On the Role of Middleware in Architecture-Based Software Development

Nenad Medvidovic

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

Abstract

Software architectures promote development focused on modular functional building blocks (components), their interconnections (configurations), and their interactions (connectors). Since architecture-level components often contain complex functionality, it is reasonable to expect that their interactions will be complex as well. Middleware technologies such as CORBA, COM, and RMI, provide a set of predefined services for enabling component composition and interaction. However, the potential role of such services in the implementations of software architectures is not well understood. Furthermore, components adhering to one middleware standard cannot readily interact with those adhering to another. In order to understand the role and tradeoffs among middleware technologies in implementing architectures and enable component interoperability across middleware platforms, we have investigated a set of techniques and conducted case studies involving a particular architectural style, C2, and its implementation infrastructure. In particular, by encapsulating middleware functionality within C2's explicit software connectors, we have been able to couple C2's existing benefits such as component interchangeability, substrate independence, and structural guidance with new capabilities of multi-lingual, multi-process, and distributed application development in a manner that is transparent to architects. Furthermore, we have demonstrated the utility of our connector-based approach in enabling components implemented on top of different middleware platforms to interoperate. Though several details of our approach derive from the characteristics of the C2 style, we believe that a number of lessons learned are more generally applicable. We argue that these lessons can help form a broader research agenda for coupling the modeling power of software architectures with the implementation support provided by middleware.

1. Introduction

The software systems of today are rapidly growing in number, sophistication, size, complexity, amount of distribution, and number of users. This information technology boom has been fueled by the increased affordability of hardware and the evolution of the Internet from a novelty gadget of the "technological elite" to a critical world-wide resource. As a result, the demand for software applications is far outpacing our ability to produce them, both in terms of their sheer numbers and their desired quality [Sta98]. Most notable about current software engineering practice is the continued preponderance of ad-hoc development approaches, driven by industry needs, commercial interests, and market pressures, rather than well-understood scientific principles. The magnitude of this situation has even been recognized at the highest levels of the U.S. Government, reflected in the recent President's Information Technology Advisory Committee (PITAC) report [P99]. In particular, the PITAC report identified several issues of large-scale software development, including component-based development, software reuse, and software interoperability, as major software engineering challenges.

These issues have been extensively explored in the past decade, resulting in numerous commercial component interoperability or *middleware* technologies that have been adopted as de facto standards (e.g., CORBA [OHE96], (D)COM [Ses97], OLE [Cha96], ActiveX [Cha96], Enterprise JavaBeans (EJB) [FFCM99], Java RMI [Sun], DCE [Sch93], SoftBench [Cag90], ToolTalk [JH93]), as well as a number of widely-studied, research-oriented middleware technologies (Q [MHO96], Field [Rei90], Polylith [Pur94], JEDI [CDF98], SIENA [CDRW98]). One can use any one of these technologies to develop software systems from existing components more quickly and reliably than was generally possible in the past. Yet ironically, the proprietary nature of these middleware technologies has served to hinder interoperability between components developed according to different technologies. For example, the developers of COM components must modify or reimplement those components for use in a system based on CORBA. Moreover, even com-

ponents implemented using different flavors of CORBA may not be interoperable. The result is a highly fragmented software component marketplace that ultimately impedes the ability of software organizations to develop systems with the highest possible quality and reliability, at the lowest possible cost. Another problem area has been the training of software engineers in the principles of component-based software development: it is currently mired in the details and peculiarities of a few chosen technologies, instead of focusing on underlying common principles and mechanisms.

Another research and development thrust, actively pursued in parallel with that on middleware, has been software development with an explicit focus on common *architectural* idioms [PW92, SG96, MT00]. In particular, software architecture research is directed at reducing the costs and improving the quality of applications by shifting the development focus from lines-of-code to coarser-grained architectural elements (components and connectors) and their overall interconnection structure (configurations). Additionally, architectures separate computation in a system (performed by components) from interaction among the components (facilitated by connectors). This enables developers to abstract away the unnecessary details and focus on the "big picture:" system-level structure and behavior, high-level communication protocols, component deployment, and so forth. Software architects also have at their disposal a number of architectural styles—collections of recurring structural, behavioral, and interaction patterns—with well-understood properties.

Architectures and middleware address similar problems—large-scale, component-based development—but at different stages of the development lifecycle. While architecture is an early model of a system that highlights the system's critical *conceptual* properties using high-level abstractions, middleware enables that system's realization and ensures the proper composition and interaction of the *implemented* components. Most existing architecture modeling and analysis approaches have suffered from the inability to map architectural decisions to the system's implementation in an automated and property-preserving manner [MT00]. At the same time, software development based purely on middleware can, in many ways, be regarded as the "assembly programming" of software composition [OMTR98]: a middleware technology provides no support for determining the application's structure and behavior, selecting the needed components, or interconnecting the components into the desired topologies.

The relationship between the two areas and their respective shortcomings suggest the possibility of coupling architecture modeling and analysis approaches with middleware technologies in order to get "the best of both worlds." Given that architectures are intended to describe systems at a high-level of abstraction, directly refining an architectural model into a design or implementation may not be possible. One reason is that the decision space rapidly expands with the decrease in abstraction levels: at the design level, constructs such as classes with attributes, operations, and associations, instances of objects collaborating in a scenario, and so forth, are identified; the implementation further requires the selection and instantiation of specific data structures and algorithms, interoperation with existing libraries, deployment of modules across process and machine boundaries, and so forth. One proposed solution to this problem has been to provide mechanisms for refining an architectural model into its implementation via a sequence of intermediate models [MQR95, LV95, RMRR98, AM99]. However, the resulting approaches have had to trade off the engineer's confidence in the fidelity of a lower-level model to the higher-level one against the practicality of the adopted technique [MT00]. Furthermore, to a large extent, the existing refinement approaches have failed to take advantage of a growing body of existing (implemented) components that may be reusable "as is."

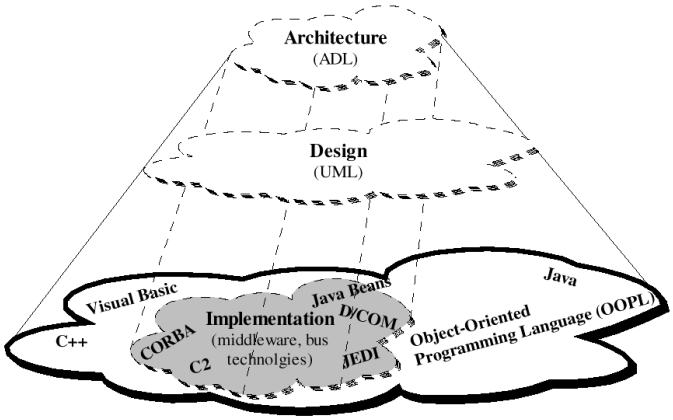


Figure 1. The decision space grows as a system is refined from an architecture to an implementation. Middleware technologies can be employed to bound the implementation space and render the mapping from architecture to implementation more tractable.

This paper pursues another strategy, depicted in Figure 1. The goal is to bound the target (implementation) space to a specific, well-defined subspace by employing (a set of) middleware technologies. The paper presents a technique for exploiting a particular architectural construct, software connector, to achieve the

desired result in a manner that minimizes the effect of the chosen middleware on the interacting components. Indeed, our approach enables, e.g., CORBA components to interact via, e.g., Java's RMI in principle. We have conducted a series of case studies to validate our hypothesis. A specific architectural style, C2, has been used as the basis for this investigation [TMA+96, MRT99]. Our initial results are promising and indicate that a successful marriage of architecture- and middleware-based techniques and technologies is indeed possible. At the same time, this initial experience also points to certain deficiencies of basing our approach solely on connectors. We use these deficiencies as the motivation for a broader study of component-based development, interoperability, and the relationship between middleware and architectures. These issues comprise a research agenda that frames our future work.

The remainder of the paper is organized as follows. Section 2 provides a brief description of the C2 architectural style and the connectors it employs. Section 3 describes our approach to providing "middleware-enabled" connectors and discusses how those connectors are used to enable the interaction of components compliant with heterogeneous middleware. A discussion of lessons learned and future work rounds out the paper.

2. Overview of the C2 Style

We chose the C2 architectural style as a foundation upon which to initially explore the issues of integrating middleware with software architectures. The C2 style is a good fit for this task for several reasons. C2 has an explicit notion of software connectors as first-class entities that handle component interactions. The style provides facilities for exploring specific properties of connectors such as filtering, routing, and broadcasting, which are also typically provided by middleware. Further, the style is well-suited to a distributed setting, allowing us to leverage the networking capabilities of middleware technologies. C2 supports a paradigm for composing systems in which components may be running in a distributed, heterogeneous environment without shared address spaces, architectures may be changed dynamically, multiple users may be interacting with the system, multiple user interface toolkits may be employed, and multiple media types may be involved.

The C2 style can be informally summarized as a network of concurrent components hooked together by connectors. All communication among components in an architecture is achieved by exchanging messages via connectors. Message-based communication is extensively used in distributed environments for which C2 is suited. Each component may have its own thread(s) of control. This simplifies modeling and implementation of multi-component, multi-user, and concurrent applications and enables exploitation of distributed platforms. A proposed architecture is distinct from its implementation(s) so that it is indeed possible for components to share threads of control. The separation of architecture from implementation is a key aspect of our approach to integrating middleware technologies with C2, as discussed in Section 3. Finally, there is no assumption of a shared address space among C2 components. Any premise of a shared address space could be unreasonable in an architectural style that allows composition of heterogeneous, highly distributed components, developed in different languages, with their own threads of control, and internal architectures.

As already discussed, central to our approach are *software connectors*, architecture-level abstractions and facilities that bind components together into an architecture and facilitate their interactions [SG96]. Connectors manifest themselves in a software system as shared variable accesses, table entries, buffers, procedure calls, remote procedure calls (RPC), network protocols, pipes, and so on [SG96, MT00]. In large, and especially distributed systems, connectors become key determinants of system properties, such as performance, resource utilization, global rates of flow, and security [MMP00]. Abstracting and encapsulating interaction details within architectural connectors has shown a lot of promise in helping to address traditional software development challenges: scalability, distribution, concurrency, runtime adaptability, code mobility, and so forth [SDK+95, SG96, AG97, OMT98, KM98].

We have extensively employed connectors in our previous research to support software modeling, analysis, generation, evolution, reuse, and heterogeneity [TMA+96, MOT97, MT97, OMT98, MRT99]. In particular, connectors in the C2 style may be connected to any number of components as well as other connectors. A connector's responsibilities include message routing, broadcast, and filtering. C2 connectors also support adaptation of messages to accommodate mismatched interfaces and interaction protocols [TMA+96, MOT97].

To support implementation of C2-style architectures, we have developed an extensible framework of abstract classes for concepts such as components, connectors and messages, as shown in Figure 2. This framework is the basis of development and middleware integration in C2.[1] As we will discuss, the framework encapsulates all access to integrated middleware, ensuring that the use of middleware is transparent to an architect, and, indeed, to the implementor of a particular architecture. The framework, together with any employed middleware, directly enables the support for automatic (partial) application generation from an architecture in our DRADEL tool suite [MRT99]. The framework implements interconnection and message passing protocols. Components and connectors used in C2-style applications are subclassed from the appropriate abstract classes in the framework. This guarantees their interoperability, eliminates many repetitive programming tasks, and allows developers of C2 applications 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. To date, the framework has been implemented in C++ and Java.

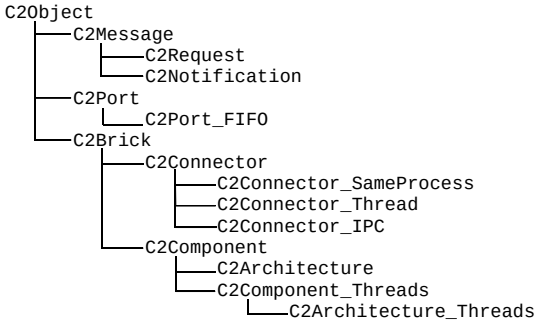
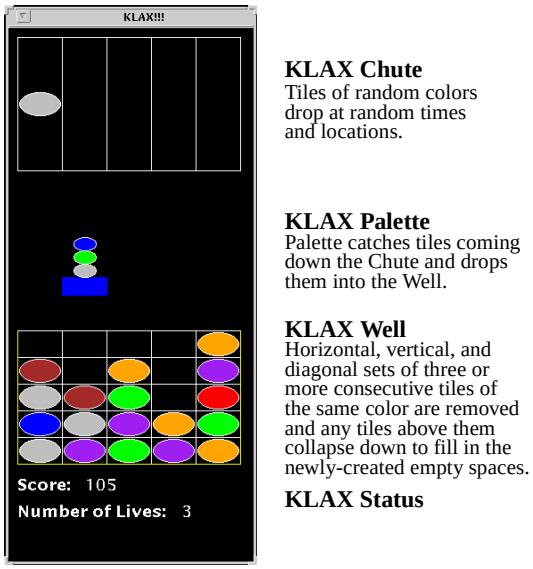


Figure 2. C2 implementation framework.

Example C2-Style Application

An application that was extensively used in our investigation of middleware integration with C2 is a version of the video game KLAX. A description of the game is given in Figure 3. This particular application was chosen because game play imposes some real-time constraints on the application, bringing performance issues to the forefront. The architecture of the application is depicted in Figure 4. The components that make up the KLAX game can be divided into three logical groups. At the top of the architecture are the components that encapsulate the game's state. The game state components respond to request messages and emit



KLAX Chute
Tiles of random colors drop at random times and locations.

KLAX Palette
Palette catches tiles coming down the Chute and drops them into the Well.

KLAX Well
Horizontal, vertical, and diagonal sets of three or more consecutive tiles of the same color are removed and any tiles above them collapse down to fill in the newly-created empty spaces.

KLAX Status

Figure 3. A screenshot and description of our implementation of the KLAX video game.

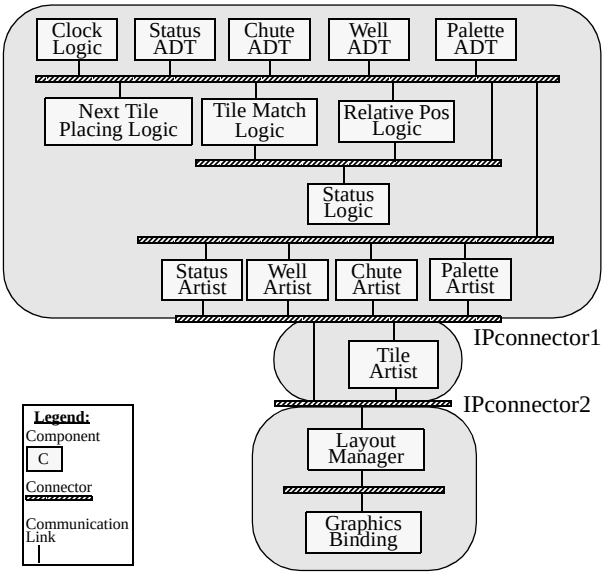


Figure 4. Conceptual C2 architecture for KLAX. Shaded ovals represent process/machine boundaries.

1. It has been argued by others [DR99, YBB99] that this framework is similar to commercial middleware platforms, such as CORBA and COM.

notifications of internal state changes. Notification messages are directed to the next level, where they are received by both the game logic components and the artist components. The game logic components request changes of game state in accordance with game rules and interpret the change notifications to determine the state of the game in progress. The artist components also receive notifications of game state changes, causing them to update their depictions. Each artist maintains the state of a set of abstract graphical objects which, when modified, send state change notifications in the hope that a lower-level graphics component will render them on the screen. The *GraphicsBinding* component receives all notifications about the state of the artists' graphical objects and translates them into calls to a window system. User events, such as a key press, are translated by *GraphicsBinding* into requests to the artist components.

We used the deployment profile shown in Figure 4 to examine the issues in using middleware technologies to implement architectures, although a number of other deployments are clearly possible. Two KLAX implementations were built using the C++ and Java versions of the framework shown in Figure 2. A variation of the architecture shown in Figure 4 was also used as the basis of a distributed, multi-player KLAX application implemented using the Java framework. In this variation, each player executes a copy of KLAX on his own machine. A player competes against other game participants by issuing requests to a remote, central *GameServer* to, e.g., add an extra tile to a given player's chute. The *GameServer*, in turn, notifies the appropriate players of the changes to their states in response to their opponent's action.

Performance of the different implementations of KLAX easily exceeds human reaction time if the *ClockLogic* component is set to use short time intervals. Although we have not yet tried to optimize performance, benchmarks indicate that the C++ framework can send 1200 simple messages per second when sending and receiving components are in the same process, with the Java framework being somewhat slower. In single-player KLAX, a keystroke typically causes 10 to 30 message sends, and a tick of the clock typically causes 3 to 20 message sends. The efficiency of message exchange across process and/or machine boundaries is a function of the underlying mechanism (i.e., middleware) used to implement the given inter-process/machine connector.

3. Employing Middleware to Implement Software Connectors

The predominant focus of component-based software development has been on designing, selecting, adapting, implementing, and integrating computational elements—software components. Since components may contain complex functionality, it is reasonable to expect that their interactions will be complex as well. Existing middleware technologies have addressed component interaction via a predefined set of capabilities (e.g., RPC) that is typically not intended to be extensible. These capabilities are usually packaged into a facility, such as an object request broker (ORB), a message broker (MOM), or a software bus [Rei90, Cag90, Pur94, ISG97, IMA98].[2] As foreshadowed above, our approach to coupling the benefits of architecture- and middleware-based development approaches will focus on component interactions and leverage ORBs. Thus, our primary hypothesis is that connectors are the proper abstraction for integrating architectures and middleware. This hypothesis is motivated by the recognition that, though different, ORBs and connectors share several key characteristics. Indeed, an ORB can be viewed as an implementation of a sophisticated connector that supports a large set of interaction protocols and services. This perspective suggests our general approach: a software architect designs an application in the most appropriate and intuitive way, selects one or more middleware platforms that are suitable for implementing the architecture, maps the architecture to a particular topology (sometimes imposed by the middleware [DR99]), selects the needed set of off-the-shelf (OTS) components, and uses the appropriate ORBs to implement the connectors in the architecture.

A simple example that illustrates this strategy is shown in Figure 5. A conceptual architecture of a system is shown at the top. In this case, the C2 style mandates that information flow only up and down through the connector (e.g., *Comp1* and *Comp3* cannot directly interact, while *Comp1* and *Comp2* can). Assume we want to implement the architecture with components bound to a given middleware and to distribute the implementation over three locations. The middle diagram depicts the resulting solution: the single ORB ensures the cross-machine interaction of its attached components, but not the topological and interaction constraints imposed by the style. Also note that, if the four components are not all built on top of the same

2. In the interest of simplicity, and as is commonly done in literature, we will refer to the different interaction facilities provided by middleware as "ORBs" in the remainder of this section.

middleware infrastructure (e.g., CORBA), the engineers will depend on existing point solutions or will have to develop the needed inter-middleware bridge as yet another point (likely, ad-hoc) solution.

Our approach, depicted on the bottom of Figure 5, enables a more principled way of integrating architectures and middleware. The approach also allows bridging middleware, i.e., the interaction of components adhering to different middleware standards (e.g., CORBA and COM). We keep connectors an explicit part of a system's implementation infrastructure, as discussed in the context of Figure 2. Each component thus only exchanges information with a connector to which it is attached; in turn, the connector will (re)package that information and deliver it to its recipients using one or more middleware technologies. Each such "middleware enabled" connector is a variant of a standard connector (recall Figure 2); it changes the underlying mechanism for marshalling and delivering messages, but externally appears unchanged. This approach minimizes the effects on a given component of varying application deployment profiles and of using components that adhere to heterogeneous middleware standards. Note that, unlike the "middleware-only" solution shown in the middle diagram, the bottom diagram of Figure 5 also preserves the topological and stylistic constraints of the application. Furthermore, the connector allows *Comp1* and *Comp2* to interact efficiently, using the in-process (i.e., C2 implementation framework) mechanisms while, at the same time, interacting with *Comp3* and *Comp4* using the inter-process (i.e., OTS middleware) mechanisms.

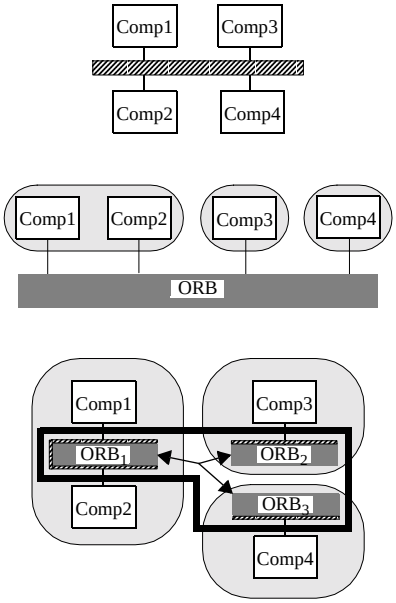


Figure 5. Realizing a software architecture (*top*) using a middleware technology (*middle*) and an explicit, middleware-enabled software connector (*bottom*).

We have developed and used two different techniques that enable us to use middleware in the context of an architecture as outlined in Figure 5. Both of these techniques consist of implementing a single conceptual software connector using two or more actual connectors that are linked across process or network boundaries via a given middleware technology. Each actual connector thus becomes a segment of a single "virtual connector." All access to the underlying middleware technology is encapsulated entirely within the abstraction of a connector, meaning that it is unseen by both architects and developers, as well as the interacting components.

We call the first approach "lateral welding." It is depicted in the top diagram of Figure 6. Messages sent to any segment of the multi-process connector are broadcast to all other segments via the underlying middleware. Upon receiving a message, each segment has the responsibility of filtering and forwarding the message to components in its process as appropriate.

While the lateral welding approach allows us to "vertically slice" a C2 application, we also developed an approach to "horizontally slice" an application, as shown in the bottom diagram of Figure 6. This approach is similar to the idea of lateral welding: a conceptual connector is broken up into top and bottom segments, each of which exhibits the same properties as a single-process connector to the components attached above and below it, respectively. However, the segments themselves are joined using the appropriate middleware.

These two techniques have been implemented using five different middleware technologies: ILU [Xerox], VisiBroker CORBA [Inpr], RMI [Sun],

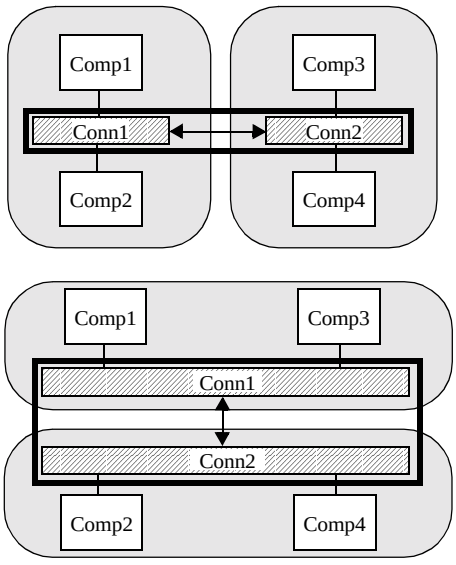


Figure 6. Connectors as a primary vehicle for interprocess communication. A single conceptual connector can be "broken up" vertically (top) or horizontally (bottom) for this purpose. Shaded ovals represent process boundaries. Each connector encapsulates an ORB (elided for simplicity).

Polylith [Pur94], and Q [MHO96]. The resulting connectors are arbitrarily composable to support any deployment profile or middleware platform. The motivation for such a composition is that different middleware technologies may have unique benefits. By combining multiple such technologies in a single application, the application can potentially obtain the benefits of all of them. For instance, a middleware technology that supports multiple platforms but only a single language, such as RMI, could be combined with one that supports multiple languages but a single platform, such as Q, to create an application that supports both multiple languages and multiple platforms.

The advantages of combining multiple middleware technologies within software connectors are manifold. In the absence of a single panacea solution that supports all required platforms, languages, and network protocols, the ability to leverage the capabilities of several different middleware technologies significantly widens the range of applications that can be implemented within an architectural style such as C2. We believe that the key challenge is to develop the inter-middleware "bridge" to allow two or more technologies to exchange data; once the bridge is developed, it is usable indefinitely thereafter. We have tested this hypothesis by combining the lateral welding and horizontal slicing techniques from Figure 6 to implement a single conceptual connector in the KLAX application (recall Figure 4) using Xerox's ILU and Java's RMI middleware. An example of this combined binding method is shown in Figure 7: our approach creates a three-process "virtual connector" using two in-process C2 connectors to bind two multi-process connectors. Note that an alternative approach would have been to create a single implementation-level connector that supported both ILU and RMI. However, the approach we adopted is compositional and, therefore, more flexible, with a slight efficiency cost due to the addition of in-process connectors to bind the multi-process connectors.

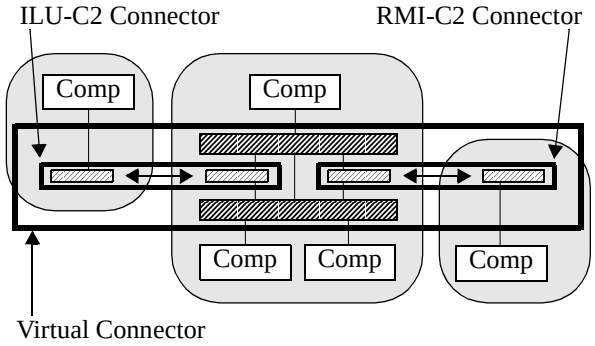


Figure 7. An example of a three-process C2 application using different middleware packages. A single virtual connector is implemented with two in-process and two multi-process connectors. The in-process connectors facilitate message passing between the multi-process connectors. Shaded ovals represent process boundaries.

In all the examples shown thus far, the components have been treated as homogeneous (i.e., they adhere to the same architectural style, are implemented in a single language, and/or on top of the same platform). While the underlying mechanisms employed to enable the components' interactions have varied, each middleware-enabled connector discussed above exports a message-based interface understood by C2 components. It is important to point out that this does not mean that we have adopted message-passing as *the* solution to the problem of software interoperability, replacing the proprietary interaction mechanisms employed by various middleware with one of our own. The approach described above provides the implementation power and flexibility to construct connectors that enable the interaction of heterogeneous components across middleware platforms. Again, the challenge is to isolate the inter-middleware bridge inside a connector, such that components on both sides of the connector can assume that they are still residing in a homogeneous (e.g., C2-only or CORBA-only) environment. One such connector we have constructed enables the interaction of C2 and CORBA components. So, for example, in the bottom diagram shown in Figure 6, *Comp1* and *Comp3* would be C2 components, while *Comp2* and *Comp4* are VisiBroker CORBA components; *Comp1* and *Comp3* assume that they are interacting with other C2 components via a C2 connector, while *Comp2* and *Comp4* assume that they are interacting with other CORBA components via an ORB.

Another aspect of heterogeneity is the components' implementation in different languages. For example, we have used Q to enable the interaction among components written in C++ and Ada. Specifically, one configuration of the KLAX application (recall Figure 4) involved the *TileArtist* component implemented in Ada, while the rest of the architecture was implemented using C2's C++ framework. The connectors on top and bottom of *TileArtist* used Q to bridge the two languages.

We should note that these two examples (interoperability between C2 and CORBA, and Ada and C++, respectively) did not require specialized solutions, but were simple variations of our solution depicted in

Figure 6. If we consider the C2 implementation framework to be a custom middleware platform for C2-style applications as argued by [DR99, YBB99], then the encapsulation of third-party ORBs inside C2's connectors is nothing more than a composition of two or more inter-middleware bridges. For example, interoperability between C2 and CORBA is achieved by composing a C2-CORBA bridge with a CORBA-C2 bridge, as depicted in Figure 8. Each such bridge (e.g., the top half of the figure) may be used independently as an inter-middleware connector. Furthermore, it is possible to invert the bridges such that, in the example shown in Figure 8, the underlying mechanism for CORBA component interaction is C2 message passing.

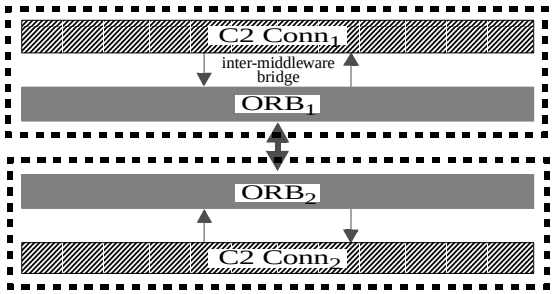


Figure 8. A middleware-enabled C2 connector is a composition of two inter-middleware connectors.

4. Conclusions

Ensuring interoperability is a critical issue in the quickly-emerging marketplace of heterogeneous software components. The unfortunate reality is that competing middleware vendors vying for market dominance have ended up constructing incompatible, proprietary component and middleware standards. The current situation can be characterized as a "component tower of Babel:" components "speaking" the same language are interoperable, while those "speaking" different languages are not. Although several technologies to bridge middleware platforms have been proposed and implemented (e.g., COM-EJB [Ver99] or CORBA-COM [Con98, Gar99]), these are usually pairwise solutions that are directly dependent upon the characteristics of the involved middleware and provide little, if any, guidance as to how a similar outcome can be achieved with a different set of middleware platforms. Furthermore, although "architecture" is a frequently used term in the context of middleware technologies, middleware providers do not focus on guiding developers to achieve a suitable architecture for their systems, but rather propose a solution based solely around their implementation-level infrastructures. This "one size fits all" mentality is also reflected in the failure of middleware providers to acknowledge that their technologies impose certain stylistic constraints on component composition and interaction [DR99].

This paper has presented an approach that has the potential to remedy this situation. The approach directly exploits architectural constructs (styles and connectors) and provides a principled, repeatable solution to the problem of bridging middleware. We have employed sets of both commercial (RMI, VisiBroker) and research (ILU, Polylith, Q) OTS technologies to test our hypothesis that software connectors are the proper mechanisms for supporting middleware-based implementation of architectures. Our results to date are very promising: note that the details of the examples discussed above varied widely, yet the same basic integration techniques, shown in Figures 6 and 8, were used across all of them. At the same time, we acknowledge that these results are not definitive and that there are several issues that must be studied in order to render this work more general and assess the true extent of its applicability:

- We must have a better understanding of the underlying properties, both shared and proprietary, of middleware technologies in order to provide general, reusable, potentially automatable solutions.
- We must understand the role of and constraints imposed by architectural styles other than C2 in enabling middleware integration.
- Finally, we must understand the issues in applying this technique to connector types beyond message passing. To this end, we intend to leverage our recent work that has resulted in a comprehensive taxonomy of software connectors [MMP00].

To address the issues identified above, our future work will involve a more comprehensive approach to investigating the role of middleware in implementing software architectures and exploiting architectural abstractions and mechanisms to enable cross-middleware interoperability. We have initiated a multi-institution project [MGR00] with three facets: improving our understanding of the relationship between middleware and software architectures [DR99], analyzing and codifying the underlying building blocks common to all middleware platforms [KG98, KG99], and further exploring the role and limitations of software connectors in achieving general solutions to the problem of heterogeneous component-based development. A criti-

cal issue we must address is the extent to which our solutions must be pairwise (in the sense of, for example, requiring N^2 inter-middleware connectors for N middleware technologies). Our initial results, discussed in Section 3, indicate that it is possible to provide compositional inter-middleware connectors such that pairwise solutions can be avoided. We intend to leverage our codification of middleware services in constructing general connectors that can accommodate multiple, arbitrarily chosen technologies at once. Ideally, we would be able to provide a single, general solution for each technology; the solutions would be reusable indefinitely thereafter, even as new interoperability technologies are defined. We must also evaluate the tradeoffs (such as reliability, quality of service, and performance) between the different cross-middleware interoperability approaches that we investigate.

The benefits of this work will accrue from large amounts of legacy software at one's disposal and the knowledge of what (types of) components can be (re)used in an application and under what circumstances. In turn, in tandem with related academic and industry-led work, this research has the potential to influence the next generation of interoperability standards and provide the underpinning of a true, open component marketplace.

5. References

[AG97] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 213-249, July 1997.

[AM99] M. Abi-Antoun and N. Medvidovic. Enabling the Refinement of a Software Architecture into a Design. In *Proceedings of The Second International Conference on The Unified Modeling Language (UML'99)*, Fort Collins, CO, October 1999.

[Cag90] M. R. Cagan. The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, vol. 1, no. 3, pp. 36–47, June 1990.

[CDF98] C. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an Event-Based Infrastructure to Develop Complex Distributed Systems. In *Proceedings of the 20th International Conference on Software Engineering*, Kyoto, Japan, April 1998.

[CDRW98] A. Carzaniga, E. Di Nitto, D. Rosenblum, and A. L. Wolf. Issues in Supporting Event-Based Architectural Styles. In *Proceedings of the Third International Worshop on Software Architectures*, Orlando, FL, November 1998.

[Cha96] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, Redmond, WA, 1996.

[Con98] R. Condon. ICL bridges COM, CORBA. Computerworld, March 1998. http://www2.computerworld.com/home/online9697.nsf/all/980317icl1DA8A

[DR99] E. Di Nitto and D. S. Rosenblum. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, May 1999.

[FFCM99] D. Flanagan, J. Farley, W. Crawford, and K. Magnusson. Java Enterprise in a Nutshell. O'Reilly, 1999.

[Gar99] D. Gardner. New ORB Pledges Faster COM-CORBA Integration. *Info World Electric*, January 1999. http://www.idg.net/crd_orb_65690.html

[IMA98] International Middleware Association. A Middleware Taxonomy: Middleware in the World of Distributed Computing. Technical Report, 1998.

[Inpr] Inprise Corp. VisiBroker. http://www.borland.com/visibroker/

[ISG97] International Systems Group, Inc. Middleware: The Essential Component for Enterprise Client/Server Applications. February 1997.

[JH93] A. Julienne and B. Holtz. *Tooltalk and Open Protocols: Inter-Application Communication*. SunSoft Press/ Prentice Hall, April 1993.

[KG98] R. Keshav and R. Gamble. Toward a Taxonomy of Integration Strategies. *3rd International Software Architecture Workshop*. November 1998.

[KG99] A. Kelkar and R. Gamble. Understanding the Architectural Characteristics Behind Middleware Choices. *Proceedings of the 1st International Conference on Information Reuse and Integration*. November 1999.

[KM98] J. Kramer and J. Magee. Analysing Dynamic Change in Software Architectures: A Case Study. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pp. 91-100, Annapolis, MD, May 1998.

[LV95] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717-734, September 1995.

[MGR00] N. Medvidovic, R. F. Gamble, and D. S. Rosenblum. Towards Software Multioperability: Bridging Heterogeneous Software Interoperability Platforms. To appear in *Proceedings of the Fourth International Software Architecture Workshop (ISAW-4)*, Limerick, Ireland, June 2000.

[MHO96] M. J. Maybee, D. H. Heimbigner, and L. J. Osterweil. Multilanguage Interoperability in Distributed Systems: Experience Report. In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, March 1996.

[MMP99] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. To appear in *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, June 2000.

[MOT97] 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.

[MQR95] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 356-372, April 1995.

[MRT99] 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*, pp. 44–53, Los Angeles, CA, May 1999.

[MT97] N. Medvidovic and R. N. Taylor. Exploiting Architectural Style to Develop a Family of Applications. *IEE Proceedings Software Engineering*, vol. 144, no. 5-6, pp. 237-248, October-December 1997.

[MT00] N. Medvidovic and R.N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. Accepted for publication in *IEEE Transactions on Software Engineering*, vol. 26, no. 1, January 2000.

[OMT98] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98*), pp. 177-186, Kyoto, Japan, April 1998.

[OMTR98] P. Oreizy, N. Medvidovic, R. Taylor, and D. Rosenblum. Software Architecture and Component Technologies: Bridging the Gap. *OMG-DARPA-MCC Workshop on Compositional Software Architectures*. Monterey, CA, January 1998.

[OHE96] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, 1996.

[P99] President's Information Technology Advisory Committee. Information Technology Research: Investing in Our Future. National Coordination Office for Computing, Information, and Communication, February 1999.

[PW92] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40-52, October 1992.

[Pur94] J. Purtilo. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 1, pp. 151-174, January 1994.

[Rei90] S. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, vol. 7, no. 4, July 1990.

[RMRR98] 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*, Kyoto, Japan, April 1998.

[Sch93] A. Schill, editor. *DCE — The OSF Distributed Computing Environment*. Proceedings of the International DCE Workshop, Karlsruhe, Germany, Springer Verlag, October 1993.

[SDK+95] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 314-335, April 1995.

[Ses97] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, 1997.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.

[Sta98] The Standish Group. The CHAOS Report, 1998. http://www.standishgroup.com/chaos.html; http://www.standishgroup.com/_compass/chao98.html

[Sun] Sun Microsystems, Inc. Remote Method Invocation. http://java.sun.com:80/products/jdk/rmi/index.html

[TMA+96] 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*, vol. 22, no. 6, pp. 390-406, June 1996.

[Ver99] C. Verbowski. Integrating Java and COM, Microsoft Corporation, January 1999. http://www.microsoft.com/java/resource/java_com.htm

[Xerox] Xerox PARC. ILU - Inter-Language Unification. ftp://ftp.parc.xerox.com/pub/ilu/ilu.html

[YBB99] 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, May 1999.