Architecture Specification Support for Component Integration

W. (Voytek) Kozaczynski, Edy S. Liongosari, Jim Q. Ning, and Ásgeir Ólafsson

Center for Strategic Technology Research, Andersen Consulting
100 South Wacker Drive, Chicago, Illinois 60606, U.S.A.
email: {voytek, edy, jning, olafsson}@andersen.com

Abstract

This paper describes an approach to automating the construction of software systems from components. We illustrate how integration-related concerns such as component interfacing, interconnection, distribution, and configuration, can be modeled with a specification language. We also show how a graphics-based design environment can be used to support visual specification and transformation of integration specifications into implementations. This approach raises the level of architecture specifications to assist the currently labor-intensive and error-prone process of system integration.

Keywords: architectures, system integration, software buses, component-based reuse

1. Introduction

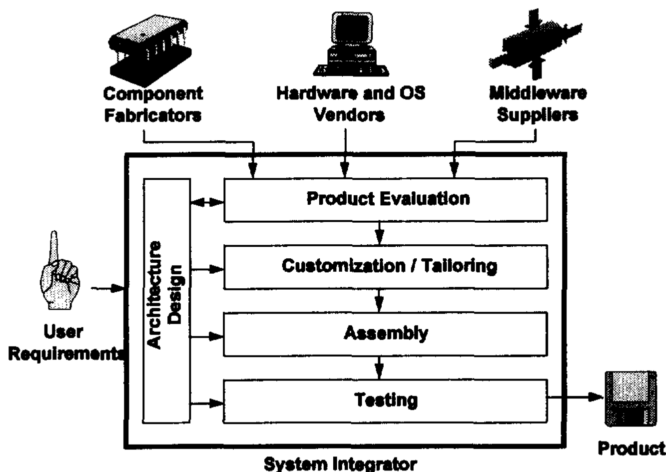

Fig. 1 The CBSE Process

This paper concerns Component-Based Software Engineering (CBSE), the process of building software systems from reusable parts. In particular, we discuss an approach that Andersen's Center for Strategic Technology Research (CSTaR) has developed recently to address component integration, a critical issue in CBSE.

General interest in constructing software systems from components has been growing as we have come to realize that software engineers would benefit from the general availability of standard parts. This is, of course, not a new idea as most mature engineering disciplines rely heavily on the existence of reusable components. By using such components, these disciplines have substantially reduced time to market, increased productivity, lowered development cost, and improved product quality, goals that the software engineering industry has always been striving for.

A high-level view of the CBSE process is depicted in Figure 1. In a component-based software industry, software development is mainly a system integration job. According to user requirements, a system integrator first comes up with a high-level architecture design. Based on the initial design, he/she evaluates and selects pre-fabricated components, an execution platform (hardware, operating system, etc.), and a communication layer (*middleware*). In the process, the architecture design is refined. The system integrator then customizes and tailors the reused components, assembles the components together via the use of the middleware layer on top of the hardware platform to generate a new system, test the system, and finally package and deliver the product.

While the process seems intuitively obvious, a number of problems faces the software engineers attempting to apply it in practice. The central problem is that interconnections among software components are much more complex than those between physical components. Coupling between

1066-1387/95 $4.00 © 1995 IEEE

different parts of a software system can be subtle and dynamic. The situation is further complicated by the fact that many systems must integrate components of different technical generations that were not designed to work together -- the heterogeneity and legacy problems.

2. Approach and Scope of Work

Our approach to solving the component interconnection puzzle is to leverage and extend the existing and emerging research ideas and technologies as much as possible. In the marketplace new products and standards are emerging which may very well lay the foundation for future component standards. At the same time, various research projects are attempting to look ahead and explore new technologies to make the CBSE a reality. In the following two sub-sections we describe the current and imminent state of the market and some of the technologies investigated by academic and industrial researchers.

2.1. State of the Market

There is a number of efforts aiming at standardizing component interfaces and services. Many of these efforts are influenced by object oriented[1] principles and specify how objects and services can be made accessible by other objects. The two best known emerging standards are the Common Object Request Broker Architecture Specification (CORBA, [OMG91]) specified by the Object Management Group[2] (OMG) and the Component Object Model (COM) specified by Microsoft as part of OLE 2 [Microsoft93]. A number of different implementations of CORBA are (or will soon be) available. These include: XShell from Expersoft [Expersoft94], SOMobjects from IBM [IBM93], DOE from Sun [SunSoft93], and ObjectBroker from DEC [DEC93].

OMG and Microsoft also specify separate standards for *object services*. OMG is currently working on a wide variety of services which should make CORBA a more complete development environment. These services address, for example, concurrency, transaction, security, object life cycle and naming support. Microsoft has built a number of services on top of COM. These include persistence and compound document support.

Both CORBA and COM support specification of object classes by using an interface definition language (IDL). IDLs define the attributes and methods supported by an object. However, they do not specify some important properties of objects such as an object's resource requirements.

The Portable Common Tool Environment (PCTE) [Groote89] also addresses problems related to component integration. PCTE may be viewed as an object repository which supports communication, concurrency control and associations between objects.

2.2. State of Research

A number of research projects have addressed different pieces of the CBSE puzzle.

Domain-specific kits are being researched at HP labs [Griss93, GW94]. A domain-specific kit is a collection of components packaged together for the purpose of constructing applications within a particular domain. A kit generally contains artifacts such as components, frameworks, generic applications, glue language, environment, documentation, and test suites. Construction of applications using a kit consists of the selection of appropriate components, instantiation of their parameters, and generating a system.

Gerhard Fischer has done work on *Domain-Oriented Design Environments* (DODEs, [Fischer94]). JANUS, one of such environments, illustrates application of Fischer's ideas to kitchen design wherein the components are sinks, stoves, refrigerators, and so on. Central to this work is a knowledge-based critiquing mechanism that warns of possible violation of composition constraints [FNOSS93].

The ARPA *Domain-Specific Software Architecture* (DSSA) program [MG92, WDSSA90] has been targeted at developing architectures, methodologies, and tools for supporting the development of similar systems in selected domains.

[1]In this paper, we use the word "component" to describe an addressable entity. Whether this entity turns out to be an object in the object oriented sense is irrelevant.

[2]The OMG is a consortium of companies interested in standardizing distributed object oriented programming. Andersen Consulting is a voting member of the OMG.

Allen and Garlan [AG94] define a formal framework for specifying component interconnections. 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.

A number of research projects are focusing on the runtime environments necessary to support and connect components distributed throughout a network. These include *Polylith* [Purtilo91] and *Regis* [MDK94]. The projects also involve research on advanced configuration languages (MIL in Polylith 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. Support for dynamic reconfiguration of components is also explored.

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 [Callahan93], which automatically determines how to integrate a diverse collection of software components based on their type and the capabilities of integration tools available in the environment.

There is a variety of other research efforts related to our approach at the technology level. Some examples are: *faceted classification and software libraries* [Prieto-Diaz91, SB93], *module interconnection languages* (MIL, [DeRemer76, HW93, PN86, Thomas76, Tichy79]), *module interface specification and analysis* [Perry89], *megaprogramming* [BS92, WWC92], and *frameworks and other object-oriented techniques* [HW93, WWW90, JR91].

2.3. Our Approach

Our analysis of existing software buses has shown that they cannot be used conveniently to support architecture-level component integration. They are typically extensions of existing programming languages with libraries providing access to inter-process communication services. For example, CORBA's IDL [OMG91] can specify data and operations that components provide to other client components, but it suffers from the following limitations:

- IDL only specifies syntactic properties of components such as parameter types of interfaces. It is too low level, however, to specify semantics-oriented design information (i.e., what a component is supposed to do).
- IDL can only specify *provided services*, operations that a component offers to other components. It does not support the explicit specification of *required services*, operations that a component, as a client, must obtain from other components. References to server operations have to be embedded in the code implementation of client components. As a result, interconnections between components are "hard-wired", greatly reducing the reusability of client components and reconfigurability of the resulting systems.
- IDL is limited to describing component interfaces only; it cannot be used to specify high-level architecture concerns such as composition, interconnection, communication, control, distribution, and configuration.

We also concluded that none of the research efforts fully addresses all of the component integration concerns. However, collectively they address most of these concerns. We therefore decided to leverage from the fact that current software buses already offer network protocol and implementation language transparency[3]. At the same time, we built upon the existing research work described in the previous section. While our approach is not as domain-specific, we borrowed architecture-driven specification, graphics-based interconnection, and knowledge-based critiquing ideas from domain-specific kits and DODEs. We have also benefited from research in distributed systems configuration represented by Polylith and Regis -- some configuration constructs in our specification language are similar to those in MIL and Darwin. The software packaging idea is an evolution of

[3] We utilize XShell [Expersoft94], a robust object request broker, to serve as a backbone for our runtime system. This software bus selection does not constrain the applicability of our approach; the packaging technology can be retargeted to different bus implementations.

similar work conducted at the University of Maryland.

We consider the breadth and depth of our research and the integration of many research ideas to be our main contribution. Our framework for specification of component integration covers a wide spectrum of architecture design concerns from interface specification to interconnection and system configuration. We extend the existing research in many ways. For example, we automate the validation of component interconnections using the interface and interconnection specifications. We also address how to interconnect heterogeneous components. But most importantly, our approach establishes a bridge between promising research ideas in integration specification to leading integration technology (CORBA-based software buses) on the market.

2.4. Scope of Work

The primary focus of our work is formal specifications of architectural properties of systems. Software architectures are a much debated topic. It is not our intention to provide here yet another definition of architectures. Suffice it to say that our discussion is mainly concerned with structural aspects of software systems. More specifically, we focus on component integration related architectural issues: how software components can be interconnected and packaged into a software systems. We explore how the work of an architect can be assisted. From this view point, integration issues broadly fall into the following three categories:

1. **Interface**. An interface of a component describes the operations it supports, services it requires, and other external characteristics. The architect has to be able to understand very well the interfaces of components in order to effectively use them. It is generally not enough to understand only the syntactic requirements of interfaces (*signatures*). The architect may need to know their semantic behaviors as well.
2. **Interconnection**. To construct a system, interconnections between components must be established. Such interconnections can, for example, tie together a component which requires a particular service to another instance which provides the service. Without automated support, the architect must use low-level protocols of inter-process communication (RPCs, for example) to support distributed interconnections, which not only demands specialized skills of architects but also makes the resulting system less portable.
3. **Configuration**. Configuration of components addresses their relative disposition within the application they form. The configuration may specify that a component runs in a thread in a process on a particular machine and that it should communicate with another component running as a process on a different machine. Thus configuration also addresses the distribution of components. Today, it is the responsibility of the architect to configure or map components onto a distributed computing environment consisting of a network of machines. In addition, the architect may have to develop start-up and shut-down scripts, reconfiguration routines, etc.

Today, the resolution of the above integration issues is manual, labor-intensive, error-prone. The research described in this paper is an attempt to automate the component integration process. Our approach is depicted in Figure 2.

Central to our approach is an Intelligent Visual Design Environment. We assume the availability of components, either newly developed or reused. Since the components may come from reuse libraries, the environment is supported by a library management sub-system. The environment provides graphical assistance in developing interface, interconnection, and configuration specifications. It also verifies the specifications for completeness and correctness. The packaging engine transforms the specifications into CORBA IDL code, compilation procedures or "makefiles", configuration procedures, startup, shutdown, and installation scripts, etc. The makefiles are used by the *make* utility to invoke preprocessors, compilers, and linkers on IDL code and source code of components to generate system executables. The configuration and runtime scripts are used to install the system code and manage its execution and reconfiguration.

More details of the design environment will be elaborated in Section 4. In the next section, we

introduce a language for specifying component integration.

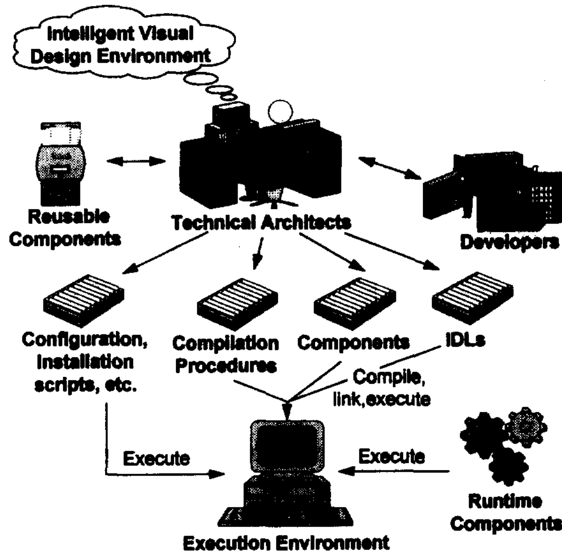

Figure 2. An approach to automating component integration process.

3. Architecture Specification Language

Our intention has been to create an Architecture Specification Language (ASL) capable of supporting the component integration design task. We do not present the syntax of the language, which has not been fully developed yet. Rather, we highlight its important features and explain why ASL is suitable for interface, interconnection, and configuration specifications. Then, we use an example to illustrate these features. The format of ASL specifications is shown in Figure 3.

3.1. Interface specification

Existing interface definition languages such as CORBA IDL and Object Database Management Group's ODL [ODMG93], are generally low-level and syntax-oriented. ASL, on the other hand, can be used to describe semantic characteristics of component interfaces independent of their implementation characteristics associated with programming languages and execution environments.

```
interface component-name : ancestors {
        services
        ports
        interface-constraints
        protocols
}

implementation component-name {
        instances
        bindings
        connectors
        rununits
        implementation-constraints
}
```

Figure 3. Format of ASL specifications.

Some key features of ASL are:

- **Services**. There are two types of services. *Provided services* are operations that a component offers to other components. They can be compared to *methods* or *member functions* in object-oriented terms. *Required services*, on the other hand, are operations that a component must obtain from other components. In concept, they are similar to function or method calls. A required service may reference a provided service either *directly* or *abstractly*. Existing programming languages only support direct references: to call a function, its name must be explicitly given. An abstract reference, on the other hand, is a specification of the semantic characteristics of a service that may be satisfied in more than one way. The use of abstract references breaks up "hard-wired" connections among software components. As a result, components can be independently developed and reused in different situations.
- **Ports**. ASL supports two methods of specifying component interfaces. The first method is to use provided and required services as described above. The second method is to use *ports*. Components use *output ports* to send data or objects to *input ports* of other components. There is an additional type of ports: *input/output ports* which can be used to both receive and send objects. Port semantics support a dataflow style of component interaction and are often

considered not very object-oriented. Nevertheless, ports add a convenient way to express certain desirable features that are otherwise difficult to express. For example, ports make it easy to implement a *publish/subscribe* protocol of communication in which certain components broadcast (*publish*) objects (usually called *events*) to other components, which listen (*subscribe*) to these events.

- **Interface constraints**. Some constraints on interface matching are obvious. A required service, for example, can only match with a provided service, but cannot match with another required service or a port. Similarly, an input port can only match with an output port, or an input/output port. Also, types or classes of data/objects passed between service or port interfaces must match. In addition, ASL supports the specification of more semantics-oriented interface constraints associated with services and ports. More specifically, *preconditions* and *postconditions* specified for services describe states before and after services are performed. Ports have only preconditions specifying states before messages are sent/received. Preconditions and postconditions are expressed in a first-order predicate logic with semantic extensions (e.g., predefined, domain-specific predicates).
- **Protocols**. In some situations, interface objects of a component must be accessed in a certain order for the component to function properly. For example, calls to a file management component must follow an Open (Read | Write)* Close sequence. We call this dependency among interface objects within a component interface protocols. A state machine representation can be used to specify protocol requirements [YS94].

3.2. Interconnection specification

Independently developed components have to be assembled or "glued" together to create new components or systems. Interconnection-related concepts are specified in the *implementation* construct. An implementation specification is associated with an interface specification by using the same component name. An interface specification may have multiple implementation specifications associated with it representing different versions of a component. Interconnections are specified via:

- **Instances**. Interface specifications of components are like type or class definitions in a programming language, from which instances can be created in implementation specifications.
- **Bindings**. Direct references between required and provided services can be established at assembly or linking time as function invocations, or at runtime by a name brokerage service. Similarly, direct port references can be established as message sending and receiving statements, or at runtime by an event-driven mechanism. Because abstract references are not name-based, *bindings* among component services and ports must be explicitly specified. The binding construct in ASL may be used to interconnect services and ports that have potentially different names. Another use of bindings is to export services or ports of instance components to the interface of their enclosing components.
- **Connectors**. A primary inhibitor to reuse is that people can never find exactly what they want when they try to reuse. When components cannot be interconnected directly with the *bind* construct, more complex connectors must be specified. Such connectors can be *adapters* that fill the gaps between slightly mismatched components. Or they can be *coordinators* to facilitate complex patterns of interactions involving multiple components.

3.3. Configuration specification

Configuration-related concepts are also described in the implementation part of component specifications:

- **Rununits**. Here, we group instances of components into *rununits* and assign them to specific machines. A rununit describes computation that executes in the same process and address space. We may also specify how the system can be reconfigured at runtime.

- **Implementation constraints**. Implementation constraints provide information concerning properties of source code such as file paths, programming language choices, platform requirements, etc.

Configuration-related concepts are also described in the implementation part of component specifications

A simple example illustrates how ASL can be used. The ASL pseudo code shown in Figure 4[4] assumes that there are two components, Hello and Print. Hello provides a service (to create a greeting message) and sends out a string to print. Its precondition specifies that the output string S will contain value "Hello, World!" at the time it is sent out. Print accepts any string with less than 80 characters and prints it. Implementation specifications of Hello and Print indicate locations of source code files and architecture assumptions.

In the example Hello and Print are combined into a new component HelloWorld, which acts as a server accepting a greeting command and printing out the message "Hello, World!". The HelloWorld component is implemented with two subcomponents h and p, which are instances of Hello and Print, respectively. The first bind statement glues h's output port (SentToPrint) to p's input port (PrintString). Notice that the two ports can be glued together even though the name of Hello's output port (SentToPrint) does not match that of Print's input port (PrintString). They are *plug-compatible* because their interface constraints (preconditions) match. The second bind statement exports the service provided by h (Greeting) to the interface of HelloWorld (GenGreeting) such that a call to GenGreeting is equivalent to a call to Greeting.

System configuration is specified in the later part of HelloWorld implementation. Two rununits, runh and runp, are specified in terms of component instances h and p that will run on workstations named sellers and fields respectively.

[4] The example used here is only meant to illustrate ideas. So it should not be taken literally as far as the syntax is concerned.

```
interface Hello {
  provide service Greeting ();
  output port SentToPrint (string S)
    PRECOND = StringEqual(S, "Hello, World!");
}
implementation Hello {
  FILE = /examples/hello.c;
  ARCH = Sparc;
}
interface Print {
  input port PrintString (string T)
    PRECOND = StrLen(T) < 80;
}
implementation Print {
  FILE = /examples/print.c;
}
interface HelloWorld {
  provide service GenGreeting();
}
implementation HelloWorld {
  Hello h;
  Print p;
  bind h.SendToPrint to p.PrintString;
  bind GenGreeting to h.Greeting;
  rununit runh { h } @sellers;
  rununit runp { p } @fields;
}
```

Figure 4. Pseudo ASL specs.

4. Visual Environment

System architectures concern structural aspects of software systems: interconnections among components, distribution topology, hardware platforms, etc. It is thus desirable to represent architecture design information graphically to facilitate specification visualization and understanding. For this purpose, we are developing a graphical design environment to support visual presentation and construction of architecture specifications and component interface specifications.

Figure 5 illustrates how interconnections are specified. Initially the architect may import reusable components from a reuse library to the design environment. By using the pallette on the left-hand side of the window, the architect interconnects components in the design area. Whenever a connection between two interfaces is established, an

interconnection verification mechanism is triggered to verify its validity. The mechanism is described in section 4.2. The design area in Figure 5 shows that the HelloWorld component is composed from the Hello and Print components. The bottom half of the screen contains the detailed information of the selected part in the design area.

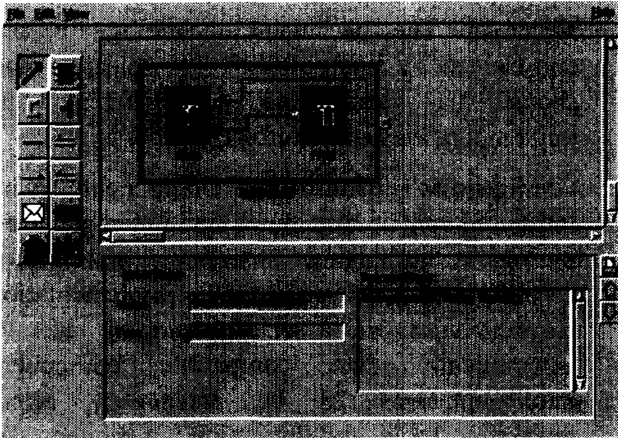

Figure 5. Interconnection window.

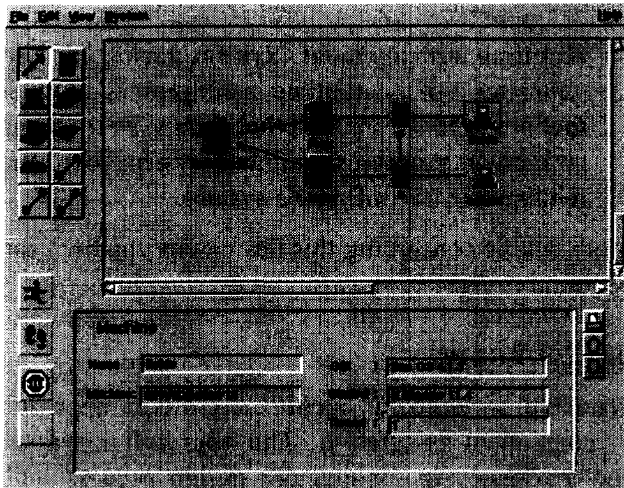

Figure 6. Configuration window.

Figure 6 illustrates the configuration window of the environment in which the architect groups components into rununits and assigns them to machines. The design area shows that the HelloWorld component consists of Print and Hello components. These two components are assigned to one rununit each. Each rununit in turn, is assigned to a particular machine.

In addition to specification assistance, the design environment also provides the following types of services:

4.1. Component management

Our design environment is associated with a library management system to assist in the storage and retrieval of reusable assets: component specifications, their implementations (code), architecture specifications, etc.

The reusable assets are organized based on the *faceted classification* [Prieto-Diaz91] scheme along several dimensions, or *facets*, such as application domain, technical architecture, functional architecture, implementation language, asset type, etc. *Terms*, or sub-facets, further discriminate the assets. Through a graphical user interface, the architect is able to conveniently classify and retrieve assets, and perform library management functions (redefine classification schemes, query usage patterns of assets, etc.).

4.2. Specification verification

Architecture design decisions specified in ASL must be verified and analyzed before they are committed into implementation. We are currently developing capabilities to support the following two types of analysis:

- **Interconnection verification.** When an interface object (a service or a port) is connected (bound) to another interface object, the *compatibility* of the objects should be checked. Existing methods mainly focus on syntactic checking such as type matching and parameter size checking. Our approach uses more semantics-based techniques. Very briefly, we say that a required service object R is compatible with a provided service object P if R's precondition *implies* P's precondition (P's invocation condition is met). Moreover, if P returns results to R, then P's postcondition must *imply* R's postcondition (R's service requirement is met by P's results). Ports do not have postconditions because they do not

produce or return results. It is only necessary to perform implication proving of preconditions for interconnected ports. In the example shown in Figure 3 where h's SendToPrint binds with p's PrintString, implication proving can be carried out between SendToPrint's and PrintString's preconditions to verify that whenever StringEqual(S, "Hello, World!") is true, StrLen(S) < 80 will be true as well.

- **Configuration analysis**. System configuration properties, such as distribution and performance, are analyzed using domain knowledge and general design principles and constraints. Domain knowledge may describe typical distribution patterns of system components in an application domain. This knowledge can be used to detect questionable configuration decisions. For example, if two components are known to be distributed in an application domain, a warning message will be generated when an attempt is made to combine them into a single rununit. In addition, many types of domain-independent, technical architecting knowledge will be used to catch design defects. An example of such a design defect is assigning large server processes to run on small machines.

4.3. System packaging

So far we have only dealt with specifications -- abstract and implementation-independent descriptions of component interfaces and system architectures. The design environment supports *system packaging*, the derivation of procedures from ASL specifications for generating an executable system. More specifically, packaging addresses the preparation of components so that they may be accessed and used by other components, e.g., by locating optimal implementations of an interface, compiling implementations when necessary, and creating stub procedures or "glue" code in order to facilitate communication between heterogeneous components. Usually such glue code would rely on the services of a software bus, which supports the instantiation of components and provides the basic communication facilities.

This packaging process produces the following information:

- **Interface definitions**. We are using a CORBA-compliant software bus XShell® [OMG91, Expersoft94] as the integration and communication layer for system execution. System packaging generates CORBA IDL (Interface Definition Language) code, which is subsequently compiled into XShell stubs and skeletons. The packaging technology is not constrained by this software bus selection and should be retargetable to different bus implementations and interface standards.
- **Configuration procedures**. Components to be integrated can be distributed and heterogeneous. System packaging uses architecture specifications, component implementation characteristics, and a set of *packaging rules* to automatically find compatible component implementations to be integrated, select integration tools, and determine the sequence of tools application. The results of system packaging are "makefiles" or compilation procedures, configuration procedures that can be used to generate executable system code.
- **Runtime management**. System packaging also generates system runtime management routines that are used to, among other things, start up and shut down processes, maintain system logs, and dynamically reconfigure a system.

More details concerning this design environment can be found in [NMK94].

5. Summary

We have presented an architecture-driven approach to component integration. The approach promotes and automates an architecture design process built around the assumption that software systems are assembled from components. The work described in this paper is still at the starting stage. Our recent focus has been on how to specify semantic aspects of architectures concerning interface constraints, protocols, and complex types of component connections.

References

[AG94] Allen, R., and Garlan, D., "Formalizing Architectural Connection," *Proceedings of 16th International Conference on Software Engineering*, Sorrento, Italy, 1994.

[BS92] Boehm, B., and Scherlis, B., "Megaprogramming," *Proceedings of the DARPA Software Technology Conference*, 1992.

[Callahan93] Callahan, J. R., "Software Packaging," *Technical Report CS-TR-3093*, U. of Maryland, 1993.

[DEC93] Digital Equipment Corporation, "ObjectBroker: Application Integration Software for Multivendor Environments Supports Digital/Microsoft Common Object Model," *Technology Overview*, Nov. 1993.

[DeRemer76] DeRemer, J., and Kron, H., "Programming-in-the-Large versus Programming-in-the-Small," *IEEE Trans. on Software Engineering*, June 1976.

[Expersoft94] Expersoft Corporation, *XShell User's Manual*, Release 3.0, March 1994.

[Fischer94] Fischer, G., "Domain-Oriented Design Environments," *Automated Software Engineering*, Johnson, L., and Finkelstein, A., eds., Kluwer Academic Publishers, Vol. 1, No. 2, June 1994.

[FNOSS93] Fischer, G., Nakakoji, K., Ostwald, J., Stahl, G., Summer, T., "Embedding Computer-Based Critics in the Context of Design," *INTERCHI'93*, 1993.

[Griss93] Griss, M. L., "Software Reuse: From Library to Factory," *IBM Systems Journal*, Vol. 32, No. 4, 1993.

[Groote89] Groote, C., "PCTE - a remarkable platform," *Information and Software Technology*, 1989.

[GW94] Griss, M. L., and Wentzel, K. D., "Hybrid Domain-Specific Kits for a Flexible Software Factory," *SAC'94*, Phoenix, AZ, March 1994.

[HW93] Hall, P., and Weedon, R., "Object-Oriented Module Interconnection Languages," *Proceedings of the Second International Workshop on Software Reuse*, Lucca, Italy, March 1993.

[IBM93] IBM, *SOMobjects Developer Toolkit User Guide Version 2.0*, June 1993.

[JR91] Johnson, R. E., and Russo, V. F., "Reusing Object-Oriented Designs," *Technical Report*, University of Illinois at Urbana-Champaign, Illinois, May 1991.

[MDK94] Magee, J., Dulay, N., and Kramer, J., "A Constructive Development Environment for Parallel and Distributed Programs," *Proceedings of the Second International Workshop on Configurable Distributed Systems*, Pittsburgh, PA, March 1994.

[Microsoft93] Microsoft Corporation, "Object Linking and Embedding (OLE) Today and Tomorrow," *Technology Overview*, November 1993.

[NMK94] Ning, J. Q., Miriyala, K., and Kozaczynski, W., "An Architecture-driven, Business-specific, and Component-based Approach to Software Engineering," *Proceedings of the 3rd International Conference on Software Reusability*, Rio de Janeiro, Brazil, November 1994.

[ODMG93] Object Database Management Group, *The Object Database Standard: ODMG-93*, Cattell, R. G. G., ed., 1993.

[OMG91] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, OMG Doc. Number 91.12.1., 1991.

[Perry89] Perry, D. E., "The Inscape Environment," *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh PA, May 1989.

[PN86] Prieto-Diaz, R., and Neighbors, J. M., "Module Interconnection Languages," *Journal of Systems and Software*, 6, pp. 307-334, 1986.

[Prieto-Diaz91] Prieto-Diaz, R., "Implementing Faceted Classification for Software Engineering," *Communications of the ACM*, May 1991.

[Purtilo91] Purtilo, J., "The Polylith Software Bus," *Technical Report TR-2469*, U. of Maryland, 1991.

[SB93] Singhal, V., and Batory, D., "P++: A Language for Large-Scale Reusable Software Components," *Proceedings of the 6th Annual Workshop on Software Reuse*, Owego, NY, November 1993.

[SunSoft93] SunSoft, Sun Microsystems Inc., *Interface User's Guide and Reference, Project DOE External Developer's Release 1*, June 1993.

[Thomas76] Thomas, J. W., *Module Interconnection in Programming Systems*, Ph.D. thesis, Brown University, June 1976.

[Tichy79] Tichy, W. F., "Software Development Control Based on Module Interconnection," *Proceedings of the 4th International Conference on Software Engineering*, September 1979.

[Wiederhold92] Wiederhold, G., "Mediators in the Architecture of Future Information Systems," *IEEE Computer*, March 1992.

[WWC92] Wiederhold, G., Wegner, P., and Ceri, S., "Toward Megaprogramming," *Communications of the ACM*, November 1992.

[WWW90] Wirfs-Brock, R., Wilkerson, B., and Wiener, L., *Designing Object-Oriented Software*, P T R Prentice Hall, New Jersey, 1990.

[YS94] Yellin, D. M. and Strom, R. E., "Interfaces, Protocols, and the Semi-Automatic Construction of Software," *Proceedings of the OOPSLA'94*, Portland, Oregon, October 1994.