Appeared in the Proceedings of the International Conference on Configurable Distributed Systems (ICCDS 4).
Annapolis, Maryland, May 4-6, 1998.

On the Role of Software Architectures in Runtime System Reconfiguration

Peyman Oreizy   Richard N. Taylor

Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
{peymano,taylor}@ics.uci.edu

Abstract

Society's increasing dependence on software-intensive systems is driving the need for dependable, robust, continuously available systems. Runtime system reconfiguration is one aspect of achieving continuous availability. We present an architecture-based approach to runtime software reconfiguration, highlighting the role of architectural styles and software connectors in facilitating runtime change. Finally, we describe the implementation of our tool suite, called ArchStudio, that supports runtime reconfiguration using our architecture-based approach.

1. Introduction

Society's increasing dependence on software-intensive systems is driving the need for dependable, robust, continuously available systems. The ability to reconfigure a system at runtime is one critical aspect of achieving continuously availability. Although operating systems and programming languages have provided programmers with the ability to evoke runtime software changes since the 1960's, such mechanisms do not guarantee that a change will have the desired effect or maintain application integrity. It is therefore imperative that we develop approaches to runtime system reconfiguration that help us (a) determine what to change, (b) facilitate reasoning about the consequences of a change, and (c) govern change to preserve application integrity. Without this, the risks introduced by runtime reconfiguration may outweigh those associated with shutting down and restarting the system for reconfiguration.

Software architectures [22, 28] have the potential to provide a foundation for systematic runtime software evolution. Architectures shift development focus away from lines-of-code toward coarse-grained components and their overall interconnection structure. This enables designers to abstract away unnecessary details and focus on the big picture: system structure, interactions among software components, assignment of software components to processing elements of the execution environment, and potentially runtime reconfiguration.

We present an architecture-based approach to runtime software reconfiguration, highlighting the role of architectural styles and software connectors in facilitating runtime change. We also describe the implementation of our tool suite, called ArchStudio, that supports runtime reconfiguration using our approach.

A unique benefit of our approach is that it enables system architects to control several critical aspects of runtime reconfiguration independent of application-specific behavior. Furthermore, our approach does not dictate a particular strategy for implementing runtime reconfigurable systems. Instead, it permits architects to specify the most appropriate strategy based on application-specific requirements.

The paper is organized as follows. Section 2 describes key aspects of runtime change management. Section 3 describes our architecture-based approach to runtime software reconfiguration. Section 4 identifies research areas relevant to this work and Section 5 summarizes the contributions of the paper.

2. Managing runtime change

While runtime software change is commonly available in operating systems (for example, dynamic link libraries in UNIX and Microsoft Windows), component object technologies, and programming languages, these facilities all share a major shortcoming—they do not ensure the consistency, correctness, or other desired properties of runtime change. *Change management* is a critical aspect of runtime system evolution that: identifies what must be changed, provides the context for reasoning about, specifying, and implementing change, and controls change to preserve system integrity. Without change management, the risks engendered by runtime modifications may outweigh those associated with shutting down and restarting a system.

Our ability to manage change in large, complex systems hinges on several critical factors:

1. *Change application policy* controls how a change is applied to a running system. A policy, for example, may instantaneously replace old functionality with new functionality. Another policy may gradually introduce change by binding invocations subsequent to the change to the new functionality, while preserving bindings previously established to the old functionality. Ideally, change application policy decisions should be made by the designer based on application requirements, not by the runtime reconfiguration methodology.
2. *Change scope* is the extent to which different parts of a system are affected by a change. One approach, for example, may stall the entire system during the course of a change. The designer's ability to localize the effects of runtime change by controlling its scope facilitates change management. The designers ability to ascertain change scope helps reason about change.
3. *Separation of concerns* captures the degree to which issues concerning functional behavior are isolated from runtime change. The greater the separation, the easier it becomes to alter one without adversely affecting the other.
4. The *level of abstraction* at which changes are described impacts the complexity and quantity of the information that must be effectively managed.

In the next section, we evaluate our architecture-based approach against these factors.

3. Architecture-based runtime system reconfiguration

We advocate an architecture-based approach to runtime software reconfiguration. Several direct benefits result when managing change at the architectural level. First, control over change application policy and scope can be placed in the hands of the system architect, where decisions can be made based on an understanding of the application requirements (factor #1 and #2). Previous approaches to runtime change either dictate a particular policy or fail to separate application-specific functionality from runtime modification. As a result, concerns over runtime change permeate system design. Second, software engineers commonly use the system architecture when describing, understanding, and reasoning about overall system behavior [22, 28]. Leveraging the engineer's knowledge at this level of system abstraction holds promise in helping manage runtime change (factor #4). Third, *architectural connectors* separate application-specific behavior from decisions regarding change application policy and scope, allowing them to be altered independently (factor #3).

Each of the following three subsections focuses on one facet of our architecture-based approach to runtime software reconfiguration: architectural style, architectural connectors, and runtime support. We illustrate the benefits provided by each facet using a simple producer-consumer application. Although the application is trivial in nature, it elucidates the important aspects of our approach. More realistic applications implemented using our approach are described in [16], [20] and [29].

3.1 Architectural style

Architectural styles are idiomatic patterns of system organization that characterize a particular application domain [22, 28]. In this way, architectural styles define a vocabulary for describing systems and a set of rules that guide their construction. In cases where the descriptions and rules can be expressed formally, overall system properties may be derived from system organization. The pipe-and-filter style, for example, consists of filter components, which read data from an input stream and produce data on two output streams, and pipes, which bind the output stream of one filter to the input stream of another. The pipe-and-filter style emphasizes sequential transformation of data, and is commonly used by Unix shell programs and traditional compiler architectures.

Not all architectural styles are equally well suited for runtime system reconfiguration. For example, the nested organization of behavior typified by layered systems and main program/subroutine styles complicates replacement of deeply nested functionality. In contrast, event-based implicit invocation styles [7] are more amenable to runtime reconfiguration since (a) components are not directly bound to one another, and (b) a component is unaware of (and unaffected by) implicitly invoked components.

Our experience indicates that several characteristics of the C2 architectural style facilitate runtime reconfiguration. Although most of these characteristics are not unique to C2, our approach to combining them is. In the following subsections, we briefly summarize the C2-style, present a C2-style architecture for the producer-consumer application, and highlight characteristics of the C2-style that facilitate runtime reconfiguration.

3.1.1 C2 architectural style

The C2 architectural style[1] can be informally summarized as a network of concurrent components bound together by connectors, i.e., message routing devices, in accordance with a set of style rules. Components and connectors both have a defined top and bottom. The top of a component may be connected to the bottom of a single connector and the bottom of a component may be connected to the top of a single connector. No direct

[1]For a more detailed discussion of the C2 architectural style and its benefits see [29].

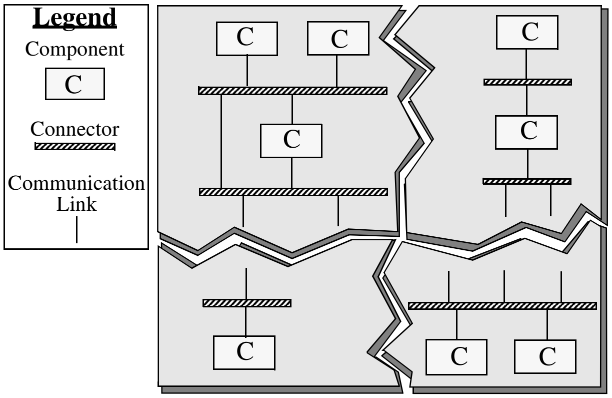

Figure 1. An abstract C2 architecture. Jagged lines represent portions of the architecture not shown.

Component Interface
- start()
- finish()
- handle(request)
- handle(notification)

Connector Interface
- start()
- finish()
- handle(request)
- handle(notification)
- addTopPort()
- removeTopPort()
- addBottomPort()
- removeBottomPort()

Architecture Interface
- start()
- finish()
- handle(request)
- handle(notification)
- addComponent(component)
- removeComponent(component)
- addConnector(connector)
- removeConnector(connector)
- weld(connector, component)
- weld(component, connector)
- weld(connector, connector)
- unweld(connector, component)
- unweld(component, connector)
- unweld(connector, connector)

Figure 2. The external interfaces of C2 components, connectors, and architectures that may be invoked during runtime.

component-to-component links are allowed. A connector may be connected to an unbounded number of other components and connectors. When two connectors are attached to one another, it must be from the bottom of one to the top of the other (see Figure 1).

Components implement application behavior and may encapsulate functionality of arbitrary complexity, maintain state information, and utilize multiple threads of control. The style does not place restrictions on the implementation language or granularity of the components. It does require that all component communication occur by asynchronous message exchange through connectors[2]. Furthermore, components cannot assume that they will execute in the same address space as other components or share a common thread of control.

Central to the architectural style is a principle of limited visibility or *substrate independence*: a component within the hierarchy can only be aware of components "above" it and is completely unaware of components which reside at the same level or "beneath" it. Notions of above and below are used to support an intuitive understanding of the architectural style. A component explicitly utilizes the services of components "above" it by sending a *request* message. Communication with components below occurs implicitly; whenever a component changes its internal state, it announces the change by emitting a *notification* message, which describes the state change, to the connector below it. Connectors broadcast notification messages to every component and connector connected on its bottom side. Thus, notification messages provide an implicit invocation mechanism, allowing several components to react to a single component's state change.

[2] While the style does not forbid synchronous communication, the responsibility for implementing synchronous message passing resides with individual components. Ideally, the most common synchronous communication patterns would be implemented in the C2 class framework and reused across applications.

We have developed an extensible class framework to facilitate the implementation of C2-style applications. The framework provides abstract classes for C2 concepts such as components, connectors, and messages, and implements default behavior for interconnecting components and connectors, message passing, and component initialization and termination. Application components (and connectors) subclass from the appropriate framework classes and override the default behavior if necessary. This eliminates many repetitive programming tasks and allows developers to focus on application-specific issues.

The framework simplifies several aspects of supporting runtime reconfiguration in application-specific components and connectors. The subset of framework methods used for runtime change are presented in Figure 2. For example, in order to support custom initialization and termination behavior, the component (and connector) framework classes implement a start() and finish() method. The start() initiates component execution. The finish() method terminates component execution. The default implementation of the finish() method does not interrupt the component's execution if the component is processing a message; it waits until the component has finished processing the message before terminating it. Application components inherit these methods, but can replace or augment their behavior if desired. Most components, for example, override the start() method to synchronize their state with that of the rest of the application. C2 connectors provide two additional methods, weld() and unweld(), for connecting and disconnecting components during runtime.

Our class framework has been implemented in C++, Java, and partially in Ada. The C++ and Ada frameworks implement a connector based on the Q interprocess communication (IPC) library [14] to enable distributed message passing. A similar connector has been implemented in the Java framework using Java's RMI (Remote Method Invocation) mechanism.

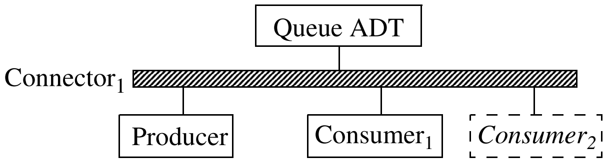

Figure 3. The C2 style architecture for a producer-consumer application in which a second consumer component, *Consumer$_2$*, is being added during runtime.

3.1.2 Example

Figure 3 illustrates the C2-style architecture for the producer-consumer application. The single connector, *Connector$_1$*, is responsible for routing messages between components. In this instance, the connector uses an interprocess communication mechanism.

The *Producer* component periodically adds an item to the *Queue ADT* by sending it an *enque(item)* request message (*item* is the message parameter). The *Queue ADT* receives the request, and updates its data structure to reflect the addition. Since its internal object is modified, it emits the notification message *enqued(item)* down the architecture. *Connector$_1$* broadcasts the notification to every component below it. If *Consumer$_1$* is not busy when it receives the notification, it attempts to retrieve the newly added item by sending a *dequeue()* request to *Queue ADT* and waiting for a response. The *Queue ADT* receives the *dequeue()* request, updates its data structure to reflect the removal, and emits a notification message, *dequeued(item)*, in response to the request. Again, *Connector$_1$* broadcasts the notification to every component below it. *Consumer$_1$* determines if the *dequeue(item)* notification is a response to its request (by checking the message's "originator" parameter); if it is, it processes the item, otherwise it ignores the notification and continues to wait for a reply. The *Producer* component ignores all notification messages.

Adding consumers (or producers) is straightforward since they do not adversely affect the other components in the architecture. For example, a second consumer component, *Consumer$_2$*, may be dynamically instantiated and welded to the bottom of *Connector$_1$*. Once added, *Consumer$_2$* effectively competes with *Consumer$_1$* for newly added items. In the case where both components are idle when a new *enqued(item)* notification is broadcast, both respond with *dequeue()* requests to the *Queue ADT*. Since the *Queue ADT* serializes access to its internal object, the first request results in a *deque(item)* notification, while the second request results in an *EmptyQueue()* notification.

Removal of a consumer (or producer) is also straightforward, though some care must be taken to ensure that the consumer has the opportunity to complete any necessary processing. In the case of *Consumer$_2$*, the finish() method waits until the component is idle (i.e., it has completed message processing and is not waiting for a response) before disconnecting the component from *Connector$_1$* and terminating its execution.

In this particular application, removing the consumer and discarding its state does not violate application consistency so long as the component is idle. This is because no components in the application depend on the services or state provided by the consumer component. Such a restriction is clearly inadequate for the *Queue ADT* since consumer components depend upon its functionality and internal state. The other facets of our approach, described in subsequent sections of the paper, address runtime changes involving the *Queue ADT.*

3.1.3 Summary

The C2-style rules that facilitate runtime reconfiguration include:

- *asynchronous message passing*—Since all communication between components is achieved by exchanging asynchronous messages through connectors, we avoid several subtle complexities inherent in supporting runtime change where components utilize synchronous service requests. This restriction has occasionally made it more difficult to implement particular component interactions, since the component must continue to respond to requests and notifications from other components while awaiting a notification message from a service request it has initiated. We are currently investigating strategies for implementing synchronous communication mechanisms on top of our asynchronous mechanism without negatively impacting runtime change.
- *no assumption of shared address space or shared thread of control*—Since components cannot assume that they will execute in the same address space as other components, complex component dependencies resulting from the use of pointer variables and global variables are avoided. Since components do not share a common thread of control, complexities involving control dependencies are similarly avoided.
- *substrate independence*—Since a component is unaware of components at the same level and "below" itself, it is oblivious to runtime changes that involve these components. Conversely, a runtime change involving a component can only affect components strictly "below" itself. Thus, substrate independence confines change scope to a subset of the architecture.

Although these characteristics facilitate runtime change, the C2-style does not, by itself, guarantee that a change will leave the application in a consistent state. We have avoided adding style rules that ensure some form of application integrity since such rules would prevent the use of the style in some application domains. The other facets of our approach address other aspects of application integrity.

Generally, determining when, how, or even if a runtime change preserves application integrity depends largely on application-specific requirements. For example, the only constraint governing runtime component removal of consumer components is that the component be idle. Such a constraint may be unnecessary is the application is designed to tolerate component failures. As a result, our approach enables architects to explicitly govern runtime reconfiguration on an as-needed basis as opposed to mandating particular rules or policies.

3.2 Architectural connectors

Connectors are explicit architectural entities that bind components together and act as mediators between them [28]. In this way, connectors separate a component's interfacing requirements from its functional requirements [24], and separate component behavior from component interaction. This is especially important when constructing systems from reusable off-the-shelf components, since the component designers cannot anticipate every context in which the component will be used.

Connectors have been used for a wide variety of purposes, including: ensuring a particular interaction protocol between components [2]; specifying communication mechanism independent of functional behavior, thereby enabling components written in different programming languages and executing on different hosts to transparently interoperate [24]; visualizing and debugging system behavior by monitoring messages between components [23]; and integrating tools by using connectors as message broadcast buses [25].

Connectors play a central role in supporting several aspects of change management. They can implement different change application policies by altering the conditions under which newly added components are invoked. For example, to support immediate component replacement, a connector can direct communication away from the old component to the new component. To support a more gradual component replacement policy, a connector can direct new service requests to the new component, while directing previously established service requests to the original component. To support a policy based on replication, a connector can direct service requests to any member of a known set of functionally redundant components.

Connectors can also be used as a means of localizing change. For example, if a component becomes unavailable during the course of a runtime change, the connectors mediating its communication can queue service requests until the component becomes available. As a result, other

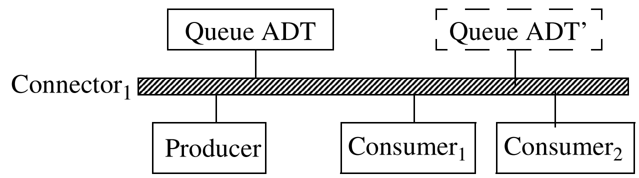

Figure 4. The C2 style architecture for the producer-consumer application in which a new Queue ADT component, *Queue ADT'*, is replacing the existing *Queue ADT* component during runtime.

components are insulated from the change. Encapsulating change application policy decisions within connectors lets designers specify the most appropriate policy based on application requirements and in a manner independent of component behavior.

In the following subsections, we illustrate how a connector can implement a gradual change application policy within the context of the producer-consumer application, and summarize the benefits of utilizing architectural connectors to support runtime reconfiguration.

3.2.1 Example

Figure 4 depicts the C2-style architecture for the producer-consumer application in which a new Queue ADT component, *Queue ADT',* is to replace the existing *Queue ADT* component. Our example illustrates one possible technique for replacing the component using a connector to implement a gradual change application policy. Many other change application policies and implementation techniques are possible. In our example, we assume that *Connector*$_1$ has been specifically implemented to support this policy by the application designer. The general problem of designing parameterized connectors that may be customized by different change application policies is a topic of future work.

When *Queue ADT'* is added and connected to the top of *Connector*$_1$ as a replacement for *Queue ADT*, *Connector*$_1$ directs subsequent *enque(item)* messages emitted from the *Producer* component to *Queue ADT'*. In this way, new items are added to *Queue ADT'* without affecting the other components in the architecture. When *Connector*$_1$ receives a *dequeue()* request from either consumer, it directs the message to the original *Queue ADT.* If *Queue ADT* responds with *dequeued(item)*, *Connector*$_1$ broadcasts the notification to all the components connected below it as before; if *Queue ADT* responds with *EmptyQueue()*, the connector re-sends the *dequeue()* message to *Queue ADT'* and broadcasts the notification to all the components connected below it. Once *Queue ADT* is empty, the connector can disconnect and remove it from the architecture.

3.2.2 Summary

Although other architectural styles and architecture description languages represent connectors as explicit entities in the design, they have traditionally been implemented as indiscrete entities in the implementation. For example, procedure call and data access connectors in UniCon are reified as linker instructions during system generation [27]. Similarly, component binding decisions, while malleable during design, are typically fixed during system generation. As a result, modifying binding decisions during runtime becomes difficult. C2 connectors, in contrast, are explicit runtime entities in the implementation. Consequentially, C2 connectors encapsulate [21]:

1. the identity of the component receiving a particular message;
2. the number of components receiving a particular message;
3. the particular runtime change application policy used when adding, removing, or altering component-connector bindings;
4. the policy used to determine which components (from a set of eligible components) receive a message. If two or more components provide similar functionality, the connector may determine the most appropriate component to receive a given message. Such a decision may be based upon communication latency, machine load, etc.;
5. the particular interprocess communication mechanism used for message passing. The connector can isolate the particular communication mechanism used to pass messages from one component to another (e.g., direct procedure calls, Unix sockets, RPC, CORBA, etc.);
6. the component's location in the network. Since components are not statically bound to one another, a component may migrate from one host to another without notifying other components;
7. the mapping from messages sent to message received. Since the connector acts as a conduit for communication, it can act as a domain translator [31] between components;
8. the message to method mapping. If a component does not process C2 messages directly, the connector can provide a message to method mapping. This mapping, like the dynamic dispatch mechanism in Lisp, can potentially be altered during runtime. In fact, the binding does not have to be one-to-one. The connector may map a single message to several methods and combine the results in an appropriate manner.

The distinction between C2 connectors and software buses, such as Polylith [24] and Field [25], is that an application may utilize multiple C2 connectors, each one of which may implement a different change application policy. A software bus, in contrast, implements one policy and requires all application components to utilize it. In this respect, C2 connectors are more flexible than software buses.

3.3 Runtime architectural support

The third facet of our approach consists of runtime architectural support for reconfiguration. Four interrelated mechanism implement this facet of our approach. They are:

- *an explicit architectural model*—In order to effectively reconfigure a system during runtime, an accurate architectural model must be available. Since changes specified in terms of the architectural model must be reified in the implementation, a mapping from the model to the implementation must also be provided.
- *describing runtime change*—Modifications are expressed in terms of the architectural model, and include operations for adding and removing components and connectors, replacing components and connectors, and changing the architectural topology. Operations for querying the architectural model should also be included since modifications may be dependent upon the particular configuration of the system.
- *governing runtime change*—Although architectural style rules and connectors may be used to prevent particular classes of runtime reconfigurations that would compromise system integrity, a mechanism that governs a broader set of application-specific changes is necessary. Constraints play a natural role in governing change, and several approaches that apply constraints to software architectures have been described in the literature (see Section 4). It should also be noted that during the course of a complex reconfiguration, the system may “move” through several invalid configurations before reaching a final valid configuration. Although constraints may legitimately restrict certain modification “paths”, doing so solely based on intermediate invalid configurations prevents some valid runtime changes. As a result, a mechanism that supports transactional modifications should ultimately be provided.
- *reusable runtime architecture infrastructure*—The runtime architecture infrastructure (a) maintains the consistency between the architectural model and implementation as reconfigurations are applied, (b) reifies changes in the architectural model to the implementation, and (c) ensures that architectural constraints are not violated.

In the following subsections, we describe our implementation of these mechanisms, illustrate how they support component replacement in the producer-consumer application, and summarize their benefits.

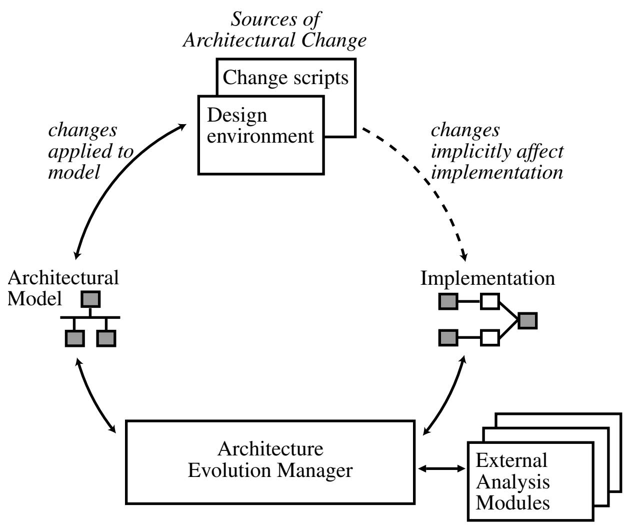

Figure 5. High-level architecture diagram for the ArchStudio tool suite.

3.3.1 ArchStudio tool suite

Our initial prototype of a tool suite that supports runtime reconfiguration is called ArchStudio. The tools that comprise ArchStudio are implemented in the Java programming language and can modify applications written using the Java-C2 class framework. Figure 5 depicts a high-level view of the ArchStudio architecture.

The *architectural model*[3] represents an up-to-date model of the application's architecture. Our current implementation encapsulates the architectural model in an abstract data type (ADT). This ADT provides operations for querying and changing the application's architectural model and is kept up-to-date during system execution. The model is stored in an ASCII file when the application is not executing. The model consists of the interconnections between components and connectors and their mapping to Java classes. Runtime modifications consist of a series of query and change requests to the architectural model and may generally arrive from several different sources.

The *Architecture Evolution Manager* (*AEM*) maintains the correspondence between the *architectural model* and the *implementation*. Attempts to modify the architectural model invoke the *AEM*, which determines if the modification is valid. The current implementation of the *AEM* uses implicit knowledge of C2-style rules to constrain changes; the addition of an architectural constraint system and external analysis tools is underway. If a change violates the C2-style rules, the *AEM* rejects the change. Otherwise, the *architectural model* is altered and the implementation mapping is used to make corresponding changes to the *implementation*.

Each change to the architectural model corresponds to a change in the implementation. For simplicity, our current implementation assumes a one-to-one mapping between components in the architectural model and implementation modules written as Java classes. This has enabled us to focus on dynamism independently of issues concerning mappings between architectures and their implementations, which is an open research problem in itself [6, 18]. For example, the addition of a new model component (or connector) corresponds to dynamically loading the Java class implementing the component (or connector), creating an instance of the class, and invoking its start() method. The removal of a model component corresponds to invoking the finish() method on the component's instance, and deallocating its instance. Adding (or removing) a connection from the model corresponds to establishing (or tearing down) a communications channel between the components and connectors involved.

ArchStudio currently includes three tools that act as *sources of architectural modification:* Argo, ArchShell, and Extension Wizard Scripts. *Argo* [26] provides a graphical depiction of the architectural model that may be directly manipulated by the architect and is similar to ConicDraw [11]. New components and connectors are selected from a palette and added to the architecture by dragging them onto the design canvas. Components and connectors are removed by selecting them and issuing a delete command. Interconnections between component and connectors are altered by directly manipulating the links between them. *ArchShell* [19] provides an interactive, textual, command-line interface for specifying reconfigurations.

Argo and ArchShell are interactive tools meant for use by software architects to describe architectures and architectural modifications. *Extension Wizard Scripts,* in contrast, provide a greatly simplified end-user interface for enacting runtime change. The Extension Wizard is deployed as a part of the application and executes modification scripts designed by architects. Modification scripts can query and alter the architectural model using the same mechanisms as Argo and ArchShell. End-users use a Web browser to display a list of available system updates, e.g. provided on the application vendor's Web site. A system update is a compressed file containing a runtime reconfiguration script and any new implementation modules. Selecting a system update causes the Web browser to download the file and invoke the Extension Wizard to process it. The Extension Wizard uncompresses the file, locates the reconfiguration script contained within, and executes it. Hall et al. [8] use a similar approach for deploying software updates.

[3] Italicized text in this section denote graphical entities in Figure 5.

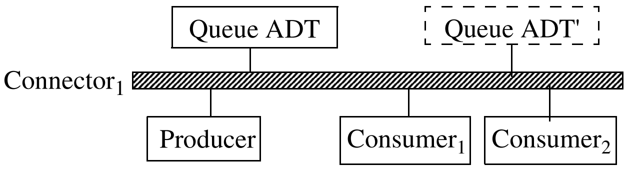

Figure 6. The producer-consumer C2 style architecture in which the *Queue ADT'* component is replacing the existing *Queue ADT* component and extracting the *Queue ADTs* state during runtime.

3.3.2 Example

Figure 6 depicts the C2-style architecture for the producer-consumer application in which a new Queue ADT component, *Queue ADT',* is to replace the existing *Queue ADT* component. In contrast to the gradual component replacement policy described in the previous section, this example illustrates an instantaneous component replacement policy whereby the internal state of the original Queue ADT is transferred to its replacement. For the purposes of this example, we use the component replacement strategy described by Hofmeister [9] in which each component exposes two additional methods: one for divulging state information, and the other for performing special initialization when replacing a component. Alternative approaches for preserving component state during runtime replacement have been proposed in the literature [4,5].

When the replacement operation is invoked to replace *Queue ADT* with *Queue ADT',* the runtime infrastructure (1) invokes any external analysis tools to determine if the replacement preserves application integrity, (2) directs *Connector*$_1$ to temporarily queue incoming messages for *Queue ADT*, (3) invokes the *Queue ADTs* special method to divulge state information, (4) disconnects *Queue ADT* from *Connector*$_1$, (5) invokes the special initialization method of *Queue ADT'* with the state of the original component, (6) connects *Queue ADT'* to *Connector*$_1$, and (7) directs *Connector*$_1$ to forward all queued and future messages for *Queue ADT* to *Queue ADT'*.

3.3.3 Summary

Notably missing from the interface to the architectural model (see Figure 2) are methods to support component replacement. Our current implementation does not currently support component replacement, though the implementation allows currently available approaches to be adopted. For example, we could adopt Hofmeister's approach by adding two methods to each component, one for divulging state and the other for initializing state. A new *replace(old, new)* method on the architectural model's ADT would direct the AEM to utilize the additional methods to make runtime changes.

4. Related issues

This section briefly outlines a number of cross cutting research issues that are pertinent to runtime architectural modification.

Architecture Description Languages (ADLs)—ADLs provide a formal basis for describing software architectures by specifying the syntax and semantics for modeling components, connectors, and configurations [17]. Since a majority of existing ADLs have focused on design issues, their use has been limited to static analysis and system generation. As such, existing ADLs support a static description of a system, but provide no facilities for specifying runtime architectural changes. Although a few ADLs, such as Darwin [13], Rapide [12], and LILEANNA [30], can express runtime modification to architectures, they require that the modifications be specified and "compiled into" the application. Our approach, in contrast, can accommodate unplanned modifications of an architecture and incorporate behavior unanticipated by the original developers. Our approach does not attempt to replace static architecture description languages. In fact, our tools can utilize current ADLs instead of our own for the static portion of the architectural model.

Architectural modification languages (AMLs)—While ADLs focus on describing software architectures for the purposes of analysis and system generation, AMLs focus on describing *changes* to architecture descriptions. Such languages are useful for introducing unplanned changes to deployed systems by changing their architectural models. The Extension Wizard's modification scripts, C2's AML [15], and Clipper [1] are examples of such languages and have many similarities.

Architectural constraint languages—Several approaches for specifying architectural constraints have been proposed. Constraint languages have been used to restrict system structure using imperative [3] as well as declarative [13] specifications. Others advocate behavioral constraints on components and their interactions [12]. Finding appropriate mechanisms for governing architectural change using constraints is an active area of ongoing research.

5. Conclusions

We have described and illustrated the beneficial role of software architectural styles and connectors in supporting runtime software reconfiguration. We have demonstrated the utility of preserving explicit architectural connectors in system implementation as well as using connectors to encapsulate different runtime change application policies.

While previous approaches either dictate a particular policy or fail to separate application-specific functionality from runtime modification, our approach enables the architect to choose or design the most appropriate runtime change application policy based on application-specific requirements. As a result, concerns over runtime reconfiguration do not change permeate system design.

We have also described a prototype tool suite that enables architects to specify, invoke, and govern runtime change at the architectural level. The unique aspects of our implementation include: an explicit architectural model deployed and kept up-to-date with the implementation as runtime changes are applied; a mechanism whereby software reconfigurations specified in terms of the architectural model are reified in changes to the implementation; and a flexible mechanism for governing runtime architectural changes.

6. Acknowledgments

We are grateful to David Hilbert, Nenad Medvidovic, Jason Robbins, and David Rosenblum for providing valuable insights on this work.

The material is based on work sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021. 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, Rome Laboratory or the U.S. Government. Approved for Public Release - Distribution Unlimited.

7. References

The role connectors play in supporting runtime architectural change (described in Section 3.2) and the description of the ArchStudio tool suite (describe in section Section 3.3.1) are summarized from [20].

[1] B. Agnew, C. R. Hofmeister, J. Purtilo. Planning for change: A reconfiguration language for distributed systems. Proceedings of the *International Workshop on Configurable Distributed Systems,* March 1994.

[2] R. Allen, D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[3] R. Balzer. Enforcing architectural constraints. *Second International Software Architecture Workshop (ISAW-2)*, San Francisco, October 1996.

[4] T. Bloom, M. Day. Reconfiguration and module replacement in Argus: Theory and practice. *IEE Software Engineering Journal*, vol 8, no 2, March 1993.

[5] O. Frieder, M. Segal. On dynamically updating a computer program: From concept to prototype. *Journal of Systems and Software*, 14:111-128. 1991.

[6] D. Garlan. Style-based refinement for software architecture. *Second International Software Architecture Workshop (ISAW-2)*. San Francisco, CA, October 1996.

[7] D. Garlan, G. E. Kaiser, D. Notkin. Using tool abstraction to compose systems. *IEEE Computer*. 25(6):30-38, June 1992.

[8] R. S. Hall, D. Heimbigner, A. van der Hoek, A. L. Wolf. An architecture for post-development configuration management in a wide-area network. *17th International Conference on Distributed Computing Systems*, Baltimore, Maryland, May 1997.

[9] C. R. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. Ph.D. Thesis. University of Maryland, Computer Science Department, 1993.

[10] J. Kramer, J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11), November 1990.

[11] J. Kramer, J. Magee, K. Ng. Graphical configuration programming. *IEEE Computer*, 22(10), 1989, 53-65.

[12] D. Luckham, J. Vera. An event-based architectural definition language. *IEEE Transactions on Software Engineering,* 21(9):717-734, 1995.

[13] J. Magee, J. Kramer. Dynamic structure in software architectures. *Fourth SIGSOFT Symposium on the Foundations of Software Engineering*, San Francisco, October 1996.

[14] 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. Also issued as CU Technical Report CU-CS-782-95.

[15] N. Medvidovic. ADLs and dynamic architecture changes. *Second International Software Architecture Workshop (ISAW-2)*, San Francisco, October 1996.

[16] N. Medvidovic, P. Oreizy, R. N. Taylor. Reuse of off-the-shelf components in C2-style architectures. *Symposium on Software Reusability*, Boston, May 1997.

[17] N. Medvidovic, R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference* together with the *Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp 60-76, Zurich, Switzerland, September 22-25, 1997.

[18] M. Moriconi, X. Qian, R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*. pp 356-372, April 1995.

[19] P. Oreizy. Issues in the runtime modification of software architectures. *UC Irvine Technical Report UCI-ICS-96-35*, Department of Information and Computer Science, University of California, Irvine, August 1996.

[20] P. Oreizy, N. Medvidovic, R. N. Taylor. Architecture-based Runtime Software Evolution. In *Proceedings of the International Conference on Software Engineering (ICSE),* Kyoto, Japan, April 1998.

[21] P. Oreizy, D. S. Rosenblum, R. N. Taylor. On the Role of Connectors in Modifying and Implementing Software Architectures. *UC Irvine Technical Report UCI-ICS-98-04*. Department of Information and Computer Science, University of California, Irvine. February 1998.

[22] D. E. Perry, A. L. Wolf, Foundations for the study of software architecture. *Software Engineering Notes*, 17(4), 1992.

[23] J. Purtilo. MINION: An environment to organize mathematical problem solving. *Proceedings of the 1989 International Symposium on Symbolic and Algebraic Computation*, July 1989.

[24] J. Purtilo. The Polylith software bus. *ACM Transactions on Programming Languages and Systems*. 16(1), 1994.

[25] S. P. Reiss. Connecting tools using message passing in the FIELD environment. *IEEE Software*. 7(4):57-67, 1990.

[26] J. E. Robbins, D. F. Redmiles, D. M. Hilbert. Extending design environments to software architecture design. *11th Knowledge-Based Software Engineering Conference (KBSE'96)*. Syracruse, New York. Sept. 1996.

[27] 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,* 20(4):314-335, 1995.

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

[29] 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,* 22(6):390-406, 1996.

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

[31] D. M. Yellin, R. E. Strom. Interfaces, Protocols, and Semi-Automatic Construction of Software Adaptors. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pp 176-190, Portland, OR, USA, October 1994.