Chapter 1

AN OBJECT-ORIENTED FRAMEWORK FOR BUILDING COLLABORATIVE NETWORK AGENTS

Ladislau Bölöni
Computer Sciences Department
Purdue University
West Lafayette, IN 47907
boloni@cs.purdue.edu

Dan C. Marinescu
Computer Sciences Department
Purdue University
West Lafayette, IN 47907
dcm@cs.purdue.edu

Contents

1. INTRODUCTION

The are various, sometimes conflicting, definitions in the literature for intelligent or autonomous agents. Stan Franklin and Art Graesser in their overview paper [10] 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.* The taxonomy provided in [10] identifies several types of agents, namely biological, robotic, software, and artificial life agents. Examples of software agents are task-specific agents, entertainment agents, and viruses. A software agent is expected to posses in some degree attributes like: reactivity, the ability to selectively sense and react, autonomy, collaborative behavior, inferential capability, adaptivity, mobility, the ability to communicate with other agents and humans using languages closer to the natural languages than the protocols used for program to program communication [4].

Other authors, like Leonard N. Foner [8] are considering the anthropomorphism of an agent an important factor. Our understanding of an agent is largely equivalent with the one of Franklin and Graesser, and we are not considering the features of user interface as a defining element of the agent, because some of the Bond agents may have immediate interaction only with other agents.

Current work in distributed object systems demonstrates the need to construct and control autonomous software agents. Although most of the software agents we are currently using have no reasoning based intelligence in the classical AI sense, it is important that the agent framework presented here allows the seamless integration of a reasoning system into the agent. The agent framework automates external control to start an agent, the acquisition of information (soft state) about the world and utilizes a generic framework for appending strategies. The strategies currently range from simple table lookup (for the monitoring agent) to relatively complicated search engines (for the metaprogram scheduling agent).

1.1 AGENT-BASED FRAMEWORKS FOR INTEROPERABILITY

We are primarily interested in the design of software agents supporting interoperability in a heterogeneous computing system. Important applications in commercial, industrial, and scientific computing areas are naturally distributed and require interoperability of various software systems developed independently and running on computing systems with various architecture and system software. For example there are several Electronic Commerce Frameworks, ECFs, developed by IBM, Sun, Oracle, Microsoft; interoperability among these ECFs must be ensured to guarantee the success of Electronic Commerce. In the scientific computing area, several projects in the area of meta-computing on a grid are under way. A computational grid consists of a number of independent nodes with different architecture and resources, interconnected by a high speed network. Metacomputing on a grid requires means to annotate network resources and to schedule activities on the nodes of the grid, [9].

CORBA, Common Object Request Broker Architecture, provides an answer of the distributes systems community to the problem of interoperability. CORBA uses a declarative language, IDL, the Interface Description Language, to describe the methods of an object, defines communication protocols for a distributed object system, ORB, Object Request Broker and IIOP, Internet Inter-Object Protocol, as well as a set of services like interface repository, directory, event, and so on. The Java platform, provides another architecture independent solution for building complex systems and for accommodating heterogeneity.

CORBA and Java represent significant steps towards interoperability in a distributed object system. The search for solutions to the more intricate aspects of automatic coordination of activities in a complex system leads to the consideration of software agents. Some of the limitations of direct manipulation interfaces identified by Bradshaw [4], namely difficulties in handling large search spaces, rigidity, and the lack of improvement of behavior, extend to most other facets of traditional approaches to interoperability. Software agents can filter large amounts of information, they can be instructed at the level of goals and strategies to find solutions to unforeseen situations, and they can use learning algorithms to improve their behavior.

In his paper on Agent-Oriented Programming, AOP, Shoham, [20] assimilates an agent with "an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices, and commitments". He argues that AOP is a specialization of the Object-Oriented Programming paradigm, OOP: "whereas OOP views a computational system as made up of modules that are able to communicate with one another and have individual ways of handling incoming messages, AOP specializes the framework by fixing the state (now called mental state), of the modules (now called agents), to consist of precisely defined components called beliefs (including beliefs about the world, about themselves, and about one another), capabilities, choices and possibly other notions.

A computation now consists of the agents informing, requesting, offering, accepting, rejecting, competing, and assisting one another."

Genesereth describes an agent-based framework for interoperability [11] and argues that current subroutine libraries provide limited support for automatic coordination and are "limited by the expansiveness of the languages used to document resources and by their lack of inferential capabilities". He underlines the importance of Agent Communication Languages, ACLs and advocates a federation architecture. In this architecture agents do not communicate directly with each other, they communicate with entities called *facilitators*, and facilitators in turn communicate with each other. The agents surrender their autonomy to facilitators. Upon connecting to a facilitator an agent provides a specification of its capabilities and needs in ACL. Agents can dynamically connect and disconnect from facilitators. Messages are sent to facilitators and the facilitators route the messages to agents. Facilitators are capable to translate a message, decompose a message into several messages and route them independently, combine several messages, they can also establish links between agents and allow them to communicate directly. ACL consists of three components, an "inner" language called KIF (Knowledge Interchange Format), a vocabulary, and an "outer" language, KQML (Knowledge Query and Manipulation Language). An ACL message is a KQML expression consisting of sentences in KIF formed using the ACL vocabulary. KIF is prefix version of the language of first order predicate calculus.

1.2 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.

Infospheres (`http://www.infospheres.caltech.edu/`) 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/replies. Djinns are running as separate threads inside a *master djinn* which provides remote activation capabilities. A related project is Übernet which provides a very flexible communication infrastructure allowing users to assemble custom protocol stacks.

IBM Aglets (`http://www.trl.ibm.co.jp/aglets/index.html`) provide 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.

Objectspace Voyager (`http://www.objectspace.com`) 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 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 its execution resumed at the remote location.

Bond (`http://bond.cs.purdue.edu`) is a project to provide middleware for network-centric computing. It consists of a core distributed object system built on a message-oriented structure, using KQML as messaging language. The system includes several frameworks for monitoring, security, software agents and metaobjects. Bond objects [1] are network objects in the sense that they can communicate with each other, can be instantiated and run remotely. An overview of the system is presented in Section 3..

2. DESIGN PRINCIPLES FOR AN AGENT-BASED SYSTEM

The problems we address in our research on software agents are:

- an open architecture integrating software agents into a distributed-object system,
- constructive models for software agents favoring software reuse, and supporting the development of a continuum of agents with various levels of inferential capabilities, adaptability, reactivity, and communication abilities,
- means to integrate multiple knowledge representation models into software agents, e.g. metaobjects and systems based upon first order predicate-logic,
- applications to the development of software agents critical for network computing e.g. scheduling agents, coordinator agents, mediator/broker agents, discovery agents, monitoring agents, wrapper agents, match-making agents and so on.

We are investigating alternative architectures integrating distributed objects systems and software agents. We start by considering systems consisting of groups of agents with known communication patterns, and limited inference ability and delegate some of the functions of the facilitators to servers. We believe that this gradual approach is more likely to be accepted by the distributed system community than an "ab-initio" strategy where the inference ability of an agent is the only measure of success of the architecture.

2.1 INTEGRATION OF AGENTS INTO A DISTRIBUTED-OBJECT SYSTEM

The first question we have to answer when designing an agent-based, distributed-object architecture is the level of integration of agents and objects. Should the agents be tightly integrated into the architecture, or can they be added on to an existing distributed-object framework?

To answer this question we examine two critical issues, communication and the complexity of the system.

Communication is a critical issue in a distributed system and though procedure-oriented and message-oriented paradigms are arguably dual to each other, most distributed-object systems favor remote method invocation and synchronous communication. The conventional wisdom is that remote method invocation is more efficient than message passing and that asynchronous communication is considerably more difficult to program and debug than synchronous one. Thus if we want to design a complex system, capable to evolve and to exhibit a good level of performance the only choice is synchronous remote method invocation.

On the other hand, agent-oriented programming favors the message-oriented approach and asynchronous communication. Since agents can exhibit a complex behavior, remote method invocation is too restrictive. Inter-agent communication is based upon message passing, using agent-communication languages like KQML. An agent has the ability to make autonomous decisions and should be able to adapt its behavior to the response to a message or to the lack of a response. In a wide-area distributed system, components including other agents or objects, as well as communication links may fail, thus the lack of a response should still lead to predictable behavior. The ability to respond to component failure is an important argument in favor of integrating software agents into network computing systems.

An approach discussed in the literature, is to support remote method invocation for regular objects and message passing for agents. We believe that this dichotomy increase the software complexity and creates additional problems when regular objects and agents need to communicate with each other. Our solution is to design a pure message passing system with all objects communicating using KQML. An agent is just another object extended with the ability to make autonomous decisions, it can communicate with any other object and the communication fabric is common to the entire object population.

Important as it may be, the fact that every object "speaks" KQML, in other words understands the syntax of a KQML message and is able to parse it, does not guarantee that two objects can cooperate with each other. An object should be able to "understand" semantically a message and respond to it in a coherent manner. The granularity of objects can differ widely, ranging from simple objects like an address, a date, or a message, to complex objects like a scheduling agent or an authentication server. A server is an active object capable to provide a well-defined set of services. Though it may be very complex, a server does not have the autonomy an agent must exhibit. The ability of an object to "understand" a message is also very different.

To solve the problem of semantic understanding of a message we introduce the concept of subprotocols. A subprotocol is a closed set of messages, an object understands semantically every single message in a subprotocol and it is able to respond with a message in the same subprotocol. This guarantees the ability of agents to cooperate. An agent may use a "discovery" subprotocol to determine a set of common subprotocols with another agent or object. Once a group of agents discover that they all understand a subprotocol any agent sending a message belonging to the common subprotocol is guaranteed to get a meaningful response. An object can determine with ease if it understands a message because every message is stamped with the subprotocol it belongs to.

A subprotocol is inherited based upon the object hierarchy. When a new object extending an existing one is created, in addition to the subprotocols inherited, new ones can be defined. The only subprotocol understood by all objects is the "property access subprotocol". All agents understand the "agent control subprotocol". The capabilities of an existing agent may be enhanced with *probes*, special objects capable to handle other aspects of a complex system, e.g. security, monitoring, accounting, and so on. We also have the ability to generate subprotocols dynamically once the communication patterns of a group of agents are known.

Another major architectural decision is to write the system in an architecture-neutral language, Java. Agent mobility may be an important consideration in some applications and code mobility is a necessary although not a sufficient condition for agent mobility.

2.2 COMPONENT-BASED AGENTS

A first distinctive feature of the Bond architecture, described in more detail in Section 3. of this paper, is that agents are native components of the system. This guarantees that agents and objects can communicate with one another and the same communication fabric is used by the entire population of objects. Another distinctive trait of our approach is that we provide middleware, a software layer to facilitate the development of a hopefully wide range of applications of network computing. We are thus forced to pay close attentions to the software engineering aspects of agent development, in particular to software reuse. We decided to provide a framework for assembly of agents out of components, some of them reusable. This is possible due to the agent model we overview now and discuss in depth in Section 4.

We view an agent as a finite-state machine, with a strategy associated with every state, a model of the world, and an agenda as shown in Figure 1.6. Upon entering a state the strategy or strategies associated with that state are activated and various actions are triggered. The model is the "memory" of the agent, it reflects the knowledge the agent has access to, as well as the state of the agent. Transitions from one state to another are triggered by internal conditions determined by the completion code of the strategy, e.g. success or failure, or by messages from other agents or objects.

The finite-state machine description of an agent can be provided at multiple granularity levels, a course-grain description contains a few states with complex strategies, a fine-grain description consists of a large number of states with simple strategies. The strategies are the reusable elements in our software architecture and granularity of the finite state machine of an agent should be determined to maximize the number of ready made strategies used for the agent. We have identified a number of common actions and we started building a strategy repository. Examples of actions packed into strategies are: starting up one or more agents, writing into the model of another agent, starting up a legacy application, data staging and so on. Ideally, we would like to assemble an agent without the need to program, using ready-made strategies from repositories.

Another feature of our software agent model is the ability to assemble an agent dynamically from a "blueprint", a text file describing the states, the transitions, and the model of the agent. Every Bond-enabled site has an "agent factory" capable to create an agent from its blueprint. The blueprint can be embedded into a message, or the URL of the blueprint can be provided to the agent factory. Once an agent was created, the agent control subprotocol can be used to control it from a remote site.

In addition to favoring reusability, the software agent model we propose has other useful features. First, it allows a smooth integration of increasingly complex behavior into agents. For example, consider a scheduling agent with a mapping state and a mapping strategy. Given a task and a set of target hosts capable to execute the task, the agent will map the task to one of the hosts subject to some optimization criteria. We may start with a simple strategy, select randomly one of the target hosts. Once we are convinced that the scheduling agent works well, we may replace the mapping strategy with one based upon an inference engine with access to a database of past performance. The scheduling agent will perform a more intelligent mapping with the new strategy. Second, the model supports agent mobility. A blueprint can be modified dynamically and an additional state can be inserted before a transition takes place. For example a "suspend" new state can be added and the "suspend" strategy be concatenated with the strategy associated with any state. Upon entering the "suspend" state the agent can be migrated elsewhere. All we need to do is send the blueprint and the model to the new site and make sure that the new site has access to the strategies associated with the states the agent may traverse in the future. The dynamic alteration of the finite state machine of an agent can be used to create a "snapshot" of a group of collaborating agents and help debug a complex system.

Agent security is a critical issue for the system because the ability to assemble and control agents remotely as well as agent mobility, provide unlimited opportunities for system penetration. Once again the fact that agents are native Bond objects leads to an elegant solution to the security aspect of agent design. Any Bond object, agents included, can be augmented dynamically with a security probe providing a defense perimeter and screening all incoming and outgoing messages.

2.3 METAOBJECTS

The world of software agents is populated with network objects: hardware platforms, communication links, sensors, programs, libraries, services, data files, and so on. Agents should be able to use and manipulate the network objects. This leads us to the question of information representation: what is the best way to store the information concerning the network objects. The large number of possible properties of a network object makes it difficult to organize the data in a tabular format, so the use of relational databases is impractical. Sometimes hierarchically organized directory services like LDAP are used to store information about network objects. Although Bond agents can use LDAP to gather the needed information from external databases, they use *metaobjects* as their internal knowledge representation method for representing network objects. The function of metaobjects is to provide the knowledge needed by the software agents to manipulate network objects.

The defining characteristics of metaobjects are:

- There is a one-to-one mapping between a metaobject and a corresponding network object. For example to a host computer corresponds a single metaobject.

- The properties of a metaobject mirror a subset of the properties of the network object.
- Unlike traditional objects, metaobjects do not have a fixed set of properties. Properties are added as they are discovered by the agent.
- Metaobjects have no fixed type, their type is evolving with their properties. For example, a metaobject which represent a "Host", later, with additional information may represent a "UnixMachine". From a practical programming point of view, of course this means that we cannot use static type checking for metaobjects.
- Metaobjects can be used to form a *distributed knowledge base*. Two agents can cooperate by (a) identifying the metaobjects which point to a common network object and (b) synchronizing the information between the two metaobjects.

There are no methods specific to a metaobject other than to set and reset their attributes. Agents are capable to manipulate metaobjects and invoke functions to determine if the metaobject provides the information needed by a specific strategy. The metaobjects provide the means to integrate object-oriented and agent-oriented programming.

To illustrate these ideas consider the scheduling agent example discussed earlier. The agent must be able to determine if a particular host is suitable for a computational task. We have the choice of creating a regular object with a number of attributes to reflect the most relevant properties of a host e.g. its IP address, the type and Id of the processor, the number of processors, the amount of main memory, and so on. We may ask ourselves what is a relevant property for a host, for example are the type of the graphics cards, or the number of geometric engines important? The answer is that in some cases the graphics hardware may not be present and in other cases we may not be interested in graphics when scheduling the task. Rather than defining a *host object* with a potentially very large set of attributes we define a metaobject with no other property but a name and add dynamically properties to it as needed.

The metaobject provides "genetic information" about a network object. Some of the traits of the network object are discovered at the time the metaobject was created others reflect changes in the network object itself discovered by some agent at a later point in time. For example a "discovery agent" will fill in the metaobject associated with a host with a number of (attribute, attribute-value) pairs, including the type and the Id of the processor. A selection strategy of the scheduling agent will match the attributes desired by the task with the ones offered by a specific host as reflected by the corresponding metaobject.

In summary, metaobjects are associated with network objects. Various strategies of one or more agents access a metaobject to augment the knowledge about the network object, or to use this knowledge when performing actions. The metaobject itself does not reflect a particular taxonomy of a class of network objects, it only provides "genetic information".

3. BOND MIDDLEWARE

This section introduces Bond, a Java-based, object-oriented middleware for network computing. Network computing is a paradigm that emphasizes the use of network resources, computing resources distributed across the network, over local resources. Network resources are hosts, communication links, programs, services and data. Middleware is a software layer that allows developers to mold systems tailored to specific needs from components and develop new components based upon existing ones. The goals of the Bond system are to:

(a) facilitate access to network resources,

(b) support collaborative activities, and

(c) accommodate software diversity and hardware heterogeneity.

Bond is a message-oriented system, it uses KQML [6], as a meta-language for inter-object communication. KQML offers a variety of message types (performatives) that express an attitude regarding the content of the exchange. Performatives can also assist agents in finding other agents that can process their requests. A performative is expressed as an ASCII string, using a Common Lisp Polish-prefix notation. The first word in the string is the name of the performative, followed by parameters. Parameters in performatives are indexed by keywords and therefore order-independent.

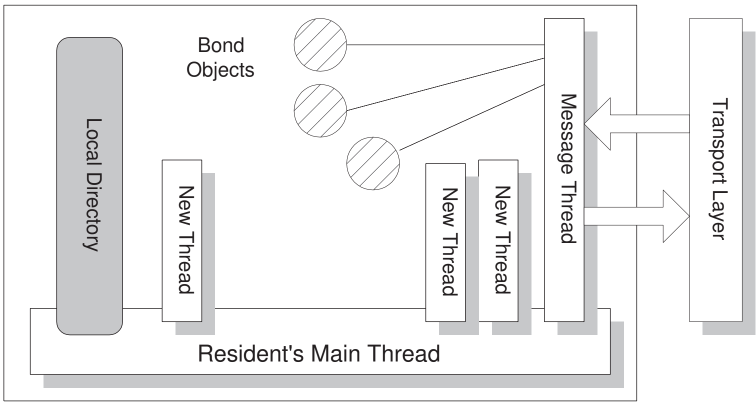

Figure 1.1 A Bond resident. All objects in the container, excluding messages and shadows are registered with the local directory. The messaging thread of the resident delivers messages to local objects identified by their `bondId`. Messages for remote objects are sent out by the messaging thread using the address of the remote container provided by the local shadow. The main thread of a resident may start up new threads as requested by various subprotocols.

3.1 BOND OBJECTS

A *Bond Object* is collection of data fields and methods, [1]. The data component of an object consists of several *fields* and a vector of *dynamic properties*. Each object has a *unique bondId*. The granularity of Bond objects ranges from simple objects, e.g. a message to complex objects like a directory server. Some objects are *passive*, e.g. a message or a metaobject associated with a data file, others are *active* they have one or more threads of control. Active as well as passive objects can communicate using the `say()` method. All objects are persistent. The full Bond object hierarchy is available at (`http://bond.cs.purdue.edu`).

A *resident* is an container consisting of several threads of control, a local directory and a set of objects. Bond objects with the exception of *light-weight objects* like messages and shadows always register with the local directory upon creation.

Figure 1.1 shows the structure of a resident. The messaging thread of a resident delivers messages from remote objects to local ones and sends out messages originating from local objects. Several other threads may be running as a side-effect of messages in subprotocols understood by the resident. An *workspace* is a passive container consisting of several, possibly many objects belonging to one user. An workspace can be saved

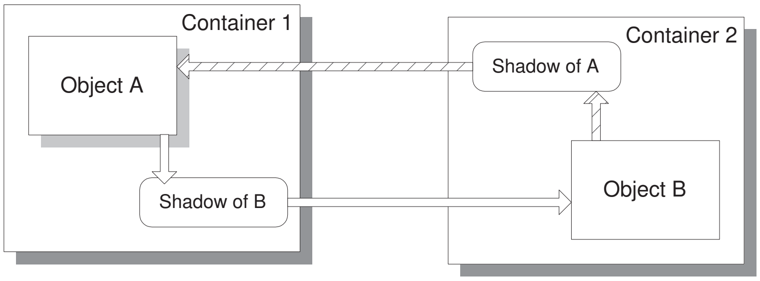

Figure 1.2 Communication between remote objects using their shadows. A shadow is a proxy for a remote object. To communicate with object B in container 2, object A in container 1 creates a shadow of B using the `find` function. Once the shadow of B is created, A applies the local `say()` method to send messages to it; then the shadow delivers the message to B. To send a message to A, B follows the same same procedure, creates a shadow of A then sends messages for A to its shadow.

to and retrieved from persistent storage (or stored on the local host and retrieved by the resident).

3.2 COMMUNICATION FABRIC

The system provides a communication fabric allowing objects to interact with one another. The communication fabric is based upon *active messages*. An *active message* is one triggering the execution of a procedure at the receiving site. Bond favors non-blocking communication. Typically the sender continues execution execution after sending a request and it is interrupted upon the arrival of the reply. There are means to block, waiting for a message (see the `waitReply()` function).

A *shadow object* is a proxy for a remote object. A shadow is an abstraction designed to allow distant objects to communicate with one another as if they were co-located as shown in Figure 1.2. A message delivered to the local shadow is guaranteed to reach the remote object. Shadow objects are the conceptual equivalents of the stubs in CORBA and RMI and can be implemented using stubs.

A *virtual network* is a local collection of shadow objects. This abstraction is necessary to create dynamic sets of semantically related objects. For example, a directory maintains a virtual network of local directories, a scheduler agent maintains a virtual network of execution agents.

A *subprotocol* is a closed subset of KQML messages, a specialized language needed to perform a specific task. The subset is closed because a subprotocol does not reference outside messages, the reply is a member of the same subprotocol with the question. Examples of generic Bond

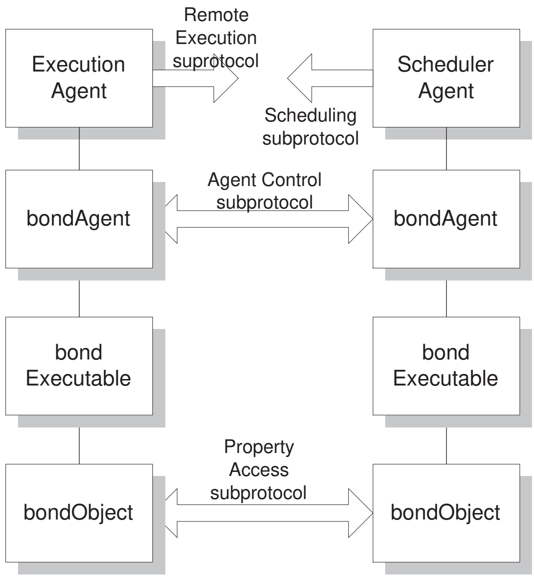

Figure 1.3 Subprotocol inheritance. A scheduling agent and execution agent understand the property access and the agent control subprotocols. In addition each one understands a specialized subprotocol related to its own functionality.

subprotocols are *property access* subprotocol, *agent control* subprotocol or *security* subprotocol, as shown in Figure 1.3. This abstraction is necessary to introduce a *structure in the semantic space of the messages*. An object can only understand sub-protocols implemented by its own class or inherited from its ancestors.

3.3 PROBES, AN ASPECT-ORIENTED APPROACH TO COMPLEX OBJECT DESIGN

Bond relies on *probes* to separate various aspects of complex object design. Probes are objects which understand a certain subprotocol and are attached to objects as dynamic properties. The implementation of `bondObject` assures that in the case a message is not understood by the object itself it searches the dynamic properties for a probe which can handle the given subprotocol and then delivers the message to it.

In the Figure 1.4 we have a scheduler agent, extended with two probes, a monitoring probe implementing the monitoring subprotocol, and a security probe implementing the security subprotocol. An incoming message which is part in the monitoring subprotocol is passed down in the inheritance hierarchy without being processed. At the bondObject level, after being checked that it is not part of the property access subprotocol, the object checks its dynamic properties for probes which implement the subprotocol of the message. In our case, the monitoring probe implements the required subprotocol, so the message is delivered to it, and from there the probe will take care of processing the message. If there is no probe implementing the subprotocol, the object replies `sorry`.

We have also the concept of a *preemptive probe* activated before searching the subprotocol hierarchy. If an object has a preemptive probe attached to it then any incoming message is delivered to this probe first. Preemptive probes can be used to implement firewall-style security systems or logging and accounting of messages.

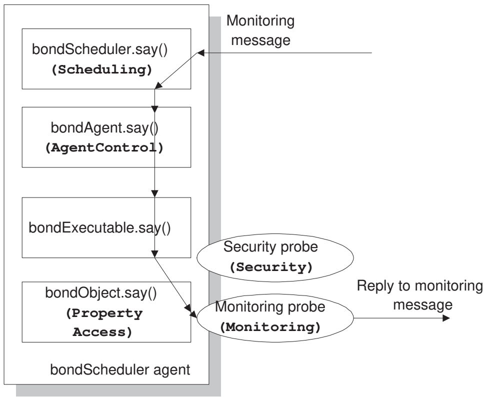

Figure 1.4 The effect of extending an object with probes. In this case a bondScheduler is extended with a monitoring probe. The extended object understands the monitoring sub-protocol and is capable of providing a meaningful reply to a monitoring message.

3.4 THE ARCHITECTURE

Figure 1.5 shows the architecture of the system and its major components:

- The transport layer - ensures message delivery to remote objects. Several communication engines are supported, one based on TCP, one on UDP, and one on Infospheres [5]. A multicast engine is under development.
- Core elements - provide the communication fabric, containers and core objects.

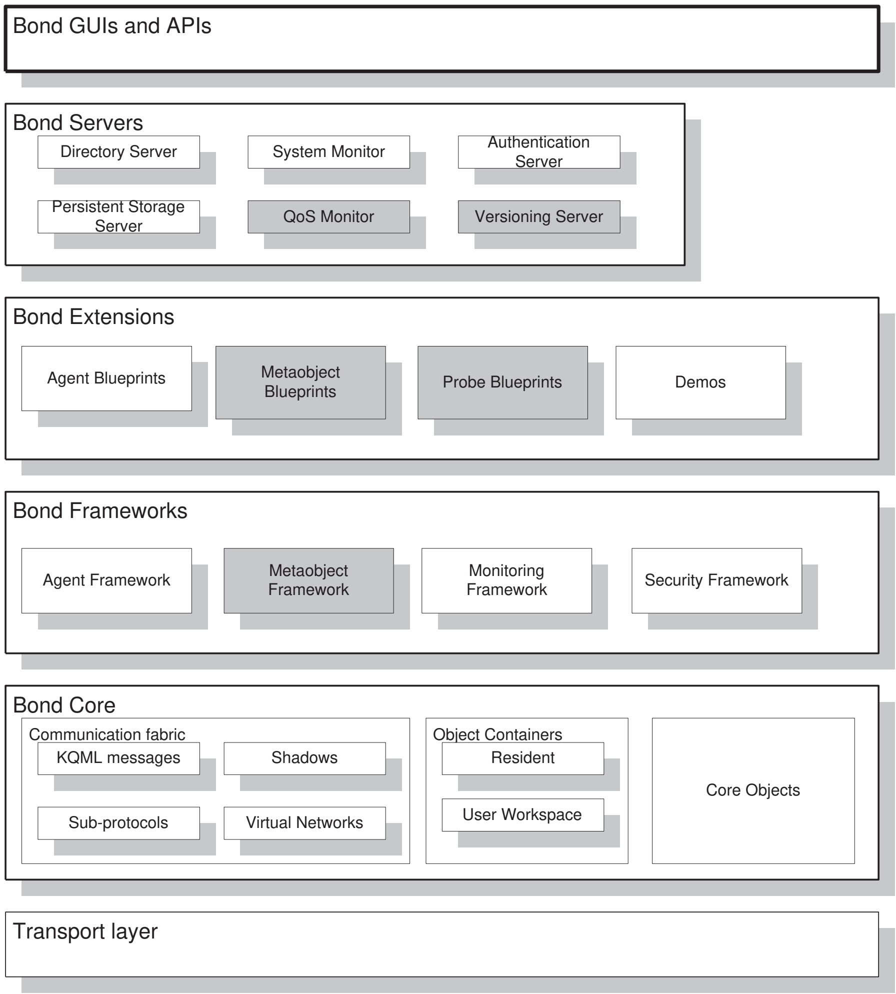

Figure 1.5 The architecture of the Bond middleware. Solid boxes are components not yet implemented.

- Frameworks - provide the infrastructure to build agents and metaobjects and augment them with monitoring, and security features.
- Extensions - provide blueprints for agents, metaobjects, probes and demos,
- Servers - provide the services needed in a Bond domain. Examples of servers: the directory server, the persistent storage server, the system monitor, the QoS monitor, the authentication server, the versioning server.
- Interfaces - GUIs and APIs provide user access to the system.

The system includes agent and metaobject factories and frameworks for monitoring and object security. The agent factory is capable to generate agents based upon blueprints describing the state-machine implemented by the agent, the strategies associated with every state, the agenda and the model of the world used by the agent.

Bond servers support the basic functionality necessary for collaborative activities. A Bond domain is a society of users with common interests. A system monitor starts up the servers and ensures fault-tolerance. Directory Servers allow objects to locate remote objects. The Persistent Storage Server is a repository for objects.

4. BOND AGENTS

The structure of the Bond agents presented in Figure 1.6 implements the theoretical model AM_1 as presented in [3].

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 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

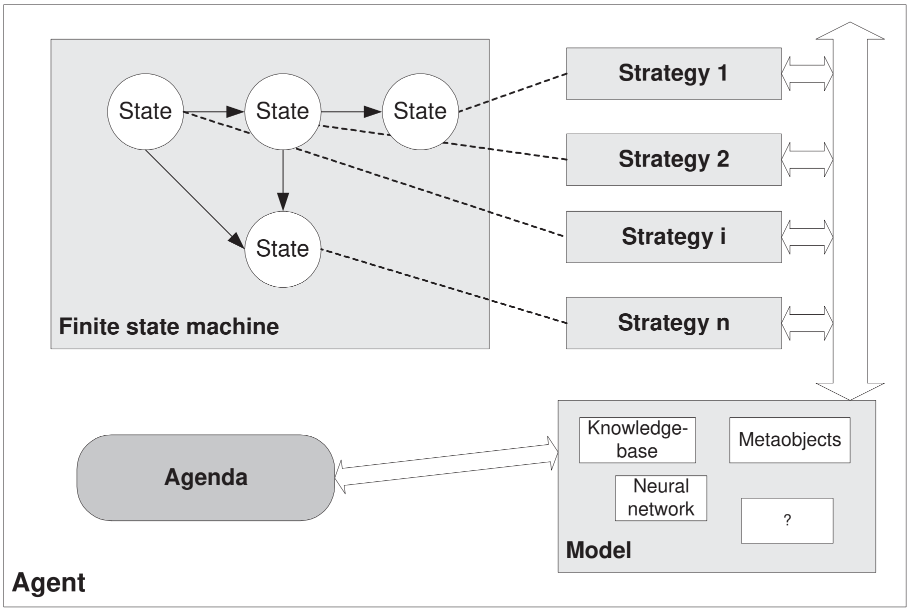

Figure 1.6 The anatomy of a Bond agent

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. 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 of 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 is a regular object and can be stored on persistent storage, or transferred over the network allowing for checkpointing and migration of Bond agents. The atomicity of the actions gives us natural checkpointing points.

4.1 SPECIFYING AN AGENT WITH THE BLUEPRINT LANGUAGE

The agent framework can be programmed at two levels. At the expert level, the developer can define 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 cannot 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 transferred across the network, stored in a repository 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.
- It is possible to modify an agent at runtime by interpreting a new blueprint script.

`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.

Compiled agents behave identically with their counterpart assembled locally at runtime. There is a small penalty to be paid for assembling an agent but the advantages of dynamic generation outweigh the disadvantage of the interpretation time.

The syntax of a blueprint program can be presented as follows:

```
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 ``;''
```

4.2 CREATING AN AGENT

A `bondAgent` can be created: (a) **Statically:** an agent derived from the *bondAgent* framework may create its components (finite state machine, states, strategies etc.) using its constructor. (b) **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.

Now we discuss the lifecycle of the agent. A Bond agent can be either *instantiated*, if it is a statically created, 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. The factory object allows us to assemble agents on remote residents:

```
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. Using the factory method agents can be instantiated remotely in a transparent manner. Another advantage of this approach 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)")
```

In this example we are instructing the agent 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. Regardless of the method used to create an agent, 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 1.7

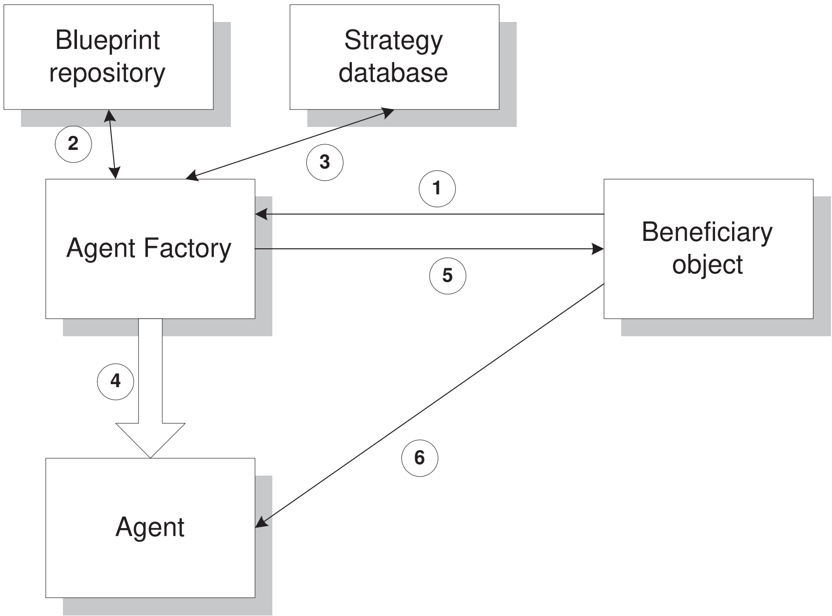

Figure 1.7 Creating an agent remotely using an agent factory. (1) The beneficiary object sends a create-agent message to the agent factory (2) The blueprint is fetched by the agent factory from a repository or extracted from the message (3) The strategies are loaded from the strategy database (4) The agent is created (5) The id of the agent is communicated to the beneficiary, and (6) The beneficiary object controls the new agent

4.3 INITIALIZATION OF AN AGENT

After creation, can send and receive 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. For example the agenda of an agent constructed to supervise the execution of a legacy application is incomplete without the name of the program to be run. During the initialization phase the beneficiary provides the agent with additional information about the world by writing it into the agent's model. For example, we can request the executor agent to start the `matlab` program:

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

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

4.4 STARTING AND RUNNING AN AGENT

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);
```

Alternatively, the agent can be started locally:

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

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

Once it enters a state an agent transfers the flow of control to the strategy associated with that state. 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 KQM-L messages sent 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 its 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 AGENTS

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.

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 inside, by the current strategy or from outside by other agents. We also present in more detail the structure of the strategies, and the possibilities offered by them to the agent programmer.

5.1 INTERNAL AND EXTERNAL CONTROL OF AN AGENT

An 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 event. These events are reserved to facilitate the assembly of agents from reusable strategies. Generating an internal event can be done in a strategy by an 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.

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.

5.2 AGENT SECURITY

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 [14].

5.3 IMPLEMENTATION OF STRATEGIES

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` and it implements the function `nextAction()`. Whenever a transition occurs in the finite state machine of the agent 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 a strategy is 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 cannot 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. There are two types of strategies:

- **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.

Strategy composition. The behavior of the agent in each state is determined by the strategy associated with the state. Usually the behavior of an agent in any given state has multiple facets. 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 repositories 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 [14], 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 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 should use disjoint 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 of 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};
  };
```

5.4 THE MODEL OF AN AGENT

We define the model of the agent as the collection of 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 of a 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.

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 repository. The example we have chosen, a legacy application wrapper is a relatively simple case, with many 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 cases where an agent should execute an external program.

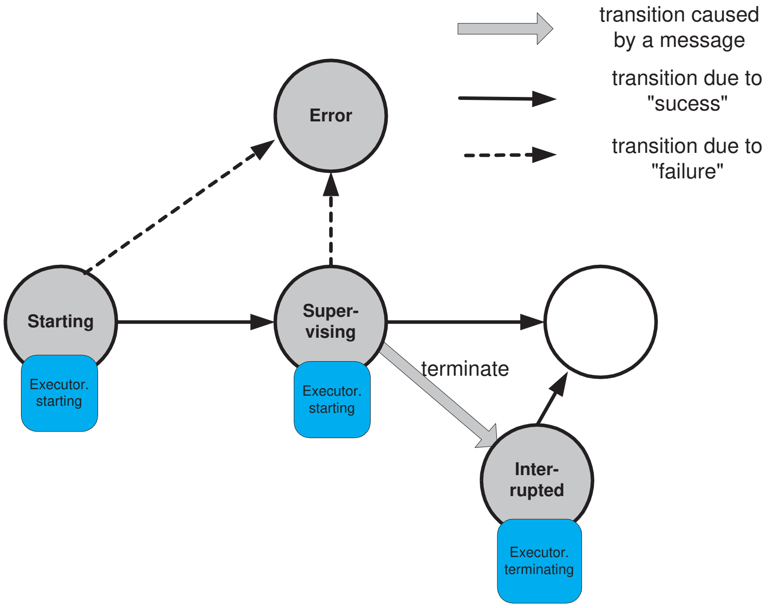

Figure 1.8 The finite state machine of the executor agent according to the initial specification

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).

```
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 1.9 The initial blueprint of the executor agent

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 1.8. 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 is to identify the strategies we need to perform the operations required in each state. First, we search the strategy repository to determine if suitable strategies exist. If this is a generic, well-known problem, like the execution of an application, there is a good chance that we will be successful in our search. In this 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 1.8. For the time being we don't have a strategy to handle errors, so we attach a null strategy there.

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 1.9.

6.2 EXTENDING AN AGENT

The agent presented above satisfies our basic requirements. We show now how to 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 also assume that the locations of remote files are known. The strategies in the `bondDataStagingStrategyGroup` are designed precisely for this purpose. We introduce a new state to handle datastaging. The updated finite state machine is shown in Figure 1.10.

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 1.11 shows the new blueprint.

7. SUMMARY OF RESULTS AND ONGOING RESEARCH

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.

We introduced an agent definition language called `Blueprint` and describe mechanisms to create dynamically agents using an agent factory

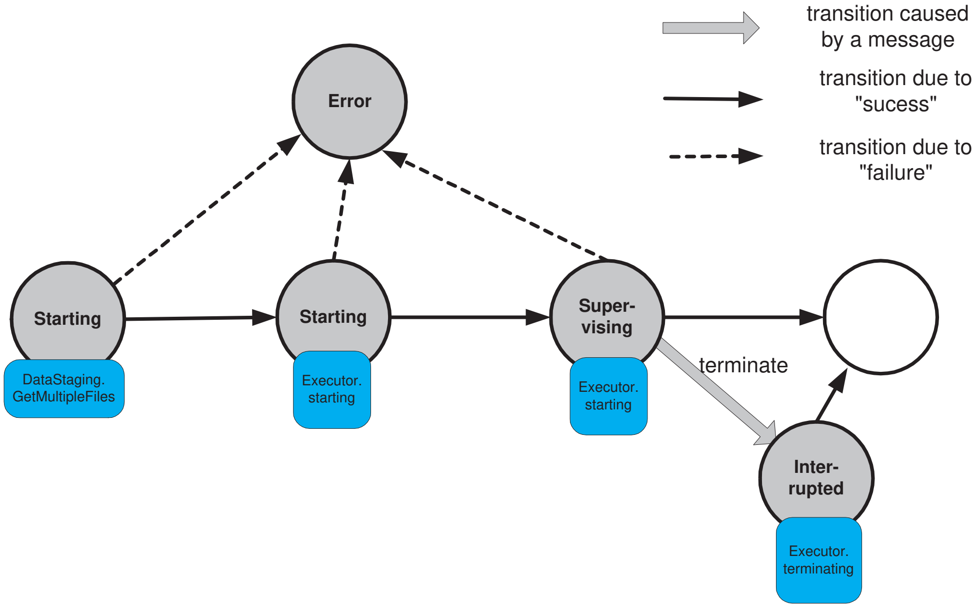

Figure 1.10 The finite state machine of the executor agent modified to include data staging

supplied with each Bond resident. An agent can be remotely controlled by its creator and possibly by other objects.

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 with an example of a wrapper agent capable to start-up a legacy application, and support its execution and communicate with

```
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 1.11 The blueprint of the executor agent modified to include data staging

other agents. A more sophisticated application a network of PDE solvers is described in [23].

The first beta version of the Bond system was released on March 25, 1999, and is available in binary and source form from the websitehttp://bond.cs.purdue.edu. This release contains all the functionality described in this chapter together with some of the new experimental features. Daily snapshots are also provided for those who want to keep the pace with the most recent improvements.

We are extending now BAF to support "spatial agents" consisting of a collection of finite-state machines. We are also investigating *federations of agents*, groups of agents that have to coordinate their activities and have known communication patterns. At the same time Bond residents are enhanced with an "awareness mechanism" allowing them to discover the existence of other residents, and exchange such *discovery* objects.

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] L. Bölöni: *Bond Objects – a white paper* Department of Computer Sciences, Purdue University CSD-TR #98-002, 1998.

[2] 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, 1999.

[3] L. Bölöni and D.C. Marinescu *A Multi-Resolution Model for Software Agents* Technical Report, Department of Computer Science, Purdue University, 1999.

[4] J. M. Bradshaw *An Introduction to Software Agents*, in J. M. Bradshaw Ed. *Software Agents*, MIT Press, pp. 3-46, 1997.

[5] K. Mani Chandy, A. Rifkin, Paolo A.G. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka and L. Weissman *A World-Wide Distributed System Using Java and the Internet* IEEE Int. Symp. on High Performance Distributed Computing, 1996.

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

[7] T. Finn, Y. Labrou, and J. Mayfield *KQML as an Agent Communication Language*, in J. M. Bradshaw Ed. *Software Agents*, MIT Press, pp. 291-316, 1997.

[8] L.N. Foner *What's an Agent, Anyway? A sociological Case Study* Agents Memo 93-01, Agents Group, MIT Media Lab, 1993.

[9] I. Foster and C. Kesselman Eds. *The Grid.Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1998.

[10] 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.

[11] M. R. Genesereth *An Agent-Based Framework for Interoperability*, in J. M. Bradshaw Ed. *Software Agents*, MIT Press, pp. 317-345, 1997.

[12] 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

[13] R. Hao, K. Jun, and D.C. Marinescu *Bond System Security and Access Control Model* Proc. of Parallel and Distributed Computing and Networks Conference, Acta Press, pp. 520-524, 1998.

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

[15] G. Kiczales, J. M. Ashley, L. Rodriguez, A. Vahdat and D. G. Bobrow *Metaobject Protocols - Why We Want Them and What Else They Can Do* in Object-Oriented Programming - The CLOS Perspective, MIT Press, 1993.

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

[17] D. Lange, M. Oshima *Programming and Deploying Java Mobile Agents with Aglets* Adison-Wesley, 1998.

[18] N. R. Jennings, K. Sycara, M. Woolridge *A Roadmap of Agent Research and Development*, in Autonomous Agents and Multi-Agent Systems, 1, pp. 275-306, 1998.

[19] Rao, A. S. and Georgeff, M. P. *Modeling Rational Agents within a BDI-architecture* Proc. Knowledge Representation and Reasoning (KR&R-91), pp. 473-484, 1991.

[20] Y. Shoham *Agent-Oriented Programming*, in J. M Bradshaw Ed. *Software Agents*, MIT Press, pp. 271-290, 1997.

[21] Singh, M. P. *Towards a Formal Theory of Communication for Multi-agent Systems* In Proc. Twelfth Int. Joint Conference on Artificial Intelligence (IJCAI-91), pp. 69-74, 1991.

[22] B. Schneiderman *Direct Manipulation Versus Agents: paths to Predictable, Controllable, and Comprehensive Interfaces*, in J. M. Bradshaw Ed. *Software Agents*, MIT Press, pp. 317-345, 1997.

[23] 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.