Appeared in the Proceedings of the International Conference on Software Engineering 1998 (ICSE'98). Kyoto, Japan, April 19-25, 1998.
http://www.ics.uci.edu/~peymano/

Architecture-Based Runtime Software Evolution

Peyman Oreizy **Nenad Medvidovic** **Richard N. Taylor**

Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
+1 714 824 8438
{peymano, neno, taylor}@ics.uci.edu

ABSTRACT

Continuous availability is a critical requirement for an important class of software systems. For these systems, runtime system evolution can mitigate the costs and risks associated with shutting down and restarting the system for an update. We present an architecture-based approach to runtime software evolution and highlight the role of software connectors in supporting runtime change. An initial implementation of a tool suite for supporting the runtime modification of software architectures, called ArchStudio, is presented.

1 INTRODUCTION

An important class of safety- and mission-critical software systems, such as air traffic control, telephone switching, and high availability public information systems, shutting down and restarting the system for upgrades incurs unacceptable delays, increased cost, and risk. Support for runtime modification is a key aspect of these systems. Existing software systems that require dynamic update generally adopt ad-hoc, application-specific approaches. Such systems would benefit from a systematic, principled approach to runtime change supported by a reusable infrastructure.

The benefits of runtime evolution are not restricted to safety-intensive, mission-critical systems. A growing class of commercial software applications exhibit similar properties in an effort to provide end-user customizability and extensibility. Runtime extension facilities have become readily available in popular operating systems (e.g., dynamic link libraries in UNIX and Microsoft Windows) and component object models (e.g., dynamic object binding services in CORBA [24] and COM [6]). These facilities enable system evolution without recompilation by allowing new components to be located, loaded, and executed during runtime.

The facilities for runtime modification found in current operating systems, distributed object technologies, and programming languages, have a major shortcoming. They do not ensure the consistency, correctness, or desired properties of runtime change. *Change management* is critical to effectively utilizing mechanisms for runtime change. Change management is a principal aspect of runtime system evolution that:

- helps identify what must be changed,
- provides context for reasoning about, specifying, and implementing change, and
- controls change to preserve system integrity.

Without change management, risks introduced by runtime modifications may outweigh those associated with shutting down and restarting a system.

Software architectures [26, 34] can provide a foundation for systematic runtime software evolution. Architecture shifts developer focus away from lines-of-code to coarse-grained components and their overall interconnection structure. This enables designers to abstract away fine-grained details that obscure understanding and focus on the “big picture:” system structure, the interactions between components, the assignment of components to processing elements, and, potentially, runtime change. A distinctive feature of software architectures is the explicit modeling of *connectors*. Connectors mediate and govern interactions among components, and thereby separate computation from communication, minimize component interdependencies, and facilitate system understanding, analysis, and evolution.

This paper presents an architecture-based approach to runtime software evolution. Several unique elements of our approach are (a) an explicit architectural model, which is deployed with the system and used as a basis for change, (b) preservation of explicit software connectors in the system implementation, and (c) an imperative language for modifying architectures. We also present our initial prototype of a tool suite that supports runtime software evolution at the architectural level.

The paper is organized as follows. Section 2 describes key aspects of effective change management. Section 3 summarizes previous approaches to runtime software change. Section 4 advocates a generic architecture-based approach to runtime change management and demonstrates how different kinds of software evolution are supported at the architectural level. Section 5 describes the role components and connectors play in supporting architectural change. Section 6 describes the particular architectural style that our tool suite, described in Section 7, supports. Section 8 identifies related research areas and Section 9 summarizes the contributions of the paper.

2 MANAGING RUNTIME CHANGE

There are several critical aspects to change management. These determine the degree to which change can be reasoned about, specified, implemented, and governed.

- *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. Approaches that dictate a particular policy may force designers to “design around” the restrictions to attain desired effects.
- *Change scope* is the extent to which different parts of a system are affected by a change. A particular 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 designer’s ability to ascertain change scope helps reason about change.
- *Separation of concerns* captures the degree to which issues concerning a system’s functional behavior are distinguished from those regarding runtime change. The greater the separation, the easier it becomes to alter one without adversely affecting the other.
- The *level of abstraction* at which changes are described impacts the complexity and quantity of information that must be effectively managed.

We refer to these aspects in subsequent sections of the paper when comparing and contrasting different approaches to runtime change.

We also distinguish between two types of change: (1) changes to system requirements, and (2) changes to system implementation that do not alter requirements.

When the requirements change, it is the responsibility of the designer to determine what to change, how to change it, and whether or not the change preserves application integrity. Once a change has been designed, implemented, and tested, it is executed on the running system. It is unrealistic to assume that any preconceived measures for maintaining system integrity would support this type of unpredictable and unrestricted change.

When changes are confined to the implementation, a preconceived set of application invariants may serve as a basis for preserving system integrity. Designers can specify these invariants as a part of the deployed system and prevent changes that violate these invariants.

The inherent difficulty of predicting likely changes during the initial software design phase necessitates that an approach to runtime software evolution support both types of change.

3 PREVIOUS APPROACHES TO RUNTIME CHANGE

Traditionally, designers have sought alternatives to runtime change altogether. A manual override in a safety critical system, for example, relinquishes computer control to a person during system maintenance. If around-the-clock system availability is not required, system updates are postponed until the next scheduled downtime. Some distributed systems employ functional redundancy or clustering as a mechanism to circumvent the need for runtime change. Web servers, for example, are upgraded by redirecting incoming network traffic to a redundant host, reconfiguring the original host in a traditional manner, and redirecting network traffic back to the original host. However, these approaches are not feasible or desirable in all cases due to the increased risk and costs they impose. Our goal is to reduce the costs and risks designers typically associate with runtime change, making it a more attractive design alternative.

Several approaches to runtime software evolution have been proposed in the literature [13, 15, 18, 27, 32]. In the following paragraphs, we describe some representative approaches and evaluate them with respect to the aspects of change management presented in Section 2. We start by discussing techniques for statement- and procedure-level runtime change and move up levels of abstraction.

Gupta et al. [15] describe an approach to modeling changes at the statement- and procedure-level for a simple theoretical imperative programming language. The technique is based on locating the program control points at which all variables affected by a change are guaranteed to be redefined before use. They show that in the general case locating all such control points is undecidable, and approximate techniques based on source code data-flow analysis and developer knowledge are required. Scaling up this approach to manage change in large systems written in complex programming languages is still an open research problem. Dynamic programming languages, such as Lisp and Smalltalk, support statement- and procedure- level runtime change. This flexibility is gained at the expense of heterogeneity and performance. Applications must be written entirely in the dynamic language to benefit from dynamism. This incurs performance overhead because every function invocation must be bound during runtime. Furthermore, application behavior and dynamism are not explicitly separated or localized. As a result, concerns regarding dynamic change permeate system design, making change management exceedingly difficult.

Peterson et al. [27] present an approach to module-level runtime change based on Haskel, a higher-order, typed programming language. Their technique requires programmers to anticipate portions of the program likely to change during runtime, and structure the program around functions that encapsulate such changes. Developers encode decisions regarding change application policy and change scope in the application source code. This technique permits fine-grained control over runtime change since designers can implement change policies tailored to the application. However, because change policies are not isolated in the application source code, they can be difficult to alter independent of application behavior. As a result, managing change in large systems becomes complex.

Gorlick et al. [13, 14] present a data flow based approach to runtime change called Weaves. A weave is an arbitrary network of tool fragments connected together by transport services. Tool fragments communicate asynchronously by passing object references (i.e., pointers). A tool fragment is a small software component, on the order of a procedure, that performs a single, well-defined function and may retain state. Each tool fragment executes in its own thread of control. Transport services buffer and synchronize data communication between tool fragments. The Weave runtime system guarantees the atomicity of data transfer between tool fragments and queues; if any problem occurs during communication, the tool fragment initiating the communication is notified and may retry the operation at its discretion. This enables the runtime reconfiguration of a weave without disturbing the flow of objects. Designers use an interactive, graphical editor to visualize and directly reconfigure a weave during runtime. Weaves does not currently provide a mechanism to check the consistency of runtime changes and no explicit support is provided for representing change policies. The designer is solely responsible for change management.

Kramer and Magee [18] present a structural-based approach to runtime change of a distributed system's configuration. In their approach, a configuration consists of processing nodes interconnected using bidirectional communication links. When a runtime change is required, a reconfiguration manager orders processing nodes directly affected by the change and nodes directly adjacent to them to enter into a "quiescent" state. While in the quiescent state, a node is expected not to initiate communication with peers. This ensures that nodes directly affected by a change will not receive service requests during the course of the change. Changes, specified in a declarative language, are induced to the running system by the reconfiguration manager. The reconfiguration manager is responsible for making decisions regarding the change application policy and its scope. It must do so based on a limited model of the application consisting of the system's structural configuration and whether or not its nodes are in quiescent states. As a result, designers must consider the reconfiguration manager's role in runtime change, and structure the system to attain desired effects.

4 RUNTIME ARCHITECTURAL CHANGE

We advocate an approach that operates at the architectural level. Four immediate benefits result from managing change at the architectural level. First, software engineers use a system's architecture as a tool to describe, understand, and reason about overall system behavior [26, 34]. Leveraging the engineer's knowledge at this level of system design holds promise in helping manage runtime change. Second, if no restrictions are placed on component internals it becomes feasible to accommodate off-the-shelf (OTS) components. Third, decisions regarding change application policy and scope are naturally encapsulated within connectors and separated from application-specific behavior. This facilitates the task of changing policies independent of functional behavior. Fourth, control over change application policy and scope is placed in the hands of the architect, where decisions can be made based on an understanding of application requirements and semantics. Previous approaches to runtime change either dictate a single policy that all systems must adopt or fail to separate application-specific functionality from runtime change considerations. As a result, concerns over runtime change permeate system design.

In the following subsections, we demonstrate how architectures can support different types of software evolution, and the circumstances under which changes may be performed. We refer to three characteristic types of evolution: corrective, perfective, and adaptive [12]. Corrective evolution removes software faults. Perfective evolution enhances product functionality to meet changing user needs. Adaptive evolution changes the software to run in a new environment.

4.1 Runtime Component Addition

Component addition supports perfective evolution by augmenting system functionality. Some design styles are more readily amenable to component addition than others. For example, the observer design pattern [9] separates data providers from its observers, facilitating the addition of new observers with minimal impact on the rest of the system. In the mediator design approach [35], new mediators may be introduced to maintain relationships between independent components. Design approaches that utilize implicit invocation mechanisms [11] are generally more amenable to runtime component addition since the invoking component is unaware of the number of components actually invoked.

In order for a component to function properly when added to a running system, it must not assume that the system is in its initial state. Typically, a component added during runtime must discover the state of the system and perform necessary actions to synchronize its internal state with that of the system.

Architectural change specifications typically specify structural changes necessary to incorporate new components. In some cases, the structural configuration changes may be implicit to the architectural style or application-domain, or derivable from externally visible properties of the component. For example, Adobe Photoshop plug-in components export a "plug-in type" property, whose value is selected from a fixed list [1]. Photoshop uses these values to determine how to interact with the plug-in.

4.2 Runtime Component Removal

Component removal enables a designer to remove unneeded behavior, potentially as a result of recent additions supplanting original behavior. Appropriate conditions governing component removal are application-specific. For example, a system's runtime environment may prohibit component removal if any of its functions are on the execution stack. Some systems, especially distributed systems communicating over inherently undependable connections, are specifically designed to tolerate sudden loss of functionality or state. As with component addition, certain design approaches and styles are more amenable to runtime removal than others.

4.3 Runtime Component Replacement

We consider component replacement as a special case of addition followed by removal when two additional properties are required: (1) the state of the executing component must be transferred to the new component, and (2) both components must not be simultaneously active during the change. Corrective and adaptive evolution are characteristic of such changes.

Component replacement is simple when components lack state or belong to systems specifically designed to tolerate state loss. Such systems typically detect state loss and switch to a degraded mode of operation while recovering. Another approach, exemplified by the Simplex architectural style [32], incorporates an "operational model" in the implementation. The model rejects upgraded components when they do not satisfy explicit performance and accuracy requirements.

In systems not specifically designed to tolerate state loss, component replacement requires additional considerations beyond those discussed for component addition and removal. Several approaches for preserving component state and preventing communication loss during runtime change have been proposed [5, 8, 17]. Hofmeister's approach [17] requires each component to provide two interface methods: one for divulging state information, and the other for performing initialization when replacing another component. These approaches are applicable only when the new component's externally visible interface is a strict superset of the component being replaced. Approaches not restricted in such a manner are an open research topic.

4.4 Runtime Reconfiguration

Structural reconfiguration of the architecture supports recombining existing functionality to modify overall system behavior. Data-flow architectures, such as UNIX's pipe and filter style and Weaves [13], provide substantial flexibility through static reconfiguration of existing behaviors. For example, UNIX's pipe-and-filter style enables construction of a rich set of behaviors through the recombination of existing behavior.

Runtime reconfiguration can be performed by altering connector bindings since connectors mediate all component communication. As with component replacement, if components assume reliable communication, it is necessary to prevent communication loss.

4.5 Summary

It is important to note that with any type of architectural change, concerns regarding the mechanics of change must be separated from the semantic effects of change on the particular application. The injudicious application of architectural changes can compromise system integrity. As a result, such changes must be verified before being applied to a running system. The use of architectural modeling and analysis tools is crucial in this regard.

5 ENABLING RUNTIME ARCHITECTURAL CHANGE

This section outlines the roles components and connectors should play in supporting the architectural changes described in the previous section. The following subsections describe the specific roles components and connectors must fulfill to support runtime change.

5.1 Components

Components are responsible for implementing application behavior. We treat their internal structure as a black box. A component encapsulates functionality of arbitrary complexity, maintains internal state information, potentially utilizes multiple threads of control, and may be implemented in any programming language. Treating components as black boxes significantly increases the opportunity for reusing OTS components. However, OTS component may not be able to participate in runtime change if it lacks certain functionality. For example, the inability to extract component state from a component prevents component replacement. We cannot circumvent these problems without modifying the component.

Components should not communicate by directly referencing one another. Instead, they should utilize a connector, which localizes and encapsulates component interfacing decisions. This minimizes coupling between components, enabling binding decisions to change without requiring component modification [29].

Each component must provide a minimal amount of functional behavior to participate in runtime change. To support runtime addition and removal, components must be packaged in a form that the underlying runtime environment can dynamically load. Most popular operating systems provide a dynamic linking capability. Dynamic linking provides a language-independent mechanism for loading new modules during runtime and invoking the services they export. Higher level mechanisms, such as CORBA [24] and COM [6], provide similar functionality. To support runtime reconfiguration, components must be able to alter their connector bindings. These additional behaviors can typically be provided in the form of reusable code libraries which act as a wrapper or proxy to the actual component (see Section 7). This alleviates the burden of implementing such functionality for every component.

5.2 Connectors

Connectors are explicit architectural entities that bind components together and act as mediators between them [34]. In this way, connectors separate a component's interfacing requirements from its functional requirements [29]. Connectors encapsulate component interactions and localize decisions regarding communication policy and mechanism. As a result, connectors have been used for a wide variety of purposes, including: ensuring a particular interaction protocol between components [3]; specifying communication mechanism independent of functional behavior, thereby enabling components written in different programming languages and executing on different processors to transparently interoperate [29]; visualizing and debugging system behavior by monitoring messages between components [28]; and integrating tools by using a connector to broadcast messages between them [30].

Although connectors are explicit entities during design, they have traditionally been implemented as indiscrete entities in the implementation. In UniCon, for example, procedure call and data access connectors are reified as linker instructions during system generation [33]. Similarly, component binding decisions, while malleable during design, are typically fixed during system generation. As a result, modifying binding decisions during runtime becomes difficult.

Connectors, like components, must remain discrete entities in the implementation to support their runtime addition and removal. They must also provide a mechanism for adding and modifying component bindings in order to support reconfiguration.[1] Supporting runtime rebinding can degrade performance in primitive connectors, such as procedure calls, since an additional level of indirection is introduced. For more complex connectors, such as RPC and software buses (e.g. Field [30]), the functionality we require can usually be integrated without a significant runtime performance penalty. Recent approaches to dynamic linking attempt to reduce or eliminate the runtime overhead associated with altering binding decisions during runtime [7]. Ultimately, designers should determine which connectors are used based on application requirements. If runtime change is not required, connectors without rebinding overhead may be used.

Connectors play a central role in supporting several aspects of change management. They can implement different change policies by altering the conditions under which newly added components are invoked. For example, to support immediate component replacement, a connector can direct all communication after a certain point in time away from the old component to the new one. To support a more gradual component replacement policy, a connector can direct new service requests to the new component, while directing previously established requests to the original component. To support a policy based on replication, service requests can be directed 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. Using connectors to encapsulate change application policy and scope decisions lets designers select the most appropriate policy based on application requirements.

6 APPLYING CONCEPTS TO A SPECIFIC ARCHITECTURAL STYLE

We are developing general techniques for runtime architecture evolution that are applicable across application domains, architectural styles, and architecture modeling notations. We are also investigating a general formal basis for architectural dynamism. However, the field of software architectures is still relatively young and largely unexplored. This is particularly true of dynamism: we can learn from traditional approaches to dynamism, outlined in Section 3, but some of the issues they raise will be inapplicable to architectures; additionally, architectures are likely to introduce other, unique problems, such as supporting heterogeneity, adhering to architectural styles, and maintaining compatibility with OTS components.

For these reasons, our initial strategy has been to address concrete problems and learn from experience. We have focused on supporting architectures in a layered, event-based architectural style, called C2 [36]. In the C2-style, all communication among components occurs via connectors, thus minimizing component interdependencies and strictly separating computation from communication. The style also imposes topological constraints: every component has a “top” and a “bottom” side, with a single communication port on each side. This restriction greatly simplifies the task of adding, removing, or reconnecting a component. A C2 connector also has a top and a bottom, but the number of communication ports is determined by the components attached to it: a connector can accommodate any number of components or other connectors. This enables C2 connectors to accommodate runtime rebinding. Finally, all communication among components is done asynchronously by exchanging messages through connectors.

Although the C2-style places several restrictions on architectures and architectural building blocks, we believe these restrictions to be permissive enough to allow us to model a broad class of applications. Narrowing our focus has enabled us to construct tools for supporting runtime architectural change. As a result, we’ve gained direct practical experience with runtime evolution of architectures and uncovered important issues in effectively supporting them.

7 TOOLS SUPPORTING ARCHITECTURE-BASED EVOLUTION OF SOFTWARE SYSTEMS

This section describes ArchStudio, our tool suite that implements our architecture-based approach to runtime software evolution. The following subsections describe our general approach to enabling evolution of software systems at the architectural level. We then present an implementation based on this approach and demonstrate its use on a simple application. We conclude by discussing the current limitations of our implementation.

7.1 Approach

Our general approach to supporting architecture-based software evolution consists of several interrelated mechanisms (see Figure 1). The mechanisms are described below. Section 7.2 describes our implementation of these mechanisms.

Explicit Architectural Model. In order to effectively modify a system, an accurate model of its architecture must be available during runtime. To achieve this, a subset of the system’s architecture is deployed as an integral part of the system. The deployed architectural model describes the

[1] Runtime rebinding can be supported without explicit connectors by essentially replacing relevant machine language instructions during runtime. This technique is highly dependent on the execution environment (memory protection, restrictions on self-modifying code, etc.) and the programming language and compiler optimizations (polymorphic functions, function inlining, etc.).

interconnections between components and connectors, and their mappings to implementation modules. The mapping enables changes specified in terms of the architectural model to effect corresponding changes in the implementation. The runtime architecture infrastructure maintains the correspondence between the model and the implementation.

Describing Runtime Change. Modifications are expressed in terms of the architectural model. A modification description uses operations for adding and removing components and connectors, replacing components and connectors, and changing the architectural topology.

This approach supports a flexible model of system evolution in which modifications are provided by multiple organizations (e.g., the application vendor, system integrators, site managers) and selectively applied by end-users based on their particular needs. By applying different sets of modifications, an end-user can effectively create a different member of the system family at her site. As a result, the modifications should be robust to variations in those systems. Facilities for querying the architectural model and using the results of the query to guide modifications should be provided as an integral part of supporting architectural change. Using the model to inform and guide modifications eliminates many accidental difficulties inherent in evolving systems.

Governing Runtime Change: Our approach to runtime system evolution supports a mechanism for restricting changes that compromise system integrity. Constraints play a natural role in governing change, and several approaches to applying them at the architectural level have been developed (see Section 8). In addition, mechanisms governing runtime change should also constrain *when* particular changes may occur.

During the course of a complex modification, the system's architecture may "move" through several invalid states before reaching a final valid state. Although constraints may legitimately restrict certain modification "paths", doing so solely based on intermediate invalid states will prevent some classes of valid runtime changes. As a result, a mechanism that supports transactional modifications should be provided.

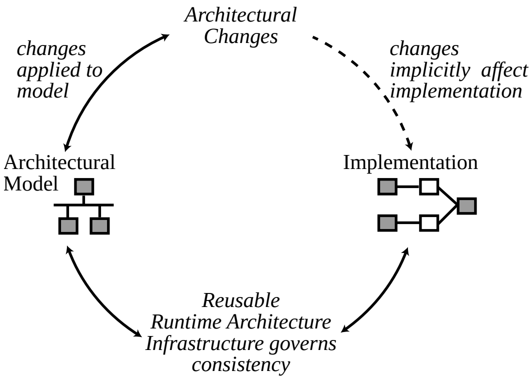

Architectural Changes
changes applied to model
changes implicitly affect implementation
Architectural Model
Implementation
Reusable Runtime Architecture Infrastructure governs consistency

Figure 1. Architectural changes applied to the model are reified into implementation by the runtime architecture infrastructure.

Reusable Runtime Architecture Infrastructure: The runtime architecture infrastructure (a) maintains the consistency between the architectural model and implementation as modifications are applied, (b) reifies changes in the architectural model to the implementation, and (c) prevents runtime changes from violating architectural constraints. As a result, the runtime architecture infrastructure can support different component addition, removal, and replacement policies and can be tailored for particular application domains. The runtime architecture infrastructure uses the architectural model's implementation mapping and the facilities of the underlying environment to implement changes.

7.2 Archstudio: A Tool Suite For Runtime Modification Of C2-style Architectures

This section describes our initial prototype of a tool suite, ArchStudio, which implements the mechanisms described in the preceding section. The tools comprising ArchStudio are implemented in the Java programming language, and can modify C2-style applications written using the Java-C2 class framework [22]. The Java-C2 class framework provides a set of extensible Java classes for fundamental C2 concepts such as components, connectors, and messages. Developers create new components and connectors by subclassing from framework classes and providing application-specific behavior. Connectors remain discrete entities in the implementation, and support runtime rebinding through a set of functions they export. Connectors that utilize intra- and inter-process communication facilities are provided with the framework.

Figure 2 depicts a conceptual view of the ArchStudio architecture. The *Architectural Model* represents an application's current architecture. Our current implementation encapsulates the architectural model in an abstract data type (ADT). This ADT exports operations for querying and changing the application's architectural model.

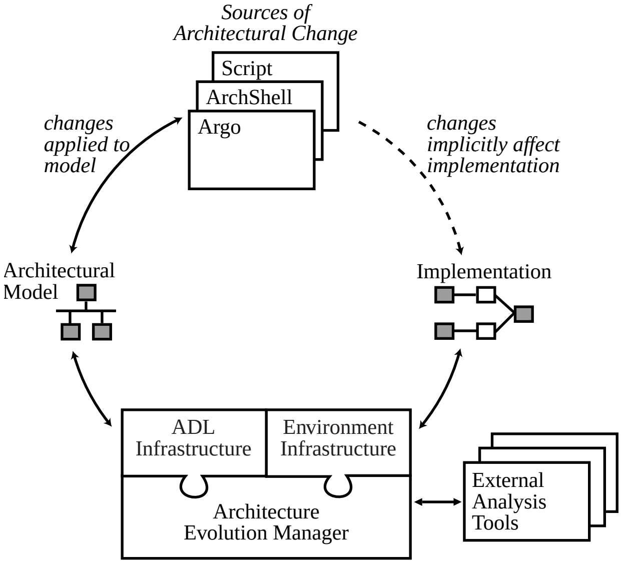
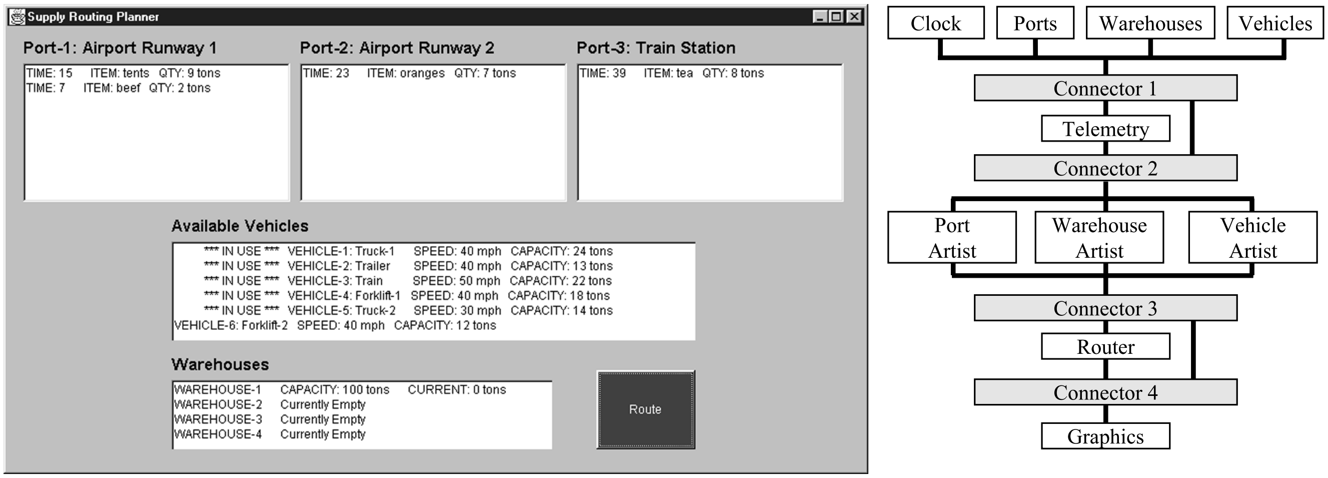

Sources of Architectural Change
Script
ArchShell
Argo
changes applied to model
changes implicitly affect implementation
Architectural Model
Implementation
ADL Infrastructure
Environment Infrastructure
Architecture Evolution Manager
External Analysis Tools

Figure 2. A conceptual architecture diagram for the ArchStudio tool suite.



Port-1: Airport Runway 1

```
TIME: 15     ITEM: tents   QTY: 9 tons
TIME: 7     ITEM: beef   QTY: 2 tons
```

Port-2: Airport Runway 2

```
TIME: 23     ITEM: oranges   QTY: 7 tons
```

Port-3: Train Station

```
TIME: 39     ITEM: tea   QTY: 8 tons
```

Available Vehicles

```
       *** IN USE ***   VEHICLE-1: Truck-1      SPEED: 40 mph   CAPACITY: 24 tons
       *** IN USE ***   VEHICLE-2: Trailer      SPEED: 40 mph   CAPACITY: 13 tons
       *** IN USE ***   VEHICLE-3: Train        SPEED: 50 mph   CAPACITY: 22 tons
       *** IN USE ***   VEHICLE-4: Forklift-1   SPEED: 40 mph   CAPACITY: 18 tons
       *** IN USE ***   VEHICLE-5: Truck-2      SPEED: 30 mph   CAPACITY: 14 tons
VEHICLE-6: Forklift-2   SPEED: 40 mph   CAPACITY: 12 tons
```

Warehouses

```
WAREHOUSE-1     CAPACITY: 100 tons     CURRENT: 0 tons
WAREHOUSE-2     Currently Empty
WAREHOUSE-3     Currently Empty
WAREHOUSE-4     Currently Empty
```

Figure 3. (a) On the left, the cargo routing system's user interface. (b) On the right, the architecture of the cargo routing system in the C2-style.

The model is stored in a structured ASCII format and maintained by the runtime architecture infrastructure. 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 *ADL* and *Environment infrastructure* insulate the AEM from changes to the ADL and runtime environment. The AEM may utilize an architectural constraint mechanism or external analysis tools to determine if a change is acceptable. The current implementation of the AEM uses implicit knowledge of C2-style rules to constrain changes; the addition of an architectural constraint mechanism and the ability to utilize external analysis tools is planned for the future. If a change violates the C2-style rules, the AEM rejects the change. Otherwise, the architectural model is altered and its implementation mapping is used to make the corresponding modification to the Implementation.

ArchStudio currently includes three tools which act as *Sources of Architectural Modification:* Argo, ArchShell, and the Extension Wizard. In addition to these tools, an application can obtain access to its own architectural model and manipulate itself using the same set of mechanisms.

Argo [31] provides a graphical depiction of the architectural model that the architect may manipulate directly. 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. The configuration is altered by directly manipulating the links between components and connectors.

ArchShell [25] is an alternative to Argo that provides a textual, command-driven interface for specifying runtime modifications. Commands are provided for adding and removing components and connectors, reconfiguring the architecture, and displaying a textual representation of the architecture. ArchShell provides two commands currently not available in Argo. The first command enables the architect to send arbitrary messages to any component or connector in the same manner as if they were sent from another component or connector. This facilitates debugging and exploration of architectural behavior.

As design tools for architects, Argo and ArchShell facilitate rapid exploration of architectural designs. They also provide valuable feedback in exploring proposed runtime architectural changes.

Argo and ArchShell are interactive tools used by software architects to describe architectures and architectural modifications. Once an architect has specified and verified a modification, she uses the *Extension Wizard* to deploy the modification to end-users. The Extension Wizard provides a simple end-user interface for enacting runtime modifications and is deployed as a part of the end-user system. The Extension Wizard is responsible for executing a modification script on the end-user's installation of the system. End-users use a Web browser to display a list of downloadable system update files, e.g., provided on the application vendor's Web site. A system update file is a compressed file containing a runtime modification 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 modification script it contains, and executes it. A similar approach for deploying system updates is used by Hall et al. [16].

7.3 The Cargo Routing System Example

We demonstrate the use of our tool suite using a simple logistics system for routing incoming cargo to a set of warehouses. Figure 3(a) shows the system's user interface. The three list boxes on the top represent three incoming cargo delivery ports, in this case two airport runways and a train station. When cargo arrives at a port, an item is added to the port's list box. The system keeps track of each

```
> add component
ClassName: c2.planner.RouterArtist
Name? RouterArtist
> weld
Top entity: Connector1
Bottom entity: RouterArtist
> weld
Top entity: RouterArtist
Bottom entity: Connector4
> start
Entity: RouterArtist
```

Figure 4. The ArchShell commands used to add the Router Artist component. Commands are denoted using bold text and command arguments are denoted using italicized text.

shipment’s content, weight, and the amount of time it has been sitting idle at the port. The text box in the center displays available vehicles for transporting cargo to destination warehouses. The system displays the vehicle’s name, maximum speed, and maximum load. The bottom most text box displays a list of destination warehouses. The system displays each warehouse’s name, maximum capacity, and currently used capacity. End-users route cargo by selecting an item from a delivery port, an available vehicle, and a destination warehouse, and then clicking the “Route” button.

Figure 3(b) depicts the architecture of the cargo routing system in the C2 architectural style. The *Ports*, *Vehicles*, and *Warehouses* components are ADTs which keep track of the state of ports, the transportation vehicles, and the warehouses, respectively. The *Telemetry* component determines when cargo arrives at a port, and tracks the cargo from the time it is routed until it is delivered to the warehouse. The *Port Artist*, *Vehicle Artist*, and *Warehouse Artist* components are responsible for graphically depicting the state of their respective ADTs to the end-user. The *Router* component sends a message to the telemetry component when the end-user presses the “Route” button and provides the end-user’s last selected port, vehicle, and warehouse. The *Graphics* component renders the drawing requests sent from the artists using the Java AWT graphics package.

We now describe the use of ArchShell and Extension Wizard in adding new functionality to the system. ArchShell is used to add a new graphical visualization of cargo routing, and an Extension Wizard script is used to add an automated planning component that assists users in making optimal routing decisions. Both changes are made during execution of the cargo routing system.

Adding the new visualization requires adding a *Router Artist* component to the architecture. We add the new router artist between Connector 1 and Connector 4 because it uses notification messages provided by the *Port*, *Warehouse*, and *Vehicle* ADTs and utilizes the *Graphics* component for drawing graphics. The architect uses ArchShell to add the component using the “add component” command, connect it to buses using the “weld” command, and signal that the component should receive execution cycles using the “start” command (see Figure 4).

Adding the automated planner involves adding a *Planner* component to the architecture. The new planner component is added below Connector 1 because it monitors the state of the ADTs to determine optimal routes. Figure 5 shows the critical portion of the modification script the Extension Wizard executes when installing the change. The script determines if the architectural model is that of the cargo routing system, then queries the model to determine the names of the connectors to which the planner component must be attached. If any of these operations fail, an exception is thrown which aborts the installation. An operation may fail if the architectural elements on which the change relies have been previously altered by other architectural modifications.

```
try {
      if (model.architectureName().equals(“CargoSystem”)) {
            Connector above = model.connectorBelow(“Ports”);
            Connector below = model.connectorAbove(“PortArtist”);
            model.addComponent("Planner", "planner");
            model.weld(above, "planner");
            model.weld("planner", below);
            model.startEntity("planner");
            return true;
      } else return false;
} catch (ArchitectureModificationException e) {
      return false;
}
```

Figure 5. A portion of the Extension Wizard script used to add the Planner component into the running system. The “model” represents the ADT interface to the system’s architectural model.

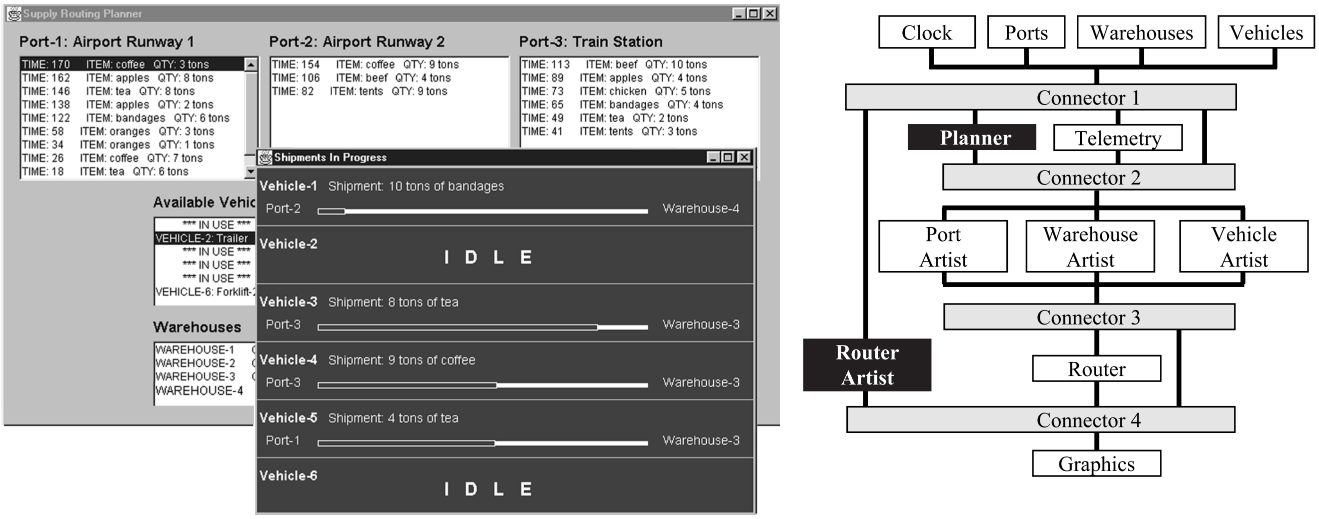
Figure 6 depicts the updated user interface and architecture after both modifications have been made.

Supporting runtime modification requires the deployment of the Architecture Evolution Manager, the Extension Wizard, and a portion of the cargo routing system’s architectural model. The Architecture Evolution Manager and the Extension Wizard consist of 38 kilobytes of compiled Java code. The cargo routing system’s architectural model consumes 2 kilobytes of disk space. The Planner system update, which consists of the modification script and the compiled Planner component, is 6 kilobytes.

7.4 Limitations and Future Work

Our prototype facilitates exploration of architectural dynamism, but has several practical limitations. Currently, all components and connectors must be written using the Java-C2 class framework. The framework, however, does not make any assumptions about execution threads and processes or message passing protocols. This has allowed us to implement runtime component addition using Java’s dynamic class loading facilities. In the future, we plan to use language independent facilities, such as those provided by CORBA and COM.

Currently, C2 components communicate by passing asynchronous message through connectors. Although techniques for emulating other communication mechanisms (e.g., method invocation, shared memory) atop a message passing substrate have been developed by researchers in the parallel algorithms domain, we are investigating how these different communication mechanisms impact dynamism.



```
Supply Routing Planner

Port-1: Airport Runway 1
TIME: 170   ITEM: coffee   QTY: 3 tons
TIME: 162   ITEM: apples   QTY: 8 tons
TIME: 146   ITEM: tea   QTY: 8 tons
TIME: 138   ITEM: apples   QTY: 2 tons
TIME: 122   ITEM: bandages   QTY: 6 tons
TIME: 58   ITEM: oranges   QTY: 3 tons
TIME: 34   ITEM: oranges   QTY: 1 tons
TIME: 26   ITEM: coffee   QTY: 7 tons
TIME: 18   ITEM: tea   QTY: 6 tons

Port-2: Airport Runway 2
TIME: 154   ITEM: coffee   QTY: 9 tons
TIME: 106   ITEM: beef   QTY: 4 tons
TIME: 82   ITEM: tents   QTY: 9 tons

Port-3: Train Station
TIME: 113   ITEM: beef   QTY: 10 tons
TIME: 89   ITEM: apples   QTY: 4 tons
TIME: 73   ITEM: chicken   QTY: 5 tons
TIME: 65   ITEM: bandages   QTY: 4 tons
TIME: 49   ITEM: tea   QTY: 2 tons
TIME: 41   ITEM: tents   QTY: 3 tons

Available Vehic
*** IN USE ***
VEHICLE-2: Trailer
*** IN USE ***
*** IN USE ***
*** IN USE ***
VEHICLE-6: Forklift-2

Warehouses
WAREHOUSE-1
WAREHOUSE-2
WAREHOUSE-3
WAREHOUSE-4

Shipments In Progress
Vehicle-1   Shipment: 10 tons of bandages
  Port-2                                  Warehouse-4
Vehicle-2
                 I D L E
Vehicle-3   Shipment: 8 tons of tea
  Port-3                                  Warehouse-3
Vehicle-4   Shipment: 9 tons of coffee
  Port-3                                  Warehouse-3
Vehicle-5   Shipment: 4 tons of tea
  Port-1                                  Warehouse-3
Vehicle-6
                 I D L E
```

Figure 6. (a) On the left, the cargo routing system user interface after the addition of the new router artist and planner components. (b) On the right, the updated cargo system architecture highlighting new components.

For simplicity, we assume a one-to-one mapping between components in the architectural model and Java implementation classes. This enables us to focus on dynamism independently of issues concerning mappings between architectures and their implementations, which is an open research area of significant complexity [10, 23].

The runtime architecture infrastructure currently supports the addition and removal of components and connectors, and the reconfiguration and querying of the architectural model. There is currently no support for component replacement, though the implementation allows currently available approaches to be adopted.

Finally, our current implementation is limited to checking invariants derived from the C2-style rules. The addition of a general purpose architectural-constraint mechanism that supports application-specific invariants is the focus of future work. Our positive experience with incorporating the C2-style invariants suggests that our approach will support a more general mechanism.

8 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. 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 static description of a system, but provide no facilities for specifying runtime architectural changes. Although a few ADLs, such as Darwin [20], Rapide [19], and LILEANNA [37], can express runtime modification to architectures, they require that the modifications be specified during design and "compiled into" the application. Our approach, in contrast, can accommodate unplanned modifications of an architecture and incorporate behavior not anticipated by the original developers. It is important to note that our approach does not attempt to replace static architecture description languages. In fact, our tools can utilize existing ADLs, instead of our own, for the static portion of the architectural model. In this way, our approach augments current ADLs with runtime change support.

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 [21], and Clipper [2] are examples of such languages and share 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 [4] as well as declarative [20] specifications. Others advocate behavioral constraints on components and their interactions [19]. Finding appropriate mechanisms for governing architectural change using constraints is an active topic of ongoing research.

9 CONCLUSIONS

Software architectures have the potential to provide a foundation for systematic runtime software modification, as opposed to brittle, "one-of-a-kind" patches. An effective approach to runtime change can reduce the risks and costs typically associated with such change. Our experience demonstrates that an architecture-based approach to runtime software evolution provides several unique benefits over previous approaches. These benefits include a common representation for describing software systems and managing runtime change, separation of computation from communication, and encapsulation of change application policies and scope within connectors.

Our work has benefited from hands-on experience with architectural dynamism. In the process, we have produced a set of results that are generally applicable to the problem of runtime software evolution. We have confirmed the central role of connectors in supporting runtime change and identified the desired characteristics of connectors that facilitate that change. We have also demonstrated the role of connectors in supporting different change policies. We have recognized the need for both architecture-specific (structural) and application-specific (behavioral) constraints in making runtime changes, as well as the need for transaction support during those changes. Finally, a simple imperative modification language has proven to be adequate for specifying a broad class of runtime changes.[1]

REFERENCES

1. Adobe Systems Incorporated. *Adobe Photoshop Plug-In SDK*. http://www.adobe.com/supportservice/devrelations/sdks.html. 1997.
2. B. Agnew, C. R. Hofmeister, J. Purtilo. Planning for change: A reconfiguration language for distributed systems. *Proceedings of CDS'94*, 1994.
3. R. Allen, D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.
4. R. Balzer. Enforcing architectural constraints. *Second International Software Architecture Workshop (ISAW-2)*, San Francisco, October 1996.
5. T. Bloom, M. Day. Reconfiguration and module replacement in Argus: Theory and practice. *IEE Software Engineering Journal*, vol 8, no 2, March 1993.
6. K. Brockschmidt. *Inside OLE 2*. Microsoft Press, 1994.
7. M. Franz. Dynamic linking of software components. *IEEE Computer*, vol 30, no 3, pp 74-81, March 1997.
8. O. Frieder, M. Segal. On dynamically updating a computer program: From concept to prototype. *Journal of Systems and Software*, vol 14, pp 111-128. 1991.
9. E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
10. D. Garlan. Style-based refinement for software architecture. *Second International Software Architecture Workshop (ISAW-2)*. San Francisco, CA, October 1996.
11. D. Garlan, G. E. Kaiser, D. Notkin. Using tool abstraction to compose systems. *IEEE Computer.* vol 25,no 6, pp 30-38, June 1992.
12. C. Ghezzi, M. Jazayeri, D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, 1991.
13. M. M. Gorlick, R. R. Razouk. Using weaves for software construction and analysis. *Proceedings of the 13th International Conference on Software Engineering*. IEEE Computer Society Press, May 1991.
14. M. M. Gorlick, A. Quilici. Visual programming-in-the-large versus programming-in-the-small. *Proceedings of the IEEE Symposium on Visual Languages*. IEEE Computer Society Press, October 1994.
15. D. Gupta, P. Jalote, G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, vol 22, no 2, February 1996.
16. 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.
17. C. R. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. Ph.D. Thesis. University of Maryland, Computer Science Department, 1993.
18. J. Kramer, J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, vol 16, no 11, Nov. 1990.
19. D. Luckham, J. Vera. An event-based architectural definition language. *IEEE Transactions on Software Engineering*, pp 717-734, September 1995.
20. J. Magee, J. Kramer. Dynamic structure in software architectures. *Fourth SIGSOFT Symposium on the Foundations of Software Engineering*, San Francisco, October 1996.
21. N. Medvidovic. ADLs and dynamic architecture changes. *Second International Software Architecture Workshop (ISAW-2)*, San Francisco, October 1996.
22. N. Medvidovic, P. Oreizy, R. N. Taylor. Reuse of off-the-shelf components in C2-style architectures. *Symposium on Software Reusability*, Boston, May 1997.
23. M. Moriconi, X. Qian, R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*. pp 356-372, April 1995.
24. Object Management Group. *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, July 1996. http://www.omg.org/corba/corbiiop.htm
25. 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.
26. D. E. Perry, A. L. Wolf, Foundations for the study of software architecture. *Software Engineering Notes*, vol 17, no 4, October 1992.
27. J. Peterson, P. Hudak, G. S. Ling. Principled dynamic code improvement. *Yale University Research Report YALEU/DCS/RR-1135*. Department of Computer Science, Yale University, July 1997.
28. J. Purtilo. MINION: An environment to organize mathematical problem solving. *Proceedings of the 1989 International Symposium on Symbolic and Algebraic Computation*, July 1989.
29. J. Purtilo. The Polylith software bus. *ACM Transactions on Programming Languages and Systems*. vol 16, no 1, Jan. 1994.
30. S. P. Reiss. Connecting tools using message passing in the FIELD environment. *IEEE Software*. vol 7, no 4, pp 57-67, July 1990.
31. 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.
32. L. Sha, R. Rajkumar, M. Gagliardi. Evolving dependable real-time systems. *IEEE Aerospace Applications Conference*. New York, NY, pp 335-346, 1996.
33. 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*, pp 314-335, April 1995.
34. M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.
35. K. Sullivan, D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology.* vol 1, no 3, pp 229-268, July 1992.
36. 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*, pp 390-406, June 1996.
37. W. Tracz. Parameterized programming in LILEANNA. *Proceedings of ACM Symposium on Applied Computing SAC'93*, February 1993.

[1] 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.