ICS

TECHNICAL REPORT

An Architecture-Centered Approach to Software Environment Integration

Nenad Medvidovic[1] **Peyman Oreizy**[2] **Richard N. Taylor**[2]

Rohit Khare[2] **Michael Guntersdorfer**[2]

Technical Report 00-11

Department of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA

March 23, 2000

[1]Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
neno@usc.edu

[2]Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
{peymano,taylor,rohit,mgunters}@ics.uci.edu

Information and Computer Science

University of California, Irvine

An Architecture-Centered Approach to Software Environment Integration

Nenad Medvidovic[1] **Peyman Oreizy**[2] **Richard N. Taylor**[2]

Rohit Khare[2] **Michael Guntersdorfer**[2]

Technical Report 00-11

Department of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA

March 23, 2000

[1]Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
neno@usc.edu

[2]Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
{peymano,taylor,rohit,mgunters}@ics.uci.edu

ABSTRACT

Software architecture research has yielded a variety of powerful techniques for assisting in the design, implementation, and long-term evolution of complex, heterogeneous, distributed, multi-user applications. Since software development environments are themselves applications with these characteristics, it is natural to examine the effectiveness of an architectural approach to constructing and changing them. We report on our experience in creating a family of related environments in this manner. The environments encompass a range of services and include commercial off-the-shelf products as well as custom-built tools. The particular architectural approach adopted is fully reflexive: the environments are used in their own construction and evolution. We also report on some engineering experiences, in particular with our use of XML as the vehicle for supporting a common and extensible representation of architectural models, including the model of the environment itself. Generally applicable lessons from the experience are described.

Keywords: Software architectures, software environments, tool integration, off-the-shelf reuse, XML

1 INTRODUCTION

A comprehensive software development environment will offer many services to help with the myriad activities associated with application development. Environments are multi-user applications; they are likely to be distributed and concurrent. Building, maintaining, and evolving such an environment is fraught with difficulties. As the scope of the environment expands, the heterogeneity of the constituent toolset is likely to expand as well: tools may come from different vendors, have varying platform requirements, vary in their size, complexity, degree of openness (from source code to APIs to closed binaries), and potentially interact in complex ways. Deciding whether or not a given tool should be considered for inclusion in an environment requires understanding complicated trade-offs. Actual integration of a tool may entail substantial effort. Inadequate understanding of the multiple and various issues has often led to failed environments or serious schedule or budget overruns.

The issues and challenges posed by environments are not unique, of course. Many complex applications exhibit the same properties. Over the past decade, an architecture-based approach to the engineering of such applications has emerged. Central to this approach is the use of architectural models. Such models separate designers' concerns between components, as loci of computation and state, and connectors, as the sole means of communication between components. These models

- focus attention on component-based development,
- aid us in reasoning about certain overall system properties, both rigorous and conceptual,
- aid in system implementation and/or generation,
- assist during system evolution, both before and during runtime, and
- aid configuration management.

Architecture-based software engineering also emphasizes architectural styles, both domain-specific abstractions and effective domain-independent patterns and idioms. These styles and the architectural models work to facilitate understanding and communication among team members and can be a key element in an effective reuse strategy.[1]

Our work has focused on examining the applicability and the utility of applying an architecture-based software engineering approach to the development of an extensible software development environment. We are thus treating the development of an environment as "just another" application development problem. Our approach involves:

- explicitly and formally representing the architecture of the environment,
- employing adaptable connectors to model and implement all the communication between tools, and

1. All architectures and architecture-based approaches are not equal, however. Support for reuse, heterogeneity, and distribution are *not guaranteed.* One-of-a-kind, homogeneous, monolithic systems also have architectures; applications with rigid connectors may have no appealing characteristics with regard to evolution; and so on.

- using an architectural style to further aid understanding, analysis, and evolution of the environment's architecture.

The explicit model employed provides a higher fidelity representation of the environment's architecture than either an architecture implicit in the environment's construction or an ad-hoc, boxes-and-arrows diagram, such as has been typical in the software environments world [13,32,34,45]. We do not simply use an architecture-based approach to explicate our system, however, but also to play a concrete technical role in the implementation and evolution of the environment. For one, if the tools in the environment support development of applications having the same architectural concepts and style as that exhibited by the environment itself, then a reflexive development environment results. That is the case with our work: the environment is used for supporting its own development and evolution, a tradition dating back at least as far as Interlisp [44]. The benefits of doing this include avoiding the need for building special-purpose tools for maintaining the environment.

Our approach is, of course, applicable on different levels of abstraction. We can use the same set of architectural principles to build custom tools within the environment as well as to build the environment as a whole. Our results also benefit the architecture community's ongoing effort to define a canonical architectural toolkit [11].

This paper proceeds by briefly describing the architectural principles underlying our approach. We then describe the approach, report on our experiences with building a family of related environments, and discuss the lessons learned in the process. Along the way, we also consider some interesting engineering issues we encountered and discuss how we addressed them. In particular, we describe how we used the Extensible Markup Language (XML) [4] to support the common internal/external representation of the architectural models, and the benefits that resulted.

2 OUR APPROACH

We have adopted an architecture-based approach to software environment integration. The approach is intended to closely parallel that for developing applications [43], where the tools in the environment correspond to components in an application (see Figure 1). In principle, this permits us to employ an environment in its own development (modeling, analysis, implementation, and evolution). It also allows us to leverage our extensive experience with architecture-based application development.

Three key concepts underline our approach. We explicitly represent the architecture of an environment. As with components in an application architecture, a high-level model of each tool's behavior (internal object in Figure 2) is provided in first-order logic [22], while tool interactions (dialog in Figure 2) are modeled with asynchronous events (also referred to as messages) [43]. This allows early analysis of an environment to establish properties of interest. Furthermore, an environment's architecture does not prescribe its implementation. Instead, an architecture may be implemented in multiple ways, allowing one to address specific non-functional requirements (e.g., performance, concurrency, distribution, and so on).

We employ explicit software connectors to model and implement all interactions among tools. The connectors are highly adaptable: they allow arbitrary addition, removal, and replacement of their attached components. Each connector is mapped into an explicit service (i.e., a module) in an environment's implementation, allowing the modeled environment to be modified both at design-time and run-time [27]. Connector implementation issues are discussed further in Section 3.

Finally, we exploit the properties of an architectural style, C2 [43]. A style specifies design rules and constraints to which a system must adhere. In turn, awareness of style rules facilitates understanding, analysis, and evolution of a system. The C2 style was selected because of its support for distribution, heterogeneity, and reuse. C2 is characterized by minimal interdependencies among components. It supports loose component integration via connectors, as depicted in Figure 1. All communication in a system occurs by exchanging asynchronous events, as depicted in Figure 2. Mismatches among component interfaces are allowed in principle [23].[2] The above properties are desirable when

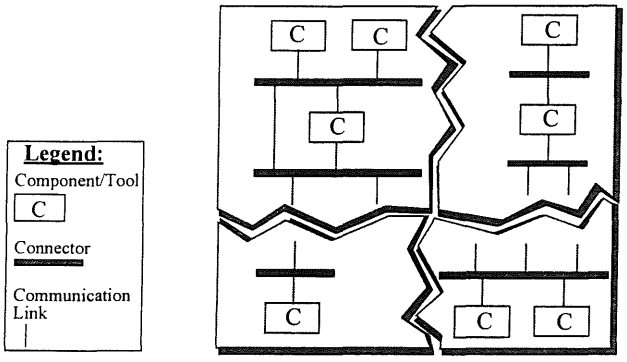

Figure 1. Overview of the adopted architecture-based approach. The architecture is structured according to the rules of the C2 style: the top (bottom) of a component attaches to the bottom (top) of a single connector; there is no bound on the number of components or connectors to which a connector attaches. Parts of the architecture are elided, as represented by the jagged lines.

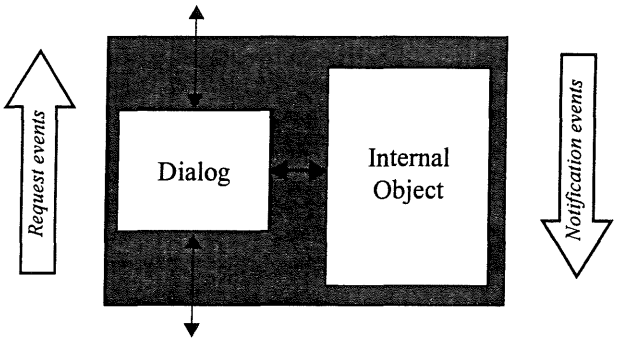

Figure 2. Internal architecture of a C2-style component (tool). The internal object contains application-specific functionality and may be a third-party tool accessible via an application programmable interface (API). The dialog engages in event-based communication with the rest of the architecture and makes invocations on the internal object. Two types of events are exchanged: *requests* of components above and *notifications* to components below.

composing large, heterogeneous, possibly third-party tools into an environment.

3 IMPLEMENTATION ISSUES

To support implementation of C2-style architectures, we have developed a light-weight, extensible framework of abstract classes for concepts such as components, connectors, and events (messages), shown in Figure 3. This framework is the basis of development and OTS reuse in C2. It implements component interconnection and message passing protocols. Components and connectors used in C2 applications are subclassed from the appropriate abstract classes in the framework. This guarantees their interoperability, eliminates many repetitive programming tasks, and allows developers to focus on application-level issues. The framework supports a variety of implementation configurations for a given architecture: the entire resulting system may execute in a single thread of control, or each component may run in its own thread of control or operating system (OS) process. The framework has been implemented in Java and C++; its subset is also available in Ada.[3]

Table 1: OTS Component Reuse Heuristics

Problem with OTS Component	Integration Method
Explicit Invocation	Wrapper
Message Interface Mismatch	Adaptor
Different Thread of Control	Inter-Thread Connector
Different Language and/or OS Process	IPC Connector
Inadequate Functionality	Source Code Modification

The base framework has been extended to provide support for reuse, distribution, and heterogeneity of applications. Reuse and heterogeneity are accomplished via explicit, flexible connectors and light-weight component adaptors and wrappers. In general, in order to incorporate any OTS component into a C2 architecture, the component can be wrapped as an internal object inside a generic framework component (recall Figure 2). An extensive series of exercises conducted to study component reuse in the context of C2 [20,23] has resulted in several other heuristics for integrating OTS components into C2-style architectures, summarized in Table 1.

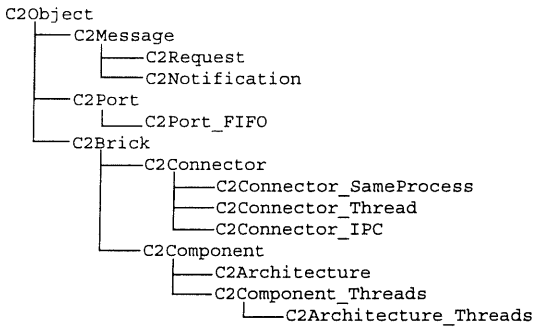

Figure 3. C2 implementation framework.

The framework supports distribution and heterogeneity through addition of connectors that supply a variety of interoperability mechanisms. To date, we have incorporated four OTS interoperability technologies into the framework: Q [17], Polylith [32], Java's RMI [42], and ILU [47]. Each technology supports one or more of the following: multi-threaded and multi-process communication, multi-lingual development, and distribution across a network [7].

We have devised two complementary strategies for incorporating an OTS technology into a connector, shown in Figure 4: a single "virtual" connector is split horizontally or vertically into two actual modules that interact using the mechanisms provided by the OTS technology. These two basic configurations can be combined to achieve any application deployment profile. For example, the architecture in Figure 4 may be configured so that each component runs on a different machine. Once such a "virtual" connector is implemented, it can be used like any other connector; its internals are entirely transparent to the developers and the interacting components.

Finally, we specified a shared data model to represent the architecture of an environment during its construction and execution. This model is used as the basis for evolution. Our approach builds on that of the Field development environment [34], which pioneered the use of shared abstract syntax trees of control flow graphs, source text, and so on, as well as event-based access to them. In particular, we applied the technologies of XML [4] for capturing syntax and C2 component packaging for concurrent access to the architecture repository.

There were several reasons to migrate from a textual architecture description language (ADL) to an XML-based one, called xADL [18]. XML offered a simpler, standardized parser and a richer, internationalized user interface in conjunction with HTML. Using separate XML Namespaces, we designed a generic vocabulary of tags and attributes useful across a range of ADLs independently of ontologies specific to particular ADLs. Furthermore, the principle of ignoring unknown tags allowed individual tools within the environment to annotate individual components and types, as well as include entirely new subtrees without affecting other tools.

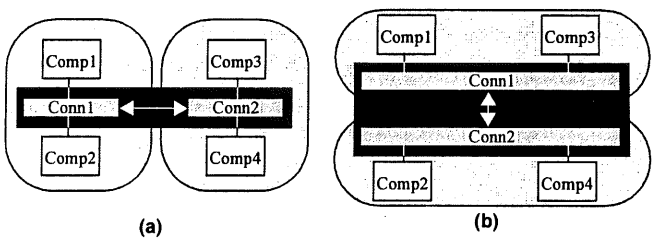

Figure 4. Connectors are a primary vehicle for distribution and heterogeneity. A single conceptual connector can be "broken up" (a) vertically or (b) horizontally for this purpose. Shaded ovals represent language, process, and/or machine boundaries.

2. The impact of a component interface mismatch on a system's operation can range from negligible to serious [23]. Our approach allows the system's architecture to be analyzed to assess the consequences of mismatches before the system is implemented and deployed [22].
3. It has been argued by others [9,46] that this framework is similar to commercial middleware platforms, such as CORBA [30] and COM [39].

Conversely, XML extended the promise of adding architectural knowledge to applications entirely outside the environment, such as XML-format drawing tools.

Fundamentally, we view architectural description as a form of hypertext. Rather than presenting a unified object-base for a project as in, say, the Montana integrated C++ environment [14], hyperlinking explicitly articulates separate evolution of each component in the architecture. One component interface might be extracted from a vendor's Web page, while connectors may be annotations in an illustration package; currently "external" documentation could be included within the project Web as well.

4 A FAMILY OF SOFTWARE DEVELOPMENT ENVIRONMENTS

During the course of the last four years, we have constructed a family of software development environments using our approach. Our first environment, ArchStudio, began as an experiment in applying the C2-style and the principles of architecture-based development to software environments. Inspired by its success, we built more comprehensive environments that integrated both commercial and research off-the-shelf (OTS) tools. We describe each of these environments in the following subsections. The purpose of this section is to reveal the complexity of the problems addressed and some of the details of how the approach was applied. In turn, this enables assessment of the general utility of the approach.

ArchStudio 1.0

ArchStudio 1.0 was the first prototype environment that embodied our approach. The initial version was constructed in 1996 and was then incrementally improved and extended. ArchStudio 1.0 provided a graphical design environment for interactively constructing and analyzing software systems at the architectural level, and a novel mechanism for interactively evolving the system *during runtime* by changing its architectural model. The environment's own architecture is depicted in Figure 5 and was implemented using the C2/Java class framework.

The two central tools in the environment are *Argo/C2* and *ArchShell*. *Argo/C2* [36] is an interactive, graphical design environment for software architectures that allows architects to drag-and-drop components and connectors from a palette onto a design canvas. *Argo/C2*'s critics continuously examine the system under design, identify errors, and non-intrusively suggest design alternatives. *ArchShell*, on the other hand, provides a text-based, interactive interface for instantiating implementations of components and connectors into an architecture and executing the architecture. *ArchShell* is novel in that it enables architects to evolve a system's running implementation by changing its architectural model [27,29].

Since *Argo/C2* and *ArchShell* were initially designed as stand-alone tools, each used its own internal representation of the architectural model. To create ArchStudio 1.0, we wrapped them to emit messages describing changes to their internal representations. *ArchADT* is a shared representation

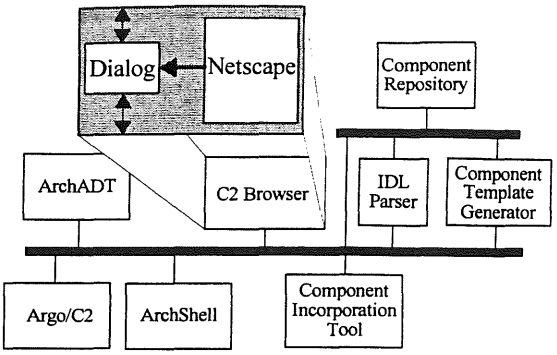

Figure 5. ArchStudio 1.0's C2-style architecture.

component that records the changes and notifies other tools in the environment of them, thus synchronizing *ArchShell* and *Argo/C2*'s internal representations.

The remaining ArchStudio 1.0 components facilitate runtime manipulation of an application's architecture. *C2 Browser* encapsulates an OTS Web browser, such as Netscape Communicator, as discussed in Section 3. *C2 Browser* allows architects to locate new C2 components (identified by a unique MIME type and file extension) over the Web. Clicking on a C2 component link causes Netscape to invoke *C2 Browser*'s dialog; in response, the dialog emits a notification message announcing a successful download of a new C2 component. *Component Incorporation Tool* reacts to the notification by unarchiving the C2 component from the downloaded file, passing its interface description to *IDL Parser*, and installing the component for use in a running application. *IDL Parser* parses the component's interface description and stores it in the *Component Repository*, a file-based repository of component interfaces, similar to CORBA's Interface Repository. Other tools can retrieve a list of available components from the *Repository* and examine their interfaces. Similarly to *C2 Browser*'s use of Netscape, *Component Repository* uses JOP [38], an OTS persistent object package, as its internal object.

Finally, *Component Template Generator* generates a Java class template when given a C2 component name. It queries the *Component Repository* for the component, examines its interface, and generates a Java method signature and body for every interface element. This capability helps to reduce the effort required to implement new C2 components.

While simplistic, ArchStudio 1.0 was constructed using the same principles as the systems it was designed to construct. In fact, the initial prototype of the environment was constructed (and modified during runtime) using *ArchShell*.

DRADEL

The DRADEL environment, initially developed in 1998, supports specification, analysis, design-time evolution, and implementation of C2-style architectures. Its architecture is shown in Figure 6. Like ArchStudio 1.0, DRADEL adheres to C2 style rules and is thus applicable on itself. The environment has been implemented in Java, using the framework described in Section 3. An early prototype of DRADEL was

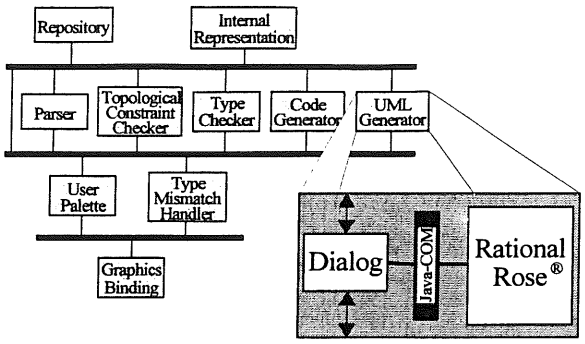

Figure 6. Architecture of the DRADEL environment.

discussed in [22]. This section briefly summarizes the environment's key features and introduces a recent addition that required integrating an OTS tool.

The *User Palette*, *Type Mismatch Handler*, and *Graphics Binding* components from Figure 6 provide a graphical front end for the environment; we will not focus on them in this paper.[4] The *Repository* component stores architectures modeled in C2's ADL, C2SADEL [22]. The *Parser* receives via C2 messages a specification of an architecture, parses it, and requests that the *Internal Representation* component check its consistency and store it. *Internal Representation* is an ADT that builds its own representation of the architecture and ensures that components and connectors are properly specified and instantiated.

Once an entire specification of an architecture is parsed and its internal consistency ensured, the *Topological Constraint Checker*, *Type Checker*, *Code Generator*, and *UML Generator* components are notified of it. *Topological Constraint Checker* ensures adherence to the topological rules of C2 discussed in Section 2. *Type Checker* is DRADEL's centerpiece, enabling two key tasks: analysis of architectures to establish (the degree of) behavioral conformance among interacting components and evolution of individual components. Both these tasks make use of the components' behavior models, provided in C2SADEL as first-order logic expressions, and implement the idea of heterogeneous typing for software architectures [18,19,22]. The *Code Generator* component is evolved from ArchStudio 1.0's *Component Template Generator*: it generates an implementation skeleton for the modeled architecture on top of the Java/C2 framework. For each component, *Code Generator* automatically generates the dialog from the component's C2SADEL specification. It also partially generates the internal object, with stubs for each method. The stubs must then be implemented manually or replaced by an OTS component.

In order to provide developers with support for implementing the internal objects, as well as for refining architectures into implementations that are independent of the C2 framework, we have introduced the *UML Generator* component. *UML Generator* implements a set of rules we have devised [21,35] for transforming an ADL specification into UML [5]. Like other C2 components, *UML Generator* internally

4. GraphicsBinding is an example of OTS reuse. It incorporates a user interface toolkit, Java's AWT [20].

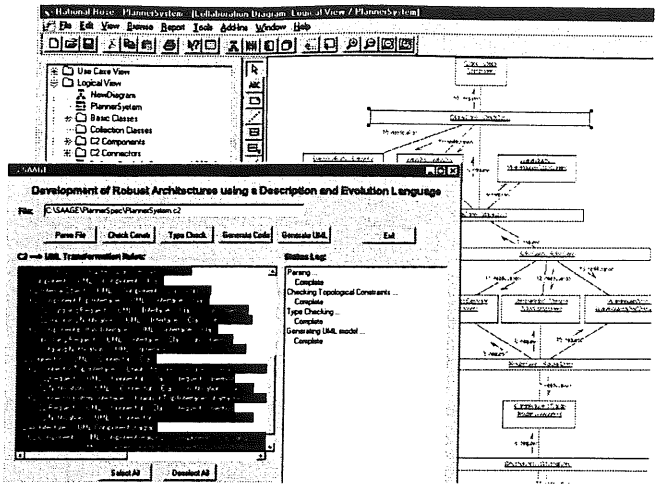

Figure 7. Screenshot of DRADEL (bottom) and the UML model produced by DRADEL's *UML Generator* in Rational Rose (top).

consists of a dialog that engages in message-based communication and an internal object that implements the component's functionality. In this case, the internal object is Rational Rose [33], an OTS environment for UML-based software development (see Figure 6).

This example illustrates the flexibility of a C2 component's internal architecture. Typically, the interaction between the dialog and the internal object is accomplished via procedure calls. However, Rose provides an API that is accessible via COM [39]. We, therefore, employed an OTS facility that enables Java applications to interact with COM objects. This facility acts as a connector internal to a component. A screenshot of the resulting tool is shown in Figure 7.

ArchStudio 2.0

ArchStudio 2.0, our most recent development environment, combines many of the tools in ArchStudio 1.0 and DRADEL with other research OTS and COTS tools, including Rational Rose and Metamata IDE. Its C2-style architecture is depicted in Figure 8. The lightly shaded tools in the figure represent individual integrations intended to assess the feasibility of the approach; they have not yet been added to the production environment. The more heavily shaded tools are part of the current environment.

A significant difference between ArchStudio 2.0 and our previous environments is our use of an extensible, persistent, XML-based shared architectural model for the system under design, xADL (see Section 3). This shared model is encapsulated as an abstract data type by *ArchADT*. Supported operations on the model include enumerating the components and connectors in the architecture, and their respective types; adding and removing architectural elements (e.g., component types and instances); and querying and modifying the architecture's topology. Additionally, the model may be extended by adding new attributes to existing elements or by adding new sub-hierarchies of elements.

xADL is a generic language framework with five basic tags:

- `<Architecture>`: a list of directed links between

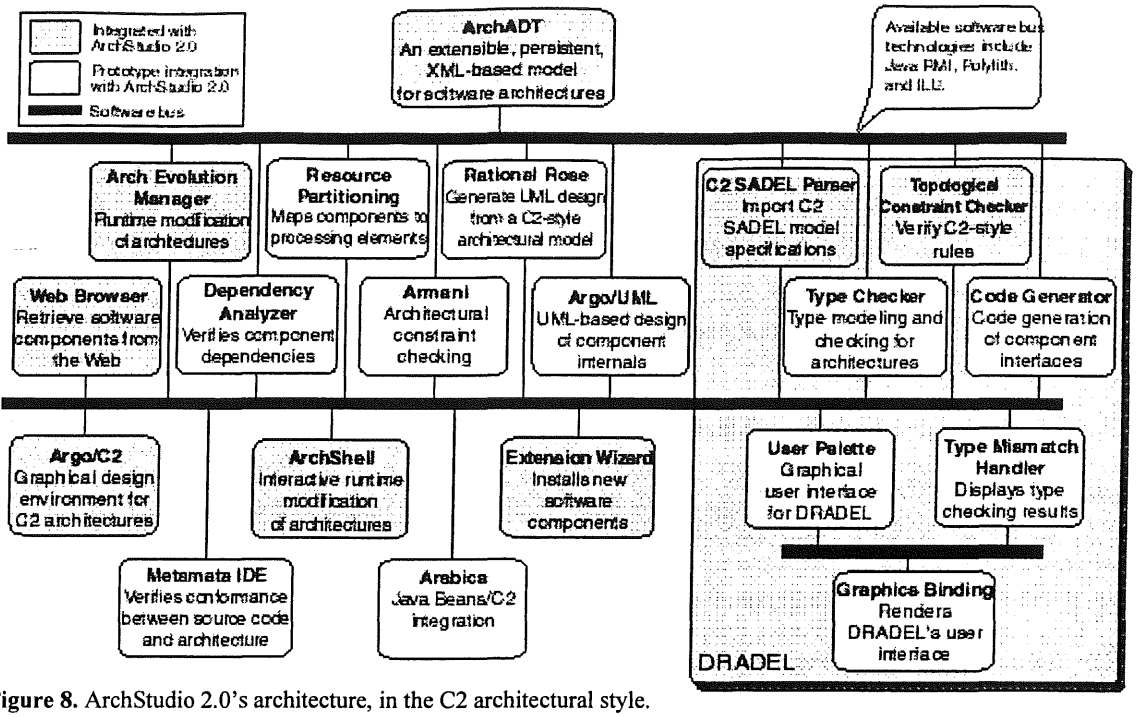

Figure 8. ArchStudio 2.0's architecture, in the C2 architectural style.

instances;

- `<Component>`: name and type(s) supported by each component instance;
- `<Connector>`: name and type(s) supported by each connector instance;
- `<ComponentType>`: name and method interfaces for each component type, and input and output parameters; and
- `<ConnectorType>`: name and method interfaces for each connector type

Composing xADL with additional C2-specific definitions yields a complete, verifiable xC2 Document Type Definition (DTD). For example, we added the `xC2:Filter` attribute and its five permissible settings [43] to `<xADL:ConnectorType>`. As discussed above, DRADEL enforces the remaining properties of an architecture the grammar alone cannot, such as type conformance.

ArchStudio 2.0's tools query and modify the model by sending C2 request messages to *ArchADT*. If a tool's request changes the model (e.g., adds a new component), *ArchADT* emits a C2 notification describing the change to the connector below it, which, in turn, broadcasts it to the tools attached below it. Each tool receives and reacts to the state change independently; typical reactions include making a corresponding change to its internal representation, updating affected graphical views, or ignoring state changes that fall outside a tool's domain of interest.

In addition to ArchStudio 1.0 and DRADEL's tools, ArchStudio 2.0 introduces several other components, giving the architect additional support for architecture modeling, analysis, implementation, reuse, deployment, and evolution.

Evolution Management: The *Arch Evolution Manager* component is responsible for changing a system's running implementation to correspond with its changing architectural model. This component was a part of *ArchShell* in ArchStudio 1.0. In ArchStudio 2.0, we have decoupled it from *ArchShell* and modified it to run as an independent component.

Resource Partitioning: One of *Argo/C2's* graphical views allows architects to assign system components and connectors to operating system processes and machine hosts. The *Resource Partitioning* tool retrieves these attributes and generates initialization and startup code for executing the system in the specified configuration. The tool relies on available OTS connector technologies, such as those described in Section 3.

Dependency Analysis: The *Dependency Analyzer* tool examines the interface of every component in the system (consisting of the messages understood and potentially emitted by the component) and the architectural topology to reveal dependencies between components. This information helps architects evaluate how components are used and the consequences of adding, removing, replacing and reconnecting components.

Constraint Management: *Armani* [25] is a language and tool set developed at Carnegie Mellon University (CMU) for specifying and checking constraints over a system's architectural topology. Our prototype integration enables architects to specify constraints over how a system's topology may evolve during runtime. If any tool (e.g., *Argo/C2*) modifies the system's model in a way that violates a topological constraint, *Arch Evolution Manager* will not perform the corresponding change to the system's running implementation.

Design Refinement: *Argo/UML* [36] is a UML design environment similar to *Rational Rose,* developed at UC Irvine.

Our prototype integration allows architects to diagram a component's internal design based on the component's interface specification. Since *Argo/UML* was developed independently of ArchStudio, it stores its diagrams in individual files, not within *ArchADT*. To relate the UML and architectural models, our integration stores the filename of each component's UML diagram as an annotation on the component's model within *ArchADT*.

Interoperability: *Arabica* [26] provides interoperability between Sun Microsystems' JavaBean components [41] and C2 components by automatically translating JavaBean events into C2 messages, and visa versa. Additionally, *Arabica* ensures C2's topological rules among connected JavaBean components.

System Generation: After generating template code for a component's interface using DRADEL's *Code Generator,* developers can edit the code using the *Metamata IDE* [24]. Using *Metamata*'s style checking feature, our integration detects source code changes that diverge from the component's interface and notifies the architect. This assures a certain degree of fidelity between the architectural model and its implementation.

Our research group is continuing to improve ArchStudio 2.0 by integrating new tools, improving its user interface, and extending its XML schema.

An Avionics Development Environment

The preceding three environments have been built by the members of our research team. This section briefly describes an environment constructed by a third-party organization, a major aerospace company, using the same principles and tools. This organization used C2 to define and implement the *System Control and Configuration Manager* subsystem of an avionics development and simulation environment. A set of the company's instrumentation and operational components was integrated with ArchStudio 1.0 and the toolset provided with Stanford University's Rapide system [16]. The resulting environment included interactive support for defining a configuration (i.e., an architecture) of a simulation, and visualizing and analyzing its run-time properties.

ArchStudio 1.0 was used to control dynamic reconfiguration of architectures of the simulated avionics systems. The C2 implementation framework provided access to the runtime events emitted by the components in a simulated system. The company defined an event visualization facility to support the identification of specific event patterns occurring during system execution. Rapide's event analysis and animation tools were notified of the observed event patterns in order to analyze the actual behavior of the simulation. For example, by capturing the run-time events, the company was able to produce a three-dimensional visualization of an in-flight replanning algorithm, which operated on the run-time trace of events associated with the algorithm's key data structures.

This environment was integrated without any reported changes to the OTS tools (ArchStudio 1.0 and Rapide tools) beyond providing adaptors and wrappers to enable their interoperation in the manner discussed in the preceding sections.

Environment Interoperability

Our work on building multiple software development environments provided us with the opportunity to explore *environment interoperability*—integrating multiple environments, each with its own tools, into a single, composite environment. The problem is one of scale. Getting two tools to interoperate is a "1 x 1 problem," where either (or both) of the tools can be adapted to suit the peculiarities of the other. Integrating a tool with an environment is an "1 x N problem" since adapting the single tool to the environment's established framework is typically easier than adapting multiple tools in the environment. Environment interoperability poses an "N + M problem" where two well-established integration frameworks must be stitched together. In this case, even the initial tasks of understanding how the disparate pieces should fit together poses significant obstacles.

We confronted these issues when we considered integrating DRADEL with a subset of ArchStudio 1.0 (specifically, *Argo/C2*, *ArchShell*, and *ArchADT*). Our goal was to allow architects to use DRADEL's analysis tools in conjunction with *Argo/C2*'s critic-based analyses, and to allow architects the option of using ArchStudio's interactive architecture specification capability or DRADEL's file-based C2SADEL specification language.

The most difficult interoperability issue concerned the incongruent architectural models maintained by each environment. DRADEL's ADT (*Internal Representation* from Figure 6) provided a rich type and component interface model, such as pre- and post-conditions on interface methods, that ArchStudio's ADT (*ArchADT* from Figure 5) lacked. On the other hand, ArchStudio's ADT modeled implementation details that DRADEL's ADT lacked, such as file system paths to component and connector binaries, and flags signaling whether or not components and connectors are running.

One obvious approach involved unifying the two architectural models and implementing a single ADT component replacement. To avoid modifying the other tools, this single ADT would have to simultaneously mimic the message interfaces of both previous ADTs—feasible, but cumbersome and error-prone. The alternative approach involved retaining both ADTs. This had the advantage of not requiring changes to existing tools. However, since the ADTs shared common elements, we would need to implement a mechanism for assuring data coherence. Our prototype implementation involved building a single special-purpose connector to replace the connectors below ArchStudio's and DRADEL's ADTs. This new connector—part adaptor and part consistency manager—dynamically translates messages to and from ArchStudio's ADT into corresponding messages understood by DRADEL's ADT, and visa versa. Since all messages intent on changing either ADT travel through this single connector, a certain degree of data coherence can be ensured. This prototype implementation allowed us to experiment with using ArchStudio 1.0 and DRADEL simultaneously, which benefited the design of ArchStudio 2.0 and its XML-based, unified data repository.

5 LESSONS LEARNED

In the previous section we discussed three environments constructed internally, one environment built by a third party, and two tool interoperability mechanisms: (1) shared, flexible data representation using XML and (2) message routing and adaptation using software connectors. We believe our experiences with these technologies are of value to software architecture and software environments researchers and practitioners, as follows:

- The use of XML as the basis of a shared, extensible representation of an evolving architecture (xADL) proved very successful. It simultaneously met the goals of providing an effective basis for information exchange between the tools without requiring a fixed format and structure. It thus addressed a critical issue that has vexed many previous environments. In the context of architecture-based design, it has done so with acceptable performance.
- The techniques for encapsulating and interacting with binary or "uncooperative" tools, such as Rational Rose and Netscape, were effective. The canonical internal architecture of a C2 component has proven adequately flexible in wrapping over twenty such tools to date.
- Extensibility and manageability of the environments was achieved via an explicit and reflective architecture, using an existing ADL. This suggests that ADLs can provide significant leverage for modeling, analyzing, implementing, and evolving solutions to problems of at least the size and scope of a software development environment.
- Some tools, such as Netscape, offered flexible and convenient interaction capabilities while others, such as Rose, demanded a very fine-grain style of interaction. In both cases, the general mechanisms offered for wrapping tools and for hooking them together with connectors not only sufficed but integrated the tools in a manner that seemed natural.
- Different sets of development environment services were successfully supported by the chosen approach: there is no single "main tool" of an environment, no single user interface, and no essential set of tasks. The tools are integrated in a manner that supports many different usage scenarios; nonetheless, a coherent view of the entire environment from an architectural perspective is maintained.
- The exercises demonstrated that architectural principles and supporting mechanisms can be effectively used to enable the composition of entire environments in much the same manner as the composition of individual tools.
- Finally, the exercises provided some new data points in the on-going evaluation of the C2 architectural style, supporting tools, and its use of software connectors that are explicit not only in the architecture, but also in the implementation. In short, the experience was positive, warranting further efforts to push and explore the boundaries of the style.

Viewing these experiences a bit differently, they motivate and reinforce some general observations: explicit connectors provide enormous leverage; XML is an effective technology; wrappers and implicit invocation are (still) effective software integration mechanisms; reflection in the context of architectures provides clear value; and architectural mismatch can be avoided (or at least minimized) through judicious use of explicit architectural models.

6 RELATED WORK

This paper has been influenced by work in several areas: software architectures, reuse, component interoperability, and software environments. Each is summarized below.

Architectures: Environment integration did not become a focus of the software architecture community until very recently [11]. With the exception of CMU's Aesop environment [10], there have been no published examples of software toolkits that employ an architecture-centric approach. Garlan et al. described in architectural terms the many problems they encountered during Aesop's construction. They coined the phrase *architectural mismatch* to denote conflicting assumptions made by components and produced a set of guidelines for avoiding architectural mismatch. These guidelines largely focus on *components*: their internal structure, adaptors to resolve conflicts, and design guidance for selecting, reusing, and composing them. We have previously discussed why C2 is well suited to address these guidelines [20].

The work described in this paper indicates that an approach centered on component *configurations* is a needed complement to the guidelines: by providing an explicit architectural model, one can assess the assumptions made by a component in the context of its "location" in the architecture. Such an approach may have resulted in early warnings of several problems Aesop's developers faced. For example, a component they used required all communication to be channelled through it. An explicit architectural model would have indicated that this assumption is in direct conflict with the envisioned configuration, in which the four major components needed to directly interact with one another. As it was, this problem was not discovered until the integration of the environment was well under way. Recently, Garlan et al. have applied their formal architecture modeling and analysis approach to the problem of tool integration [1].

Reuse: Two approaches to component reuse have influenced our work: component packaging and domain-specific software architectures (DSSA). Shaw has discussed a set of mechanisms for (re)packaging components to facilitate their use in new contexts [40]. This work has been embodied in a recent approach by DeLine that separates a component's essence from its packaging [8]. However, DeLine's approach does not support reuse of legacy components, but instead requires their reengineering. On the other hand, C2 employs several of Shaw's mechanisms, such as import/export converters, and adaptors/wrappers. Other mechanisms are obviated by C2's rules. For example, multilingual components are inherently supported by C2's use of implicit invocation and explicit connectors.

Another relevant approach to reuse, DSSA, exploits the shared characteristics of applications within a domain. DSSA have proven successful at supporting reuse, but have at times been overly restrictive in that support [20]. A representative example is GenVoca [3], which requires that all components be custom built for its style of interaction and

composition. We have tried to leverage the successes of DSSA, while providing more flexible rules: C2 eliminates assumptions of shared address spaces and threads of control; it supports event-based communication through connectors; and it separates an architecture from its implementation.

Interoperability: Component interoperability technologies (e.g., Field/SoftBench [6,34], Q [17], ToolTalk [12], CORBA [30], COM [39], Enterprise JavaBeans [41]) provide a set of communication services and protocols to enable component interactions. Though they provide effective implementation-level support for environment integration, these technologies often unduly influence the properties of systems they are used to construct [9]. Thus, they must be accompanied by a set of higher-level, compositional guidelines (e.g., an architectural style) and models (e.g., an architectural description) that clearly specify structure, behavior, and other properties of a system [28]. As discussed in Section 3, we have exploited several interoperability technologies to provide support for multi-lingual, multi-process, and multi-platform development of C2-style architectures.

Environments: We have been strongly influenced by the work on software environments. In particular, we have drawn a number of lessons from our involvement with the Arcadia environment project [13]. These lessons included the need for an environment's explicit blueprint (i.e., architecture); the necessity of tool reuse, heterogeneity, and minimal tool interdependencies; trade-offs between tight and loose tool integration; and benefits of applying an environment on itself. Although the environments discussed in Section 4 are smaller in scope than Arcadia, the approach we adopted for their integration has resulted in comparatively quicker development, greater flexibility (e.g., "plug-and-play"), increased support for incorporating OTS technologies, and easier adaptation to changing requirements. Our approach also has roots in DSSA environments (e.g., ADAGE [2]). DSSA environments exploit the properties of a given application domain, while we exploit the (broader) characteristics of a style. Finally, as discussed in [22], DRADEL in particular has drawn inspiration from the Inscape environment for software modeling and evolution [31].

7 CONCLUSIONS

We have been able to successfully construct a family of software development environments using an architecture-based approach. This lends credence to our hypothesis that an architectural approach can be applied to software environments—conglomerates of tools—in addition to "traditional" software applications and individual tools. Furthermore, since our environments have been designed to support development of applications using the same fundamental concepts as those used in the environments' own construction, we can apply the environments "on themselves." The environments have been constructed in the C2 architectural style and integrate a variety of tools, both reused off-the-shelf and developed in-house.

The key elements of our approach include:

- explicit and formal modeling of an environment's architecture. This provides a higher fidelity representation of the environment's architecture as compared to ad-hoc boxes-and-arrows diagrams used in the past;
- using multiple, explicit, adaptable connectors to model *and* implement all the communication between tools. This reduces tool coupling and facilitates the runtime evolution of the environment's architecture; and
- using an architectural style to further aid understanding, analysis, and evolution of the environment's architecture.

Another novel aspect of our approach is that we view a model of an architecture (including the environment's) as a form of hypertext, explicitly capturing the components' and connectors' separate, and possibly distributed, descriptions and evolution. To this end, we designed and incorporated xADL, an extensible XML-based architecture-neutral schema, as the shared data structure for the ArchStudio 2.0 environment. Using XML offered the added benefit of reusing several OTS tools, such as XML parsers, IBM's Xeena to edit our xADL schema and validate our architectural models against it, and Microsoft's Internet Explorer 5 to visually navigate our architectural models.

Our approach reveals that some sources of architectural mismatch may be prevented through architecture modeling and analysis, and provides further proof that OTS reuse must be planned. Architectural models are the necessary first step as they provide a framework for understanding and reasoning about the gross properties of systems, whether applications or environments.

8 ACKNOWLEDGMENTS

We wish to acknowledge the following individuals for their participation in the work described in this paper. ArchStudio 1.0 was developed by P. Oreizy and N. Medvidovic. DRADEL was developed by N. Medvidovic. ArchStudio 2.0 was developed P. Oreizy, R. Khare, M. Guntersdorfer, K. Nies, E. Dashofy, Y. Kanomata, R. Natarajan, A. Hitomi, R. Klashner, L. Pan, M. Dias, M. Vieira, S. Devanathan, and J. Robbins. We also wish to acknowledge the important contributions of G. Johnson and G. Brannum.

Effort sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-94-C-0195, F30602-97-2-0021, and F30602-99-C-0174. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory or the U.S. Government.

9 REFERENCES

1. R. J. Allen, D. Garlan, and J. Ivers. Formal Modeling and Analysis of the HLA Component Integration Standard. In *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, Orlando, FL, November 1998.
2. D. Batory, L. Coglianese, S. Shafer, and W. Tracz. The ADAGE Avionics Reference Architecture. In *Proceedings of AIAA Computing in Aerospace 10*, San Antonio, TX, 1995.
3. D. Batory and S. O'Malley. The Design and Implementation of

Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, October 1992.

4. T. Bray, J. Paoli, and C. M. Sperberg-McQueen, eds. Extensible Markup Language (XML) 1.0. World Wide Web Consortium Recommendation, 1998.
5. G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
6. M. R. Cagan. The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, June 1990.
7. E. M. Dashofy, N. Medvidovic, and R. N. Taylor. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
8. R. DeLine. Avoiding Packaging Mismatch with Flexible Packaging. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
9. E. Di Nitto and D.S. Rosenblum. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. To appear in *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
10. D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch, or, Why It's Hard to Build Systems out of Existing Parts. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, WA, April 1995.
11. D. Garlan, J. Ockerbloom, and D. Wile. Towards an ADL Toolkit. EDCS Architecture and Generation Cluster, December 1998. http://www.cs.cmu.edu/~spok/adl/index.html
12. A. Julienne and B. Holtz. *Tooltalk and Open Protocols: Inter-Application Communication*. SunSoft Press/Prentice Hall, April 1993.
13. R. Kadia. Issues Encountered in Building a Flexible Software Development Environment: Lessons Learned From the Arcadia Project. In *Proceedings of ACM SIGSOFT '92: Fifth Symposium on Software Development Environments*, Tyson's Corner, VA, December 1992.
14. M. Karasick. The architecture of Montana: an open and extensible programming environment with an incremental C++ compiler. In *Proceedings of The Sixth International Symposium on the Foundations of Software Engineering*, Orlando, FL, November 1998.
15. R. Khare and A. Rifkin. The Origin of (Document) Species. In *Proceedings of the 8th International WWW Conference*, published as Computer Networks and ISDN Systems, Volume 30 (1998), issues 1-7.
16. D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, September 1995.
17. M. J. Maybee, D. H. Heimbigner, and L. J. Osterweil. Multilanguage Interoperability in Distributed Systems: Experience Report. In *Proceedings of the Eighteenth International Conference on Software Engineering*, Berlin, Germany, March 1996.
18. N. Medvidovic. Architecture-Based Specification-Time Software Evolution. Ph.D. Dissertation, University of California, Irvine, December 1998.
19. N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, San Francisco, CA, October 1996.
20. N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability* and *Proceedings of the 1997 International Conference on Software Engineering*, Boston, MA, May 1997.
21. N. Medvidovic and D. S. Rosenblum. Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. In *Proceedings of the First Working IFIP Conference on Software Architecture*, San Antonio, TX, February 1999.
22. N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
23. N. Medvidovic and R. N. Taylor. Exploiting Architectural Style to Develop a Family of Applications. *IEE Proceedings Software Engineering*, October-December 1997.
24. Metamata IDE. Metamata Corp. http://www.metamata.com/
25. R. T. Monroe. Armani Language Reference Manual. Technical Report CMU-CS-98-163, Carnegie Mellon University School of Computer Science, October 1998.
26. R. Natarajan and D. S. Rosenblum. Merging Component Models and Architectural Styles, *Proc. Third Int'l Software Architecture Workshop*, Lake Buena Vista, FL, Nov. 1998.
27. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering*, April 1998, Kyoto, Japan.
28. P. Oreizy, N. Medvidovic, R. N. Taylor, and D. S. Rosenblum. Software Architecture and Component Technologies: Bridging the Gap. In *Proceedings of the Workshop on Compositional Software Architectures*, Monterey, CA, January 1998.
29. P. Oreizy. Issues in the Runtime Modification of Software Architectures. Technical Report UCI-ICS-96-35, Department of Information and Computer Science, University of California, Irvine, August 1996.
30. R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, Inc., 1996.
31. D. E. Perry. The Inscape Environment. In *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh, PA, May 1989.
32. J. Purtilo. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, January 1994.
33. Rational Software Corporation. *Rational Rose 98: Using Rational Rose*.
34. S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software, July 1990.*
35. J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum. Integrating Architecture Description Languages with a Standard Design Method. In *Proceedings of the 20th International Conference on Software Engineering*, April 1998, Kyoto, Japan.
36. J. E. Robbins and D. F. Redmiles. Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML. *Construction of Software Engineering Tools (CoSET'99).*
37. J. E. Robbins, D. M. Hilbert, and D. F. Redmiles. Extending Design Environments to Software Architecture Design. In *Proceedings of the 1996 Knowledge-Based Software Engineering Conference (KBSE)*, Syracuse, NY, September 1996.
38. D. Rothwell. Java Object Persistence Package. http://www.magna.com.au/
39. R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, New York, NY, 1997.
40. M. Shaw. Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging. In *Proceedings of IEEE Symposium on Software Reusability*, April 1995.
41. Sun Microsystems, Inc. Enterprise Java Beans 1.1, Draft Specification. http://java.sun.com/products/ejb/newspec.html
42. Sun Microsystems, Inc. Remote Method Invocation. http://java.sun.com:80/products/jdk/rmi/index.html
43. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI

Software. *IEEE Transactions on Software Engineering*, June 1996.

44. W. Teitelman and L. Masinter. The Interlisp Programming Environment. *IEEE Computer*, April 1981.
45. I. Thomas. Tool Integration in the Pact Environment. In *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh, PA, May 1989.
46. D. Yakimovich, J. M. Bieman, and V. R. Basili. Software Architecture Classification for Estimating the Cost of COTS Integration. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
47. Xerox PARC. ILU — Inter-Language Unification. ftp://ftp.parc.xerox.com/pub/ilu/ilu.html