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 beneficial role of architectural styles and software connectors in facilitating runtime change. We conclude by describing 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 [1, 2] 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.

This paper presents an architecture-based approach to runtime software reconfiguration, highlighting the beneficial 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 specify several critical aspects of runtime

reconfiguration separate from application-specific behavior. Furthermore, our approach does not dictate particular strategies for implementing or governing runtime reconfigurable systems. Instead, it permits architects to utilize the strategies most appropriate to the application domain and 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, using 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 change may outweigh those associated with shutting down and restarting a system for reconfiguration.

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. An alternative policy may gradually introduce change by binding invocations subsequent to the change to the new functionality, while preserving previously established bindings to the old functionality. Ideally, change application policy decisions should be made by the application designer based on their intimate knowledge of the application domain and 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. An alternative policy may insulate portions of the system not directly affected by a change. The designer’s ability to localize the effects of runtime change by controlling its scope facilitates change management. The designer’s 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 above). 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 [1, 2]. Leveraging the engineer's knowledge at this level of system abstraction holds promise in helping manage runtime change (factor #4). Third, *architectural connectors* help 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 the design of a video game application called KLAX[1]. Other applications implemented using our approach are described in [3] and [4].

3.1 Architectural style

Architectural styles are idiomatic patterns of system organization that characterize a particular application domain [1,2]. 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 input streams and produce data on 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 [5] 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

[1]KLAX is a trademark of Atari Games.

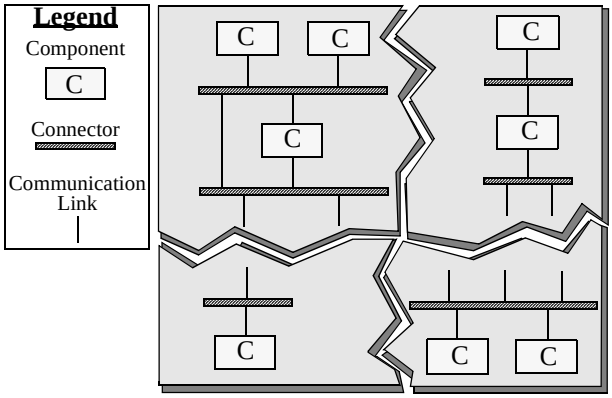


Figure 1. An abstract C2 architecture. Jagged lines represent portions of the architecture not shown.

subsections, we briefly summarize the C2-style, present a C2-style architecture for the KLAX video game application, and highlight characteristics of the C2-style that facilitate runtime reconfiguration.

3.1.1 C2 architectural style

The C2 architectural style[2] 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 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[3]. 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.

[2]The description of the C2 architectural style presented here is summarized from [4] — a more detailed description of the style and its benefits can be found therein.

[3] 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.

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)
isExistingComponent(component)
enumComponents()
addConnector(connector)
removeConnector(connector)
isExistingConnector(connector)
enumConnectors()
weld(connector, component)
weld(component, connector)
weld(connector, connector)
unweld(connector, component)
unweld(component, connector)
unweld(connector, connector)
isWelded(entity, entity)
entitiesBelow(entity)
entitiesAbove(entity)
enumWelds()

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

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.

We have developed an extensible class framework to facilitate the implementation of C2-style applications [6]. 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, and the handle() methods to respond to messages sent from other components. The C2 connector class provides four additional methods—addTopPort(), removeTopPort(), addBottomPort(), removeBottomPort()—for connecting to and

disconnecting from components (or other connectors) during runtime.

The C2 architecture class is a subclass of the component class, thereby allowing the implementation of a component to consist of a lower-level architecture. The architecture class also provides several short-hand methods for starting and stopping all the components and connectors in the architecture, as well as methods for adding, removing, enumerating, and determining the existence of components and connectors in the architecture. The weld() and unweld() methods connect and disconnect components and connectors to one another, respectively. The entitiesBelow() and entitiesAbove() methods enumerate the components and connectors directly connected to a specific component's or connector's top or bottom port, respectively; and the enumWelds() method enumerates all the connections in the architecture. The methods of the architecture class are typically used by architectural tools, but application components could conceivably invoke the same methods.

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 [7] to enable distributed message passing. A similar connector has been implemented in the Java framework using Java's RMI (Remote Method Invocation) mechanism.

3.1.2 Example

Figure 3 illustrates the C2-style architecture for the KLAX video game application. The user interface and the game rules are illustrated in Figure 4. The connectors, $Connector_{1..6}$, are responsible for routing messages between components. For our example, we assume that the connectors use an interprocess communication mechanism.

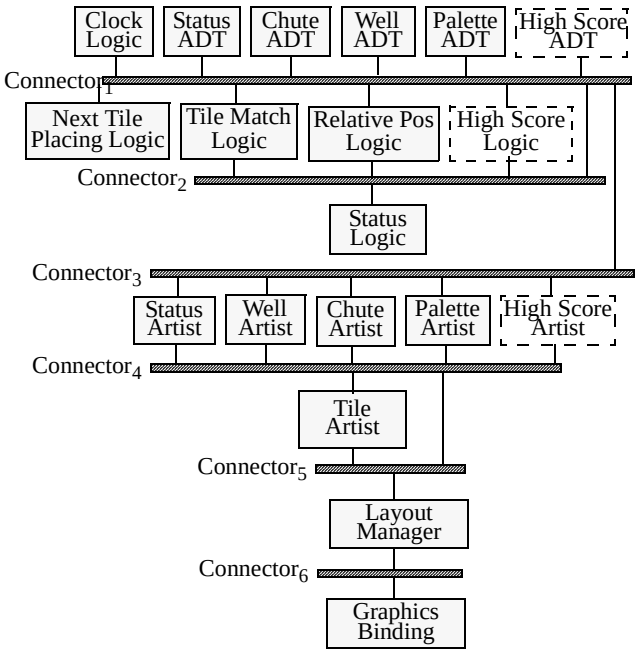
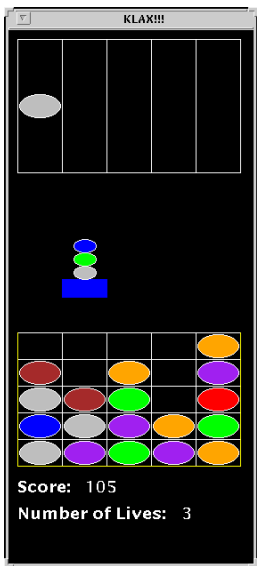


Figure 3. The C2-style architecture for KLAX in which three new components that implement a high score list are added during runtime.



KLAX Chute
Tiles of random colors fall 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 Area

Figure 4. The KLAX user interface and game rules.

The components that make up the KLAX game can be divided into three logical groups. The *game state* components are at the top of the architecture. These components receive no notifications, but respond to requests and emit notifications of internal state changes. The *game logic* components request changes of game state in accordance with game rules and interpret the resulting notifications to determine the state of the game in progress. The *artists* also receive notifications of game state changes, causing them to update their depictions. Each artist maintains a set of abstract graphical objects which, when modified, send state change notifications in hope that lower-level graphics components (in this case, *GraphicsBinding*) will render them. *GraphicsBinding*, in turn, translates user events, such as a key press, into requests to the artist components.

We demonstrate the benefits of the C2 style for runtime reconfiguration by dynamically adding a high score list to KLAX. Our high score list implementation adds three new components: a *High Score ADT* component that maintains a persistent list of the top ten player scores and names, a *High Score Logic* component that decides when the high score list needs to be changed, and a *High Score Artist* component that provides a user interface for displaying the list. Adding the three components that implement the high score list feature is straightforward since they do not adversely affect the other components in the architecture. The *High Score Logic* component waits until the *Status ADT* emits an *endOfGame()* notification, at which time it queries the *Status ADT* for the final game score and the *High Score ADT* for the list of high scores. If the final game score is greater than the lowest score in the high score list, it sends a *setNewHighScore(finalScore)* request to *High Score ADT,* causing the ADT to update its internal state and broadcast a state change notification on $Connector_1$. *High Score Artist* receives this notification (since the notification is also broadcast on $Connector_3$) and responds by creating a dialog box for retrieving the players name and subsequently sending a *setPlayerName(name)* request to the *High Score ADT.*

Removal of the components is also straightforward, though some care must be taken to ensure that each of the high score components has the opportunity to complete any necessary processing. In the case of all three components, 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 components from their respective connectors and terminating its execution.

In this particular application, removing the high score list components and discarding their state does not violate application consistency so long as the components are idle. This is because no components in the application depend on the services or state provided by these components. Such a restriction is clearly inadequate for some of the other components, such as the *Status ADT* since several other 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 components such as the *Status 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 run-time 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 particular forms of application integrity since such rules would prevent the use of the style in some application domains. The other facets of our approach address 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 an artist component is that it be idle. Even this constraint may be unnecessary if the other components in the system are designed to tolerate sudden component failures.

3.2 Architectural connectors[4]

Connectors are explicit architectural entities that bind components together and act as mediators between them [2]. In this way, connectors separate a component's interfacing requirements from its functional requirements [8], and separate component behavior from component interaction. This is especially important when constructing systems from reusable

[4]The roles connectors play in supporting runtime architectural change are summarized from [3].

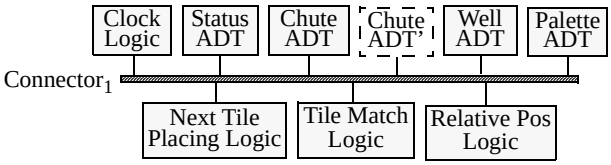


Figure 5. A subset of the KLAX architecture illustrating the gradual replacement of the *Chute ADT* component with *Chute ADT'* component during runtime.

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 [9]; specifying communication mechanism independent of functional behavior, thereby enabling components written in different programming languages and executing on different hosts to transparently interoperate [8]; visualizing and debugging system behavior by monitoring messages between components [10]; and integrating tools by using connectors as message broadcast buses [11].

Connectors play a central role in supporting several aspects of runtime 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 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 KLAX application, and summarize the benefits of utilizing architectural connectors to support runtime reconfiguration.

3.2.1 Example

Figure 5 depicts a subset of the C2-style architecture for the KLAX application in which a new Chute ADT component, *Chute ADT',* is to replace the existing *Chute ADT* component during runtime. 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 *Chute ADT'* is added and connected to the top of *Connector*$_1$ as a replacement for *Chute ADT, Connector*$_1$ directs subsequent *addTile(tile)* requests emitted from the *Next Tile Placing Logic* component to *Chute ADT'*. In this way, new tiles are added to *Chute ADT'* without affecting the other components in the architecture. Once the *Chute ADT* is empty, *Connector*$_1$ can safely disconnect and remove *Chute ADT* 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 [12]. 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 [13]. Consequentially, C2 connectors facilitate runtime reconfiguration by encapsulating:

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 [14] 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 [8] and Field [11], 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. Several other systems described in the literature, such as Conic [15] and DEIMOS [16], also maintain an explicit model of the system to support runtime reconfiguration.
- *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. Several languages for describing change at the architectural level have been described in the literature (see Section 4).
- *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.

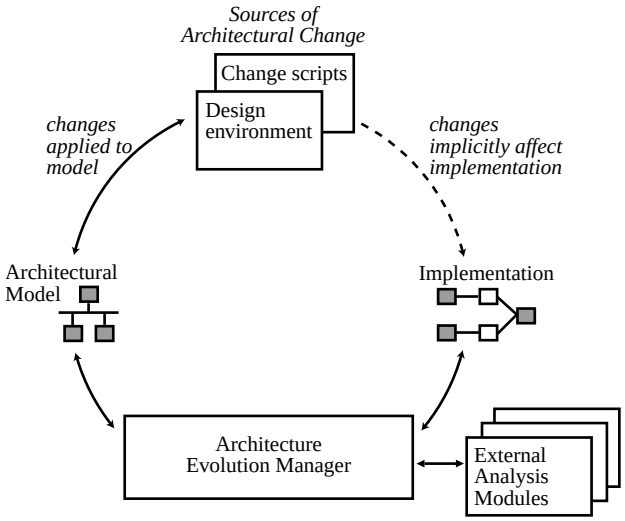


Figure 6. High-level architecture diagram for the ArchStudio tool suite.

- *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 KLAX video game application, and summarize their benefits.

3.3.1 ArchStudio tool suite[5]

Our initial prototype of a tool suite that supports runtime reconfiguration is called ArchStudio. The tools it comprises are implemented in the Java programming language and can modify applications written using the Java-C2 class framework (described in Section 3.1.1). Figure 6 depicts a high-level view of ArchStudio's architecture.

The *architectural model*[6] 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 two mechanisms to constrain runtime changes to the architecture: (1) implicit knowledge of C2-style rules, and (2) an external architectural constraint analysis module called Armani [17]. If a change violates any C2-style rules or any of the constraints specified in Armani, the *AEM* rejects the

[5]The description of the ArchStudio tool suite is based upon the one that appears in [3].

[6]Italicized text in this section denote graphical entities in Figure 6.

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 [18, 19]. 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 (see Figure 2)**.** 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* [20] provides a graphical depiction of the architectural model that may be directly manipulated by the architect and is similar to ConicDraw [21]. 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* [22] provides a similar set of commands using an interactive, textual, command-line interface as opposed to the graphical one of Argo.

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. The reconfiguration script queries and modifies the *architectural model* as necessary, and like other tools, is prevented from violating the constraints enforced by the *AEM*. Hall et al. [23] use a similar approach for deploying software updates.

3.3.2 Example

As in the previous example, we demonstrate how the *Chute ADT* component can be replaced during runtime (refer to Figure 5). 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 Chute ADT is transferred to its replacement. For the purposes of this example, we use the component replacement strategy described by Hofmeister [24] in which each component exposes two additional methods: one for divulging state information, and the another for performing special initialization when replacing a component. Alternative approaches for preserving component state during runtime replacement have been proposed in the literature [25,26].

When the replacement operation is invoked to replace *Chute ADT* with *Chute 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 *Chute ADT*, (3) invokes the *Chute ADT*s special method to divulge state information, (4) disconnects *Chute ADT* from $Connector_1$, (5) invokes the special initialization method of *Chute ADT*' with the state of the original component, (6) connects *Chute ADT*' to $Connector_1$, and (7) directs $Connector_1$ to forward all queued and future messages for *Chute ADT* to *Chute 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 directly 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 during replacement.

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 [27]. 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 [28], Rapide [29], 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. Examples include Clipper [31], Extension Wizard's modification scripts (described in Section 3.3.1), C2's AML [32], and Gerel [33]. All of these languages are operational and utilize many similar constructs.

Architectural constraint languages—Several approaches for specifying architectural constraints have been proposed. Constraint languages have been used to restrict system structure using imperative [34] as well as declarative [17, 28] specifications. Others advocate behavioral constraints on components and their interactions [29]. Configuration graphs that explicitly represent component dependencies have also been proposed [35]. 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 identified several rules of the C2 style that facilitate runtime reconfiguration. We have also 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 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[7]

We are grateful to David Hilbert, Andre van der Hoek, Nenad Medvidovic, Jason Robbins, and David Rosenblum for providing valuable insights on this work. The anonymous referees provided valuable suggestions and criticism.

[7]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

[1] PERRY, D. E. and WOLF, A. L: 'Foundations for the study of software architecture,' Software Engineering Notes, 17(4), 1992.

[2] SHAW, M. and GARLAN, D: 'Software Architecture: Perspectives on an Emerging Discipline', (Prentice-Hall, New York,1996)

[3] OREIZY, P. and MEDVIDOVIC, N. and R. N. TAYLOR: 'Architecture-based Runtime Software Evolution,' Proceedings of the 20th International Conference on Software Engineering (ICSE-20), April 1998, Kyoto, Japan, pp. 177-186.

[4] TAYLOR, R. N. and MEDVIDOVIC, N. and ANDERSON, K. M. and WHITEHEAD, E. J. and ROBBINS, J. E. and NIES, K. A. and OREIZY, P. and DUBROW, D. L.: 'A component- and message-based architectural style for GUI software,' IEEE Transactions on Software Engineering, 1996, 22(6) pp. 390-406.

[5] GARLAN, D. and KAISER, G. E. and NOTKIN, D.: 'Using tool abstraction to compose systems,' IEEE Computer. 1992, 25(6) pp. 30-38.

[6] MEDVIDOVIC, N. and OREIZY, P. and TAYLOR, R. N.: 'Reuse of Off-the-Shelf Components in C2-Style Architectures,' *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)*, May 17-19, 1997, Boston, MA, pp. 190-198.

[7] MAYBEE, M. J. and HEIMBIGNER, D. H. and OSTERWEIL, L. J.: 'Multilanguage Interoperability in Distributed Systems: Experience Report,' Proceedings of the Eighteenth International Conference on Software Engineering, March 1996, Berlin, Germany.

[8] PURTILO, J.: 'The Polylith software bus,' ACM Transactions on Programming Languages and Systems, 1994, 16(1), pp. 151-174.

[9] ALLEN, R. and GARLAN, D.: 'A formal basis for architectural connection,' ACM Transactions on Software Engineering and Methodology, 1997, 6(3) pp. 213-249.

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

[11] REISS, S. P.: 'Connecting tools using message passing in the FIELD environment,' IEEE Software, 1990, 7(4), pp. 57-67.

[12] SHAW, M. and DELINE, R. and KLIEN, D. V. and ROSS, T. L. and YOUNG, D. M. and ZELESNIK, G.: 'Abstractions for software architecture and tools to support them,' IEEE Transactions on Software Engineering, 1995, 20(4) pp. 314-335.

[13] OREIZY, P. and ROSENBLUM, D. S. and TAYLOR, R. N.: '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.

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

[15] KRAMER, J. and MAGEE, J.: 'Dynamic Configuration for Distributed Systems,' IEEE Transactions on Software Engineering, April 1985, 11(4) pp. 424-436.

[16] CLARKE, M. and COULSON, G.: 'An Architecture for Dynamically Extensible Operating Systems,' International Conference on Configurable Distributed Systems, May 1998, Baltimore, MA.

[17] MONROE, R. T.: 'Capturing Software Architecture Design Expertise with Armani: The Armani Language Reference Manual version 1.0,' CMU Technical Report CMU-CS-98-163, School of Computer Science, Carnegie Mellon University, October 1998.

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

[19] MORICONI, M. and QIAN, X. and RIEMENSCHNEIDER, R. A.: 'Correct architecture refinement,' IEEE Transactions on Software Engineering, 1995, 21(4), pp. 356-372.

[20] ROBBINS, J. E. and REDMILES, D. F. and HILBERT, D. M.: 'Extending design environments to software architecture design,' 11th Knowledge-Based Software Engineering Conference (KBSE'96), Syracruse, New York. Sept. 1996.

[21] KRAMER, J. and MAGEE, J. and NG, K.: 'Graphical configuration programming,' IEEE Computer, 1989, 22(10), pp. 53-65.

[22] OREIZY, P.: '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.

[23] HALL, R. S. and HEIMBIGNER, D. and VAN DER HOEK, A. and WOLF, A. L.: 'An architecture for post-development configuration management in a wide-area network,' 17th International Conference on Distributed Computing Systems, Baltimore, Maryland, May 1997.

[24] HOFMEISTER, C. R.: 'Dynamic Reconfiguration of Distributed Applications,' Ph.D. Thesis. University of Maryland, Computer Science Department, 1993.

[25] BLOOM, T. and DAY, M.: 'Reconfiguration and module replacement in Argus: Theory and practice,' IEE Software Engineering Journal, 8(2), March 1993.

[26] FRIEDER, O. and SEGAL, M.: 'On dynamically updating a computer program: From concept to prototype,' Journal of Systems and Software, 1991, 14(2) pp. 111-128.

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

[28] MAGEE, J. and KRAMER, J.: 'Dynamic structure in software architectures,' Fourth SIGSOFT Symposium on the Foundations of Software Engineering, San Francisco, October 1996.

[29] LUCKHAM, D. and VERA, J.: 'An event-based architectural definition language,' IEEE Transactions on Software Engineering, 1995, 21(9) pp. 717-734.

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

[31] AGNEW, B. and HOFMEISTER, C. R. and PURTILO, J.: 'Planning for change: A reconfiguration language for distributed systems,' Proceedings of the International Workshop on Configurable Distributed Systems, March 1994.

[32] MEDVIDOVIC, N.: 'ADLs and dynamic architecture changes,' Second International Software Architecture Workshop (ISAW-2), San Francisco, October 1996.

[33] ENDLER, M. and WEI, J.: 'Programming Generic Dynamic Reconfigurations for Distributed Applications,' Proceedings of the International Workshop on Configurable Distributed Systems, March 1992, London, UK, pp. 68-79.

[34] BALZER, R.: 'Enforcing architectural constraints,' Second International Software Architecture Workshop (ISAW-2), San Francisco, October 1996.

[35] HILTUNEN, M. A.: 'Configuration Management for Highly-Customizable Services,' International Conference on Configurable Distributed Systems, May 1998.