Subprotocols: an object oriented solution 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

September 14, 1998

Abstract

In a 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 article discusses message handling in the Bond distributed object system. Bond uses KQML as a messaging 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 [7]. 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 can be designed using two paradigms, remote method invocation or message passing. The two paradigms are dual, the same functionality can be achieved by with one of them, can be provided by the other, yet the use of one over the other may be proved to be advantageous in some cases. In this paper we present one of these cases.

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 discover 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. To communicate, two objects must know few words of common language, and use this initial knowledge to gradually discover each other's properties. This is the approach taken in Bond.

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.

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, [8, 9]. Intended as an inter-agent communication language by its designers, KQML is used in Bond as an inter-object communication language. In Bond all objects can receive and send messages.

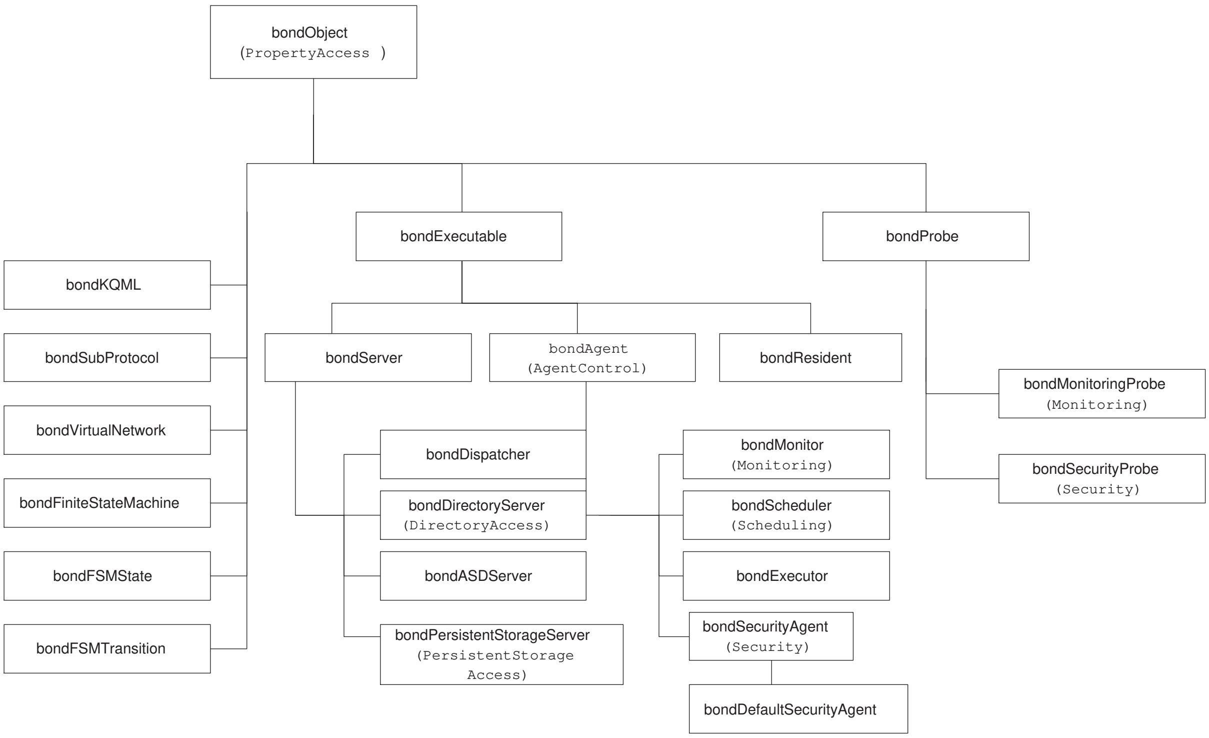

Figure 1: The upper part of the Bond object hierarchy. In parenthesis there are the subprotocols implemented in the class definitions of each object. Every object understands the property access subprotocol, implemented in bondObject

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. 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.

Although every Bond object understands the full syntax of KQML, their understanding is limited to messages related to their functionality. Instead of the language/ontology pair we introduce a different approach based on *subprotocols*. This extention is fully compatible with the KQML standards.

2 Motivation for subprotocols

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 would imply a serious 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 which 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 it's 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 acknowledment is always a member of the same subprotocol with the question. The only exception is 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.*

To create a fully functional distributed system, a typical object should implement a number of subprotocols. We call a *message pattern* the totality of messages a distributed object system should use in order to accomplish a certain task.

Two objects can communicate using messages which are members of the subprotocols implemented by both objects. Every Bond object implements at least the *property access subprotocol* which allows to remotely interogate and set the properties of an object. The subprotocols implemented by an object is also a property of the object. If two objects want to communicate without having any previous knowledge about the other, the first thing to do is to interogate the `SubprotocolsImplemented` property. After this, they can communicate using the intersection of the subprotocols implemented by both of them.

3 Implementation of subprotocols

In the following we discuss the three ways in which a Bond object can implement a subprotocol: static implementation, acquiring subprotocols by probes and generating and learning new subprotocols.

3.1 Static implementation of Bond subprotocols

Some of the subprotocols are intrinsic properties of certain Bond objects. 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 subprotocols 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 than passed to the `say()` function of the immediate ancestor in the object hierarchy. This way every Bond object inherits all the subprotocols implemented by the objects above it in the Bond object hierarchy. 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 `bondSceduler` 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 paranthesis. 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

The subprotocols presented in the previous section are intrinsic properties of the objects. However there are other properties which although can be part of any object their actual implementation in the object would place an unnecesarry overhead on the object. For example every Bond object can be monitored by an external monitor using the monitoring subprotocol. To allow any object to be monitored, the monitoring subprotocol should be implemented at the top of the hierarchy, in the bondObject class. Practically, however, only a few objects are monitored in any given system. Another example is the security subprotocol: there may be systems where the user does not need any additional security besides that provided by the native operating system. In other cases, additional authentication and encryption features are necessary.

The Bond system implements the concept of *probes* to allow one to dynamically add a new subprotocol to the object. Probes are objects which understand a certain subprotocol and are

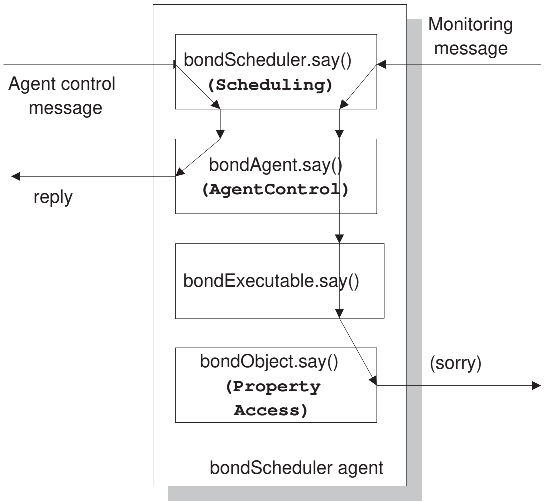

Figure 2: Message processing by a Bond scheduler agent. The incoming messages are handled by the say() function of each object, and if not understood passed to the say() function of the parent. In paranthesis we have the subprotocol implemented by the corresponding say() function. The processing sequence 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.

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. If there is no probe either, the object replies with `sorry`.

It is important to understand that a implementing a subprotocol means not only syntatic but also the semantic understanding of the subprotocol.

Examples of probes are the `bondMonitoringProbe` which, when attached to an object, allows a remote monitor object to monitor the properties of the object, or the `bondSecurityProbe` which, when attached to an object, allows the object to understand encrypted messages.

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 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`.

This construction is roughly similar in scope to the Decorator design pattern as presented in [11], 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* [12, 13]. Mixins are generally implemented as

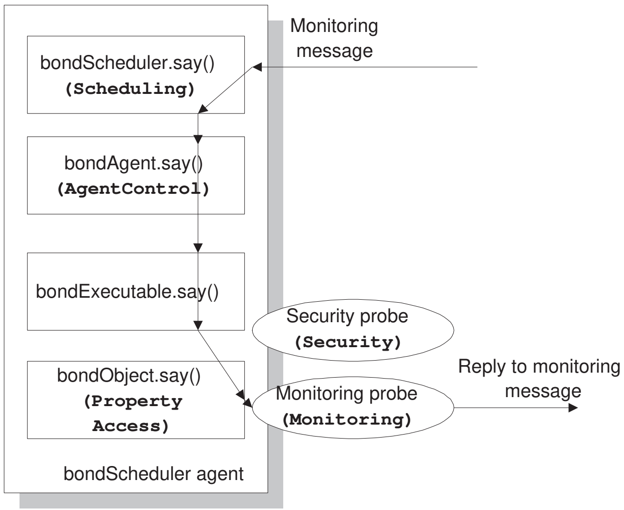

Figure 3: The effect of extending an object with probes. In this case a bondScheduler is extended with a monitoring probe. The extended object understands the monitoring subprotocol and is capable of providing a meaningful reply to a monitoring message.

abtract 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 aquisition of subprotocols is based on subprotocols defined by the human designer of the objects and referring to a semantics known at design time. However there may be some cases when the semantics and the subprotocol itself are defined during runtime. The current example for this is the control protocol of the agents, but further applications are envisaged.

The main components of a Bond agent are a *finite state machine* which defines the internal states of the agent and the transitions between the states, together with *strategy objects* which define the behaviour 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. We call the totality of these messages the *control subprotocol* of the agent. The semantics of these messages is described by the transitions triggered by them.

All the components of the agent are persistent objects, so a factory object can dynamically generate a new, custom agent. For example an air conditioning agent can be created by loading a heating strategy, a cooling strategy and creating the corresponding states and transitions. A different user may want also one or more "economy" settings with different strategies. The structure of the Bond agent framework is presented elsewhere, however the only aspect of interest here is that the agent control subprotocol for the dynamically generated agents is also generated dynamically.

Figure 4 presents the sequence of steps for the dynamic subprotocol generation. In step 1 the agent factory is provided with inputs for generating an agent. The most important is the the user input, which specifies the nature of the agent. If the requirements can be satisfied with states and strategies already available, they are loaded from the persistent storage by the agent factory.

In step 2 the agent factory creates the agent, generating the finite state machine based on user specification and annotating it with the corresponding strategies.

In step 3 the control subprotocol is generated from the finite state machine of the agent. The messages of the subprotocol correspond to the transitions of the finite state machine.

In step 4, a copy of the control subprotocol object is transferred to the controlling authority (which usually is a user interface or another agent), which in this way "learns" the subprotocol. More than one external object can learn the newly generated subprotocol.

In step 5, the newly created agent and the controlling authority are communicating using the control subprotocol.

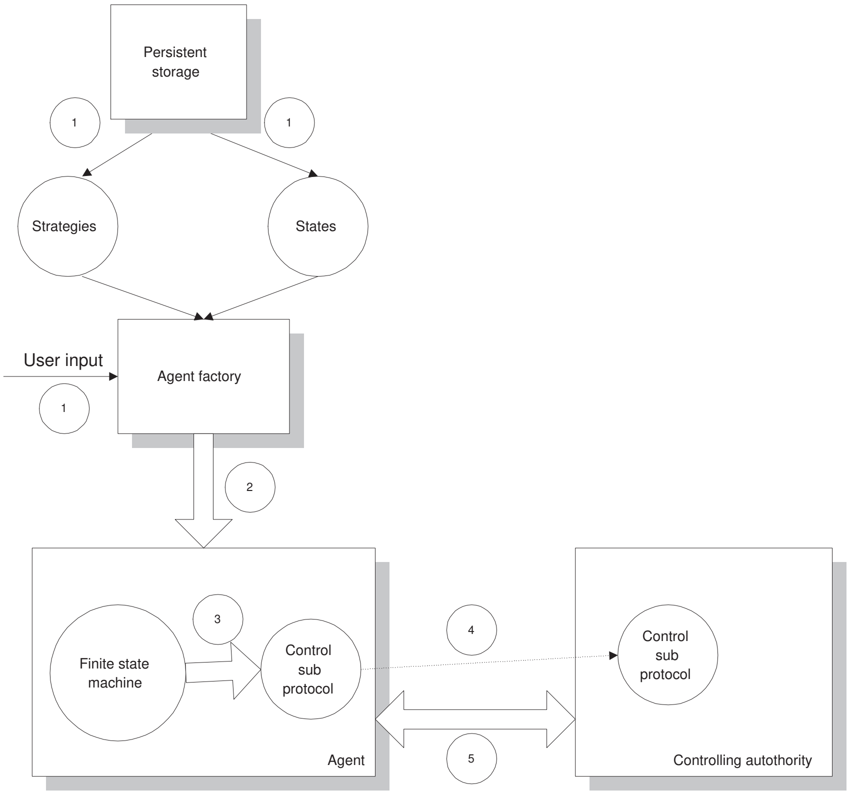

Figure 4: The sequence of operations needed to generate a custom agent and the corresponding control subprotocol.

For example in Figure 5 we have the finite state machine associated with the air conditioning agent described above. In each of the state there is a different stategy, which determines the actions of the agent.

The subprotocol generated from this finite state agent is presented in the following table:

:content
`stopped_to_heating`
`heating_to_stopped`
`heating_to_economy`
`economy_to_heating`
`stopped_to_cooling`
`cooling_to_stopped`

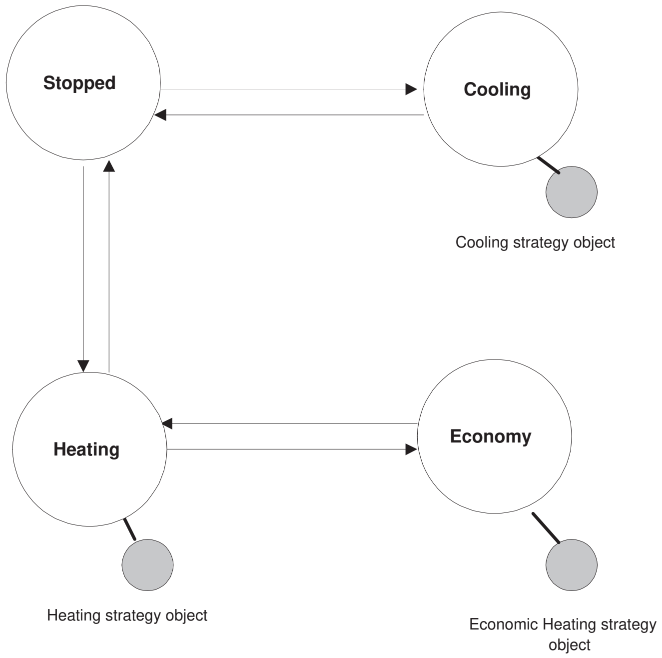

Figure 5: The finite state machine of a hypothetical air conditioning agent. Some of the states have different strategies associated with them.

In this case, the generated subprotocol is still human readable, because the transitions have a correspondence in one of the most commonly used household switches. Even in the case of a more complicated agent, where some of the transitions will be more obscure, at least we can associate messages with the transitions in the agent's finite state machine.

4 Examples of subprotocols

In this section we present some of the subprotocols implemented in the Bond system. We will also provide the description of the property access subprotocol, the basic subprotocol implemented by every Bond object. For a detailed description of the other subprotocols the reader should consult the document http://www.cs.purdue.edu/homes/sb/papers/subprotocols.ps.

Table 1 lists the defined subprotocols in Bond.

The property access subprotocol is used by a Bond object A to access(to read and write) the properties of a remote object B. Table 2 lists the definition of messages in this subprotocol.

For example, if an object A wants to obtain the value of the property i of the object B, it will send the following KQML message:

```
(ask-one :sender A :receiver B :subprotocol PropertyAccess
         :content get :property i
         :reply-with 43567)
```

Let's assume that the property i of the object B has the value 7. The object will reply with the following message:

```
(ask-one :sender B :receiver A :subprotocol PropertyAccess
         :content value
```

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 [6] to obtain information about the current state of the object.
Checkpoint/restart	Supports checkpoint/restart facilities
Agent control	Allows a Bond object to start, stop and control a remote agent.
Database access	Supports creation, deletion and updating of objects from databases.
Scheduling	Supports scheduling of a contract
User access	Allows the External agent to control the execution of a legacy program running under the control of a wrapper.
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: Subprotocols in Bond

```
:value 7
:in-reply-to 43567)
```

The parameters `:reply-with` and `:in-reply-to` are identifiers generated by the sender, and are used to match the reply with the question. This is needed by the asynchronous operation of message passing. This matching is done automatically in Bond by the use of *reply-waiting slots* [5].

Performative	:content	Parameters	Description
ask-one	get	:property *name*	asks the value of the property *name*
achieve	set	:property *name* :value *new_value*	asks to set the value of the property *name* to value *new_value*
tell	value	:value *value*	reply to "get"
tell	ok		reply to "set"
sorry		:error *error-name* :description *description*	an error occured

Table 2: The property access subprotocol

5 Conclusions and future work

In this article we have presented an approach to implement message understanding in a distributed object system. The system is already implemented, and is undergoing a testing in the Bond Lab at the Computer Sciences Department of Purdue University. We expect a public release of the code in early November 1998.

References

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

[2] 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 International Symposion on High Performance Distributed Computing, August 1996.

[3] K. Mani Chandy *Caltech Infospheres Project Overview: Information Infrastructures for Task Forces*

[4] M.Lewis and A. Grimshaw *The Core Legion Object Model* Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society Press, Los Alamitos, California, August 1996.

[5] L. Bölöni, K.K. Jun, T. Daniels and D.C. Marinescu: *Message patterns in the Bond Distributed Object System* Purdue University CSD-TR #98-004

[6] K. Jun, L. Bölöni, R. Hao and D.C. Marinescu *Bond System Monitor* Purdue University CSD-TR #98-026

[7] R. Orfali, D. Harkey, J. Edwards *Instant CORBA* Wiley Computer Publishing 1996

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

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

[10] D. Flanagan *Java in a Nutshell*, O'Reilly & Associates Inc. 1996

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

[12] G. Bracha and W. Cook *Mixin-based inheritance* ECOOP/OOPSLA 90, 303-311

[13] Y. Smaragdakis, D. Batory *Implementing Layered Designs with Mixin Layers*, 12th European Conference on Object-Oriented Programming (ECOOP 98).