Software Architecture and Component Technologies: Bridging the Gap

Peyman Oreizy **Nenad Medvidovic** **Richard N. Taylor** **David S. Rosenblum**

Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
{peymano,neno,taylor,dsr}@ics.uci.edu
http://www.ics.uci.edu/pub/arch/

1 INTRODUCTION

Engineering large-scale software systems is fundamentally different from programming in the small. A programming language statement is inadequate as the unit of development. Instead, *components* must become the building blocks of software. Component-based development of software has become an area of intense research and commercial focus, resulting in several component interoperability models, such as CORBA, ActiveX, and JavaBeans. These models have the potential to help practitioners cope with the increasing complexity of software systems.

Software architecture is another promising approach to controlling software complexity. Software architecture research is directed at reducing costs of developing applications and increasing the potential for commonality between different members of a closely related product family. Architectures enable developers to center on the “big picture” in developing a system and to adopt a component-based development philosophy as opposed to always building a system from scratch. Architectures do this by making a software system’s structure explicit, separating the computation of components from their interactions in a system, and providing a high-level model of a system that can be manipulated and analyzed before any changes are effected in an actual implementation.

In this paper, we argue that the full benefit of component interoperability models can only be achieved if complemented by explicit architectural models. We first outline a set of driving factors in selecting an approach to engineering and evolving large, complex, distributed applications. We then evaluate a particular software architectural style, C2 [TMA+96], and differentiate it from component models based on these goals. Finally, we briefly discuss the issues which are likely to determine the role and prominence of an architectural approach in the formation of a software component marketplace.

2 REQUIREMENTS FOR COMPOSITIONAL SOFTWARE ARCHITECTURES

Any approach to effectively designing, implementing, and evolving large, complex, distributed systems, potentially from pre-existing, heterogeneous parts, must support at least the following goals:

- Allow communication about a system among multiple stakeholders: customers, architects, managers, component developers, system integrators, users, and so on.
- Allow understanding of and reasoning about a system at a level of abstraction above source code and closer to the stakeholders’ mental models of the system.
- Narrow the gap between system requirements, which are in the “problem” space, and software designs, which are in the “solution” space.
- Support reuse and families of applications as opposed to custom and “one of a kind” solutions.
- Enable codification of successful design and evolution properties from legacy projects.
- Allow upstream analysis to correct errors early and reduce costs associated with those errors.
- Allow reconfigurability of software both before and during runtime.
- Allow components of varying granularities, implemented in different programming languages.
- Support distributed, heterogeneous environments with multiple address spaces, threads of control, and operating system processes.

The architecture community has developed the consensus that, to achieve these properties, a technology must be employed which provides explicit, high level system models and support for capturing recurring properties of an application domain. Our experience indicates that software architectures, and particularly architectural styles that separate computation from communication through explicit communication elements, in fact provide such a technology. In the following section, we describe the C2 architectural style [TMA+96], which has been designed to support a large subset of the goals outlined above.

3 EXPERIENCE WITH THE C2 ARCHITECTURAL STYLE

The C2 style is primarily concerned with high level system composition issues, rather than particular component packaging approaches; the style can employ multiple component middleware technologies. Building blocks of C2 architectures are *components* (computational elements) and *connectors* (interconnection and communication elements). This separation of computation from communication enables the construction of flexible, extensible, and scalable systems that can evolve both before and during runtime. The style places no restrictions on the implementation language or granularity of components and connectors, potentially allowing it to use multiple interoperability technologies. Central to the C2 style is a principle of limited visibility or *substrate independence*: components are arranged in a layered fashion in a C2 architecture, and a component is completely unaware of components that reside “beneath” it. Substrate independence has a clear potential for fostering substitutability and reusability of components across architectures. Components communicate only by exchanging messages through connectors, which greatly simplifies the problem of control integration issues; this property also facilitates low-cost interchangeability of components to construct different members of the same family. Two components cannot assume that they will execute in the same address space; this eliminates complex dependencies, such as components sharing global variables, and simplifies modification of architectures. Conceptually, components run in their own thread(s) of control, allowing components with potentially different threading models to be integrated into a single application. Finally, a conceptual C2 architecture can be instantiated in a number of different ways. Many potential performance issues or variations in functionality can be addressed by separating the architecture from actual implementation techniques.

Most of these properties have been demonstrated in various applications built using the C2 style. C2 has been used successfully in several application domains. We are currently exploring its applicability to other domains. Examples of C2’s use are:

- *Family of video game*s — An architecture for a family of interactive video games was designed in the C2 style. A library of interchangeable components in multiple programming languages was produced and several off-the-shelf components were reused to enable over 500 variations of the video game. Several dozen such variations were built. We also demonstrated the ability to substitute components and change from one variation of the game to another “on the fly”.
- *Mission-critical logistics application* — The suitability of the style for building mission-critical applications was demonstrated by designing and implementing a cargo-routing application, and adding extensions to it at runtime.
- *Simple software development environment* — The tool suite used in designing and implementing C2 applications has itself been implemented in the C2 style; each tool in the environment is treated as a C2 component and all communication among them occurs via C2 connectors.
- *Avionics system simulation environment* — Northrop Grumman is currently designing a USAF B-2 aircraft simulation environment in the C2 style.

4 UTILIZING EXISTING COMPONENT MIDDLEWARE TECHNOLOGIES IN CONCERT WITH SOFTWARE ARCHITECTURE TECHNOLOGIES

Existing component middleware technologies, such as CORBA, ActiveX, and Java Beans, are *component-centric*: they are primarily concerned with standardizing external component properties—interfaces, packaging, binding mechanism, intercomponent communication protocols, and expectations regarding the runtime environment. Software architectures and styles, in contrast, are *system-centric*: they focus on specifying systems of communicating black-box components, analyzing resulting system properties, and generating “glue” code that binds system components.

Both are crucial aspects of component-based software development, yet there has been surprisingly limited interaction between the two domains. The different foci suggest a possible strategy for bridging the gap between the two domains: use existing component middleware technologies to implement systems modeled with architectural technology.

Several key technological challenges must be overcome before a seamless transition is possible:

- *a shared model combining existing architectural models and component middleware technologies* — The two domains use similar, but incompatible, models of components and component bindings. ActiveX, for example, supports multiple functional interfaces for each component, whereas most architectural modeling notations support a single functional interface per component. Many of these inconsistencies are revealed by comparing the interface definition languages (IDLs) of each domain. Other inconsistencies require careful comparison of the capabilities provided by the component infrastructure and architectural description language (ADL). ActiveX’s *dispatch interfaces* and CORBA’s *dynamic method invocation*, for example, enable components to dynamically locate, load, and invoke services of other components, allowing the system to evolve as new components are introduced. Some ADLs, in contrast, typically assume that a system’s architecture is static and does not evolve after system generation. The C2 ADL, among others, does provide dynamic modeling capabilities.
- *a mapping from architectural entities to the implementation components* — Components at the architectural-level have a potentially straightforward mapping to components at the implementation level. It is unclear how other architectural elements are mapped onto the implementation. An architectural connector providing a synchronous request-reply interaction maps naturally to CORBA’s static method invocation mechanism. More complex connectors, such as asynchronous message broadcast connectors, would most likely have to be implemented as separate CORBA components.

- *modeling infrastructure services* — The component interactions in a complex system constructed using existing component middleware technologies provide only a partial model of the system. This is because components typically make extensive use of the services provided by the middleware infrastructure (e.g., CORBA's persistence, events, and transaction services) and by the operating system, which are both critical to understanding the system. These services should also be represented at the architectural level.

These technological challenges are significant and complex, yet their solution does not guarantee a successful software component marketplace. We examine some of the other issues in the subsequent section.

5 A SOFTWARE COMPONENT MARKETPLACE

There are a myriad of factors beyond the architectural issues we have raised thus far that affect the commercial success of a component marketplace. Our previous work examines successful component marketplaces, including commercial markets such as Visual Basic VBXs, and non-commercial markets such as UNIX filters [WRMT95]. Several of the key *architectural* requirements exhibited from these successful markets include support for:

- *Multiple component granularities* - The architectural infrastructure must support components that are both small and large, from simple data structures to large databases. While most larger components would undoubtedly be constructed from smaller components, larger components can provide a more meaningful packaging of functionality for designers.
- *Substitutability of components* - The architectural infrastructure must provide support for removing one component and substituting it for an equivalent component. This allows competition based on features and price for a component.
- *Parameterized components* - The architectural infrastructure must support components that can be parameterized and customized during design. Ideally, the parameterization process should be easy to perform, facilitating the use of the component during design. A provision enabling the parameterization functionality to be removed for system release should be available to reduce the size of the application.
- *Component development in multiple programming languages* - Since different programming languages have strengths for different application domains, and since new languages emerge periodically, the architectural infrastructure should support components developed in different programming languages.
- *Component-specific help* - In order to reduce the barrier to use components, it should be possible for software designers and component users to receive help regarding the use of a particular component.
- *User interface composition* - While there are many components that do not present a user-interface, there are some domains and components that do have stereotypical user interfaces. The architectural infrastructure should support the composition of multiple component user interfaces into a single, uniform, integrated, user interface.
- *Easy distribution of components* - It should be easy to package and distribute a component. Ideally, the architectural infrastructure should support component packaging and distribution in various forms.
- *Support for multiple sales models* - Existing software has multiple sales models, ranging from single sale, single-user (isolated PC model), to single sale, multiple user (network-based PC model). The architectural infrastructure should not be biased to a particular sales model, and should ideally support different licensing and pricing models.

Additionally, the marketplace must satisfy certain requirements, such as supporting the task of locating relevant components. It should not be forgotten that many of the critical factors resulting in success are non-technical, such as monetary backing, support by key software vendors, companies willing to develop and purchase components, market timing, market strategy, a cost effective distribution channel, and pricing. However, addressing the technological issues presented here can substantially enhance the chances a given component architecture has in becoming a successful commercial marketplace.

6 CONCLUSION

Component middleware technologies alone do not adequately address certain system-wide aspects of engineering large, complex, distributed software systems. Software architecture research, on the other hand, typically has not focused on component development, packaging, and interoperability. These different but complementary foci indicate an opportunity for an effective marriage of the two areas, where one can couple the benefits of explicit architectural models with those of component interoperability models. Such a unified approach would form a solid basis on which a successful software component marketplace can built.

7 REFERENCES

[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, pages 390-406 (June 1996).

[WRMT95] E. J. Whitehead, Jr., J. E. Robbins, N. Medvidovic, and R. N. Taylor. Software Architecture: Foundation of a Software Component Marketplace. In David Garlan, ed., Proceedings of the First International Workshop on Architectures for Software Systems, pages 276-282, Seattle, WA, April 24-25, 1995.