An object-oriented approach for semantic understanding of messages in a distributed object system

Ladislau Bölöni, Ruibing Hao, Kyungkoo Jun, and Dan C. Marinescu
(Email: boloni, hao, junkk, dcm@cs.purdue.edu)
Computer Sciences Department
Purdue University
West Lafayette, IN, 47907, USA

March 2, 2000

Abstract

In message-oriented distributed object system, cooperating objects exchange messages and various functions are carried out as a side effect of message exchanges. As objects range from an icon on the screen to a full featured server, the set of messages they can understand is also different. An object may not know the nature of all the objects it is communicating with. The possibility that objects acquire new functionality at runtime makes the problem even more difficult.

This paper presents message handling in the Bond distributed object system. Bond uses KQML as a communication language allowing every object to parse every message. The solution for the semantic understanding of messages are the *subprotocols* - highly specialized, closed set of commands. Being object instances themselves, subprotocols can be generated dynamically or stored into persistent storage. Objects implement the handling of subprotocols in their class definitions, can be extended in runtime with *probes* to understand new subprotocols or (in some specific cases) can dynamically generate new subprotocols or learn them.

Keywords: distributed object systems, KQML, message oriented middleware

1 Introduction

The success of the World Wide Web provided the impulse to develop a World Wide object Web for cooperating objects [?]. This new approach to distributed computing relies on new technologies, object oriented design, network programming languages like Java, and architecture frameworks like CORBA.

Distributed object systems rely on a networking paradigm which is usually either remote method invocation or message passing. Other paradigms, like virtual shared memory or tuplespaces (like Linda [?] or JavaSpaces) were also proposed. Message passing and remote method invocation are dual, the same functionality which can be achieved by with one of them, can be provided by the other. As a practical implementation however they impose different programming styles, and thus influence the design of the systems. It is generally accepted that message passing is advantageous for asynchronous, loosely coupled systems, while remote method invocation allows programmers to (partially) hide the difference between local and remote objects for synchronous, tightly coupled systems.

For both paradigms a critical issue in the design of distributed object systems is interface discovery. As new objects are created their usefulness to the community is determined by the ability of existing objects to interact with the newly created ones.

Any member of the community wishing to interact with the newly created object invokes an *interface discovery service* and learns how to interact with the object. In a remote method invocation system like CORBA the interface discovery amounts to information on the names and types of the methods and their parameters required for the method invocation. CORBA provides a solution to this problem by means of an interface repository service. Once an object is created, it uses this service to advertise its interfaces to the entire object community.

Turning now to message-oriented distributed object systems we recognize a similar problem. In layman's terms, a message oriented interface defines the language the object understands. The recipient of the message cannot possibly understand the syntax and semantics of an arbitrary message. Understanding the syntax is a relatively minor problem, and can be solved by choosing a suitable meta-language like KQML, FIPA or most recently, XML. However, the problem of semantic understanding still has to be addressed.

This article proposes the notion of *subprotocols*, a solution for the semantic understanding of messages, implemented in the Bond distributed object system [3]. The article is structured as follows. The problem of message understanding, the ontology based solution in KQML and the motivation and the idea of subprotocols is introduced in section 2. The three methods of implementing subprotocols (statically, dynamically through probes and runtime generation are introduced in section 3. Conclusions are drawn in section 5.

2 Abstractions for message understanding

2.1 Interface-based message understanding

Interfaces are the basic method for method understanding for remote method invocation systems. An interface represent a collection of methods implemented by (or messages understood by) a server object. While the actual implementation can differ, the interface represents essentially a *contract* between the server and the client object, where the client relies on the fact that the server implements a set of functions with the corresponding signatures.

Interfaces usually have a formal description in form of an interface description language (IDL). IDL's normally describe the syntax of interaction (the signatures of the methods involved). Nevertheless a certain amount of semantic is also involved in the interface contract, without being formally specified - developers rely on the fact that functions like `add(Object)` and `delete(Object)` perform the common-sense operations given by their name.

Interfaces are a natural solution for remote method invocation systems like CORBA, RMI or DCOM. While usually used in a static definition way, interfaces can be learned on the client side using interface repositories. Component object models like enterprise Java Beans allow dynamic aggregation of interfaces on the server side.

2.2 Ontology-based message understanding

The basic message understanding paradigm for multi-agent systems is relying on common *ontologies*. We present the approach taken in the KQML interagent communication language.

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, [?].

KQML messages, called performatives encode basic abstractions like asking, replying, achieving, subscribing or notifying. There are several classes of performatives: informative like tell and deny, database performatives e.g. insert, and delete, basic query performatives as evaluate, reply, ask-if, ask-about, ask-one, ask-all, sorry, effector performatives like achieve and unachive, notification performatives as subscribe and discard, networking performatives as register, unregister, forward, broadcast, pipe and facilitator performatives e.g. broker-one, broker-all, recommend-one, recommend-all, recruit-one, recruit-all.

An advantage of KQML is that messages can be easily interpreted by humans. Debugging a distributed system is still a very difficult task, and the ability to identify the source, the destination and relation to other messages is very helpful.

KQML is a meta-language in the sense that understanding its syntax is not equivalent to understanding the semantics of the message. Of course, the performative itself is giving a hint on the nature of the message, thus allow certain services like brokers or forwarders to be implemented which can function without having a full understanding of the KQML message.

In a typical deployment scenario, the parameters of a KQML performative (and specially the `:content` parameter) are statements written in a specialized command language, or even in a full featured interpreted programming language like Prolog. This language may be specified in the `:language` parameter of the message. The message can specify an *ontology*, in the `:ontology` parameter. The ontology is the background knowledge needed to understand the message, typically implemented in the form of a knowledge base.

Ontology-based message understanding is the natural choice for systems based on logical programming languages like Prolog. For these cases the content field of a KQML message can be directly interpreted by the interpreter in the context of the current ontology.

2.3 Subprotocol-based message understanding

The Bond distributed object system is built on a message oriented structure, using KQML as messaging language. Bond objects [1] are network objects in the sense that they can communicate with each other, can be instantiated and run remotely. Intended as an inter-agent communication language by its designers, KQML is used in Bond as an inter-object communication language. All Bond objects can receive and send messages.

KQML allows us to express the attitude communicated by a message (question, command, answer and so on), yet the full understanding of the semantics of a message would require a full featured programming language and force us to embed a new interpreter for Prolog or Scheme in every Java based Bond program - a very serious overhead. (Java is not interpreted in this sense of the word: Java statements should be compiled to bytecode before executing, so there is no simple way to execute a string written in Java).

Another possible alternative is to design our own command language to be used in every message. This also implies a major overhead - Bond objects range from a simple icon on the screen, to full featured agents or servers. Another problem is the extensibility of the language. The users should be able to create their own objects with new semantics based on the object-oriented library provided by Bond. The agent framework provides the best arguments for the need of some objects to acquire new properties at runtime. An agent is a program capable to independently perform a well defined set of actions in order to pursue its own agenda, and also it has to be controlled by other agents in a specific way. The object-oriented agent framework allows the runtime creation of agents by loading the control finite state machine, agenda and strategy from an object database and combining them in various ways. This also implies that the corresponding commands should be generated during runtime and learned by the controlled agent and the controlling authority.

Our solution to these problems is the introduction of *subprotocols*. 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. The only exceptions are the `(sorry)` and `(error)` KQML performatives, possible replies to messages of any subprotocol.

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 (see Table 1). An alternative formulation would be that subprotocols introduce a *structure in the semantic space of the messages.*

Subprotocols differ from interfaces in that they do not necessarily reflect a client-server approach, but can capture a more complicated sequence of messages e.g. publish-subscribe, multiobject interaction. On the other hand, subprotocols do not need an underlying knowledge base like ontology-based understanding.

To create a fully functional distributed system, a typical object should implement a number of subprotocols. We call a *message pattern* (analogous to design patterns [5]) the totality of messages a distributed object system should use in order to accomplish a certain task.

Subprotocols are *self-describing*: independently of there method of implementation, a *subprotocol object* can be always created, which lists the messages in the subprotocol, their parameters and description.

Subprotocol objects then can be passed over the network and they provide the means for object to learn new subprotocols.

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 set of 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.

Subprotocols are not a complete solution for the understanding of messages, they just push the problems one level higher. As an analogy, in human communication, if the answer to "Parles vous francais" is "Oui", than we are assuming that from that moment, everything we say in French will be understood by our partner. Similarly, if an object answered that he does speak "AgentControl", we must assume that it will perform correctly in response to commands like "start", "checkpoint" or "migrate". While objects can be extended with new subprotocols, or even subprotocols can be generated during runtime, there must exist either a human-written code which interprets the subprotocol (in the case of *probes*) or a previous convention which governs the rules of the runtime generation of subprotocols (like the multi-plane state machine structure at the core of Bond agents).

3 Understanding subprotocols

In the following we discuss the three ways for a Bond object to understand a subprotocol: static implementation, acquiring the ability to understand subprotocols by probes and generating and learning new subprotocols.

3.1 Subprotocols as a static property

Subprotocols are intrinsic properties of Bond classes and objects inherit subprotocols from their ancestors. The object hierarchy presented in Figure 1 indicates also the subprotocol implemented in the corresponding classes. For example every Bond object understands the property access subprotocol and every Bond agent understands the agent control subprotocol. The handling of the commands in these subprotocols is implemented by the methods of the corresponding objects. The messaging thread of a Bond executable delivers every incoming message to the `say()` function of the corresponding object. If the message is not understood, it is usually passed to the `say()` function of the immediate ancestor in the object hierarchy. This is basically a chain of responsibility design pattern, where the chain of responsibility basically is the `say()` functions of the ancestors of the object. Every Bond object inherits all the subprotocols implemented by the objects above it in the Bond object hierarchy, but this inheritance can be overwritten by the programmer. For example, the scheduler agent object implements the scheduling subprotocol and inherits the agent control subprotocol implemented by the `bondAgent`, and the property access subprotocol implemented by the `bondObject`.

Figure 2 shows two examples of messages delivered to a `bondScheduler` object, which extends a `bondAgent`, which in turn extends a `bondExecutable`, which in turn extends a `bondObject` (see Figure 1). The subprotocols specified at each level are specified in parenthesis. The `bondScheduler` object does not understand a monitoring message (it does not inherit a monitoring subprotocol), so after being passed all the way in the hierarchy, the `say()` function of the `bondObject` class answers with `sorry` indicating that it does not understand it.

3.2 Acquiring new subprotocols through probes

The subprotocols presented in the previous section are intrinsic properties of the objects. However in some instances subsets of objects from different classes need to understand the same subprotocol. For example agents and servers involved in a critical experiment need to authenticate all messages, any object can be

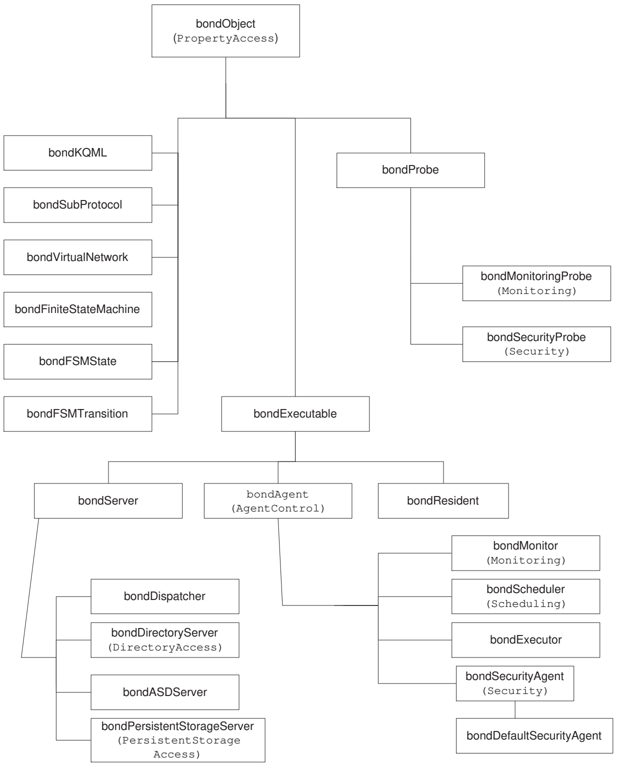

Figure 1: The upper part of the Bond object hierarchy. In parenthesis there are the subprotocols implemented in the class definitions of each object. Every object understands the property access subprotocol, implemented in bondObject

monitored. The implementation of the monitoring or security subprotocol in all objects would place a serious overhead.

The Bond system implements the concept of *probes* to dynamically add a new subprotocol to the object. Probes are objects which understand a certain subprotocol and are attached to objects as dynamic properties. The message delivery object of the Bond system automatically modifies the chain of responsability for the handling of messages when a probe is attached or removed. Regular probes are attached at the end of the chain of responsibility, so only messages which are not handled statically by the object itself are reaching them. There are two types of special probes: the preemptive probe and the autoprobe. *Preemptive probes* are inserted at the top of the chain of responsibility and act as filters. They can be used for the implementation of object-level firewalls, for logging and accounting, for authentication, for queuing or traffic shaping etc. An example of use of a preemptive probe is presented in Figure 4. The lightweight *autoprobe* uses the reflection features of Java and an instantiation table to instantiate and install on demand a probe able to understand a message the first time when the message of the given subprotocol arrives. For example the Resident object should be able to understand a large variety of messages, from monitoring to agent control. The use of the autoprobe allows the agent factory, a relatively complex object to be instantiated only when and if a message in the agent control subprotocol is received by the Resident object. This process is presented in figure 5.

In the Figure 3 we have the same scheduler agent, this time extended with two probes, a monitoring probe implementing the monitoring subprotocol, and a security probe implementing the security subprotocol. An incoming message in the monitoring subprotocol is passed down in the inheritance hierarchy without being processed. At the `bondObject` level, after establishing that the incoming message does not belong to the property acccess subprotocol, the object checks its dynamic properties looking for probes

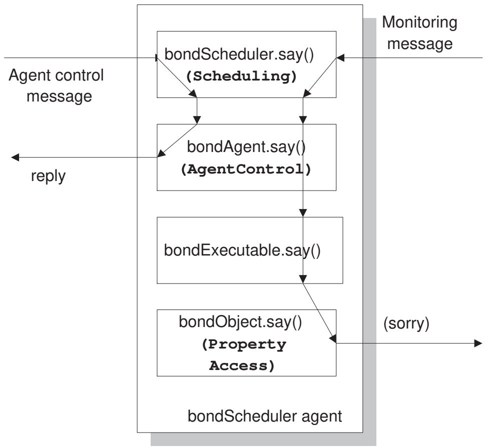

Figure 2: Message processing by a Bond scheduler agent. The incoming messages are handled by the `say()` function of each object, and if not understood passed to the `say()` function of the parent. In parenthesis we have the subprotocol implemented by the corresponding say() function. The processing sequence is then presented for two messages: an agent control message, understood by every object which inherits from `bondAgent` and a monitoring message, which is not understood by this instance of the `bondScheduler` object.

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`.

This construction is roughly similar in scope to the Decorator design pattern as presented in [5], allowing to dynamically extend the functionality of an object without subclassing. However the implementation is different - instead of a wrapper which captures the function call, we have a dynamically appended object which is consulted only in the case when the message does not make sense for the object itself. The difference in implementation is due to the message oriented nature of the objects: the higher flexibility and looser coupling between objects communicating by messages.

Another object-oriented structure which allows objects to acquire new functionality after "programming time" is the notion of a *mixin* [4]. Mixins are generally implemented as abstract classes, with reserved functions for future functionality. As such, the programmer needs at least a rough idea about the nature of the functionality with which the object may be extended. In our special case, the probes offer a larger flexibility, of course at the cost of the additional processing time to syntactically and semantically interpret the messages.

3.3 Generating and learning new subprotocols

Both the static implementation and dynamic acquisition of subprotocols is based on subprotocols defined by the human designer of the objects and referring to a semantics known at compile time.

In modern distributed programming however, functionalities or services may be created dynamically, on demand. This required that the semantics of the subprotocol is created at the moment of the creation of the associated functionality. Moreover, in the case of *mutable programs* the subprotocol itself can be modified together with the modification of the functionality. In both cases, the subprotocol thus created or modified can be understood by other objects by a learning procedure.

In the following we present the generating and learning of the subprotocols for Bond agents. The structure of the Bond agent framework is presented elsewhere, [2], we will summarize here only concepts which are relevant for subprotocol generation and learning.

Bond agents are active objects which follow their *agenda* by generating *actions* according to some

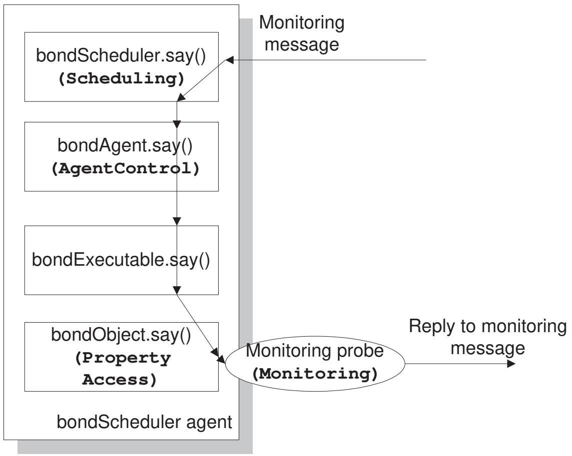

Figure 3: The effect of extending an object with probes. In this case a `bondScheduler` is extended with a monitoring probe. The extended object understands the monitoring sub-protocol and is capable of providing a meaningful reply to a monitoring message.

strategy active in the given moment. The main components of a Bond agent are a *multiplane state machine* which defines the internal states of the agent and the transitions between the states, together with *strategy objects* which define the behavior of the agent in various states. Transitions from a state to another are triggered by *internal events* generated by the agent itself or *external events* caused by messages sent to the agent. The totality of these messages define the subprotocol of the agent. The semantics of these messages is described by the transitions triggered by them.

The agent is a composed object, assembled from loosely connected active components into a data structure (the multiplane state machine). This operation is performed by an agent factory based on a text mode description in a language called `blueprint`. This data structure can also be modified during runtime, an operation called *agent surgery*, described by a surgical blueprint.

Figure 6 presents the sequence of steps performed between various entities which lead to the creation of a new agent. The process starts with the beneficiary object sending a message to the agent factory requesting the creation of the agent. The blueprint of the agent is either provided by the beneficiary or retrieved from a *blueprint repository*. Based on the blueprint the agent factory assembles the new agent, instantiating the strategies from a *strategy repository*. The subprotocol of the agent is defined by the structure of the agent. The subprotocol object is created using a reflection procedure on the agent structure and is transferred to the beneficiary, which thus *learns* the subprotocol of the agent. Other objects or agents can learn the subprotocol of the agent in a similar way.

As the structure of Bond agents can be surgically modified, the subprotocol can be modified too during the life-cycle of the agent. In these cases the learning process has to be repeated.

4 Example: agent control subprotocol

The Bond system uses more than twenty subprotocols (not counting the dynamically generated subprotocols of the agents). Table 1 lists some of the general purpose subprotocols used Bond.

We present in more detail the agent control subprotocol, which governs the operations related to the life-cycle of agents. The messages of the agent control subprotocol are listed in Table 2. The agent control subprotocol is implemented in Bond by the `bondAgentFactory` object. This object is usually used as a probe appended on the resident. For external objects this means that they can send their messages to the resident, which understands agent control through the probe. If a developer wants to implement its own version of the agent factory, he has to implement the subprotocol completely. Still the subprotocol

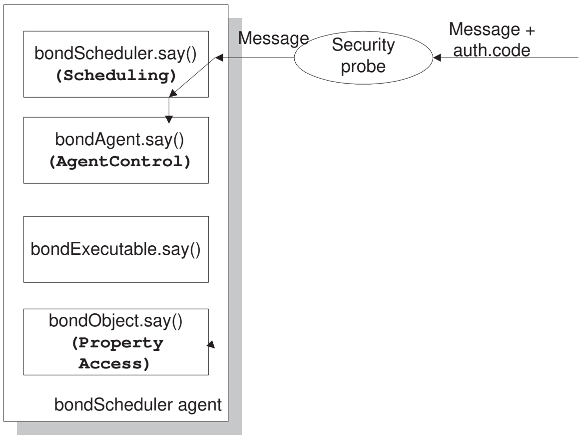

Figure 4: Extending an object with preemptive probes. In this case a security probe is used to authenticate the message. If the message contains authentication code, it is verified and stripped from the message, which then is delivered to the object. If the message can not be authenticated, it will not be delivered to the object. Thus, security probes allow the separation of the security code from the objects behavior.

functionality should not be confused with the complete implementation of every detail referred in some messages of the the subprotocol. If a specific agent factory does not implement surgery, it still has to correctly recognize what was required from him, and it should provide a correct (negative) answer.

5 Conclusions

In this paper we present an approach to message understanding in a distributed object system, based on subprotocols. Subprotocols offfer a larger flexibility than interfaces and impose comparably small changes to the object oriented programming style, and they can be used in situations where the ontology approach is not applicable.

The techniques presented in this article form the basic message understanding technique of the Bond system developed at the Computer Sciences Department of Purdue University. The Bond system is released under an open source licence (LGPL) and is downloadable from `http://bond.cs.purdue.edu`.

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

References

[1] L. Bölöni. Bond objects – a white paper. Technical Report 002, Department of Computer Sciences, Purdue University, 1998.

[2] L. Bölöni and D. C. Marinescu. A Multi – plane State Machine Agent Model. Technical Report CS TR 99-027, Computer Sciences Department, Purdue University, 9 1999.

[3] L. Bölöni and D. C. Marinescu. An Object-Oriented Framework for Building Collaborative Network Agents. In A. Kandel, K. Hoffmann, D. Mlynek, and N.H. Teodorescu, editors, *Intelligent Systems and Interfaces*. Kluwer Publising House, 1999.

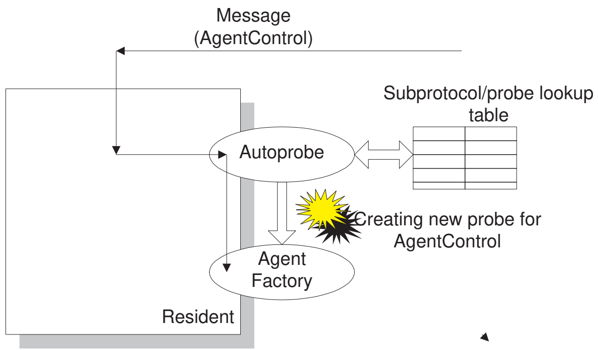

Figure 5: Using the autoprobe to instantiate probes on demand. The resident receives a message in the AgentControl subprotocol. As this message is not understood neither by the Resident itself, neither by its existing probes, is delivered to the autoprobe. The autoprobe instantiates a new probe to handle the message, based on a subprotocol/probe table and delivers the message to the new probe (in this case, the agent factory). Subsequent AgentControl messages will be delivered to the new probe.

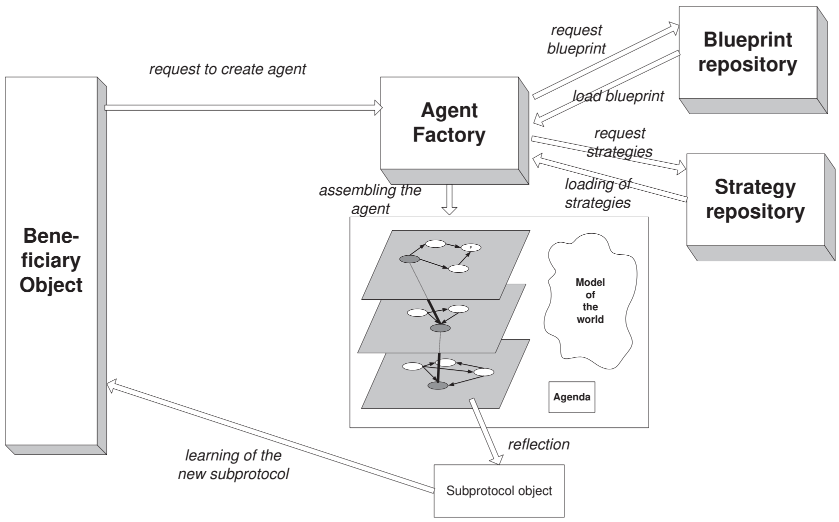

Figure 6: The sequence of operations needed to create an agent and the corresponding control subprotocol

[4] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, October 1990. ACM Press.

[5] E.Grama, R. Helm, R. Johnson, and J.Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Inc, 1995.

Subprotocol	Function
Property access	Supports read/write access to all properties of a Bond object.
Security	Used to establish trust relationship amongst Bond objects.
Monitoring	Allows a SystemMonitor agent to obtain information about the current state of the object.
Agent control	Allows a Bond object to start, stop and control a remote agent.
Scheduling	Supports scheduling of a contract
Persistent Storage	Allows Bond objects including BondWorkSpace to be save to or load in from the Persistent Storage Server
Data Staging	Allows Bond objects to move files and directories between hosts which has Bond resident running on it
Registration	Allows Bond Executables to register to SystemMonitor and Directory Server

Table 1: General purpose subprotocols used in the Bond system

Performative	:content	Parameters	Description
achieve	assemble-agent	:blueprint *address*	requires the agent factory to create an agent based on the blueprint at *address*
tell	agent-created	:bondID *bondID* :address *address*	confirms the creation of a new agent and transmits its id and address to the beneficiary
achieve	modify-agent	:blueprint *address* :agent *id*	requires the agent factory to perform surgery on the agent *id* using the surgical blueprint at *address*.
achieve	migrate-from-here	:agentid *agentid* :remote-address *remote address*	sent by the beneficiary, initiates the migration process between two agent factories
achieve	migrate-agent	:agentid *agentid* :modelid *modelid*	send by the agent factory at the original location to the agent-factory at the remote location, contains the generated blueprint of the agent to be migrated.
tell	migrated	:agentid *agentid*	the remote agent factory reports the successful creation of the agent. Triggers the disposal of the original agent at a local site.
achieve	checkpoint	:checkpointfile *filename*	checkpoints the current state of the agent in the file *filename*
achieve	checkback	:checkpointfile *filename*	restores a previously saved agent state from file *filename*
achieve	start-agent		starts the execution of the agent
achieve	stop-agent		performs a soft stop on the agent.
achieve	kill-agent		kills the agent
ask-one	get-state		asks the agent about it's current state
tell	state	:state *state*	a reply for the get-state message

Table 2: The agent control subprotocol