COORDINATING DISTRIBUTED OBJECTS WITH DECLARATIVE INTERFACES

NARINDER P. SINGH
Computer Science Department, Stanford University
Stanford, California 94305, USA
singh@cs.stanford.edu

and

MARK A. GISI
Software Technology Lab
Hewlett-Packard Laboratories
Palo Alto, California 94305
gisi@hplabs.hp.com

Recieved March 24
Revised March 24, 1995

ABSTRACT

This paper presents an architecture that supports coordination among loosely coupled distributed objects. The architecture has two components: objects that provide a declarative specification of their interface, and system programs that reason with these specifications to provide sophisticated interoperation services. Traditional object-oriented interoperation technologies rely on procedural interface specifications, which do not address the semantics of the operations supported by the object. In addition, traditional approaches provide limited support for automatic interoperation in a dynamic environment. For instance, a resource that is available at compile time may not be available at runtime, or a better resource may become available at runtime. Interoperation based on machine-processible specification of object interfaces reduces the coupling (interdependence) between a client and server, and also shifts the burden of coordination from the programmer to the system.

Keywords: Objects, Software Interoperation, Specification Sharing, Declarative Interfaces

1. Introduction

A number of object-oriented technologies have been developed to support interoperation among applications distributed in an enterprise (e.g., CORBA [14, 12, 13]). They enable objects residing in one application to be accessed by objects residing in another. They also enable an entire application to be represented as a single object.

These technologies provide a number of system services that support a model of programming that is similar to programming in a single process (address) space.

Although objects may be distributed over a network, conceptually they are viewed as belonging to a single application. Programs are constructed by one or a small group of programmers who have complete knowledge about the kinds of interactions that take place among the objects. The relationship between objects can be described as being tightly coupled and static.

However, the single-process space abstraction is not appropriate for modeling interactions that occur among distributed objects supporting different functions of an enterprise. Distributed systems that support one or more enterprises continually change and evolve over time, creating a dynamic environment. Furthermore, although objects that support different functions of an enterprise are required to interoperate from time to time, they often maintain high degrees of autonomy. That is, they do not relinquish total control to a central authority. Instead of viewing a collection of enterprise objects as belonging to a single application, it is more appropriate to view each object as maintaining a high degree of autonomy, that interoperates with other enterprise objects to support different threads of activity. For example, an activity might be a request to print a file, process a customer order, perform group scheduling, or obtain the latest temperature reading of a patient. The relationship between enterprise objects can be described as being loosely coupled and dynamic.

It is impossible for one or a small group of people to have complete knowledge about the different kinds of interactions that can take place during runtime. There is too much information and it changes too rapidly. A resource that is available at compile time may not be available during runtime, or a better resource may become available. Implementing objects that interact in dynamic runtime environments is a difficult task. We need to develop more sophisticated system services that assist object implementors in dealing with incomplete knowledge.

System services should be able to acquire, store, update and reason about existing resources and object capabilities. In order for them to reason about such information, we need a precise and semantically rich way of describing an object's capabilities and properties. Unfortunately, conventional procedural interfaces provide little semantic information in a machine-processible form.

We address this limitation by providing objects with a second declarative interface so that system services can utilize simple inferencing techniques to reason about them. A declarative interface can be used to provide a semantic description of a server object's capabilities and properties. It also enables client objects to describe their requests declaratively. A client can state "what" it would like as opposed to specifying "how" to achieve it. That is, the client describes what it would like, and it is the system's task to determine how to satisfy the request. This reduces the complexity of a client's implementation in two ways. First, the steps required to satisfy a request can change as the environment changes; therefore this logic does not need to be imbedded in the client's implementation. Second, the client doesn't have to know about the precise details of the interfaces of the servers that are employed to satisfy the request, thereby reducing the coupling between client and server.

Consider the following situation where a client would like to print a file in Microsoft WORD format. There is no server that can handle the request directly, but there are two servers that, when used together, can satisfy the request. Suppose

```
Interface PrintServer
{
    void PrintPostscript(in file theFile)
}

Interface WordServices
{
    void TranslateWordToPostscript (in file wordFile, out file postscriptFile)
}
```

Figure 1: Interface Definition Language (IDL) Example.

there is a server that advertises it can handle requests to print a PostScript file and another object (a version of Microsoft WORD) that advertises it can handle translating a WORD file to PostScript. The system has the necessary logic to combine (glue) these services together to satisfy the request. If the client were required to know "how" to achieve this, then it would need to know which servers to invoke, obtain references to those services, and know the exact details of the interfaces of those servers, all of which could change over time.

The ideas presented in this paper have been developed within a project focused on constructing distributed intelligent autonomous agents [8]. We believe a more restricted form of this work could be developed to extend existing distributed object models (e.g., CORBA and OLE). We describe the benefits of providing objects with a semantically rich declarative interface. We also describe services that utilize simple inferencing techniques to reason about existing resources that can assist object implementors in dealing with the complexities associated with large, evolving distributed environments.

2. Related Work

The CORBA model [1, 14, 16] provides a number of system services that enables a client to find a reference to an object at runtime. Each server object publishes its interface to the Interface Repository (IR) using CORBA's Interface Definition Language (IDL). Figure 1 provides two simple interface definitions, one that can print a PostScript file and the other that translates a Microsoft WORD file to PostScript.

Before a client can invoke a method of a server object, it must first acquire a reference to the object. The client can obtain the reference either statically (at compile time) or dynamically (at runtime). At runtime the client has two options: it can utilize a name server, or it can utilize the Interface Repository and Dynamic Invocation Interface (DII) services. The name server is referred to as the "white pages" because of its similarity to the telephone white pages -provide it a name and you get back a reference.

Obtaining a reference at runtime is more flexible but less efficient than obtaining

```
Interface PrintServer
{
    void PrintPostscript(in file theFile)
    Properties:
    {
     Building : {A, ..., F}
     Floor : {1, ..., 5}
     size8by11 : {true, false}
     size11by17 : {true, false}
     color : {true, false}
     }
}

Interface WordServices
{
    void TranslateWordToPostscript (in file wordFile, out file postscriptFile)
}
```

Figure 2: IDL Extended to Include Attributes.

the reference at compile time. It is more flexibile because the client postpones binding to a service instance until runtime, thereby reducing the coupling between the client and a particular instance of a server. It is less efficient because the client must do a considerable amount of work during runtime to obtain the reference. This also adds complexity to the client's implementation.

Object interface specifications with the simple IDL shown in Figure 1 cannot take into account the properties of objects. For example, it is not possible to describe the location of an object. This may be important if the object provides printing services, and the client would like to print a file at a particular location.

The ANSA project has developed an architecture that supports interoperation among heterogenous telecommunication services and distributed computer applications [10]. ANSA recognized the limited expressiveness of IDL, and therefore, a server can augment its specification with a collection of properties (name/value pairs). Properties provide semantic information that describe differents instances of a service. For example, different instances of the print server in Figure 1 could have properties describing its location within an enterprise (e.g., building and floor), the different paper formats it can handle (e.g., 8x11, 11x17, A4) and whether it can print in color or not. Figure 2 illustrates this extension to IDL.

In addition to supporting more expressive interface specification, ANSA also provides a system service called a trader [3], that utilizes this information to facilitate interoperation. A server registers an interface specification (along with a collection of properties) with the trader, and clients make requests to the trader to help find a particular instance of a server. With each request, the client provides values for those properties it cares about. The trader uses this information to search its repository for an instance of a server that matches the client's request.

The trader then passes a reference to the server back to the client. The advantage is that the reference is located at runtime, when the lookup is delegated to the trader. This reduces the complexity of a client's implementation and adds flexibility because binding takes place at runtime. The service provided by a trader is sometimes referred to as the "yellow pages" because it can be used by a client to find an instance (vendor) belonging to a service class (merchant class).

3. Declarative Interfaces and an Intelligent System Service

When an IDL specification is augmented with name/value pairs, an object's specification begins to resemble a declarative representation. Unfortunately, there are limits to how far IDL can be extended because it is inherently a syntactical representation. In this section we describe declarative interfaces and present a sophisticated system service called the facilitator, which utilizes simple inferencing techniques to reason about an object's capabilities. The facilitator is a system service that bridges the gap between objects that can provide services and objects that request those services.

Individual programmers can create objects without knowledge of the data structures and algorithms of other objects, and without knowledge of the hardware configuration in which the objects will run (this is similar to CORBA). However, in traditional systems, the object interface descriptions are in IDL, and the meaning of the IDL interface cannot be defined in IDL (e.g., the fact that a given interface returns the list price of a product). The meaning of the service provided by an object must be known by external means, and there is no support for standardizing or defining the semantics of the interfaces. This requires manual coordination between programmers in a standard CORBA-like environment to agree on the meaning of interfaces. In a large setting, no human can have such global knowledge, and the traditional coordination approaches break down. A declarative interface addresses these issues directly by providing a fixed semantics and by providing a mechanism for examining and defining the vocabulary used.

The system services provided by the facilitator are well suited to environments that are dynamic. In a dynamic environment, resources come and go, and it impossible to make decisions at compile time. The services provided by the facilitator are based on explicit runtime dynamic data about available services, including other applications, resources, and so forth. Facilitation services use these dynamic specifications at runtime to determine the best way to service a request. The machine-processible specifications of objects permit greater flexibility and enables the system to provide many high-level coordination services.

In this section we describe the language for declarative specifications and the simple inference techniques used by the facilitator to provide intelligent system services.

3.1. Agent Communication Language

The language we use for describing a declarative interface, developed in the artificial intelligence (AI) community to support interoperation among distributed autonomous agents [7, 8], is called Agent Communication Language (ACL).

An object provides the facilitator with a declarative description of its capabilities and properties at runtime by sending it a collection of ACL messages. ACL has three components: (1) a vocabulary (a domain-specific semantic part), (2) a content language called KIF (Knowledge Interchange Format), and (3) a communication (wrapper) language called KQML (Knowledge Query and Manipulation). An ACL message is a KQML message which consists of a communication directive and a semantic content expressed in terms of the vocabulary in KIF. The communication directive instructs the facilitator on how to process the content of the message. We describe the three components in the following subsections.

KIF and KQML were developed as part of the ARPA knowledge sharing effort, and both are being used by a number of different research groups. KIF has been standardized by ANSI and is currently under consideration for standardization by the ISO. KQML is being evaluated by the Object Management Group (OMG).

3.1.1. Vocabulary

The vocabulary of ACL is listed in a large and open-ended dictionary of words appropriate to common application areas (e.g., electronic commerce, medical industry, and so forth) [9]. Each word in the dictionary has an English description for use by humans in understanding the meaning of the word, and each word has formal annotations for use by programs. The dictionary is open ended to allow for the addition of new words within existing areas and in new application areas.

Sharing information across a community demands agreement about the meaning of symbols, e.g., the meaning of the vocabulary for objects, functions, and relations in ACL. In a small setting (e.g., among a small group of programmers), it may be possible to mandate a single vocabulary for all objects to use in communication. However, in a large setting, this is impossible. More than likely, different communities in such an environment assign different meanings to the same symbols. Such an environment requires support for a collection of vocabularies. To disambiguate the meanings of symbols, all communication between objects must specify the vocabulary used. In order to support communication between objects using different vocabularies, the system also needs the definitions of articulation axioms to relate the symbols in one vocabulary to the symbols in another vocabular [9, 15].

3.1.2. KIF

KIF [6] is a prefix version of first-order predicate calculus with various extensions to enhance its expressiveness.

First and foremost, KIF provides for the expression of simple data. For example, the sentences shown below encode two tuples in a personnel database. The first argument in each is the social security number of an individual, the second argument is the department within which the individual works, and the third argument is the individual's salary.

```
(salary 015-46-3946 widgets 72000)
(salary 415-32-4707 fidgets 42000)
```

More complicated information can be expressed through the use of complex terms. For example, the following sentence states that one chip is larger than another:

```
(> (* (width chip1) (length chip1))
   (* (width chip2) (length chip2)))
```

KIF includes a variety of logical operators to assist in the encoding of logical information (such as negations, disjunctions, rules, quantified formulas, and so forth). The expression shown below is an example of a complex sentence in KIF. Note that all symbols beginning with `?` are universally quantified variables that can be instantiated to match any expression (subject to the rules of unification). It asserts that a number `?x` raised to another `?n` is positive if `?x` is real and`?n` even.

```
(<=  (> (expt ?x ?n) 0) (real-number ?x) (even-number ?n))
```

One of the distinctive features of KIF is its ability to encode knowledge about knowledge, using the `^` and `,` operators and related vocabulary. For example, the following sentence asserts that object accountant can handle requests for salary information. The use of commas signals that the variables should not be taken literally. Without the commas, this sentence would say that the accountant object can only handle the sentence `(salary ?x ?y ?z)` instead of its instances.

```
(handles accountant ^(salary ,?x ,?y ,?z))
```

KIF can also be used to describe procedures, i.e., to write programs or scripts for agents to follow. Given the prefix syntax of KIF, such programs resemble Lisp or Scheme. The following is an example of a three-step procedure written in KIF. The first step ensures that there is a fresh line on the standard output stream; the second step is to print Hello! to the standard output stream; the final step is to add a carriage return to get to a new line.

```
(sequence (fresh-line t) (print ''Hello!'') (fresh-line t))
```

The semantics of the KIF core (KIF without rules and definitions) is similar to that of first-order logic. There is an extension to handle nonstandard operators (like `^` and `,`), and there is a restriction to models that satisfy various axiom schemata (to give meaning to the basic vocabulary in the format). Despite these extensions and restrictions, the core language retains the fundamental characteristics of first-order logic, including compactness and the semidecidability of logical entailment.

KIF defines a set of objects, functions, and relations whose meaning is fixed, e.g., numbers and arithmetic functions. However, it is important to note that KIF is open ended, i.e., users are free to define the meanings of any other symbols that are not predefined. The purpose of defining the predefined set of symbols is to ensure that all objects can start communicating in a simple base language, which they are free to extend.

3.1.3. KQML

While it is possible to design an entire communication framework in which all messages take the form of KIF sentences, this would be inefficient. Because of the contextual independence of KIF's semantics, each message would have to include any implicit information about the sender, the receiver, the time of the message, message history, and so forth. The efficiency of communication can be enhanced by providing a linguistic layer in which context is taken into account. This is the function of KQML.

As used in ACL, each KQML message is a list of components enclosed in matching parentheses [4]. The first word in the list indicates the type of communication. The subsequent entries are KIF expressions appropriate to that communication, in effect the "arguments".

Intuitively, each message in KQML is one piece of a dialog between the sender and the receiver, and KQML provides support for a wide variety of such dialog types.

The expression shown below is the simplest possible KQML dialog. In this case, there is just one message– a simple notification. The sender is conveying the enclosed sentence to the receiver. In general, there is no expectation on the sender's part about what use the receiver will make of this information.

```
A to B: (tell (> 3 2))
```

The following dialog is a little more interesting. In this case, the first message is a request for the receiver to execute the operation of printing a string to its standard i/o stream. The second message tells the sender that the request has been satisfied.

```
A to B: (perform (print ''Hello!'' t))
B to A: (reply done)
```

In the dialog shown below, the sender is asking the receiver a question in an `ask-if` message. The receiver then sends the answer to the original sender in a reply message.

```
A to B: (ask-if (> (size chip1) (size chip2)))
B to A: (reply true)
```

In addition to the simple notifications, commands, and questions illustrated here, KQML also contains support for delayed and conditional operations, subscriptions, requests for bids, offers, promises, and so forth.

KQML defines a set of performatives whose meaning is fixed, e.g., `ask-if`, `tell`, perform. However, it is important to note that, similar to KIF, KQML is open ended, i.e., users are free to define the meanings of any other performatives that are not predefined. The purpose of defining the predefined set of performatives is to ensure that all objects can start communicating in a simple base language, and they are free to extend this.

3.2. Facilitators

A facilitator is a system service that can extend a distributed object model in the same manner a name server or a software trader can (see Figure 3). A facilitator

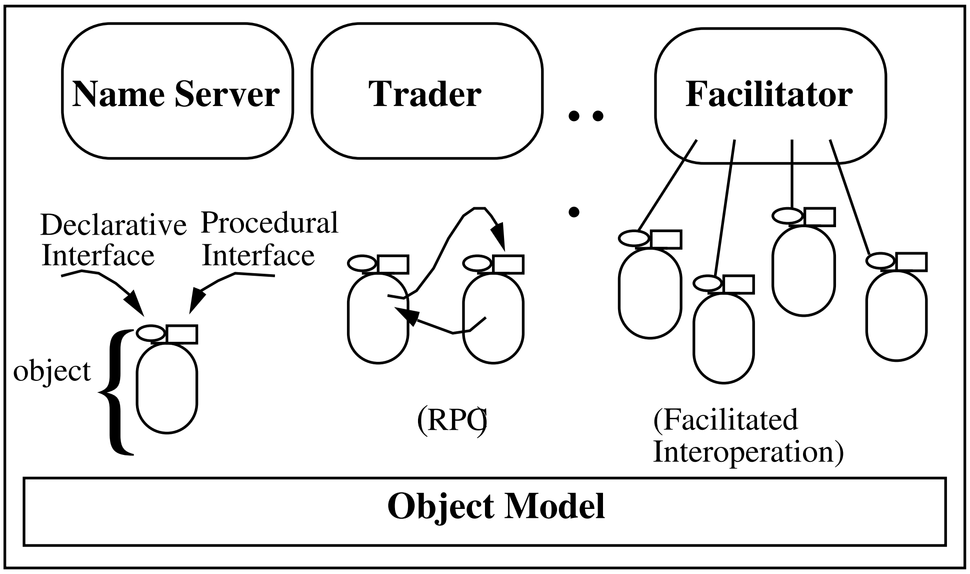

Figure 3: Multiple system services extending an object model.

is a relatively lightweight service that is visible to those objects that wish to utilize it, and transparent to those that do not. Objects are free to register services and/or make requests to a facilitator or access objects directly using conventional procedural invocations.

In this section, we illustrate (with examples) several services provided by a facilitator, the interface specifications of objects, and the processing of object requests by a facilitator. To those clients that utilize the facilitator, it appears that there exists a single system service, the facilitator, that handles all requests directly. The facilitator can be conceptually viewed as a virtual server providing all the services that are provided by all the server objects that are registered with it. A more detailed description of this architecture appears in [8].

The examples illustrate the automated coordination services provided by a facilitator, which are referred to as content-based routing. Key elements are the declarative specifications of object interfaces (a meta-description) and the declarative specifications of other background information. The power of the coordination services is made possible by the combination of these two types of information. In all of the examples there a single facilitator, though in general there can be an arbitrary number of facilitators (typically there is one facilitator per machine).

3.2.1. Example 1: Manual Coordination

This example illustrates a simple scenario where a client object manages all the coordination activities to satisfy its own request. In this example, there is a single facilitator running on a Unix workstation; there is an object named Textures running on a Macintosh that can translate a file from Latex to DVI format; and there is an object, $\mathtt{A}_1$, that accepts and processes requests on a user's behalf running on the same Unix workstation. Objects send their capabilities and requests to the facilitator. In this example, object $\mathtt{A}_1$ wants to translate a given file from Latex to DVI format.

Initially there are no services registered with the facilitator. The Textures object registers with the facilitator, specifying its interface and any additional background facts to the facilitator. All communications between the objects and the facilitator are restricted to be in ACL. The Textures object must inform the facilitator that it can translate arbitrary Latex files into DVI format.

Before describing the interface we must choose a vocabulary for describing files and translations. Files are described by using the function `file`. A file is described by terms of the form:

```
(file <name> <type>)
```

For example, `(file ''f1.tex'' latex)` denotes a file in Latex format whose name is `''f1.tex''`.

Translations between files in different formats are defined using the relation `translation`. The relation between a file in one format and a file in a different format is given by facts of the form:

```
(translation <file1> <file2>)
```

For example,

```
(translation (file ''f1.tex'' latex) (file ''f1.dvi'' dvi))}
```

states that the translation of file `''f1.tex''` (which is in Latex format) is the file `''f1.dvi''` (which is in DVI format).

The following KQML message is transmitted by the Textures object to the facilitator, and it defines the declarative interface for Textures:

```
(tell (handles textures ^(ask-one ,?y (translation (file ,?x latex)
                                                    (file ,?y dvi)))))
```

This message informs the facilitator that the Textures object can be asked to find one translation of a file in Latex format to a file in DVI format. The embedded argument to the tell KQML performative is the fact:

```
(handles textures ^(ask-one ,?y (translation (file ,?x latex)
                                              (file ,?y dvi))))
```

This is the declarative meta-description of the capability of Textures. The performative handled is the standard `ask-one` KQML performative, which takes two arguments: a pattern argument and a sentence. In order to handle this performative, Textures must return the first argument instantiated by a binding for the variables in a proof of the second sentence argument. The pattern in this example is `?y`, and the sentence is `(translation (file ,?x latex) (file ,?y dvi))`. So the previous fact states that the Textures object, when asked for one translation of a Latex file to a DVI file, will return the name of the DVI format file (`?y`).

The user interface object $\mathtt{A}_1$ next registers with the facilitator. It has no capabilities, so it does not transmit any interface specifications to the facilitator. However, $\mathtt{A}_1$ would like to find the translation of a file from Latex format to DVI format. In this example, $\mathtt{A}_1$ will do all the work to perform the translation. It does this by first asking the facilitator if there is any object that can perform the translation (using the yellow-pages services of the facilitator), and it then sends the request to an object that can handle it.

Object $\mathtt{A}_1$ first asks the facilitator if the following fact is true:

```
(handles ?a ^(ask-one ?ans (translation (file ''f1.tex'' latex)
                                        (file ?ans dvi))))
```

The facilitator uses its repository of information, which includes object interface specifications, to find a proof of this fact using backward inference. The proof is direct, since this fact matches the previous specification for object Textures (repeated below) with the variable `?a` bound to Textures:

```
(handles Textures ^(ask-one ,?y (translation (file ,?x latex) (file ,?y dvi))))
```

The facilitator informs object $\mathtt{A}_1$ that Textures can handle the request. $\mathtt{A}_1$ next sends the following request to Textures (using the communications support provided by the architecture):

```
(ask-one ?ans (translation (file ''f1.tex'' latex) (file ?ans dvi))))
```

Textures translates the file `''f1.tex''` in Latex format to the file `''f1.dvi''` in DVI format, and returns this as the answer to $\mathtt{A}_1$.

3.2.2. Example 2: Simple Content-Based Routing

This example is the same as the previous, except that the object representing the user ($\mathtt{A}_1$) does not manage the coordination, but leaves this up to the facilitator. The example is the same as the previous example up to the point where $\mathtt{A}_1$ makes the request. In the previous example, $\mathtt{A}_1$ used the yellow-pages service of the facilitator to find an object that could handle a request. In this example, the facilitator gives $\mathtt{A}_1$ the illusion that it is directly achieving the request, and $\mathtt{A}_1$ is unaware that Textures will actually be used in the translation.

$\mathtt{A}_1$ makes the following request to the facilitator:

```
(ask-one ?ans (translation (file ''f1.tex'' latex) (file ?ans dvi)))
```

The facilitator tries to find an object that can handle the quoted request:

```
(handles ?a ^(ask-one ?ans (translation (file ''f1.tex'' latex)
                                        (file ?ans dvi)))
```

The facilitator uses backward inference to try to prove this fact. The proof is direct, since this fact matches the previous specification for Textures (repeated below) with the variable `?a` bound to textures:

```
(handles textures ^(ask-one ,?y (translation (file ,?x latex)
                                              (file ,?y dvi))))
```

The facilitator passes the request from A_1 to Textures. Textures translates the file ``f1.tex'' in Latex format to the file ``f1.dvi'' in DVI format. The Textures object returns the answer ``f1.dvi'', and this is the answer returned to A_1.

Note that in this example, A_1 makes the following request to the facilitator:

```
(ask-one ?ans (translation (file ``f1.tex'' latex) (file ?ans dvi)))
```

while in the previous example, A_1 made the following request to the facilitator:

```
(handles ?a ^(ask-one ?ans (translation (file ``f1.tex'' latex)
                                        (file ?ans dvi)))
```

and the second request to textures:

```
(ask-one ?ans (translation (file ``f1.tex'' latex) (file ?ans dvi))))
```

This example illustrates a small improvement with content-based routing; however, in general an arbitrary amount of intermediate effort can be saved by content-based routing, as illustrated by the next example.

3.2.3. Example 3: Simple Problem Decomposition

This example illustrates a case requiring problem decomposition. The scenario is the same as the previous example, except that there is a third object dvi2ps running on a different Unix workstation that can translate DVI format files to PostScript files. In this example, the user object A_1 requests the facilitator for the translation of a Latex file into a PostScript file. This requires first translating the file from Latex to DVI, and then translating the file from DVI to PostScript. A_1 is unaware of the sequence of operations taking place. It appears to A_1 that the facilitator is performing all the translations directly.

The example is the same as before up to the point when Textures registers with the facilitator and sends its capabilities. Following this, dvi2ps registers with the facilitator and sends the following message:

```
(tell (handles dvi2ps ^(ask-one ,?y (translation (file ,?x dvi)
                                                  (file ,?y ps)))))
```

This is similar to the message sent earlier by Textures to the facilitator. For this example, object dvi2ps also transmits the following message to the facilitator:

```
(tell (<= (translation (file ?x latex) (file ?y ps))
          (translation (file ?x latex) (file ?z dvi))
          (translation (file ?z dvi)   (file ?y ps))))
```

The general form for a conditional rule is `(<= A B C)`, which means that `A` is true if `B` and `C` are true. The fact argument to the tell states that the translation of file `?x` in Latex format is file `?y` in PostScript format, if the translation from file `?x` in Latex format is file `?z` in DVI format, and the translation of file `?z` in DVI format is file `?y` in PostScript format. This rule in effect defines a simple transitivity relation.

The user interface object $\mathtt{A}_1$ next registers with the facilitator. It has no capabilities, so it does not transmit any interface specifications to the facilitator. However, $\mathtt{A}_1$ would like to find the translation of a file from Latex format to PostScript format. It sends the following message to the facilitator:

```
(ask-one ?ans (translation (file ``f1.tex'' latex) (file ?ans ps)))
```

The facilitator tries to find an object that can handle the quoted request:

```
(handles ?a ^(ask-one ?ans (translation (file ``f1.tex'' latex)
                                        (file ?ans ps))))
```

The facilitator uses backward inference to try to prove this fact. Unfortunately, no object can handle this request directly. However, the facilitator can use the transitive rule to decompose this request into the conjunction:

```
(and (translation (file ``f1.tex'' latex) (file ?z dvi))
     (translation (file ?z dvi) (file ?ans ps)))
```

The first conjunct is handled by the Textures object, and it binds the variable `?z` to the file `` ``f1.dvi'' ``. This binding for `?z` is plugged into the second conjunct, which is handled by the dvi2ps object, and it returns the variable `?ans` bound to `` ``f1.ps'' ``.

The problem decomposition rule for this example is applicable only in the special case of translating Latex files to PostScript by first translating them to DVI format. The following, more powerful rule allows an arbitrary intermediate format (not just DVI):

```
(<= (translation (file ?x latex) (file ?y ps))
    (translation (file ?x latex) (file ?z ?f)
    (translation (file ?z ?f)   (file ?y ps))))
```

Note that this is a recursive rule, which can applied an arbitrary number of times. Content-based routing, in the presence of such rules, can tremendously reduce the complexity of problem solving compared with manual coordination.

3.2.4. Example 4: Defining Performatives

In this example, we illustrate the definition of new performatives. The previous examples illustrated the use of the `tell` and `ask-one` primitive performatives of ACL. This example illustrates the definition of a conditional performative, though it possible to define simpler performatives without any conditions.

Performatives in ACL describe actions, as opposed to informational facts. It is important not to confuse informational facts with facts about actions. The purpose of informational facts is to define what is true, while the purpose of facts about actions is to define a procedure. For example, the fact `(and P P)` can be proved by proving `P` alone, while the action `(sequence P P)` cannot be satisfied by performing action `P` alone.

In this example, there is a single object called Scribe that can print files in PostScript format, as long as the size of the file is less than one megabyte. Scribe connects to the facilitator and sends it the following message:

```
(tell (<= (handles scribe ^(print (file ,?x ps)))
          (< (file-size ?x) 1000000)))
```

The fact inside the tell in the above message states that Scribe can print any PostScript file as long as the size of the file is less than one million (bytes). The function file-size takes a file name as an argument and maps it to the size of the file in bytes.

The user object A_1 makes a request to the system to print a PostScript file by sending the facilitator the following message:

```
(print (file ''f1.ps'' ps))
```

The facilitator first searches for an object that can handle the quoted form of the above expression. It uses backward inference to find a proof for the fact:

```
(handles ?a ^(print (file ''f1.ps'' ps)))
```

The backward inference uses the interface description to conclude that the Scribe object can print the file ''f1.ps'' if the size of ''f1.ps'' is less than one megabyte. Assume that this is so, i.e., the following fact is true:[1]

```
(< (size ''f1.ps'') 1000000)
```

The facilitator, therefore, concludes that Scribe can print the file ''f1.ps''. Next the facilitator sends a message to Scribe to print the file. Scribe prints the file, and when it is done, the successful indication of this is reported to the user object A_1.

3.2.5. Example 5: Defining Procedures

This example illustrates defining procedures in ACL. The previous examples have illustrated the definition of informational facts; this example focuses on facts that specify a sequence of actions. The primitive actions are the KQML performatives, that include **ask**, **tell**, and other user-defined primitive performatives. From these primitives, it is possible to define more complex performatives as compositions of these (just as it is possible to define complex procedures from primitive statements in a programming language).

Every performative returns a value, e.g., the performative **(ask-one <exp> <fact>)** returns **<exp>** instantiated by a binding for the variables which make **<fact>** true. Performative definitions can be nested, where the evaluation is inside-out and a component performative is replaced by its value. This is similar to the inside-out evaluation of (+ (* 2 3) 4) in programming languages.

This example is identical to the previous, except that the user object A_1 wishes to print a Latex file. In this case A_1 sends the following message to the facilitator:

[1] The size of a file is computed using procedural attachments, similar to the attachments for built arithmetic functions like addition and division.

```
(print (file ''f1.tex'' latex))
```

Unfortunately the facilitator is unable to find an object that can handle this request, since the file is in Latex format, and the printer Scribe can only print files in PostScript format. One way of overcoming this is to define a procedure for printing that defines a sequence of actions: to print an arbitrary file, first translate it to PostScript format, and then print it. This is specified by the following fact describing a procedure:

```
(define ^(print (file ,?n ,?f) )
        ^(print (file (ask-one ?n2 (translation (file ,?n ,?f)
                                                (file ?n2 ps))) ps)))
```

The general format for defining procedures is: (define ^<performative-defined> ^<performative>). A sequence of actions is defined by (sequence <a$_1$> ... <a$_n$>), and conditional actions are defined by (cond <condition$_1$> <action$_1$> ... <condition$_n$> <action$_n$>).

In this example the definition of the new performative is nested. In this case, the name of the translated file is given by the inner performative:

```
(ask-one ?n2 (translation (file ,?n ,?f) (file ?n2 ps)))
```

The request from A_1 can now be handled. The facilitator cannot find an object that can handle the request directly; however, the facilitator uses the procedural definition above to find that the file can be printed if: 1) the file can be translated from Latex to PostScript (the inner performative in the definition), and 2) some object can print the resulting PostScript file. The process of handling these two steps was illustrated in previous examples. The transitive translation rule is used to translate the file from Latex to DVI, and then from DVI to PostScript. The PostScript file ''f1.ps'' can be printed directly by the printer Scribe since it is less than one megabyte long. The two steps are performed directly by the facilitator, and the indication of success is passed back to A_1.

3.2.6. Example 6: Translation

Hiding details about implementations reduces the coupling between a client and server. Unfortunately, with procedural interfaces, interaction between a client and server requires the client to have specific knowledge about the details of a server's interface. When a client makes a declarative request to the facilitator, the client is not required to know specific details about the interface of the server(s) that satisfies it. This further reduces the coupling between the client and server, which is particularly important when supporting interoperation among loosely coupled objects interacting in a dynamic environment. This example illustrates the use of translation to match a request with a server that can handle it. A request is made with one vocabulary, while a service is provided with a different vocabulary. A translation rule is used automatically by the facilitator to map the request to the service. Such translation is not possible with a simple syntactic matching scheme.

Suppose that previously there was a different vocabulary for file translations, which is given below:

```
(file-translation <file1> <file2> <format1> <format2>)
```

For example, previously we would have used:

```
(file-translation ''f1.tex'' ''f1.ps'' latex ps)
```

while the current use is:

```
(translation (file ''f1.tex'' latex) (file ''f1.ps'' ps))
```

The new vocabulary makes explicit the concept of a file by using the function `file`. When the vocabulary was updated, the interface for Textures was also updated (as illustrated in the previous examples). Unfortunately, there is an object $\mathtt{A}_0$ (among others) that continues to use the old vocabulary. Textures would like to continue supporting the old vocabulary. It does this by communicating the following message to the facilitator:

```
(tell (<= (file-translation ?file1 ?file2 ?format1 ?format2)
          (translation (file ?file1 ?format1) (file ?file2 ?format2))))
```

The fact embedded in the `tell` defines the mapping between the old and new vocabularies. Consequently, when $\mathtt{A}_0$ makes the request to the facilitator:

```
(ask-one ?ans (file-translation ''f1.tex'' ?ans latex ps))
```

The facilitator uses the previous translation rule in the process of its backward reasoning to translate the request. The request is translated to:

```
(translation (file ''f1.tex'' latex) (file ?ans ps))
```

which is handled by Textures. The processing from this point on is identical to that in the previous example.

This example illustrates that it is possible to modify the interface of an object, and still have the object support its old interface by using the translation capabilities of the facilitator.

These examples have illustrated the complex coordination behavior that is provided by the facilitator automatically in response to a request from a client. The client is not aware of the complicated sequence of events used to process the request.

It should be noted that simple computations are performed by the facilitator if simple interface specifications are given to it. The operations of the facilitator are very efficient in these situations. The clients and servers can make the trade-off on how much work they want the facilitator to perform by controlling the facts communicated to the facilitator and by controlling the power of the inference methods used by the facilitator (simple pattern matching, backward chaining without recursion, backward chaining, or model elimination).

4. Status

There is an API for ACL in C, C++, and Lisp. It hides all the details associated with network communication from the programmer. Each API uses TCP to communicate with the facilitator, but changing the communication protocol (e.g., to DCE) is a trivial task. The API has been used to support interoperation among applications running on UNIX and Windows NT. A prototype of the facilitator is implemented in Lisp and runs on Hewlett-Packard Series 300/700 workstations and the Apple Macintosh.

The Logic Group in the Stanford University Computer Science Department, in conjunction with HP Labs in Palo Alto California, has been developing the facilitation architecture over the last five years. This system has been used in a collection of interoperation experiments, including an integrated design, manufacture, and diagnosis system for digital circuits[5], a multi-domain simulation of a robotic arm [2], and integrated CAD tools for civil engineering [11]. It is currently being used in the CommerceNet project [18], which provides smart search for product information using heterogeneous on-line catalogs, ordering, billing, etc.

5. Conclusion

In this paper we have presented an interoperation architecture based on declarative specification of object interfaces. Given machine-processible object specifications, it is possible to develop new system services that utilize simple inferencing techniques to facilitate interoperation among loosely coupled distributed objects. These services provide additional capabilities (content-based routing, problem decomposition, translation, etc.), and the automation of these services enables using the best information available at runtime.

The reduction in the coupling between a client and server is a consequence of describing requests declaratively. That is, a client describes a request in terms of what it would like, and not how to achieve it. This has the following advantages:

- The client is not required to know about which servers are needed to satisfy the request. This logic has been factored out of the client's implementation and placed in the facilitator, thereby reducing the complexity of the client's implementation.
- The client is no longer responsible for obtaining references to particular instances of the different servers.
- The client is not required to know the exact details about a server's interface.

There are a number of issues that must be addressed in order for declarative interfaces to gain acceptance. Standardizing on KIF and KQML is a good start, however, two problems arise: (1) traditional system programmers are not accustomed to providing declarative specifications, and (2) the syntax of KIF and KQML, which is based on nested parentheses, is not easy for humans to manipulate. The first is harder to solve, since it requires programmers to think about the objects, functions, and relations handled by an object. The second is easier to solve, since KQML and KIF are not meant to be processed by humans directly (this is similar to PostScript– humans use a drawing program to generate PostScript output). The problem can be addressed by developing APIs that map a traditional programming syntax into KQML and KIF.

Facilitated interoperation provides considerable flexibility, but at the expense of runtime efficiency. In small distributed environments, this approach may not be acceptable or needed. As the distributed systems become larger, more dynamic, and more decentralized, this approach may become not only reasonable but necessary.

We have just begun investigating how simple inferencing techniques can be deployed in very large distributed object models. The initial results are promising. As distributed object technologies mature and become more pervasive in supporting highly decentralized distributed systems, we can learn more about the requirements of the problem domain. We can then use those experiences to better focus the development of this approach.

References

[1] Betz, M., "OMG's CORBA," *Dr. Dobb's Special Report,* Winter 1994/95.

[2] Cutkosky, M. et al., "PACT: An Experiment in Integrating Engineering Systems," *Computer* 26, 1(1993), 28-37.

[3] Deschrevel, J. P., "The ANSA Model of Trading and Federation," Architecture Projects Management, Cambridge, 1993.

[4] Finin, T., and Weiderhold, G., "An Overview of KQML: A Knowledge Query and Manipulation Language," available through the Stanford University Computer Science Department, 1991.

[5] Genesereth, M., "Designworld," in the *Proceedings of the 1991 IEEE International Conference on Robotics and Automation,* Sacramento, California, April 1991, pp. 2785-2788.

[6] Genesereth, M. R., Fikes, R., et al., "Knowledge Interchange Format Version 3 Reference Manual" Logic92-1, Stanford University Logic Group, 1992.

[7] Genesereth, M. R., and Ketchpel, S. P., "Software Agents," *Communication of the ACM,* Vol. 37, No. 7 July 1994.

[8] Genesereth, M., Singh, N., and Syed, M., "A Distributed Anonymous Knowledge Sharing Approach to Software Interoperation," in the *Proceedings of the International Symposium on Fifth Generation Computing Systems,* 1994, pp. 125-139.

[9] Gruber, T., "Ontolingua: A Mechanism to Support Portable Ontologies," KSL-91-66, Stanford Knowledge Systems Laboratory, 1991.

[10] Herbert, A., "An ANSA Overview," *IEEE Network,* January/February 1994, pp. 18-23.

[11] Khedro, T., and Genesereth, M., "The Federation Architecture for Interoperable Agent-Based Concurrent Engineering Systems," *International Journal on Concurrent Engineering, Research and Applications,* Volume 2, pp. 125-131, 1994.

[12] Miscrosoft Corporation, *OLE 2 Programmer's Reference: Creating Programmable Applications with OLE Automation,* Volume 2, Microsoft Press, Redmond, Wash., 1994.

[13] Microsoft Technical Backgrounder OLE 2.0, 1994.

[14] The Object Management Group, "The Common Object Request Broker: Architecture and specification," Revision 1.1, TC Document Number 91.12.1, December 1991.

[15] Tawakol, O., and Singh, N. "A Name Space Context Graph for Multi-Context Systems," Logic-95-1, Stanford University Logic Group, 1995.

[16] Vinoski, S., "Distributed Object Computing with CORBA," C++ Report, July/August 1993.

[17] Wiederhold, G., "The Architecture for Future Information Systems," Stanford University, Computer Science Department, 1989.

[18] Wong, W., and Keller, A., "Developing an Internet Presence with Online Electronic Catalogs," Stanford Center for Information Technology, `http://www-db.stanford.edu/pub/keller/1994/cnet-online-cat.ps`