

# Toward Software Plug-and-Play

Francois Bronsard, Douglas Bryan, W. (Voytek) Kozaczynski, Edy S. Liongosari,  
Jim Q. Ning, Ásgeir Ólafsson, and John W. Wetterstrand

Andersen Consulting  
Center for Strategic Technology Research  
3773 Willow Road  
Northbrook, Illinois 60062-6212, U.S.A.  
+1 847 714 2537  
jning@cstar.ac.com

## ABSTRACT

The growing size and complexity of systems has revealed many shortcomings of existing software engineering practices, for example, lack of scalability. This in turn raised interest in component-based and architecture-driven software development. In all likelihood, component-based software will form the foundation on which future systems will be built. The shift toward developing systems from components has been more evolutionary than revolutionary. It has its roots in accepted architectural principles such as layering, modularization, and information hiding. But it also introduces its own principles and concepts and presents new challenges. This paper discusses research ideas and technologies that will facilitate the transition toward component-based software development by leveraging object-oriented middleware technologies such as CORBA and OLE. We also present an innovative component-based development environment to illustrate the ideas we introduce.

## Keywords

Software architectures, components, middleware, formal analysis

## INTRODUCTION

For many years software researchers have been looking for ways of assembling systems from components: the “LEGO block” style of software construction. Such an approach to software building should result in systems of higher quality and lower costs. *Software plug-and-play* also promises more evolvable systems — a result of the capability to easily replace constituent components. Over the last decade component-related research could be found under subjects such as *module interconnection languages* [27], *module interface specification and analysis* [26], *megaprogram-*

Permission to make digital/hard copy of part or all this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.  
SSR '97 MA, USA

© 1997 ACM 0-89791-945-9/97/0005...\$3.50

*ming* [5, 32], *domain-specific software architectures* [17, 9], *software generators* [2], and *architecture description and configuration* [10, 15]. A common theme to all this research is a shift of emphasis from programming-in-the-small to programming-in-the-large and from program development to development of distributed systems. The two key concepts that have emerged are: (i) *components*, large-grain functional units of systems and (ii) *architectures*, blueprints describing system composition.

Component-Based Software Engineering (CBSE) is a research project<sup>1</sup> conducted at Andersen Consulting to develop technologies enabling software plug-and-play. A primary motivation of this work is dealing with rapidly growing complexity and size of business software systems. Today, a large business system is built by a team of more than 150 people over the time of more than a year. Its major components are developed separately or purchased off the shelf. Yet, these individually developed or acquired components must work together. Hence, there is a need for a framework for plug-and-play.

The need for software plug-and-play also arises from the increased flexibility that businesses demand of themselves and consequently, of their software systems. Typical business imperatives like productivity, quality, time-to-market, and the ability to adapt to changes, are well-served by componentized, flexible systems in-house. For example, it may be reasonable for a business to try to out-source its billing while keeping other information systems. However, if its systems are not componentized and are tightly coupled, a reasonable business decision may be inhibited by software inflexibility. The goal of CBSE is to make changes like the one described above simple.

Despite the research efforts mentioned above, the research community has not yet offered practitioners a practical and actionable approach to component-based software build-

---

<sup>1</sup> The work reported in this paper is sponsored in part by the U.S. National Institute of Standards and Technology's Advanced Technology Program on Component-Based Software.

ing. At the same time software designers and developers, feeling the need to address complexity and scale issues, began to develop their own pragmatic solutions. In particular, they started leveraging various forms of *middleware*. Middleware is a layer of software that insulates application software from system software and other technical or proprietary aspects of underlying run-time environments. Among the more advanced middleware solutions are *distributed object middleware*. The two best known object middleware de-facto standards are the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [23] and the Microsoft's OLE [4] with its underlying Component Object Model (COM) [19]. A number of implementations of CORBA are commercially available. Similarly, a workstation version of OLE is available and a distributed version should be available soon.

The use of middleware, especially object middleware, provides the technology underpinning for component-based architectures. However, object models and component models differ. As a result, the object models underlying CORBA and OLE do not contain all the necessary concepts for describing components and their assembly. The plug-and-play vision encounters the "plug-and-pray" reality.

The objective of this paper is to present ideas and technologies central to software plug-and-play. We discuss these ideas in the context of how existing research and technologies, such as OLE and CORBA, can be leveraged to support component-based systems construction. The discussion in the next three sections centers around the following essential attributes of plug-and-play software:

- *Context independence*. Software components should state the assumptions made about their context. In other words, components need not know which other specific components they will interact with or be "hard-wired" with them. We should be free to connect components at assembly time as long as connections have been verified.
- *Location transparency*. A component's implementation should not assume any specific location or configuration. It is essential that we can assemble components in different configurations and assign them to execute in different threads of the same process or separate processes distributed over networks.
- *Heterogeneous connection*. It is not realistic to assume that components must be homogeneous to be plugged together. Heterogeneity of components is caused by their dependencies on the technical architectures of their run-time environments: programming languages, communication protocols, hardware platforms and operating systems. We should support composition of heterogeneous components.

- *Evolvability*. A key premise of plug-and-play is that systems will evolve easily. Thus, we should allow components to be easily added, removed, or replaced after the deployment of a component-based system.

The penultimate section introduces a prototype Architecture Design Environment (ADE) that we have developed at Andersen Consulting's research center. We use a scenario to illustrate the features of ADE and demonstrate our ideas. The last section summarizes the paper and presents some of the anticipated directions of the plug-and-play research.

## COMPONENT MODEL

This section introduces our component model and explains how it addresses some of the software plug-and-play attributes. The model is a set of system architecture modeling abstractions. It includes a set of modeling concepts, their respective semantics, and rules of how to combine the concepts into system architecture specifications. The model is accompanied by an Architecture Specification Language (ASL) which the CBSE project developed to support component-based architecture specification. While the model has been fully developed, we cannot present it here in its entirety. We focus only on the elements most relevant to component-based software: *components, component interfaces, bindings, and configurations*.

### Components, Interfaces and Bindings

A component is a significant functional unit of a system. Components can be anything from a high-level business function like "customer order processing" to a low-level technical task such as "roll back transaction." A component can be either *atomic* or *composite*. A composite component contains other components as its sub-components, which are themselves atomic or composite. Therefore, the architecture of a complete system can be described as a hierarchical organization of atomic and composite components. Moreover, a component is a reusable and sharable unit; that is, it can be handed from one project to another.

The functionality of a component is defined by its interfaces. In our model, a component has one or more *provided* and *required* interfaces. Syntactically, interfaces are similar to object class descriptions: they contain attributes and operations. Semantically, provided interfaces specify those services or capabilities that a component offers to other components. Conversely, the required interfaces specify those services that a component must receive from other components in order to carry out its own responsibilities.

The concept of required interfaces is essential to enabling software plug-and-play. They expose operation invocations that a component makes to other components. Such invocations are typically embedded in a component's implementation and are not easily identified. Required interfaces explicitly specify the server prerequisites of a com-

```

interface ImprovedPolicy : Policy {
  states { stInitial(init), stAuthorized };
  void Authorize(in string name, in string password) raises (Unauthorized)
    precondition { stInitial; };
    postcondition { stAuthorized; };
  void IncreasePremium(in float percent)
    precondition { percent > 0.0; stAuthorized; };
};

```

**Figure 1.** Interface specification

ponent<sup>2</sup>. A component does not have to be “hard-wired” to a specific component that provides the required services. Instead, it is possible to attach it to any components offering those matching services.

The interface in our model is a strict extension of the CORBA IDL interface [23]. As in IDL, it supports features such as operations, attributes, exceptions, multiple inheritance and name spaces. However, our interface also has important additional semantic-oriented features of *pre- and post-conditions*, *invariants*, and *states*.

Figure 1 gives an ASL example of an interface specification. This specification defines the ImprovedPolicy interface that inherits from the Policy interface and adds the operations Authorize and IncreasePremium.

The precondition for IncreasePremium requires that the input parameter percent be greater than zero and the state be stAuthorized. (The operation Authorize sets the state.) States are used to specify the legal sequences of operation invocations on the interface, i.e., they specify the *protocol* of the interface. It is also worth pointing out that the inheritance between the two interfaces is *interface inheritance*. Unlike inheritance in a typical object-oriented programming language such as C++, the implementation of Policy is not inherited by ImprovedPolicy. Problems with implementation inheritance in distributed programming have been sufficiently explored elsewhere [18]. The left-hand side of Figure 2 illustrates how the interface given in Figure 1 is assigned to two components as required and provided interface respectively.

As mentioned earlier, components can be connected together to form *composite components*. A connection is realized by *binding* a required interface of one component with a provided interface of another component. This is illustrated by the right-hand side of Figure 2.

The ability to express component bindings explicitly is essential to modeling of large software systems. A component composition represents the *architecture* of a system. One use of system architectures is the optimal allocation of

computing resources. For example, if high bandwidth communication is required between two components, we might choose to put them on the same network segment, same machine, or even in the same process space.

Representing component interconnections at the interface-level, rather than at the port- or operation-level [15, 27], reduces the complexity of modeling system architectures. Interfaces group related attributes and operations together. In our model one connection between a required and a provided interface represents many connections at the operation-level.

### Configurations

In general, configuration specifications provide the information necessary to integrate heterogeneous component instances into an executable distributed system. The issue we address with configurations is, that component implementations have dependencies on their development (programming languages, object model, etc.) and execution (platforms, middleware, etc.) environments. We call these dependencies component *configuration properties*. These properties are specified in ASL as shown in Figure 3.

In the example, the configuration of the Client component specifies language (C++) and operating system (Windows NT) properties. CPPApplication then states that for the clientInstance of the Application component, the CPPClient configuration is used. CPPApplication also declares that this particular configuration of the Application component may be accessed via a naming service of the underlying communication middleware (e.g., CORBA’s naming service) using the name Application. There are other properties that can be specified such as host machines and middleware requirements. Default values are assumed for necessary properties that are not specified. Multiple configurations may be specified for a single component and used in system assembly.

<sup>2</sup> Informally, a server is a component with a provided interface, and a client is a component with a required interface.

<pre> <b>component</b> Client {   <b>requires</b> ImprovedPolicy requiredPolicy; }; <b>component</b> Server {   <b>provides</b> ImprovedPolicy providedPolicy; } </pre>	<pre> <b>component</b> Application {   Client clientInstance;   Server serverInstance;   <b>bind</b> clientInstance.requiredPolicy   <b>to</b> serverInstance.providedPolicy; }; </pre>
---	---

**Figure 2.** Interface assignment and component bindings

### SPECIFICATION ANALYSIS

Before two components can be connected, or *bound* together, it must be shown that the required interface of the client component is *satisfied* by the provided interface of the server. The objective of specification analysis is to prove that such is true and assist in binding components that have interface mismatches.

In the simplest case, the provided and required interfaces are instances of the same interface, so they match exactly. Another simple case is when the provided interface specification inherits from the required interface specification. However, to truly address the context independence and evolvability properties of plug-and-play software, we cannot restrict ourselves to these simple cases alone. Instead, we must be able to analyze two arbitrary interfaces to establish whether one of them, as a provided interface, satisfies the specification of the other, as a required interface. In this section we describe *logical subtyping* which we introduce to analyze if two interfaces are bindable. For more detailed discussions on the comparative merit of inheritance versus analysis, we refer the reader to [11, 13, 21].

In cases where analysis fails, we may still be able to connect two interfaces by inserting an *adapter* component between them. An adapter supplies the exact required interface needed, using the functionality offered by the provided interface. Most importantly, an adapter *template*, a partially complete adapter that can be completed by system developers, can be generated automatically. This section also discusses generation of adapter templates.

#### Logical Subtyping

We say that a specification  $S_j$  is a *logical subtype* of specification  $S_k$  if the objects satisfying  $S_j$  also satisfy  $S_k$ . Objects that belong to a logical subtype  $S_j$  can be *substituted* for objects satisfying  $S_k$  *transparently*. For example, consider an operation that requires a structure with two fields, date and name, of type string. Logically, this operation should work when given a structure with three fields, date,

name, and ld, of type string. Although the second structure has more fields, it has the two that are expected. Thus, the replacement of the first structure by the second structure is transparent — the second structure is a logical subtype of the first. Similarly, a caller of an operation whose precondition requires a parameter to be an integer greater than 1 will be satisfied by an operation whose precondition requires that the parameter be greater than 0.

Our definition of *logical subtyping* combines *structural subtyping* [8, 12] and *behavioral subtyping* [14]. We also took inspiration from the concept of *design by contract* [16] and from work on *specification matching* [33]. The major elements of the structural subtyping relation, written  $\leq_s$ , are:

- Subtyping relation over basic types which includes relationships such as short  $\leq_s$  long and float  $\leq_s$  double.
- An extension over structures and interfaces is defined recursively as follows: structure  $A$  is a subtype of structure  $B$  (written  $A \leq_s B$ ), if for any field  $a$  occurring in  $B$  with a type  $T$ , there is also a field named  $a$  occurring in  $A$  whose type is a subtype of  $T$ . The case for interfaces is similar but quantification is over operations. (Interface attributes are treated as their corresponding *set* and *get* operations.) To handle recursive types, we assume that  $A$  is a subtype of  $B$  when making the subtyping check over the fields (or operations) of  $B$ .
- An extension over operations is also defined recursively: operation  $f$  is a subtype of operation  $g$  if all input parameters of  $f$  are super-types of the corresponding input parameters of  $g$ , and all output parameters as well as the return type of  $g$  are super-types of the corresponding parameters and the return type of  $f$ . In the above definition, “corresponding” means that the parameters have the same name or else, they are at the same position in the parameter list.

<pre> <b>configuration</b> CPPClient of Client {   <b>property</b> OS NT;   <b>property</b> language CPP; }; </pre>	<pre> <b>configuration</b> CPPApplication of Application {   CPPClient clientInstance;   <b>property</b> name Application; }; </pre>
---	--

**Figure 3.** Configuration specifications

```

interface Policy1 {
    states {Unauthorized(init), Authorized};
    void Authorize(in string agent, in string password)
        postcondition {Authorized;};
    void ChangePremium(in string client_name, in long year , in double percent)
        precondition {Authorized;};
};
interface Policy2 {
    void ChangePremium(in string password,
        in string client_name, in short year, in float percent);
};

```

**Figure 4.** Policy1 is a structural subtype of Policy2

The structural subtyping relation is a simple and efficient mechanism. It provides the flexibility needed to allow binding of interfaces of independently developed components. The provided interface, or a subset of it, must still be a close match with the required interface, but it doesn't have to be an exact match. In particular it does not have to be the same as or inherit the required interface specification. An example of structural subtyping is given in Figure 4. In this example  $\text{Policy1} \leq_s \text{Policy2}$ .

Structural subtyping, however, has limitations. It requires the names of operations in the provided and required interfaces to be identical. Similarly, the names of fields in the structures used in these interfaces must be identical. This is a strong restriction. It is easy to imagine that different developers will use different names for equivalent operations, arguments and results. Thus, a future area of work would be to define appropriate generalization mechanisms to lessen our dependency on name matching.

On the other hand, one can argue, that structural subtyping is too permissible and may allow one to bind interfaces that use identical names for different meanings. This could become troublesome with a generalized name-matching mechanism. To address this issue, we use a more constrained *behavioral subtyping* relation. Behavioral subtyping uses the structural subtyping as a base, but in addition uses semantic information such as preconditions and postconditions of operations. Specifically, the following constraints are added:

- Operation  $f$  is a subtype of operation  $g$  only if the precondition of  $g$  implies the precondition of  $f$ , and the postcondition of  $f$  together with the precondition of  $g$  imply the postcondition of  $g$ .
- Interface  $A$  is a subtype of interface  $B$  only if the invariants of  $A$  are equivalent to the invariants of  $B$ , and the *protocol* of  $B$  is a subset of the *protocol* of  $A$ ; i.e., the sequences of operations accepted by  $B$  include the sequences of operations accepted by  $A$ .

For instance, a behavioral subtyping check for the interfaces in Figure 4 would reveal the inconsistencies between the protocols of the two interfaces.

One important difference between structural subtyping and behavioral subtyping is that the latter is not a decidable relation. Establishing logical implications as required in the constraints above is an undecidable problem in general. Nevertheless, approximate mechanisms exist and are often sufficient in practice. In our analysis tool, we use the automated reasoning capability of the OTTER system [31] with some customizations.

#### Adapters

Adapters are needed in two cases. The first case is when one component provides an interface whose specification is a subtype of another component's required interface, yet we still cannot allow the two components to communicate directly. For example, an operation that expects its first parameter to be a floating-point number and its second parameter to be an integer may be a subtype of an operation where the first parameter is an integer and the second a floating-point number. However, before we invoke either operation we need to get the parameters in the order assumed by the operation's implementation. In such cases our analysis tool automatically generates *adapters* that enable components to communicate. An *adapter* is a component that provides an interface identical to the required interface, and is implemented using the provided interface. For two interfaces that are in a logical subtyping relationship, the role of the adapter is merely to forward the operation calls to the provided interface, and return the results, after realizing whatever structural transformation is necessary to make the operation call well-formed.

The second case is when the subtyping check fails, but whatever partial success it had provides valuable information on how to adapt the provided interface to satisfy the required interface's specification. Our analysis tool uses this information to construct adapter templates. This is an important side-effect of our tool. Even when it fails to validate the binding of interfaces, it can partially build an adapter for these interfaces. Suppose, that the subtyping

check fails, but it reveals that most, let's say nine out of ten, of the operations of the required interface are supplied by the provided interface. An adapter template generated by the tool will support those nine required operations. It is a simple matter to recognize for which operations and attributes the subtyping check failed. For these operations, and only these, we ask developers to provide the appropriate code. In other words, a template will implement the required operations using the provided interface when possible, and for operations for which the subtyping check failed, it will rely on developers to fill-in implementations.

In summary, we establish the validity of binding two interfaces using structural and behavioral analysis of their specifications instead of using merely a class-inspired inheritance relationship. This gives us the flexibility needed to allow components to be developed or maintained independently. Further, our analysis tool is used to completely or partially build adapters for interfaces that cannot be bound directly.

### COMPONENT INTEGRATION

Component integration addresses two issues: (i) packaging components so that they can be connected at run-time, and (ii) connecting, disconnecting and re-connecting components at run-time. This section describes our approach to handling these issues and explains how that supports software plug-and-play.

Components must be "packaged" into executables in such a way that they can inter-operate with the other components to which they are connected [7]. For example, packaging a component for CORBA requires a *skeleton* and a *stub*. The skeleton connects the component's application logic to the middleware, while the stub connects component clients to the middleware and thus, indirectly, to the skeleton. We have built a tool, called the *Packager*, which determines how to package components. Simply put, the Packager:

1. analyzes a component-based architecture specification written in ASL to determine if consistent packages are feasible,
2. selects a *manufacturing plan* for generating the packages based on what is feasible, and
3. outputs instructions for executing the plan.

That which is "feasible" is changing rapidly. For example, Andersen Consulting as a technology integrator builds systems using a variety of market solutions. Although tools like C++, COM/OLE [19], and CORBA are becoming common practice, new ones like Java Beans [30], D-COM [3] and ActiveX [20] soon follow. Our approach to the rapidly changing technology problem is to treat it as a given, rather than attempt to design a new implementation language or middleware layer. Thus, a major requirement is that the Packager be easy to modify.

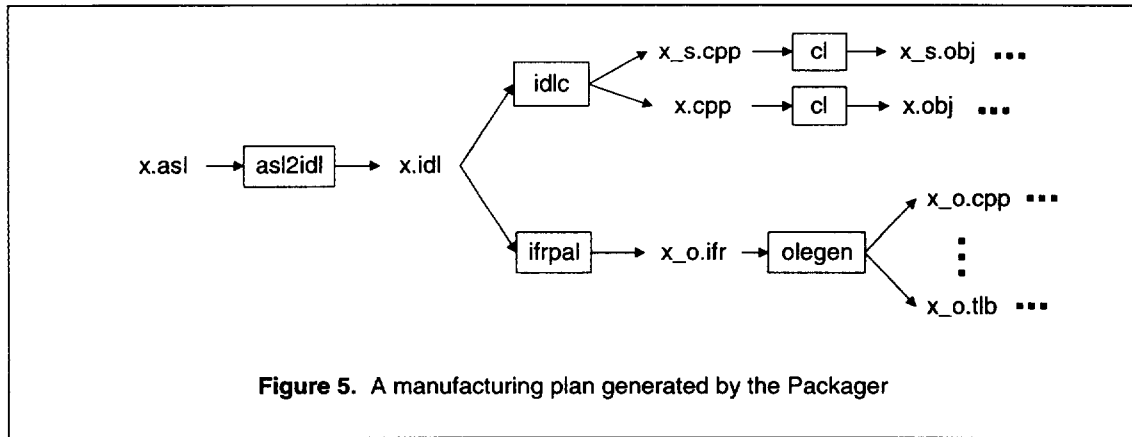
Input to the Packager is a set of configuration specifications and the bindings between the corresponding components. Configuration specifications, described earlier, define properties of component implementations, and thus constrain how components may be packaged. Bindings further constrain packages since bound components must inter-operate.

Not all packages are feasible. For example, executing a Visual Basic component under Windows NT on a Sun UltraSPARC machine is not. Similarly, connecting CORBA and OLE components has limitations, depending on which CORBA and OLE products one decides to use. The configuration properties that affect our Packager the most are (1) programming language, (2) object model (e.g., CORBA, COM, and OLE automation), and (3) middleware. Architects may also set operating system and machine type (e.g., Intel 386 or SPARC) properties. Of course, all these properties are related since, for example, not all programming languages are available for all middleware tools.

Usually there are also many feasible manufacturing plans for an architecture. Differences between plans can range from low-level details like the number of object files generated, to high-level concerns such as the number of workstations needed to deploy a system. When these differences are multiplied, the number of feasible manufacturing plans grows rapidly. Indeed, combinatorial explosions often occur. Picking one plan over another is often subjective, or based on project-specific criteria. In most cases, there is no one "best" plan.

Given the combinatorics of finding packages, the fact that there is no optimal solution for most architectures, and the ever-changing tools market, we built the knowledge of our Packager using an expert system rule-base. Some of the rules define what is feasible by defining *manufacturing capabilities*. A manufacturing capability describes one tool (or one aspect of a tool) by constraining its inputs and defining its outputs. For example, a C++ compiler takes any number of C++ source files and generates object files, and a linker for C and C++ takes any number of object files, containing one and only one "main" function, and generates an executable. Other rules in the rule-base define user or project preferences, such as the preferred programming language, or whether to minimize the number of component implementations per executable. The expert system searches for manufacturing plans by repeatedly applying manufacturing capabilities, and picks one plan if multiple plans are possible. In effect, the Packager orchestrates the application of tools.

An example of a manufacturing plan is partially shown in Figure 5. This manufacturing plan is for an architecture containing a CORBA-based C++ server and an OLE-based Visual Basic client. The boxes represent tool invocations and the labels represent files. `asl2idl` is a tool that we de-



veloped that maps ASL specifications to CORBA IDL. `idlc` is a commercially available tool that maps IDL to C++ code. `cl` is a C++ compiler. `ifrpal` creates interface repository files, and `olegen` generates OLE type libraries (tlb files) and OLE proxies for CORBA objects. Currently we output manufacturing plans in the form of make-files.

Our rule-based approach supports context independence by automating the search for a manufacturing plan. For example, suppose two components are initially implemented in C++ and share a process. Later, we replace one of them with a Smalltalk component. If Smalltalk and C++ cannot share a process, we simply derive a new manufacturing plan that creates separate processes. Location transparency and heterogeneity are supported by utilizing middleware that provides these features.

As new, commercial tools become available, their manufacturing capabilities are added to the rule-base. Occasionally we do need to build some custom tools (e.g., `asl2idl`) to bridge gaps between commercial tools. By using a rule-base to derive manufacturing plans, we can build small tools that do simple things well. The rule-base then orchestrates the new tools, together with others tools, to generate manufacturing plans that were not previously feasible.<sup>3</sup>

The second aspect of component integration is run-time integration. Our run-time requirements mainly have to do with binding the provided and required interfaces of components. Components that have required interfaces need to obtain an object that implements the interface. Similarly, components that provide an interface need to make it available. In simple, static systems, bindings may occur at start-up time as component instances are created. To replace a component or extend the architecture, the system is simply brought down, reconfigured, and restarted.

<sup>3</sup> It is Andersen Consulting's experience that mid-sized expert systems themselves are quite maintainable.

Many systems are not so static and shutting down a widely distributed, mission critical system may be a formidable task. Also, in many client-server architectures the significance of creating a component instance varies greatly depending on the component. For example, a client instance might be created on each workstation as users log-in each morning, or clients might be short-lived and started by a World Wide Web browser. While small, stateless servers may be also short lived and created on demand, large, data-rich servers often are only started or stopped manually, due to the large number of users affected by such an event.

To support extensibility, components may be replaced and the architecture may be extended while a system is running. Our approach to this is a simple and pragmatic one. Components (clients) requiring an interface call a middleware service to locate servers. Clients maintain a reference to servers as long as they like. Servers may be safely replaced when no client has a reference to it. In this approach, we simply allow the client to decide when to look-up the server (and thus potentially obtain a newer revision) and when to release it. We do not require all clients to use some, possibly expensive, locking mechanisms such as a transaction monitor. The implementation of this run-time service will vary with specific products. For example, OMG's factory-finders [24] and trader services [25] were designed for this purpose. Commercial implementations of these services are becoming available.

#### ARCHITECTURE DESIGN ENVIRONMENT: PULLING EVERYTHING TOGETHER

The previous sections introduced some of the key concepts and technologies we have been working on and discussed why they are important to plug-and-play software. This section attempts to pull all the material together by using a scenario to describe the Architecture Design Environment (ADE) developed by the CBSE project team.

The primary user of the ADE is a *system architect*, whose main responsibility is to understand the functional and technical requirements of the system and develop a system architecture that meets these requirements. We will use a scenario consistent with the ASL examples given in second section. The architect wishes to design a simple client-

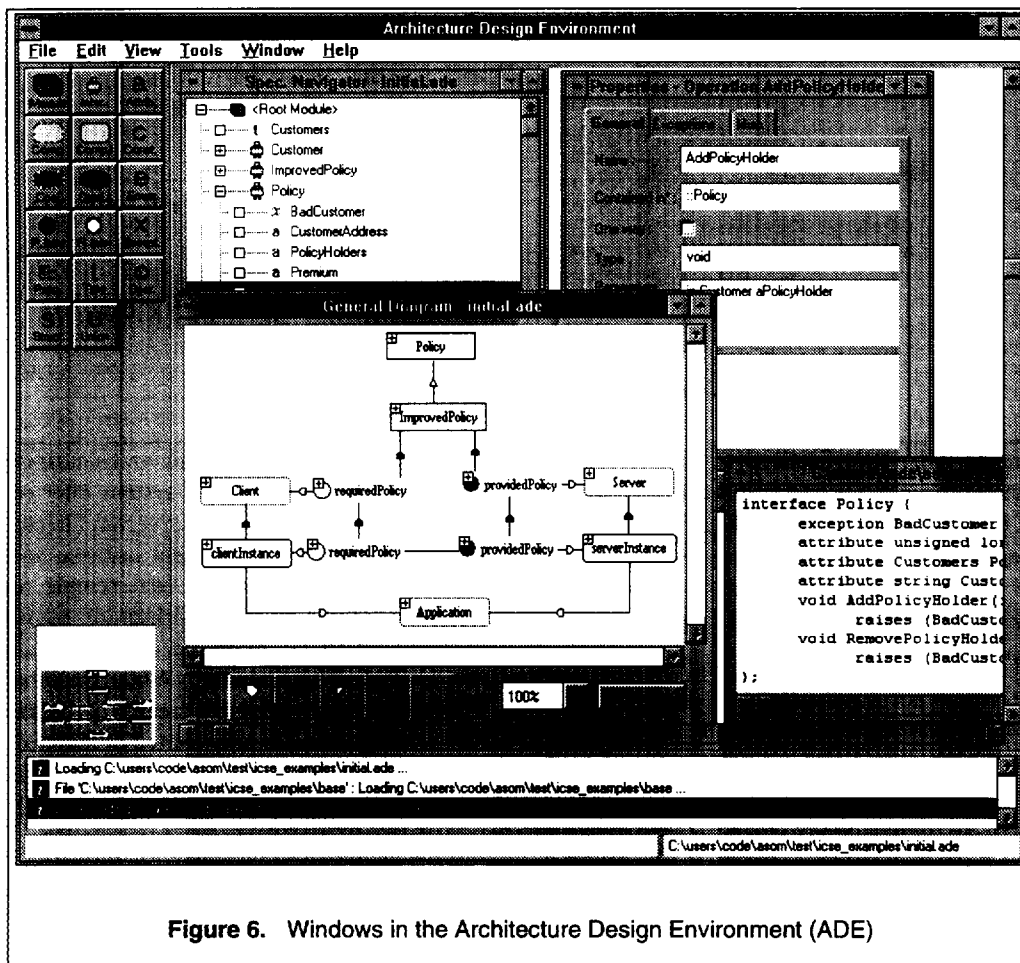


Figure 6. Windows in the Architecture Design Environment (ADE)

server system for an insurance company. The system consists of three components which we will refer to as **Client**, **Server** and **Application**. The **Client** requires an interface that provides information on insurance policies and the **Server** provides one. The **Client** and **Server** are atomic components. The **Application** is a composite component containing an instance of **Client** and an instance of **Server**, and it binds the two component instances together.

Once the ADE has been started, the architect has two options: (1) use a text editor to write an ASL specification, or (2) enter it interactively through the ADE's GUI front-end using its icons, buttons and drag-and-drop features. It is possible to mix both modes if desired. Let us assume that there is a standard **Policy** interface available which has been specified using OMG IDL<sup>4</sup>. As IDL is a subset of ASL, the architect can import the IDL file containing the **Policy** specification.

Now that the architect has acquired the **Policy** interface she may use it as she sees fit. Realizing that the interface does not completely satisfy the system's requirements, she ex-

tends the interface using inheritance by selecting the interface button in ADE and entering the relevant information, such as name (**ImprovedPolicy**) and the inheritance relationship with **Policy**. She then proceeds to add an **Authorize** and an **IncreasePremium** operation using drag and drop gestures supported by the environment. She also adds information on states, preconditions and postconditions, resulting in the specification shown in Figure 1.

Figure 6 shows ADE at the point where the architect has created the components, component instances and the necessary bindings. The window in the upper-left corner is called a **Specification Navigator**. In this window, the architect can create and browse specification objects such as interfaces, components, and their instances. A directory tree metaphor is used to help the architect navigate through the specification structure. Every specification object has an associated **Property Window** (see upper-right window) that can be used to view or edit information. The text window partially shown below the **Property Window** contains the IDL representation of the **Policy** interface. While the **Specification Navigator** window is useful to quickly traverse a specification, it is limited in that it only shows containment relationships. The **General Diagram** window (under the **Specification Navigator** window) does not have this limitation, as it can show all the relationships between

<sup>4</sup> This is not an unreasonable assumption given current standardization efforts for Business Objects within the OMG.

specification objects of interest. In order to avoid a cluttered diagram, the architect may use the Filter button to select the types of objects and relationships to be shown. The General Diagram in Figure 6 shows five object types: *interfaces* (*Policy* and *ImprovedPolicy*), *provided interfaces* (two instances of *providedPolicy*), *required interfaces* (two instances of *requiredPolicy*), *components* (composite component *Application*, atomic components *Client* and *Server*), and *component instances* (*clientInstance* and *serverInstance*). It also shows four types of relationships: *inheritance* (triangle), *containment* (polygon), *instantiation* (filled polygon), and *binding* (the straight line between the required and provided interfaces of *clientInstance* and *serverInstance*). At the bottom of the ADE screen there is an information window showing a history of user-triggered operations and informational messages.

Now that the architect has described the architecture, she is ready to make configuration decisions to be used by the Packager. In this example, she uses ADE to specify a few properties related to implementation language, naming and operating system. An ASL representation of these configurations was given in Figure 3. Finally the developer invokes the Packager which generates the necessary platform- and middleware-specific files. These include: OMG IDL files, C++ files and makefiles. However, the application logic of the components is still missing as the architect is not reusing any existing components in this scenario. The team responsible for component implementation should insert the necessary code into the code skeletons generated by the Packager.

To continue the scenario, suppose that a new component supporting the *Policy* interface is introduced to the market. While it does not provide the *ImprovedPolicy* interface the architect would prefer, it does perform much better under heavy loads and using it would significantly increase system throughput. ADE's interface analysis tool goes a long way toward assisting the system architect in this situation. The architect can import the new component, *NewServer*, create a new instance of it within the *Application* component and then let the analysis tool compare the required interface of *clientInstance* to the provided interface of *newServerInstance*. An adapter component and a partial implementation of it would be automatically generated.

## RELATED WORK AND SUMMARY

Naturally, CBSE is not the only project studying technologies that enable software plug-and-play. In previous sections, we mentioned a number of related projects. In summary, the projects that are most influential to our work are the following. Some work focuses on the runtime environments necessary to support and connect components distributed throughout a network. These include Polyolith [28] and Regis [15]. These projects also involve research

on advanced configuration languages (MIL in Polyolith and Darwin in Regis) to complement the runtime systems. These configuration languages allow the dynamic instantiation and movement of components throughout a network, as well as specifying the connections between different components.

Another area of research in component integration is *software packaging*. The most representative work in this area is the software packager developed at the University of Maryland by Callahan and Purtilo [6, 7]. (Purtilo is also a consulting member of the CBSE project team.) This packager automatically determines how to integrate a diverse collection of software components based on their types and the capabilities of integration tools available in the environment.

The *architecture* of software systems is another related area of research. Projects in this area include UniCon [29], and Wright [1]. These projects study the various types or roles that components and bindings can have in the architecture of a system. In particular, a formal framework for specifying component interconnections is defined for Wright. The key idea of their approach is a definition of architectural connectors in terms of a collection of protocols that characterize participant's roles in an interaction. They also show how interconnection compatibility can be checked based on semantic information.

While the concepts related to software plug-and-play have been under research for many years, the need for real plug-and-play capability is just starting to be felt in the marketplace. The tools supporting such capabilities have yet to emerge and mature. On the other hand, there are already technologies, especially middleware, that are widely accepted by the software development practice and can potentially be used to address certain aspects of component-based development.

The focus of this paper has been on how to leverage and build upon capabilities offered by object middleware such as CORBA and OLE. In particular, the paper focused on three technology pieces:

1. A component-based architecture model. Central to this model are the ideas of provided and required interfaces, binding or interconnection of components, and system configuration. An architecture specification language (ASL), which extends CORBA IDL and uses OLE/COM's multiple interface concept, has been developed to support this model.
2. A component specification analysis tool. Architecture design decisions need to be verified before they are committed into implementation. We developed a tool that checks the compatibility of interfaces between components and generates adapters to bridge incompatible interfaces.

3. A component integration tool (the Packager). This tool offers capabilities to transform abstract component-based architecture specifications into executable code that can run on distributed communication infrastructures such as CORBA or OLE. The automation support provided by the Packager reduces skill requirements in building distributed and heterogeneous systems from components.

We also described ADE, a tool that ties the above technology pieces together under a graphical, interactive, and user-oriented environment. Prototype versions of the tools have been completed. We are currently in the process of testing and enhancing them, and verifying our design assumptions.

CBSE research and technology development is still in an infancy stage. We recognize that there are a number of additional issues that must be addressed before software plug-and-play is ready to make a real, practical impact. Our future research will primarily focus on the following two areas:

- *Process support.* CBSE promotes a new paradigm for building software systems. It emphasizes explicit representation of many new concepts such as interfaces, components, and architectures. Conventional software development models, methodologies, and processes will have to be enhanced to fit this new paradigm. For example, a developer would need process guidance in recognizing components from requirements, just as one has to know how to identify classes and objects in an object-oriented analysis phase. A more detailed discussion on CBSE process issues can be found in [22].
- *Dealing with other architectural models and styles.* The component model described in this paper is most appropriate for developing static configuration-based, client-server systems. Our initial experience with systems that Andersen built for our telecommunication clients suggests that it is not always realistic to determine, and therefore specify, bindings between components at system construction-time. In many cases, bindings can only be resolved dynamically at run-time based on certain run-time constraints and events. Further, the emergence of new architecture and configuration styles such as mobile computing and Internet-based computing pose new challenges.

## REFERENCES

1. Allen, R., and Garlan, D., "Formalizing Architectural Connection," in *Proceedings of 16th International Conference on Software Engineering*, IEEE Computer Society Press, pp. 71–80, May 1994.
2. Batory, D., and Geraci, B. J., "Validating Component Composition in Software System Generators," in *Proceedings of the Fourth International Conference on Software Reuse*, IEEE Computer Society Press, April 1996.
3. Box, D., "Introducing Distributed COM and the New OLE Features in Windows NT 4.0," *Microsoft Systems Journal*, pp. 19–38, May 1996.
4. Brockschmidt, K., *Inside OLE*, 2nd ed., Microsoft Press, Redmond, Washington, 1995.
5. Boehm, B., and Scherlis, B., "Megaprogramming," in *Proceedings of the DARPA Software Technology Conference*, 1992.
6. Callahan, J. R., "Software Packaging," Ph.D. thesis, Technical Report CS-TR-3093, Univ. of Maryland, 1993.
7. Callahan, J., and Purtilo, J., "A packaging system for heterogeneous execution environments," *IEEE Transactions on Software Engineering*, 17(6):626–35, June 1991.
8. Cardelli, L., and Wegner, P., "On understanding types, data abstraction, and polymorphism," *ACM Computing Surveys*, 17(4):471–522, December 1985.
9. Fischer, G., "Domain-Oriented Design Environments," *Automated Software Engineering*, Johnson, L., and Finkelstein, A., eds., Kluwer Academic Publishers, Vol. 1, No. 2, June 1994.
10. Garlan, D., and Perry, D. E., "Introduction to the Special Issue on Software Architecture," *IEEE Transactions on Software Engineering*, 21(4):269–74, April 1995.
11. Hölzle, Urs, "Integrating independently-developed components in object-oriented languages," in *Proceedings of the European Conference on Object-Oriented Programming*, Springer-Verlag, July 1993.
12. Katiyar, D., Luckham, D., and Mitchell, J., "A type system for prototyping languages," in *21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, January 1994.
13. Linton, M., Vlissides, J. and Calder, P., "Composing user interfaces with InterView," *IEEE Computer*, 22(2):8–22, February 1989.
14. Liskov, B. H., and Wing, J. M., "A behavioral notion of subtyping," *ACM Transactions on Programming Languages and Systems*, 16(6):1811–41, November 1994.
15. Magee, J., Dulay, N., and Kramer, J., "A Constructive Development Environment for Parallel and Distributed Programs," in *Proceedings of the Second International Workshop on Configurable Distributed Systems*, March 1994.

16. Meyer, B. *Object-Oriented Software Construction*. Prentice-Hall, 1988.
17. Mettala, E., and Graham, M. H., eds., "The Domain-Specific Software Architecture Program," Technical Report CMU/SEI-92-SR-9, Carnegie Mellon University, June 1992.
18. Microsoft Corporation, "Object Linking and Embedding (OLE) Today and Tomorrow," *Technology Overview*, November 1993.
19. Microsoft Corp., "The Component Object Model: Technical Overview," *Dr. Dobbs Journal*, December 1994. Also as [http://www.microsoft.com/oledev/olecom/com\\_modl.htm](http://www.microsoft.com/oledev/olecom/com_modl.htm).
20. Microsoft Corp., *ActiveX Resources Area*, <http://www.microsoft.com/activex/>, 1996.
21. Muckelbauer, P. A., and Russo, V. F., "Structural Subtyping in a Distributed Object System," Technical Report, Purdue Dept. of Computer Sciences, <http://www.cs.purdue.edu/Renaissance/LF-overview.ps>, Purdue University, 1996.
22. Ning, J. Q., "A Component-Based Software Development Model," in *Proceedings of Twentieth Annual International Computer Software and Applications Conference*, August 1996.
23. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, July 1995. Also as <http://www.omg.org/corbask.htm>.
24. Object Management Group, *CORBA Services: Common Object Services Specification*, document number 95-3-31, March 1995.
25. Object Management Group, *OMG RFP5 Submission, Trading Object Service*, document orbos/960506, May 1996.
26. Perry, D. E., "The Inscape Environment," in *Proceedings of the 11th International Conference on Software Engineering*, IEEE Computer Society Press, May 1989.
27. Prieto-Diaz, R., and Neighbors, J. M., "Module Interconnection Languages," *Journal of Systems and Software*, 6:307-334, 1986.
28. Purtilo, J., "The Polyolith software bus," *ACM Transactions on Programming Languages and Systems*, 16(1):151-74, January 1994.
29. Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G., "Abstractions for Software Architecture and Tools to Support Them," *IEEE Transactions on Software Engineering*, 21(4):314-35, April 1995.
30. Sun Microsystems, Inc., *Java Beans: A Component Architecture for Java*, <http://splash.javasoft.com/beans/WhitePaper.html>, July 1996.
31. Wos, L., Overbeek, R., Lusk E., and Boyle J., *Automated Reasoning: Introduction and Applications*, McGraw-Hill, 2nd ed., 1992.
32. Wiederhold, G., Wegner, P., and Ceri, S., "Toward Megaprogramming," *Communications of the ACM*, November 1992.
33. Zaremski, A. M., and Wing, J. M., "Specification Matching of Software Components," Technical Report CMU-CS-95-127, School of Computer Science, Carnegie Mellon University, March 1995.