

Specifying Architectural Constraints on Components

Technical Report LU-CSE-03-004

Wayne DePrince Jr.
Lehigh University
19 Memorial Drive West
Bethlehem, PA 18015 USA
01.610.758.6972
wayne@jawnee.com

Christine Hofmeister
Lehigh University
19 Memorial Drive West
Bethlehem, PA 18015 USA
01.610.758.4103
hofmeister@cse.lehigh.edu

ABSTRACT

Research to improve component reuse has focused on providing the specification of various behavior properties. In this paper we present our approach to this problem, which focuses not so much on specifying the behavior of the component, but instead on its architectural constraints. We introduce our research project “lips”, a language for formally capturing these usage constraints and a toolset for automatically providing for their enforcement at runtime. Our approach captures the architectural constraints that are local to a particular component. In this way we express these restrictions on its reuse independent of an actual client or application. We then embed these constraints within the component’s specification, and use it to generate code which will enforce the constraints at runtime.

Categories and Subject Descriptors

1.1 D.2.1 [Requirements/Specifications]: Languages – *abstract data types, polymorphism, control structures.*

General Terms

Documentation, Experimentation, Languages, Verification.

Keywords

Component, Component Model, Component Specification, Architectural Constraints, CORBA, IDL, EJB, Architecture Description Language.

1. INTRODUCTION

That software components need better specification is a well known fact [5],[13]. Better understanding of a component is critical in being able to reuse a component correctly. In the development of any component, the developer makes implicit assumptions about the ways in which it will be reused [9]. In essence, these assumptions are part of the component’s design. The problem is that these assumptions are hidden in the implementation, but not actually specified formally.

Classically, the only constraints on the reuse of a component were specified as operation signatures with supporting documentation. An example of this would be CORBA’s Interface Definition Language (IDL). Unfortunately, operation signatures only provide syntax and data type constraints, while human generated documentation is open to a host of problems, including language and interpretation differences. Problems with these standard component specifications are described in other research [9],[10],[13],[14].

Modern approaches to this problem of component reuse look to use descriptions of the component’s behavior to help clients better understand how a component functions [5]. Current research in this area includes Architecture Description Languages (ADLs) and component quality attribute specification languages.

Research with component specification languages, such as [3],[11],[12],[14], provide formal languages in which behavior, non-functional attributes, etc., can be specified. These languages aim to describe the behavior of a component, often concentrating on providing information about various qualities, including performance, memory requirements, and security. These properties, or quality attributes, are then applied to statically evaluate the utility of a particular component for some application. They also allow potential clients to better understand how the component behaves.

ADLs also provide richer component specifications. However, since an entire architecture is being described, ADLs like [1] concentrate more on specifying where and how a component can be placed within an architecture. Constraints exist within the architecture on the connections between components, thereby constraining the interaction among, and the instantiation of, all components.

In our work we look to tackle the problem of correct component reuse by taking a simple, more practical approach. As in other component specification languages, we look to augment the description of a component with additional information. But instead of specifying how a component behaves, we instead specify the component’s architectural constraints on its reuse, and then enforce those constraints at runtime.

In this paper we present our research project “lips”, a specification language used to formally specify implicit assumptions, and a toolset for their automatic enforcement at runtime. We define these restrictions on reuse as *usage constraints*. Usage constraints are constraints on component instantiation and connections in an architecture, as well as constraints on how a component can be interacted with. This is similar to the approach taken by ADLs. However, unlike ADLs, the architectural constraints are local to a particular component, independent of the rest of the system. The independent usage constraints for each component can then be dynamically enforced as the component and clients interact.

The remainder of the paper is as follows: in Section 2 we present an example component to illustrate some problems with its reuse. Section 3 introduces the types of usage constraints we have identified as important for correct reuse. In Section 4 we discuss the importance of usage constraint enforcement and how different

techniques effect the specification. Finally, we summarize our approach and presented positions in Section 5.

2. EXAMPLE

To illustrate some of problems that can occur if reuse assumptions are ignored, and the usage constraints that capture these restrictions, we consider a simple CORBA component named “Chat” [2]. The Chat component is an open source implementation of a chat room, provided as an example of using Java with CORBA 2.x [6]. The IDL interface of Chat is very simple and shown in Figure 1.

```
interface ChatClientSvcs
{
    /*...*/
}

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 ChatClientSvcs
     * 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 ChatClientSvcs 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 IDL interface.

As mentioned earlier, the IDL interface provides very simple information on the correct use of Chat. We can see in Figure 1, the operation signatures of the components are the only types of formal information presented. The comments may provide some useful additional information, but again they are open to (mis)interpretations and inconsistencies. Some of the concerns not addressed by this interface include:

- Can the operations be invoked asynchronously?
- Should clients share an instance of the Chat component, or must each client connect to a unique instance?
- What are the valid values for input and output parameters?
- Is there a particular sequence for invoking operations?

These concerns represent basic usage constraints on the Chat component that are ignored by the formal specification provided with the component. However, they are critical in reusing this component correctly, since they embody assumptions made by the developer that are encoded into its very implementation.

Unfortunately, the answers to these usage constraint questions had to be inferred from the source code and context of Chat. And though source code can be considered a formal language, the reader will agree that it should not be used a means for communicating usage constraints. Furthermore, for many of the commercial and COTS components whose use we are trying to facilitate, the source code is not even available.

In our analysis of Chat’s source code, we discovered answers to the questions raised above about how to reuse the component correctly. Again, the source code was the only place that these restrictions were “specified”. Here we give some of the restrictions identified in Chat:

- An instance of the Chat component must be shared by all clients that wish to participate in the chat session.
- An instance of the Chat component can only support a fixed number of connected clients.
- A client, after obtaining a reference to an instance of Chat, 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 invoke close() when it no longer wishes to chat.
- An instance of the Chat component can have only one operation executing at a time.
- An instance of Chat invokes putc() on every connected client when any client invokes putc() on that instance.

These constraints, though seemingly simple, are critical for any client of Chat. That these restrictions exist for a component as straightforward as Chat shows the fundamental level of these constraints. Before tackling other specification issues, we feel that this problem must be solved. Now let’s look at two types of usage constraints that we have identified.

3. USAGE CONSTRAINTS

Software components make many restrictions on their reuse, which we call usage constraints. As previously mentioned, our approach associates with each component a formal specification about the correct ways by which it can be reused. The specification of these usage constraints is manifested as restrictions on the client’s use of a component’s operations.

After analyzing Chat (and other components), we have identified two types of usage constraints. The first type constrains the process by which clients obtain references to an instance of a component. The second type specifies how to use the operations of a component correctly. These two classes of usage constraints are similar to the constraints provided by ADLs, restricting the instantiation and connection of components. Of course, the usage constraints presented here apply to a particular component’s reuse, not the connection between two or more components. Our

research projects defines a specification called a *usage policy*, which is used to formally define these types of usage constraints.

3.1 Activation Constraints

For a client to interact with a component, it first must obtain a reference to an instance of that component. A component instance is an executing image of the component that contains state, analogous to an object of a class in object oriented programming. So a client of a component needs a method by which it can obtain a reference to a component instance. Once obtained, this reference can be used by the client to invoke operations of the component. We define the process by which a client obtains a reference to a component instance as *activation*. Activation is similar to object instantiation, with one important difference. Every activation performed by a client may or may not result in the creation of a new instance of the component.

We classify assumptions on a client's activation of a component instance as *activation constraints*. In our example component, Chat, we identified different activation constraints. These include the requirement that all clients wishing to chat with each other must reference the same Chat instance. So in essence, for each logical chat room there exists only one instance of Chat, which must be shared among all clients. To enforce this constraint, only the first activation request by a client would result in a new Chat instance. Subsequent activations would return a reference to the same instance. This requirement must be communicated and enforced, so an application using Chat will be structured correctly.

Usually a constraint like this is implemented in source code. In COM, for example, this constraint would be manifested in the ClassObject of a COM component. By moving these constraints into a specification language, not only is the information more clearly communicated to clients, but it can also be used to automatically generate the code which will enforce it. Within a *usage policy*, an *activation policy* contains the formal specification of these types of constraints. We now list the types of activation constraints that can currently be specified in an activation policy:

- Limits on the number of 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.

A similar idea is being used in the Enterprise Java Beans (EJB) component model [8]. In EJB, components declare themselves as a certain type of component, which indicates a certain activation policy. Then, the EJB container enforces the activation constraints associated with that particular component type. However, in our usage policy, we provide more fine grained control of activation restrictions. Also, our *usage policy* specification language is not platform specific.

During our investigation of usage constraints, we noted a distinct difference between the concept of a client, as seen from the

component's and client's point of views. Consider the example component Chat, where one instance of the component is meant to be shared by multiple clients. An instance of Chat must keep track of each individual client, in order to handle connections, disconnections, and chat messages. Now consider an instance of a multithreaded client which activates the Chat component twice, each time on a separate thread. Each activation will return a reference to the same instance of Chat, as specified by the activation constraints. To the Chat instance, it seems as if two clients are now connected. However, these two logical clients are actually the same physical client instance.

To represent this distinction, we define a new concept called a *virtual client*. A virtual client represents a component's view of a logical client, indicated by an activation request. This facilitates the specification of activation constraints by allowing restrictions to be expressed in terms of virtual clients, which is how a shared component instance naturally views clients.

For Chat, our activation policy captures the constraint on instantiation which limits the total number of virtual clients that an instance can support to five, and restricts the number of Chat instances to one.

3.2 Interaction Constraints

Once a virtual client has obtained a reference to a component instance, the virtual client and component instance can now interact via operation invocations. The other type of usage constraint we are presenting in this paper deals with these types of interactions, and is called an *interaction constraint*. Interaction constraints restrict the use of operations by virtual clients. These interaction restrictions are similar to the connection constraints of ADLs, but only specify them for a single component.

For the Chat component, various interaction constraints exist. First, the Chat component expects each virtual client to follow a certain operation invocation sequence, in essence ensuring each client registers and receives a unique name. It is important to note that this interaction constraint is applied per-virtual-client, meaning each virtual client must independently follow this sequence. We call this characteristic the scope of the interaction constraint. The scope indicates the applicability of a restriction, which can be per-virtual-client, per-instance, or per-component (all instances of the component).

Another very important interaction constraint illustrated by Chat deals with concurrency. Nowhere in the Chat IDL interface is it indicated if the Chat implementation is safe for asynchronous operation invocation, whether by the same virtual client (multithreaded), or different virtual clients. This is particularly important for this component, as the very nature of a chat room implies asynchronous invocations by multiple virtual clients. After analysis of the Chat implementation, we discovered that an instance of Chat could only have one operation executing at a time, due to per-virtual-client state shared within an instance of Chat. Thus, an interaction constraint exists which restricts the maximum concurrency of operation invocations to one, per-instance.

These types of interaction constraints are contained in an *interaction policy*. The interaction constraints that can currently be specified in an interaction policy are:

- 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.

3.3 Chat Usage Policy

After presenting some of the activation and interaction constraints we identified in our example component, Chat, we now present an excerpt from the lips usage policy for Chat in Figure 2 (note the specification of operations and their parameter constraints has been omitted for space). This usage policy captures the constraints we identified earlier.

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

    ActivationPolicy
    {
        PublicName = "SimpleChat";
        VirtualClientsPerInstance = "5";
        VirtualClientsPerComponent = "5";
        InstancesPerComponent = "1";
        InstanceTimeToLive = "unbounded";
        ComponentTimeToLive = "unbounded";
        InstanceInactivityTimeout = "unbounded";
        ActivationOperation
        {
            Name = "activate";
        }
    }
    /*...*/
    InteractionPolicy
    {
        Concurrency
        {
            MaximumConcurrentInvocations = "1";
            Scope = "per instance";
        }

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

            Transitions
            {
                getUniqueName;
                open;

                repeat
                {
                    select
                    {
                        putc;
                        close
                        {
                            end;
                        }
                    }
                }
            }
        }
    }
}
```

Figure 2. lips activation and interaction policy excerpts from the Chat usage policy.

In the ActivationPolicy section, we first see how the various activation constraints we discovered within Chat are specified. First, to indicate that only one instance of the Chat component should exist, we set the InstancePerComponent property to one. To constrain the number of virtual clients that an instance can support, VirtualClientsPerInstance is set to five (Note the VirtualClientsPerInstance and VirtualClientsPerComponent properties are equal since there will only ever be one instance of the component).

The InteractionPolicy section of the usage policy firsts constrains the allowable concurrency of an instance of Chat. This is specified in the Concurrency subsection, which limits the maximum concurrent invocations to one for the whole instance (indicated by the Scope property being set to “per instance”). Next, an *interaction protocol* is presented which states the valid transitions that each virtual client must independently follow (again specified by the Scope). Note the requirement that the Chat instance invoke putc() on all virtual clients after having its putc() invoked is not specified. We are currently looking at ways to support this constraint, since it complicates not only the specification of these usage constraints but support for their enforcement as well.

4. ENFORCEMENT

4.1 Implementing the Specification

Formally specifying usage constraints is a big step in ensuring the correct reuse of a component. But even with a usage constraint specification, a component implementation being used by unknown and untrusted clients (as is the case with many COTS and commercial components) must still be sure that the assumptions are being satisfied. Without enforcement of the usage constraints, they are not very useful. Trusting the clients to simply follow the restrictions is not sufficient, as malicious clients will purposely ignore the constraints, while ignorant clients will misapply them. What is needed is a way to enforce these usage constraints, thereby checking that a client is honoring the restrictions a component has placed on its reuse.

For a standalone component to ensure its correct reuse, a developer must create additional code to enforce the usage constraints. This code is often thought of as “error checking” code, and is usually only implemented if “there’s time”. We feel this additional effort is one of the reasons why these constraints are not checked by developers. In fact, the Chat component only verified a single usage constraint, limiting the number of virtual clients that a Chat instance can support.

To illustrate the additional effort needed to implement usage constraint enforcement, we developed two additional implementations of the example component. In the first, named Chat2, we manually provided source code for the runtime enforcement of the usage constraints listed in Section 2. The second, named Chat3, we used the lips toolset to automatically generate the runtime enforcement code from the Chat usage policy. Table 1 gives a comparison of the amount of effort, measured in lines of code (LOC), needed for each implementation. A LOC is considered enforcing a usage

constraint if without that LOC 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

As shown in Table 1, the effort to manually enforce usage constraints in the Chat2 component implementation is not trivial. Though we understand Chat2 is only a simple component implementation, the fact that the percentage of code needed to implement the restriction checking is so large, this leads us to believe that the amount of effort for any size implementation would be a significant part of the total effort.

It is also important to state that the original Chat implementation was a simple example used to illustrate some programming concepts. Thus it follows that little effort would be exerted to make the component robust. Also, we must remember that Chat is not a standalone component. The same developer creating the component implementation also creates the client. Thus Chat is part of a trusted application, and so perhaps had no need to enforce these usage constraints. Even still, by checking these restrictions in trusted applications, programming and design errors may be discovered.

4.2 Dynamic vs. Static

We feel that because of the techniques used in enforcing component specifications, other approaches have simplified or ignored the usage constraints we present here. This is due to the fact that the enforcement technique often used is static verification. By committing to static verification, even the simplest of constraints becomes difficult to specify and often impossible to check. Of course it would be nice to know if these constraints will be honored before the component is used. However, it is not only very impractical, but can also be very limiting.

With static verification of a component's usage constraints, certain limitations are placed on the type of restrictions that can be specified. For starters, to statically verify the use of a component, one must also know the possible uses of the component by virtual clients. This is seen in the need for the client's source code in [11], or the behavior specification of both sides of a connection in [1]. As such, any static verification not only requires the specification of how to use a component correctly, but also the specification of how each virtual client will use the component. This information is often not available at development time for components which are created to be reused in a binary form at some later time. With runtime enforcement of usage constraints, however, the specification of a virtual client's use of a component is not necessary.

Another benefit of dynamic enforcement is that additional properties of a component can be specified. So a specification language which targets dynamic enforcement can be very expressive about many types of usage constraints. In fact, for the types of usage constraints we are interested in, dynamic enforcement is the practical way that they can be checked.

One of the main reasons for the shunning of dynamic enforcement of usage constraints is due to concerns over performance. Adding code to dynamically check the constraints at runtime will obviously increase the execution time of a component. However, we feel that this drawback is easily outweighed by the practicality, expressiveness, and simplicity that dynamic enforcement provides for a usage constraint specification language. For any developer who wishes to implement the enforcement of the usage constraints presented here, they would have to implement the same dynamic checks that we automatically generate from a specification (as we did in our implementation of Chat2). This suggests that dynamic enforcement is more practical for general purpose software development.

We see no reason to dismiss dynamic enforcement as others do [3],[4], especially when one considers the increase in constrainable properties and simplicity of specification that dynamic enforcement provides over static verification. Of course, if performance is an issue, the dynamic checks can be enabled and disabled at runtime for trusted applications.

4.3 Automatic Dynamic Enforcement

One of the main goals of our research project (and hopefully any specification language) is not only to specify the correct reuse of the component, but simplify the development of the component by using the specification to automatically generate code which will dynamically enforce the constraints.

In Table 1, we also showed the LOC for another implementation of the Chat interface, Chat3. Chat3 makes use of the lips toolset to generate the code which will dynamically enforce the Chat usage policy. It is important to point out that much of the LOC used for the enforcement in Chat3 is actually libraries, which the generated code depends on. However, either way it is code the developer does not have to create.

Automatic generation of enforcement code provides many well known benefits to the component developer, including simplified implementations and more correct code. A critical concern with specification languages is maintaining their correspondence to implementation code. With automatically generated enforcement code, this problem is simplified, as the implementation is generated from the specification language.

In lips, the generated usage constraint enforcement code is embodied in a container. A container is similar to a proxy design pattern, allowing interactions on a component instance to be monitored and constrained. The commercial component models EJB and CORBA Component Model (CCM) [7],[8] both use this concept to provide services to component developers. In [15], containers are used to provide runtime assertion checking for components.

Our lips toolset is currently in progress. The design of the lips toolset is based on a lips container, which is component model neutral. This container is mostly implemented by a Java class library we developed for handling various usage constraint tasks

like thread synchronization, queuing, and scheduling, interaction protocol enforcement, and parameter checking. Then, a usage policy is processed by the lips translator which generates code to bind the lips container to a particular component and component model. We currently can generate code to allow the lips container to constrain the activation, check parameter values, and enforce the interaction protocols for CORBA 2.3 components implemented in Java. We are also working towards providing bindings for the JavaBeans component model as well.

5. SPECIFY CONSTRAINTS ON REUSE TO ENSURE CORRECT REUSE

In the pursuit of correct component reuse, understanding the behavior of a component and the global requirements of the architecture are very important. However, we feel that a more fundamental and often overlooked issue is understanding the implicit assumptions that constrain a component's reuse.

Other work on this problem, namely ADLs and quality attribute specifications, differs in several aspects. First, ADLs, while providing methods for formally specifying constraints on activation and interaction, focus on describing the entire architecture. Most ADL constraints are specified in terms of the architectural connections between many components, such as interface compatibility. We allow for the specification of very partial architectures: interactions with a single component. This information is carried in the component's specification.

Other component specification languages do allow for the description of properties on a per component basis. However, these properties are mostly used for helping a client better understand how a component behaves, and to evaluate how well a component will fit into an existing architecture, application, or design.

Our approach instead concentrates on specifying the constraints a component makes on its reuse. We summarize our approach as follows:

- We capture usage constraints locally for each component, thus the constraints apply to all uses of the particular component in any architecture.
- These constraints are embedded in each individual component's specification.
- By enforcing these usage constraints at runtime, the specification can describe many more properties than statically verified descriptions. Also, no specification of a virtual client's interaction is necessary.

6. REFERENCES

- [1] Allen, R., Garlan, D. A Formal Basis for Architectural Connection. ACM Trans. Software Engineering and Methodology, Vol. 6, No. 3, July 1997, 213-249.
- [2] Berg, C. Chat Java/CORBA component. <http://www.digitalfocus.com/ddj/code/>.
- [3] Bokowski, B. IPDL - Interaction Protocols for Distributed Objects. Proceedings of the KI-96 Workshop on Agent-Oriented Programming and Distributed Systems. DFKI Document D-96-06. Saarbrücken, 1996.
- [4] Cernuda, A., Labra, J. E., Cueva, J. M. Itacio: A Component Model for Verifying Software at Construction Time. III ICSE Workshop on CBSE. 5-6 June.
- [5] Compare, D., Inverardi, P., Wolf, A. Uncovering Architectural Mismatch in Component Behavior. Science of Computer Programming, Vol. 33 No. 2. 1999, 101-131.
- [6] CORBA 2.3. <http://www.omg.org/cgi-bin/doc?formal/98-12-01>.
- [7] CORBA 3.0: CORBA Component Model. <http://www.omg.org/cgi-bin/doc?orbos/99-07-01.pdf>.
- [8] Enterprise Java Beans 2.0 Specification. <http://java.sun.com/products/ejb/docs.html#specs>.
- [9] Garlan, D., Allen, R., Ockerbloom, J. Architecture Mismatch: Why reuse is so hard. IEEE Software Vol. 12 No. 6. November 1999, 17-26.
- [10] Han, J. An Approach to Software Component Specification. Proceedings of 1999 International Workshop on Component Based Software Engineering. Los Angeles, USA, May 1999.
- [11] Liu, C., Richardson, D. Towards Discovery, Specification, and Verification of Component Usage. ASE 99: Proceedings of the 14th International Conference on Automated Software Engineering, sponsored by IEEE Computer Society, Boca Raton FL. October, 1999.
- [12] Schmidt, H. Trusted Components – Towards Automated Assembly with Predictable Properties. Proceedings of the 4th ICSE Workshop on Component-Based Software Engineering. Toronto, Canada, 2001.
- [13] Shaw, M. Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does. Proceedings of the 8th International Workshop on Software Specification and Design, March 1996.
- [14] Stafford, J., Wallnau, K. Predicting Feature Interactions in Component-Based Systems. Proceedings of the Workshop on Feature Interaction of Composed Systems, Budapest, Hungary, June, 2001.
- [15] 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.