

Enforcing a lips Usage Policy for CORBA Components

Wayne DePrince jr. and Christine Hofmeister

Lehigh University

wayne@jawnee.com hofmeister@cse.lehigh.edu

Abstract

Software components promise easy reuse, dependability, and simplified development. Problems arise when implicit assumptions about the use of the component are encoded in the implementation but not communicated to the user. One solution to this problem is to formally specify these constraints about a component's use. Once specified, these usage constraints can be statically verified or dynamically enforced. This dynamic enforcement code can be provided by either the developer or automatically generated. Our research project, lips, is a language for specifying usage constraints and a toolset for automatically generating dynamic code to enforce them. In this paper we present the notion of a virtual client and show how this is critical for ensuring correct usage of a component. We discuss our experiences providing automatic enforcement of usage constraints for CORBA components: while much of the needed support can be provided easily using a container concept, support for virtual clients requires more fundamental changes in a component model such as CORBA.

1. Introduction

Component-based systems are increasingly popular, as components promise simplified reuse by mandating well defined interfaces and supporting clear physical boundaries [2],[21]. However, in practice designing applications built with components proves to be a difficult task. These problems are due to many different factors, with a major one being the misunderstanding of how to correctly use a component [6],[12].

Correct usage requires a client to understand the behavior of the component it is using, including its functional and non-functional behavior. Our focus is on certain aspects of the non-functional behavior that we call its *usage policy*. The usage policy describes how clients may instantiate and invoke operations on a component, and how the component behaves in terms of instance sharing and concurrent execution of operations. Note that this is a subset of the full behavior of a component, which we do not address.

When using a traditional object, the client knows that each instantiation (“new”) results in a new instance that is not shared unless the client passes the object reference

elsewhere. The client also knows that operations execute in the thread in which they are invoked, so operations do not execute concurrently unless the client invokes them concurrently.

When using today’s component models and middlewares, such as CORBA 2.3, CORBA Component Model 3.0 (CCM), and Enterprise Java Beans (EJB) [7], there is no guarantee about this behavior, either from the client side or the component side.

In CORBA 2.3, a component instance is by default shared by all clients that activate it. The developer may have added code to the component to give each client a separate instance, but this is not specified in CORBA [17]. The client does not know whether the operations it invokes will execute concurrently with other clients’ or whether its own concurrent invocations will be serialized. The component does not know which client has invoked an operation, or even how many clients are using the instance.

In EJB, different types of components (applet, servlets, JavaBeans, Enterprise beans, Session beans, etc.) have different instantiation characteristics, so that by knowing the characteristics of the component type, both client and component know how it will be instantiated. Some information, e.g. about concurrency of operation invocations, is available in the deployment descriptor.

However, none of today’s component models provide a way to fully describe how the component operations may be invoked. They support the specification of an operation’s signature, but not the legal order in which a client may invoke operations, or its *interaction protocol*. The interaction protocol is also part of the usage policy.

Our approach to interaction protocols builds on previous work that uses regular expressions (path expressions) ([5],[15],[23]), or CSP ([1]) to augment interface specifications. Unlike this earlier research, which can only specify constraints in terms of the previously invoked operations, our interaction protocols can rely on dynamic information, such as operation parameter values. In this way, the valid sequence of operation invocations can change dynamically as the component executes. Another limitation of these earlier techniques is that they require a specification of the client’s use of the component in addition to the component specification. A third difference is that our interaction protocols are scoped to indicate the applicability of a particular restriction, e.g. whether it

applies for each client, or across the set of clients using the component instance.

To support the proper usage of components, we have developed the lips language and toolset. The language allows a component developer to specify the usage policy of a component in a language that is independent of any particular component model technology. The usage policy is useful in communicating to a developer how to correctly use the component. In addition, the lips toolset generates runtime support to enforce the usage policy.

The runtime support of course varies depending on the underlying component model used. In this paper we describe our experiences providing this support for CORBA 2.3.^{1 2}

We use an example component to explain the lips approach: the Chat component. This is introduced in Section 2, which focuses on the specification of the usage policy. Section 3 discusses the enforcement of a usage policy, and Section 4 presents our experiences providing usage constraint enforcement in CORBA. Section 5 summarizes related work, and Section 6 concludes the paper.

2. Specifying the Usage Policy

The Chat component is part of a freely distributed application available on the internet [4], developed to illustrate how to use CORBA with Java. It provides a “chat room” capability in which a client posts a message to the Chat component, and all connected clients see the message. In order to create a lips specification for the Chat component, we must first determine its usage policy.

An instance of a CORBA component usually exists in a remote address space from its clients, requiring clients to first locate and connect to an instance via a lookup service (for example the NameService, TraderService, IOR, etc.). Given the name of a component, the lookup service provides the client with a reference to a running instance. Client invocations on that reference are transparently routed by the CORBA ORB runtime using stubs and skeletons generated from an Interface Definition Language (IDL) file.

This IDL is the only mechanism for formally specifying usage constraints in CORBA. Other commercial component models, such as CCM, COM/DCOM, COM+, and EJB use a similar type of interface, but these interfaces provide a very limited

specification, namely the signatures of component operations.

Thus the first source of information about the Chat usage policy is its IDL interface, an excerpt of which is listed in Figure 1. Comments can supplement the operation signatures, as can meaningful names, but these are of course not reliable sources of information.

```
interface Chat
{
    /*
     * Allocate a unique user name, which contains
     * the base name. We would like to do this as
     * part of open(), but we cannot, because we
     * cannot open() until the client has created a
     * ChatClientSvc instance, and it cannot do
     * that until it has a unique name to use.
     */
    string getUniqueName(in string baseName)
    raises (CannotEstablishConnection);

    /*
     * Create a new connection with the server.
     * clientName - a unique name for the client's
     * callback service.
     * cc - the client; used for making callbacks
     */
    void open(in string clientName,
             in ChatClientSvc cc)
    raises (CannotEstablishConnection);

    /*
     * Close the connection.
     */
    void close(in string clientName)
    raises (UnidentifiedUser);

    /*
     * Send a character. The character is broadcast
     * (via callback) by the server to all clients.
     */
    void putc(in char cin);
};
```

Figure 1. Chat component interface from Chat.IDL file

Other sources of information about the usage constraints are domain knowledge (we have an idea of how a chat application should behave), experimentation with the application, and the source code itself.

Each instance of Chat maintains a list of the connected clients. From this we deduce that for clients to chat with each other, they must be referencing the same instance of the Chat component. Thus an application must have a single instance of Chat for each “chat room” desired, and an instance of the Chat component must be shared by multiple clients.

We now summarize the behavior of Chat that is related to its usage policy:

1. A Chat instance must be shared by all clients that wish to chat with each other, and only five clients can chat at a time.
2. Various input and output parameters have restrictions on their values.
3. A Chat instance can have only one operation executing at a time.

¹ In this paper, we refer to CORBA 2.3 objects as components, though in reality the component concept provided by CORBA 2.3 is a weak one when compared to that provided in other technologies, such as EJB.

² Any reference to CORBA is assumed to be CORBA 2.3 unless explicitly stated otherwise.

4. A client, after obtaining a reference to a Chat instance, must first invoke `getUniqueName()` to receive its chat user name. It must then use that name, along with its own reference, as input to an invocation of `open()`. The client may now repeatedly invoke `putc()` to chat. Finally, the client must use that name when invoking `close()` in order to stop receiving chat messages.
5. A Chat instance invokes `putc()` on every connected client when any client invokes `putc()` on that instance.

2.1. Activation Policy

Part of the usage policy is the *activation policy*, which specifies how a client requests access to a component instance, constraints on how an instance is shared among clients, and constraints on the lifetime of an instance. Activation is not the same as instantiation, since an activation may or may not result in the creation of a new component instance.

In CORBA, clients “activate” component instances by using a public name to identify a particular instance. Standard CORBA activation (activating without a developer-implemented factory or POA `ServantManager`) only supports the binding of a public name to a single instance of a component, resulting in a shared component instance.

Another desirable type of activation gives each client activation a reference to a unique component instance. (This is not appropriate for Chat, since clients must share a Chat instance in order to communicate.)

When clients share a component instance, the component instance may need to distinguish one client from another. To support this we introduce the concept of a *virtual client*. Each activation results in the creation of a new virtual client for the component instance.

We call this a ‘virtual’ client because it may not represent a separate physical client: if a client performs an activation twice (perhaps in two separate threads), these will appear to the component as two separate virtual clients. Conversely, if a client performs an activation and passes the resulting reference to another client, the component will treat these as a single virtual client. The client is responsible for knowing that each activation results in a reference associated with a new virtual client, and for managing these references appropriately.

A lips activation policy describes activation constraints in terms of a virtual client. These are the types of activation constraints that can currently be specified:

- Limits on the number of virtual clients per-instance and per-component.
- Restricts the number of instances per-component.

- Limits on the time an instance or a component may exist, including an inactivity timeout.
- The name by which clients may activate the component.
- Activation operations which allow parameterized activation of the component.

2.2. Interaction Policy

The other part of the usage policy is the *interaction policy*, which describes how a client may use the component’s operations after activation. A central part of the interaction policy is the interaction protocol.

For the Chat component, each virtual client of a Chat instance must individually follow the interaction protocol, which has a per-virtual-client scope. The notion of a virtual client originally arose from a need to support interaction constraints and their scopes; thus the virtual client concept is essential to both activation and interaction policies.

The kinds of interaction constraints that can currently be specified in an interaction policy are as follows:

- Allowable sequences of operation invocations, or interaction protocols, per-virtual-client or per-instance.
- The maximum number of concurrent invocations, per-virtual-client or per-instance.
- Constraints on the values of an operation’s input and output parameters.
- Throughput limits on the number of invocations made per some time period, per-virtual-client, per-instance.

2.3. Usage Policy for Chat

The usage policy is comprised of an activation policy and an interaction policy. In lips, a usage policy is associated with a component and describes its restrictions on reuse by clients.

From the operations specified in the Chat IDL and its source code, we derived the usage policy shown in Figure 2. Note that the specification of operations and their associated parameter constraints has been omitted for brevity. However, lips provides for the declaration of pre-conditions on input and output parameter values and interaction protocols for all operations provided by the component.

```
UsagePolicy
{
    Name = "Simple Chat";

    ActivationPolicy
    {
        PublicName = "SimpleChat";
        ClientsPerInstance = "5";
        ClientsPerComponent = "5";
    }
}
```

```
InstancesPerComponent = "1";
InstanceTimeToLive = "unbounded";
ComponentTimeToLive = "unbounded";
InstanceInactivityTimeout = "unbounded";
ActivationOperation
{
  Name = "activate";
}
} /* end ActivationPolicy */

InteractionPolicy
{
  Concurrency
  {
    MaximumConcurrentInvocations = "1";
    Scope = "per instance";
  }
}

Protocol
{
  Name = "chat";
  Scope = "per virtual client";
  Repeatable = "true";
}

Transitions
{
  getUniqueName;
  open;

  repeat
  {
    select
    {
      putc;
      close
      {
        end;
      }
    }
  } /* end repeat */
} /* end Transitions */
} /* end Protocol */
} /* end InteractionPolicy */
} /* end UsagePolicy */
```

Figure 2. Chat Usage Policy in lips

The ActivationPolicy section specifies that all clients must share the same instance. Currently lips has a fixed set of property/value pairs that must be specified in an activation policy. The InstancePerComponent property ensures that only one instance of the Chat component exists. To ensure that only five clients can be using the Chat instance at the same time, the ClientsPerInstance property is set to five. The ClientsPerComponent and ClientsPerInstance properties are equal since there will only ever be one instance of the component.

The InteractionPolicy section first lists the limit on the number of concurrent invocations of operations per instance (in the Concurrency sub-section). The scope here indicates that none of the operation invocations can be concurrent. The InteractionProtocol sub-section lists the allowable order of operations that a virtual client may invoke.

Certain interaction behavior is not specified in the Usage Policy, namely the fact that the Chat instance invokes putc() on all virtual clients in response to the invocation of its putc(). Although specifying interaction

dependencies across a set of components is important for detecting properties such as deadlock, lips does not currently support this.

3. Enforcing the Usage Policy

A lips usage policy is used to automatically generate code that checks the constraints at runtime. The two biggest advantages of runtime enforcement are: the interaction policy can depend on runtime information, and it is not necessary to specify the client's usage of the component's operations. Since this enforcement code replaces some of the code ordinarily provided by a component developer, the performance penalty is reduced.

3.1. Developer Provided Enforcement

Developers of components typically write code to make sure certain usage constraints of the component are not violated. However, developers may not consider all the kinds of usage constraints, or may not expend the effort to check them.

Of the usage constraints for the Chat component, the only one originally enforced was that a Chat instance is shared by up to five clients. Another type of constraint, on the values of input and output parameters, was not checked in the original implementation, although typically a developer does check this in the code. A third type of constraint, regarding serialization of operations ("thread safety"), is often considered by the developer but may not always be correctly enforced.

Constraints on the interaction protocol are seldom checked fully, if at all. In the case of Chat, because the putc() operation does not identify the client, an unregistered client could directly invoke putc(), posting messages even though it is not known to Chat as a client. Another problem is that a client can disconnect another client by invoking close() with the other client's name. These problems can't be detected by the Chat component, since it cannot reliably determine the identity of the client invoking the operation.

Thus Chat has a number of constraints that are not checked, and some that cannot be checked in the existing implementation. We developed Chat2, an implementation of Chat that enforces the usage constraints we identified. Chat2 shows what the developer should have done to enforce the usage constraints properly.

We had to alter the interface for Chat2 in order to keep track of the client. An input parameter was added to every operation to identify the client making the invocation.

The design of Chat2 not only pushes the problem of client tracking back onto the client, but also trusts the clients to provide the correct information. If a client

provides the wrong client identification (intentionally or unintentionally), enforcement is silently bypassed.

To illustrate the increased effort in implementing usage constraint enforcement, we compared different implementations of Chat: the original, Chat2, and Chat3, which uses the lips toolset to automatically generate the enforcement code. Table 1 lists the lines of code (LOC) of each implementation. The LOC counts for Chat2 do not include the extra coding effort requirement placed on clients. A LOC is considered enforcing a usage constraint if without it the constraint would not be checked.

Table 1. Comparison of LOC used to check usage constraints in original Chat, Chat2, and Chat3

Component	Total LOC	LOC for usage policy enforcement	% of total LOC for usage policy enforcement
Chat	76	5	6.579%
Chat2	246	154	62.602%
Chat3	595	500 *	84.033%

* combination of libraries and automatically generated enforcement code

The results in Table 1 show that even for the simple Chat component, enforcing the usage constraints is a significant effort. Since the functional behaviors of Chat, Chat2, and Chat3 are identical, it is fair to state that most of the additional LOC are due to checking the usage constraints. Of course we understand that for this simple implementation, the enforcement code is a larger proportion of the source code than is typical. However, the enforcement code will be a significant effort for any component. Also, though the original Chat component was developed as a part of a trusted application and may not need to protect itself from malicious clients, enforcement of usage constraints can also help the developer detect programming and design errors.

Because enforcement is a significant effort and is often incomplete, lips provides tools to generate this enforcement code automatically.

3.2. Automatic Generation of Enforcement Code

To enforce the usage policy we use a container. A container is a concept similar to the Proxy design pattern [11]. It is currently being used in some newer commercial component models, such as CCM, EJB, and COM+. A container is substituted for the component instance during client activation. Because the container implements the same interface as the component, clients transparently invoke operations of the component through

the container. By putting the enforcement code in a container, we separate the core functionality of the component from its usage constraint enforcement.

The lips compiler checks the usage policy, then generates a lips container. This lips container is implemented in Java and is component-model independent. A second lips tool, the deployer, then generates additional code to integrate the lips container into a particular technology such as CORBA.

In Table 1, we showed the LOC for our lips version of the Chat component, Chat3. This implementation makes use of the lips toolset to automatically generate the enforcement code. Many of the LOC used for enforcement in Chat3 are actually part of a generic container library we developed, which the generated code makes use of. However, it is still code that the developer does not need to write or maintain.

In the lips generated container for Chat3, the identity of the virtual client invoking an operation is known. With this information, the container can ensure that only the same Chat client can disconnect itself, and no unregistered client can post messages.

4. The lips Container for CORBA

To enforce Chat's activation and interaction constraints in CORBA, the lips deployer must insert enforcement code that intercepts the activation of the component and each subsequent invocation of its methods. Our usage constraint enforcement code is embodied within the lips container, which includes the lips activator for handling activation constraints.

4.1. Supporting the Activation Policy in CORBA

In CORBA the standard activation mechanism supports only one type of activation: a shared component instance. With the addition of the Portable Object Adapter (POA) specification, CORBA also supports per invocation activation. However, to support other types of activation and to identify virtual clients, the lips container must intercept all client activations. Rather than returning a reference to the component, the container returns a reference to itself, thus ensuring that subsequent invocations of operations will be intercepted by the container.

Our solution makes use of the standard CORBA activation services by registering an *Activator* instance rather than the component instance with the NameService. The Activator is part of the lips container, and it controls the component activation, identifies virtual clients, and sets up the interception of operation invocations.

An Activator, which is generated by lips for each type of component, is similar to a Factory [11]. Clients use the

same public name and CORBA lookup service, but now obtain a reference to the component's Activator instance rather than the component instance. Clients then invoke activation operations to activate and obtain a reference to the container.

The activation operations are specified in the activation policy, and they can include input parameters. This Activator, working with the container and executing on the same host as the component instances, enforces the activation policy the component specifies, and creates (as necessary) new instances of the component. The Activator itself is implemented as a CORBA component that is generated by the lips deployer from the usage policy (more specifically the activation policy). Clients are provided with the generated IDL for the Activator.

Though the Activator-based activation alters the standard activation sequence of CORBA, the change is minimal. These changes are necessary because CORBA provides no transparent way to intercept the standard CORBA activation process.

To illustrate the differences between the standard CORBA activation (used by Chat, Chat2) and lips activation (Chat3), we show the original code in Figure 3, with the activation statement in bold, italicized type. Figure 4 shows the two statements that replace the original activation statement in order to use lips.

```
// Initialize the ORB.
orb = org.omg.CORBA.ORB.init(args, null);

// Get the NamingService.
org.omg.CORBA.Object obj =
  orb.resolve_initial_references("NameService");

NamingContext ncx;
ncx = NamingContextHelper.narrow(obj);

// Lookup Chat instance via public name.
NameComponent nc =
  new NameComponent("ChatRoom", "");

NameComponent path[] = {nc};
Chat svr = ChatHelper.narrow(ncx.resolve(path));

// Start using Chat.
String name = svr.getUniqueName("christine");
```

Figure 3. Standard CORBA activation used by clients of Chat and Chat2

```
ChatActivator act =  
ChatActivatorHelper.narrow(ncx.resolve(path));

// Now, activate instance of Chat.
Chat svr = act.activate();
```

Figure 4. Changes required for clients of Chat3 (using lips)

Since the Chat activation policy specifies that all virtual clients share the same instance, the generated ChatActivator simply creates an instance upon the first

invocation of activate() (Chat's only activation operation). Subsequent invocations return a reference to the same instance. If five virtual clients are connected, then any additional activations will fail.

4.2. Supporting the Interaction Policy in CORBA

In order to support interaction constraints, the container must manage the threading of operation invocations, check the interaction protocol, and check the parameters and return values of operation invocations. Because the threading and interaction protocol are potentially scoped per-virtual-client, the container must know the virtual client associated with each operation invocation.

The approach taken with Chat2 was to modify each operation by adding a parameter identifying virtual client. This is not acceptable for lips, because it changes the component interface and requires potentially extensive changes to the client. It is also unsafe to have clients keep track of references. Since we are trying to provide the usage constraint support automatically, this virtual client information must be transparently communicated.

One possible solution is to insert the container inside the ORB serving the component instances. This would be accomplished using the Portable Interceptors service of CORBA [8]. Portable Interceptors allow code registered within an ORB to be executed when invocation requests are received. These interceptors provide a perfect place for the container to be transparently inserted in the invocation pathway. These interceptors are also portable, allowing lips containers to be used with any ORB and any CORBA supported programming language.

Unfortunately, the information needed by the container, namely the identity of the virtual client making the invocation, is not available within a Portable Interceptor handler. In fact, CORBA treats all virtual clients making invocations through a single ORB as one client [3]. This means that five separate instances of Chat clients using the same ORB would be seen as a single client within an interceptor. Thus CORBA has no standard support for tracking virtual clients.

Another solution is to modify the stubs and skeletons generated from the IDL. This approach has the drawback of negating CORBA's interoperability when using lips.

Our solution is to create a new VirtualClient object in the container upon each activation, similar to the way a web server tracks sessions. The Activator returns this VirtualClient reference to the client. The VirtualClient object implements the component's interface by forwarding each invocation to the container, along with its virtual client identification. The container then uses the extra information added by the VirtualClient object to track a virtual client's use of a component instance. Thus, transparent virtual client tracking is added to CORBA.

4.3. Status

Our implementation of the lips toolset is in progress. It is composed of the following:

- The lips container library, written in Java, provides services used by the container, including thread synchronization and queuing, parameter constraint enforcement, and interaction protocol checking.
- The lips compiler generates a component-model-independent container in Java from a usage policy. This container is delivered as a set of Java classes, packaged together in an archive.
- The lips deployer, given a lips container and usage policy, generates the code needed to integrate the container into a particular technology, such as CORBA (Activator IDL, VirtualClient, etc). It also creates code to enforce activation constraints.

Initially we implemented the container features by hand in order to test our designs.

In the current version of the toolset, the lips compiler generates containers that support constraints on parameter values, concurrency, and interaction protocols. The deployer provides integration support for CORBA. We have manually adapted containers to work with JavaBeans components, and are currently implementing a JavaBeans deployer.

The version of the lips language presented here is the first version. We have begun extending the language and toolset to support interactions that span multiple components. Thus we are adding the ability to specify required interfaces in addition to provided interfaces.

5. Related Work

Our usage policy is related to other work in improving component specification. ADLs, such as [1],[16],[18], also constrain the instantiation and connection on components. However, these restrictions are concerned with the overall architecture of many connected components. The usage policy instead specifies only the local architectural constraints of a single component. Other component specification languages, like [13] and [20], provide ways to indicate high level properties about a component, such as security and performance, which can be used to evaluate a component's potential use in an application. Instead, lips specifies more fundamental constraints on component reuse.

In [17], the ADL Darwin is used to specify the architecture of a CORBA system. From the Darwin specification, code is generated which implements the particular connections between CORBA components. In

lips, we instead focus on the local constraints of each component. Via FSP, Darwin can support interaction constraints within connectors, but these are specified within the context of an application, while in lips we specify them in the context of a single component. In addition, the lips interaction protocol, being dynamically evaluated, allows the valid protocols to change at runtime.

Previous work has been done on evaluating support for various types of usage constraints. In [14], COM/DCOM interfaces are augmented with behavior specifications used to detect deadlock, as well as automatically generate connectors which are deadlock free. This work, unlike lips, requires the specification of not only the component's behavior, but the client's as well. Currently lips does not provide deadlock detection, but it could be inserted into the lips container using a dynamic technique such as [10].

The work in [5] extends the CORBA IDL to allow the specification of message-based constraints on the interaction of CORBA components. This information, along with specification of a client's use of that component, is used to statically verify the restrictions. However, no concurrency or activation constraints are supported. Also, our specification is technology independent, as shown by our previously mentioned work with JavaBeans.

Path expressions are used in [23] to specify valid message collaborations between objects, from which code for runtime validation is generated. However, the activation policy and concurrency of operations are not supported.

In [22], the authors add an assertion capability to an open source EJB container, allowing the runtime maintenance of preconditions, postconditions, and invariants. While providing for dynamic enforcement of constraints, these assertions must be coded as plugins into the EJB container, rather than generated from a specification, as is done in lips.

Many developers recognize the need for factories and use them to support functionality similar to the Activator [19]. The CORBA Lifecycle Service provides a framework that can be used to implement factories which enforce certain activation constraints. COM also provides a type of Activator, known as a ClassObject. These solutions allow developers to implement their activation policy, but only manually in code. The lips Activator provides a richer set of activation properties, and automatic generation of enforcement code.

The CORBA Portable Object Adapter (POA) provides basic concurrency protection. The POA ThreadPolicy only gives the developer a choice of serialized invocations or other, known as ORB controlled. Since there is no standard way to control how the ORB handles threading, this results in an ORB specific solution, which may or may not support multithreaded operation

invocations. In lips, the component can control the exact number of concurrent invocations for each virtual client, instance, and component, and have those constraints enforced automatically.

Both the POA and Lifecycle services are code level entities, which must be programmatically invoked and implemented. This does not aid in the communication of usage constraints, and still requires developer implementation.

Our work in lips is related to the Object Constraint Language (OCL) [REF] in that we support pre-conditions on operations. However, OCL does not provide direct support for constraining the use of those operations.

[this last paragraph is a bit out of the blue—put it in the context of contracts.]

6. Conclusion

The goal of the lips research project is to simplify component development and correct component reuse through the formal specification of usage constraints and their automatic enforcement. Usage constraints are expressed in a component's usage policy, which describes the activation policy and interaction policy of the component.

Some of the concepts and services covered by a lips activation policy are also being addressed by the newer component models, including CCM and EJB. Both of these technologies have strong activation concepts, improved specification of activation constraints, and automatic service support. However, one important difference between lips and CCM / EJB is the virtual client concept. Without support for virtual clients, activation and interaction constraints can not be properly supported. Another important difference is that they do not support an interaction policy.

A key part of the lips interaction policy is the interaction protocol, which has several characteristics not found in previous work. First, the client's use of the operations does not need to be specified. Second, a lips interaction protocol is scoped to indicate whether it applies per-virtual-client, per-component-instance, or per-component. And finally, a lips interaction protocol may be defined in terms of information, such as parameter values or timeouts, that is not known until runtime.

One of the most interesting results of this work is the introduction of the virtual client concept. This also turned out to be one of the challenges for supporting lips with CORBA. In implementing the enforcement of lips usage policies for CORBA, we needed to:

- intercept client activation (which creates a new virtual client)
- intercept invocations of operations
- identify the virtual client invoking an operation

While CORBA provides a mechanism to transparently intercept operation invocations, it does not provide the ability to transparently intercept activations. Our solution requires a small change to the client code in order to support all of the features in a usage policy. Because the interaction policy depends on identifying a virtual client, the activation must let both client and container know the identity of the new virtual client. In CORBA this cannot be done without intercepting activations.

Thus the major weakness in CORBA's ability to support lips usage policies is its lack of a virtual client concept. As we have illustrated in this paper, virtual client support is essential in providing the enforcement of activation policies. These restrictions themselves are expressed in terms of virtual clients, as opposed to client instances, as used in other component models. Enforcement of interaction policies is also impossible without mechanisms to identify virtual clients. Component models wishing to support virtual client usage must provide a method by which a virtual client can be tracked by its use of component operations.

The next steps for this research are to support additional component models. We are currently working on supporting JavaBeans, a component type with very different characteristics from the components in CORBA. Prototype containers have been written for JavaBeans, and the automated generation of these containers is in progress. Another future platform to be supported is CCM.

Future considerations include specifying interaction protocols that span multiple components and required interfaces.

7. References

- [1] Allen, R., Garlan, D.: Formalizing Architectural Connection. In: Proceedings of the Sixteenth International Conference on Software Engineering. Sorrento, Italy (May 1994) 71-80.
- [2] Bachman, F., Bass, L., Buhman, C., Santiago, C., Long, F., Robert, J., Seacord, R., Wallnau, K.: Volume II: Technical Concepts of Component-Based Software Engineering. CMU/SEI-2000-TR-208 SEI Technical Report (May 2000).
- [3] Baldoni, R., Macheetti, C., Verde, L. : CORBA Request Portable Interceptors: Analysis and Applications. In: Proceedings of the 3rd International Symposium of Distributed Object Applications (DOA '01). Rome, Italy (September 2001).
- [4] Berg, C. : Chat Java/CORBA component. <http://www.digitalfocus.com/ddj/code/>

- [5] Bokowski, B.: IPDL - Interaction Protocols for Distributed Objects. In: Proceedings of the KI-96 Workshop on Agent-Oriented Programming and Distributed Systems. DFKI Document D-96-06. Saarbrücken (1996).
- [6] Compare, D., Inverardi, P., Wolf, A.: Uncovering Architectural Mismatch in Component Behavior. In: Science of Computer Programming, Vol. 33, No. 2. (1999) 101-131.
- [7] CORBA 2.3. <http://www.omg.org/cgi-bin/doc?formal/98-12-01>
- [8] CORBA Portable Interceptors. <http://www.omg.org/cgi-bin/doc?ptc/2001-03-04>
- [9] DePrince Jr., W., Hofmeister, C.: Analyzing Commercial Component Models. In: Bosch, J., Gentleman, M., Hofmeister, C., Kuusela, J. (eds.): Proceedings of WICSA3 - Software Architecture - System Design, Development and Maintenance. Kluwer July (2002).
- [10] Engen, A., Bradshaw, M., Oostendorp, N.: Extending Java to Support Shared Resource Protection and Deadlock Detection in Threads Programming. In: ACM Crossroads 4.2 (1997) : 9-17.
- [11] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. 16th edn. Addison-Wesley Reading, MA 1998.
- [12] Garland, D., Allen, R., Ockerbloom, J.: Architectural Mismatch or Why it's hard to build systems out of existing parts. In: Proceedings of the 17th International Conference on Software Engineering. Seattle, Washington. ACM SIGSOFT (April 1995).
- [13] Han, J.: An Approach to Software Component Specification. In: Proceedings of 1999 International Workshop on Component Based Software Engineering. Los Angeles, USA (May 1999).
- [14] Inverardi, P., Tivoli, M. Automatic Synthesis of Deadlock free connectors for COM/DCOM Applications. ESEC/FSE 2001, Vienna, Austria.
- [15] Liu, C., Richardson, D. Towards Discovery, Specification, and Verification of Component Usage. In: ASE 99: Proceedings of the 14th International Conference on Automated Software Engineering, sponsored by IEEE Computer Society Boca Raton, FL. October (1999).
- [16] Luckham, D.C.: Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. In: DIMACS Partial Order Methods Workshop IV, Princeton University, July 1996.
- [17] Magee, J., Tseng, A., Kramer, J.: Composing Distributed Objects in CORBA. In: Proc. of ISADS'97, IEEE Computer Society Press (Berlin, Germany, April 1997) 257-263.
- [18] Magee, J., Dulay, N., Eisenbach, S., Kramer J.: Specifying Distributed Software Architectures. In: 5th European Software Engineering Conference (1995) 137153.
- [19] Merle, P., Gransart, C., Roos, J., Geib, J. : CorbaScript: A Dedicated CORBA Scripting Language. In: CHEP'98 Computing in High Energy Physics. Chicago, Illinois, USA (1998).
- [20] Stafford, J., Wolf, A.: Annotating Components to Support Component-Based Static Analyses of Software Systems. In: Grace Hopper Celebration of Women in Computing. Hyannis Massachusetts (September 2000).
- [21] Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Harlow, England (1999).
- [22] Vecellio, G., Thomas, W., Sanders. R. Containers for Predictable Behavior of Component-based Software. Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering. Orlando, USA, 2002.
- [23] Yellin, D., Strom, R. Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors. OOPSLA, 1994.