.N. Foner *What's an agent, anyway? A sociological case study* Agents Memo 93-01, Agents Group, MIT Media Lab.

[3] L. Bölöni: *Bond Objects – a white paper* Department of Computer Sciences, Purdue University CSD-TR #98-002

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

[5] Y. Labrou, T. Finin *A Proposal for a new KQML Specification* UMBC TR-CS-97-03

[6] R. Hao, L. Bölöni, K. Jun, and D.C. Marinescu *An Aspect-Oriented Approach To Distributed Object Security* Technical Report, Department of Computer Science, Purdue University, CSD-TR#98-038

[7] G.Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, J. Irwing *Aspect oriented programming* Proceedings of the European Conference of Object-Oriented Programming (ECOOP), June 1997

[8] 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* Submitted to the Fourth IEEE Symposium on Computers and Communications (ISCC'99)

[9] L. Bölöni and D.C. Marinescu *A multi-resolution model for building agents* Technical Report, Department of Computer Science, Purdue University.

[10] P. Tsompanopoulou, L. Bölöni and D.C. Marinescu *The Design of Software agents for a Network of PDE Solvers* Technical Report, Department of Computer Science, January, 1999.