Issues in the Runtime Modification of Software Architectures

Peyman Oreizy

Department of Information and Computer Science
University of California, Irvine, CA 92697
peymano@ics.uci.edu
http://www.ics.uci.edu/~peymano/

UCI-ICS-TR-96-35

August 1996

Abstract

Existing software architecture research has focused on static architectures, where the system architecture is not expected to change during system execution. We argue that the architectures of many systems, especially long running or mission critical systems, evolve *during* execution, and thus cannot be accurately modeled and analyzed using static architectures. To overcome these problems, we propose the use of *dynamic architectures*, where the system architecture may change during execution. In this paper, we identify the issues involved in supporting dynamic architectures. Although some of these issues may be addressed by augmenting current models (i.e., adding constructs that support dynamism to existing architectural description languages), many are new to dynamic architectures (i.e., runtime support for modifying architectures). We describe an initial implementation of our tool, ArchShell, that supports the runtime modification of C2-style software architectures.

1. Introduction

Current software architecture research assumes that a system's architecture is "static", in the sense that the architecture does *not* change during execution. Many architecture modeling notations and tools, such as Wright [AG94] and UniCon [SDK+95] are based on this assumption. It is our belief that the architectures of many systems evolve during execution, and thus cannot be accurately modeled and analyzed using a static architecture.

For example, consider the high costs and risks associated with shutting down and restarting long running safety or mission critical systems. Many such systems employ elaborate mechanisms that allow system extension during execution.

But the benefits to runtime evolution are not constrained to safety intensive, mission critical systems. A broadening class of systems and end-user applications are beginning to exhibit similar properties in an effort to provide end-user customizability and extensibility. Runtime extension facilities have become readily available in many popular operating systems (e.g., dynamic link libraries in UNIX and Microsoft Windows) and as a part of component object models (e.g., runtime component loading facilities in CORBA [OHE96] and COM [Broc94]). These facilities enable "system evolution without recompilation" by allowing new libraries or components to be located, loaded, and executed during runtime. Unlike program overlays, the identity of the libraries or components that a system will utilize is not necessarily known until runtime. The architecture of these systems *evolves* during the course of execution.

Static architectural modeling notations lack the constructs to express runtime change, thereby making it unnecessarily difficult or impossible to describe and analyze such systems. If we are to model the architectures of dynamically changing systems, we must augment our notations to describe runtime architectural changes and construct tools to help us analyze their unique properties. We refer to the unique aspects of such architectures as *dynamic architectures*.

The remainder of this paper is organized as follows. Section 2 discusses the unique issues in supporting dynamic architectures. Section 3 presents an experimental tool we have constructed that supports dynamic architectures. Section 4 discusses related work.

2. Dynamic Architectures

Dynamic architectures require formalisms and tools beyond those of static architectures. Adequate notational constructs are needed to describe runtime change, analysis tools are needed to help verify their unique properties, and runtime support libraries are needed to reduce the costs associated with their implementation.

Our investigation into supporting dynamic architectures has revealed several important issues that need to be addressed. We present our preliminary set of issues below.

Modification time: We distinguish between four periods during which architectural modifications may occur. The first, change at design time, represents the current focus of architecture research. The last, change at runtime, represents flexible runtime change. The two intermediate periods are distinguished for practical reasons. They provide less flexibility as compared to the pure dynamic case, but they are easier to analyze and implement.

1. *Design time:* Modifications are made before the architectural model and its associated source code are "compiled" into an application. Such modifications are relatively well understood since only an abstract model of the architectural is altered.

2. *Pre-execution time:* Modifications are performed after compilation but before execution. Since the application is not running, modification can be made without regard to the application state. Such modifications require that the application's architectural model be included with the application (e.g., by embedding it in the executable binary), and mechanisms for adding and removing components to the application be provided[1].
3. *Constrained run-time:* Modifications are performed only when certain pre-specified constraints are satisfied. Constraints based on the program state allow changes to be made when the application is in a known "safe" state with respect to the modification. Constraints based on the application topology allow changes to be made in the presence or absence of specific structural properties.
4. *Run-time:* Modifications are performed during runtime where assumptions about the state of the application or its architectural topology are not made beforehand.

Modification operations: The operations for describing change include: adding components, removing components, upgrading or replacing components, changing the architecture topology by adding or removing connections between components, altering the mapping of components to processing elements, querying for properties of architectural elements (e.g., to obtain versioning information), and querying the current architectural topology.

A typical change will require several modification operations. Consider for example the corrective upgrade of the text processing component of a word processor. Before making the change one may need to verify that (a) the existing component is older than the upgraded component, (b) components dependent on the text processing component will not be adversely affected, and (c) components that the new text processing component depend on are present. If steps (b) or (c) fail, other modification operations may need to be performed to upgrade existing components.

This necessitates a facility for grouping modification operations into atomic sets, such that if any one operation fails to complete, the entire set of changes are uncommitted to avoid leaving the system in an unstable state.

Modification constraints: By enabling change within system boundaries, architectural modifications risk compromising system integrity. Thus, mechanisms that maintain system integrity in light of architectural modifications are needed.

Modification constraints provide a means to specify limits on what aspects of an architecture may change. They may restrict change based on modification operation, particular components, modification time, or a combination thereof. They may also require that functional properties be verified (e.g., by using an analysis tool) before a change is committed. Relating constraints to behavioral properties of the system allows trade-offs to made if all constraints cannot be met simultaneously.

For example, consider a system whose integrity relies on satisfying real-time constraints. If adequate tools are not available to verify such properties at runtime, architectural modifications may be restricted to the subset of the system not operating under real-time constraints.

---

1. System "patches" are a common form of pre-execution time change. But they are extremely brittle because they operate at the level of byte streams.

Architectural modification language: Existing architecture definition languages (ADLs) such as Wright [AG94] and UniCon [SDK+95] only describe static architectures. In order to support dynamic architectures, two other descriptions are necessary:

- An architectural modification language (AML) is needed to describe modification operations. It describes the set of operations that must be performed to modify the architecture. [Med96] describes our initial attempt at supporting architectural modifications for C2's ADL. An architectural modification language would most likely be operational[2].
- An architectural constraint language (ACL) is needed to describe modification constraints. It describes the constraints under which architectural modifications must be performed. The application designer will typically provided the set of architectural constraints, which may change as components are added and removed from the architecture. An architectural constraint language would most likely be declarative.

Although these two languages are conceptually distinct, there is an important interaction between the two: modifications specified in the AML must be verified against constraints specified in the ACL.

Understandability and analyzability of these languages is particularly important. Users will be the primary authors of the descriptions, and tools must be able to verify modification operations against modification constraints effectively.

Optimization in the presence of change: Runtime modifications precludes the use of some classes of static optimizations. For example, in a static architectural model, a common optimization is to implement architectural connectors using direct procedure calls [SDK+95]. Such optimizations cannot be performed on a dynamic architecture since new components may be added to the connectors at runtime. But by specifying constraints on the architecture, such as "new components cannot be added to or removed from connector X", flexibility can be exchanged for performance. Thus, the application designer may consider the trade-offs between flexibility and performance as needed.

Optimization techniques based on runtime profiling information, such as adaptive compilation [Holz94] and load balancing, may be appropriate. Further investigation into determining the applicability of these techniques to dynamic architecture is needed.

Runtime system: The runtime system is the engine of architectural modification. Runtime system responsibilities include:

- Maintaining the system's architectural model. Since the architectural model may be needed to direct runtime change, the model must be included as a part of the delivered system and maintained by the runtime system.
- Ensuring that architectural modifications are within modification constraints.
- Enacting architectural modifications using facilities provided by the environment, e.g., dynamic loading and linking of components, process migration facilities for moving a component to a new processing element, interprocess communication mechanisms for component

2. Although the architecture modification language may be interpreted, the components themselves can be compiled. This is significant because since it enables flexibility without incurring the performance costs associated with program interpretation.

communication, etc. Environments lacking some of these facilities limit the scope of dynamic change. For example, an environment without dynamic linking facilities must use interprocess communication mechanisms between the application and newly added components.

The runtime system may take many forms. It may be compiled into the application as a library during compilation, it may be packaged as a part of the modification operations, or even as a separate application. The form it takes depends on many factors including the types of modification operations supported, the types of modification constraints that need to be enforced, properties of the execution environment, etc.

Concurrency: Complex software systems have architectures involving heterogeneous OTS components, of varying granularity, written in different programming languages, and running in a distributed, heterogeneous environment. It is naive to assume that execution may be suspended during architectural modification.

Techniques supporting change in such an environment need to be investigated. Locking components involved in a modification during a change to prevent access to them is one potential approach.

System state: A component's internal state changes during its lifetime and affects its execution behavior. Replacement during execution entails transferring (and possibly translating) the state of the component to its replacement. Failure to do so is likely to compromise system integrity. Care must be taken to ensure that component state does not change during the transfer. Techniques for transferring the state of a running component to another need to be investigated.

3. ArchShell: An Environment Supporting Dynamic Architectures

We have built a prototype tool, called ArchShell, that supports the construction and runtime modification of software architectures. ArchShell has provided valuable feedback in understanding dynamic architectures and the tools that are needed to support them.

ArchShell supports the interactive construction and execution of C2-style architectures[3] in a manner similar to the way in which a UNIX shell (e.g. csh) allows construction and execution of pipe and filter architectures. However, unlike a UNIX shell, ArchShell also supports the modification of the architecture at runtime. It does this by dynamically loading and linking new architectural elements into the architecture.

ArchShell provides a simple command oriented interface for issuing commands to the system. The architect constructs the initial architecture by selecting components and connectors out of a component repository and connecting them to one another. Once the initial architecture has been constructed, the architect initiates its execution. The architect is then free to interact with the newly constructed application and simultaneously issue ArchShell commands to modify its architecture. At runtime, new components and connectors may be added to the architecture, and connections between components and connectors may be added or modified. Newly added components or connectors do not begin executing until explicitly initiated by the architect. If a non-executing component or connector receives a message, the message is put on

3. Briefly, C2 is a component- and message-based architectural style for constructing flexible and extensible software systems. A C2 architecture is a hierarchical network of concurrent components linked together by connectors (message routing devices) in accordance with a set of style rules. See [TMA+96] for a detailed exposition on the C2-style.

the component's message queue and is delivered when the component begins execution.

Figure 1 shows a sample session with ArchShell in which a simple visualization of a Stack data structure is constructed. New components and connectors are added to the architecture using the "new" command, and connected to one another using the "weld" command. The "start" command initiates the execution of the architecture, which is depicted in Figure 2(a). The

```
DevShell
> new arch
ClassName? c2.framework.SimpleArchitecture
Name? StackArch
StackArch> new component
ClassName: c2.comp.StackADTThread
Name: StackADT
StackArch> new component
ClassName: c2.comp.StackArtistGraphics
Name: StackArtist
StackArch> new component
ClassName: c2.comp.graphics.GraphicsBinding
Name: GraphicsBinding
StackArch> new connector
ClassName: c2.framework.ConnectorThread
Name: bus1
StackArch> new connector
ClassName: c2.framework.ConnectorThread
Name: bus2
StackArch> weld
Top entity: StackADT
Bottom entity: bus1
StackArch> weld
Top entity: bus1
Bottom entity: StackArtist
StackArch> weld
Top entity: StackArtist
Bottom entity: bus2
StackArch> weld
Top entity: bus2
Bottom entity: GraphicsBinding
StackArch> start
Entity: StackArch

StackArch> new component
ClassName: c2.comp.StackPieArtist
Name: StackPieArtist
StackArch> weld
Top entity: bus1
Bottom entity: StackPieArtist
StackArch> weld
Top entity: StackPieArtist
Bottom entity: bus2
StackArch> start
Entity: StackPieArtist
```

Fig. 1. An interactive session with ArchShell. The text in bold represents commands issued by the user.

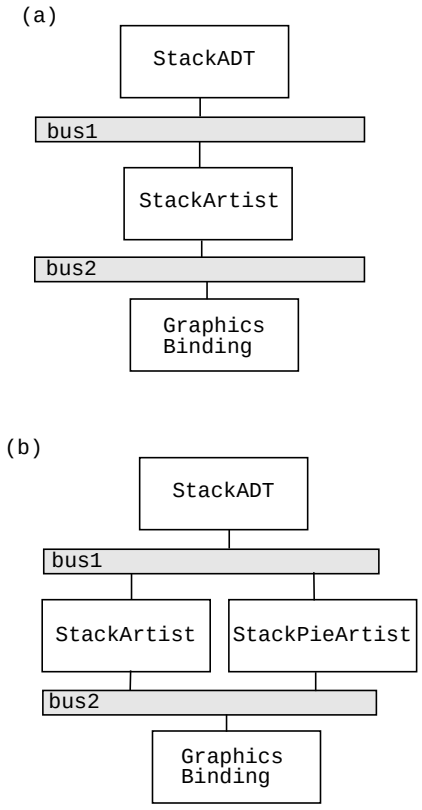

Fig. 2. The StackADT component encapsulates the stack abstract data type, and the StackArtist provides a graphical depiction of the stack using the GraphicsBinding component. (a) A graphical depiction of the initial executing architecture. (b) After the StackPie Artist has been added and connected in the architecture.

architect then uses ArchShell to add and connect the Stack Pie Artist component, which provides a different depiction of the same stack ADT. The second "start" command initiates the execution of the Stack Pie Artist. The modified architecture is depicted in Figure 2(b). On startup, the Stack Pie Artist queries the current state of the stackADT so that it may be accurately depicted.

ArchShell is written in the Java programming language using the Java C2 class framework [MOT97]. The C2 framework is extensible and provides abstract classes for C2 concepts such as

components, connectors, and messages. It implements several component interconnection and message passing protocols. Components and connectors used in C2 applications are subclassed from the appropriate abstract classes in the framework. 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

Our initial prototype of ArchShell has facilitated exploration, but has many practical limitations. It currently assumes that all components and connectors are written in Java using the C2 framework. This allowed us to use the dynamic loading facilities provided by Java. In the future, we plan on using language independent facilities, like those provided by CORBA and OLE. ArchShell only operates on architectures constructed and executed within the tool. We plan on extending ArchShell to enable it to attach itself to an executing application in much the same as the GNU debugger, *gdb*, can attach itself to an application started outside the debugger.

Currently, the runtime modifications supported include the addition of new components and connectors, and reconfiguration of the architecture topology. Specifically, we do not currently support unloading unused components or replacing components by translating and transferring their internal state information. In the future, we plan on extending ArchShell to support more diverse architectural modifications and to integrate it with Argo, our graphical design environment [RR95].

4. Related Work

4.1 Architecture Description Languages

Architecture Description Languages, or ADLs, provide the formalism necessary to describe software architectures. ADLs provide the syntax and semantics for modeling components, connectors, and their interconnections. Since the focus of existing ADLs has been as a design notation, their use has typically been limited to static analysis and system generation. As such, existing ADLs typically provide a static description of the system, and provide no facilities for making changes to the architecture at runtime.

Three notable exceptions are Rapide's *where* connection conditions [LV95], LILEANNA's *make* statements [Tra93], and Darwin's *dyn* construct [MK96]. Although these constructs enable runtime change, the modification must be "coded into" and compiled with the application. Since the modifications must be described and planned for at system design time, these ADLs do not support our notion of dynamic architecture.

4.2 Dynamic Language Environments

Many dynamic language environments provide dynamism similar to that discussed here. For example, The InterLisp environment [TM81] built using and for the Lisp programming language allows system source code to be edited during execution.

These environments gain dynamism at the expense of heterogeneity. The entire application must be written in the same programming language using either a single thread of control or multiple threads in a shared address space. As previously discussed, many large-scale complex systems cannot be constructed or rewritten using these assumptions.

Performance overhead is another concern of dynamic languages. Dynamism of dynamic architectures stems from flexible component composition. The components themselves may be

compiled and written in a static language. The *language* designer dictates granularity of dynamism for a dynamic language whereas dynamic architectures leave the choice to the *system* designer.

5. Acknowledgments

The Java framework was built as a part of a class project with the help of Neno Medvidovic. Richard Taylor, Neno Medvidovic, and Jason Robbins provided valuable comments on this work.

6. References

[AG94] R. Allen, D. Garlan. Formal Connectors, *CMU Tech. Report 94-115.* March 1994.

[Broc94] K. Brockschmidt. *Inside OLE 2*. Microsoft Press, 1994.

[Holz94] U. Holzle. Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming. *Dissertation,* Stanford University, 1994.

[Med96] N. Medvidovic. ADLs and Dynamic Architecture Changes. *Second International Software Architecture Workshop (ISAW-2)*, San Francisco, October 1996.

[MOT97] N. Medvidovic, P. Oreizy, R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. *To appear at the Symposium on Software Reuse*. May 1997.

[MK96] J. Magee, J. Kramer. Dynamic Structure in Software Architectures. *Fourth SIGSOFT Symposium on the Foundations of Software Engineering*, San Francisco, October 1996.

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

[RR95] J. Robbins, D. Redmiles. Software Architecture Design from the Perspective of Human Cognitive Needs. *Proceedings of the California Software Symposium.* Los Angeles, California, 1996.

[SDK+95] M. Shaw, R. DeLine, D. V. Klien, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, pages 314-335, April 1995.

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

[GMW95] D. Garlan, R. Monroe, and D. Wile. ACME: An Architectural Interconnection Language. Technical Report, CMU-CS-95-219, Carnegie Mellon University, November 1995.

[LV95] D. Luckham and J. Vera. An Event-based Architectural Definition Language. *IEEE Transactions on Software Engineering*, pages 717-734, September 1995.

[Tra93] W. Tracz. Parameterized Programming in LILEANNA. *Proceedings of ACM Symposium on Applied Computing SAC'93*, February 1993.

[TM81] W. Teitleman, L. Masinter. The InterLisp Programming Environment. *IEEE Computer*, pages 25-33. April 1981.

[TMA+96] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, pages 390-406, June 1996.