Architecture-Based Runtime Software Evolution

Peyman Oreizy, Nenad Medvidovic, Richard N. Taylor

Department of Information and Computer Science
University of California, Irvine, CA 92697

Technical Report 97-39

September 1, 1997

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.

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

ABSRTACT

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

For 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. Supporting 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 broadening class of commercial software applications is beginning to exhibit similar properties in an effort to provide end-user customizability and extensibility. Runtime extension facilities have become readily available in many popular operating systems (e.g., dynamic link libraries in UNIX and Microsoft Windows) and as parts of component object models (e.g., dynamic object binding services in CORBA [23] and COM [6]). These facilities enable system evolution without recompilation by allowing new components to be located, loaded, and executed during runtime.

The kinds of support for runtime modification found in operating systems, distributed object technologies, and programming languages, have a major shortcoming however. Enabling dynamism is, by itself, not sufficient to ensure the consistency, correctness, or desired results of runtime change. Support for *change management* is critical to effectively utilizing mechanisms for runtime change. Change management is a principle aspect of runtime system evolution that:

- helps identify what must be changed,
- provides a 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 [25, 33] have the potential to provide a foundation for systematic runtime software evolution. Architectures shift the development focus away from lines-of-code to coarse-grained components and their overall interconnection structure. This enables designers to abstract away unnecessary details and focus on the "big picture:" system structure, communication protocols of interacting components, assignment of software components to processing elements of the execution environment, and, potentially, runtime change. Central to architectures, in our view, are *connectors*, which mediate and govern interactions among components. Using connectors separates computation from communication, minimizes component interdependencies, and facilitates system understanding, analysis, and evolution.

Architectures provide three key enables for effective change management.

- Explicit system structure allows dynamism to be modeled separately from the functional behavior of components.
- Separation of architecture from implementation enables architectural changes to be analyzed before they are applied to the running system.
- Separation of computation from communication allows designers to focus on either runtime change to the functionality (components) or to communication mechanisms and protocols (connectors).

This paper presents an architecture-based approach to runtime software evolution. Three 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 implemented systems, and (c) an imperative language for modifying architectures. We have completed an initial prototype of a tool suite, ArchStudio, 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 an 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 roles components and connectors play in supporting these architectural changes. Section 6 briefly describes the particular architectural style that our tool suite, described in Section 7, supports. Section 8 identifies research areas relevant to this work and Section 9 summarizes the contributions of the paper.

2 MANAGING RUNTIME CHANGE

This section describes several critical aspects of managing change. These aspects determine the degree to which change can be reasoned about, specified, implemented, and governed.

- *Change policy* controls how a change is applied to a running system. A particular 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, and preserving bindings previously established to the old functionality. Ideally, policy decisions should be made by the designer based on application requirements. Dictating a particular policy negatively influences design since designers must "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.
- *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 managed plays a significant role in determining the complexity 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.

3 PREVIOUS APPROACHES TO RUNTIME CHANGE

Traditionally, designers have sought alternatives to runtime change all together. Safety critical systems, for example, typically provide manual overrides that relinquish 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 to circumvent the need for runtime change. A Web server, for example, can be 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 may not be feasible in some cases due to the increased risk and costs they impose. Our hope 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 [14, 17, 26, 31]. In the following paragraphs, we describe several 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. describe an approach to modeling changes at the statement- and procedure-level for a simple theoretical imperative programming language [14]. The technique is based on locating 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 popular programming languages is still an open research problem. Dynamic programming languages, such as Lisp and Smalltalk, also support statement- and procedure- level runtime change. However, this flexibility is gained at the expense of heterogeneity and performance. Applications must be written entirely in the given language to benefit from dynamism. This incurs performance overhead since every function invocation must be bound dynamically. 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. present an approach to runtime change at the module-level based on Haskel, a higher-order, typed programming language [26]. The technique requires programmers to anticipate portions of the program that might change during runtime, and structure the program around functions that encapsulate such changes. Application programmers encode decisions regarding change policy and scope in the application source code. Permitting fine-grained control over runtime change enables designers to implement change policies best suited for the application. However, requiring that these policies be implemented in the source code makes them difficult to alter independently of application behavior. As a result, managing change in large systems becomes overly complex.

Kramer and Magee present an approach to runtime change at the structure-level based on distributed system configurations [17]. Configuration in a distributed system consists of processing nodes interconnected using unidirectional communication links, called connections. To make a particular runtime change, nodes directly affected by the change as well as their adjacent nodes are directed to enter into a "quiescent" state by a reconfiguration manager. While in the quiescent state, a node voluntarily agrees 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 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 when managing change at the architectural level. First, software engineers commonly use the system architecture when describing, understanding, and reasoning about overall system behavior [25, 33]. Leveraging the engineer's knowledge at this level of system design holds promise in helping manage runtime change. Second, accommodating commercial-off-the-shelf components becomes feasible as long as no restrictions are placed on component internals. Third, decisions regarding change policy and scope are separated from application-specific behavior allowing them to be altered independently. Fourth, control over change 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. 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.

In the following subsections, we demonstrate how architectures can support different types of software evolution and the circumstances under which the change may or may not be performed. Three characteristic types of evolution are 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 through augmentation of 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 to the rest of the system. In the mediator design approach [34], 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 configuration changes for incorporating new components. In some cases, the structural configuration changes may be implicit in the architectural style, 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 when determining how to interact with the plug-in.

4.2 Runtime Component Removal

Component removal supports the extraction of unneeded behavior, potentially as a result of recent additions supplanting original behavior. Appropriate conditions governing component removal are application-specific. For example, the system's execution model 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. 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 of runtime change 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 largely trivial 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 [31], 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 already discussed for component addition and removal. Several approaches for preserving component state and preventing communication loss during runtime change have been proposed [5, 8, 16]. Hofmeister's approach [16] requires each component to provide two additional interface methods: one for divulging state information, and the other for performing special initialization when replacing another component. These approaches are generally 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. Systems based on data-flow architectures, such as UNIX's pipe and filter style and Weaves [13], have attained significant flexibility by supporting static reconfiguration of existing behaviors. UNIX's pipe-and-filter style, for example, enables the construction of a rich set of behaviors through the recombination of existing behavior.

Since connectors mediate component communication, runtime reconfiguration can be performed by altering connector bindings. As with component replacement, preventing communication loss may be necessary if components assume reliable communication.

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. If any of the architectural changes discussed above are applied injudiciously, system integrity can be compromised. This is precisely why 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 we believe components and connectors should play in supporting the kinds of architectural changes described in the foregoing 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 state information, potentially utilizes multiple threads of control, and may be written in any programming language. Treating components as black boxes significantly increases the opportunity for reusing off-the-shelf (OTS) components. There are obvious limitations if the OTS component does not export adequate functionality needed to support some types of runtime change. For example, the inability to extract component state prevents component replacement. We cannot circumvent these problems without modifying the component.

Components do not directly reference one another when communicating. Instead, they utilize a connector, which localizes and encapsulates component interfacing decisions. This minimizes coupling between components, enabling binding decisions to change without requiring component modification [28].

Every component must provide a minimal amount of additional 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 [23] and COM [6], may also be utilized. 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 [33]. In this way, connectors separate a component's interfacing requirements from its functional requirements [28]. 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 [28]; visualizing and debugging system behavior by monitoring messages between components [27]; and integrating tools by using a connector to broadcast messages between them [29].

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 [32]. 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. 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 [29]), the functionality we require can usually be integrated without a significant runtime performance penalty. New 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 the 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. Encapsulating change policy and scope decisions within connectors lets designers select the most appropriate policy based on application requirements.

6 APPLYING CONCEPTS TO A SPECIFIC ARCHITECTURAL STYLE

Our goal is to develop a technique for runtime architecture evolution that is 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 a number of its facets remain largely unexplored. This is certainly the case with dynamism: we can learn from traditional approaches to dynamism, outlined in Section 3, but many of the issues they raise will be irrelevant to architectures; at the same time, 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 our efforts on supporting architectures in a layered, event-based architectural style, called C2 [35]. 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 "top" and "bottom" sides, with a single communication port on each side. This greatly simplifies the task of adding, removing, or reconnecting a component in an architecture. A C2 connector also has a top and a bottom, but the number of communication ports, and hence the interface it exports, 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.

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. At the same time, 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.

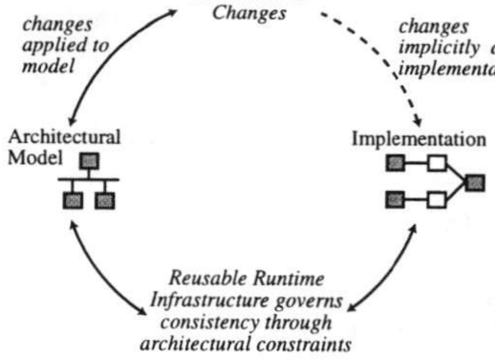
7 TOOLS SUPPORTING ARCHITECTURE-BASED EVOLUTION OF SOFTWARE SYSTEMS

This section describes our tool suite, ArchStudio, 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 our initial implementation based on this approach and demonstrate its use on a simple application. We conclude the section by discussing the current limitations of our implementation.

7.1 Approach

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

Explicit Architectural Model. In order to effectively modify an evolving system, an accurate model of its architecture must be available. We deploy a portion of the system's architecture as an integral part of the system. The



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

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

deployed model includes the interconnections between components and connectors, and their mappings to implementation modules. The mapping enables changes applied to this partial model to effect corresponding changes to the implementation. As architectural modifications are performed, the correspondence between the model and the implementation must be maintained.

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

Our desire is to support 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, each end-user effectively creates a different member of the system family at their site. As a result, the modifications should be robust to variations in those architectures. 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: A principled approach to runtime system evolution must support 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 to constraining architectural changes, mechanisms governing runtime change should also constrain *when* those changes may occur.

During the course of a complex modification, the system structure 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 runs the risk of preventing valid runtime changes. As a result, a mechanism supporting transactional modifications should be provided.

Reusable Runtime Infrastructure: The runtime infrastructure is responsible for (a) maintaining the consistency between the architectural model and implementation as modifications are applied, (b) reifying changes to the architectural model into implementation, and (c) checking architectural constraints. The runtime 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 [21]. 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. The framework is structured such that components can execute in a shared or private thread of control by subclassing from different framework classes. 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 high-level 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. The model is stored in a structured ASCII format and maintained by the runtime 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 current implementation of the AEM uses implicit knowledge of C2-style rules to constrain changes. The AEM allows the use of architectural constraints or external analysis tools; the incorporation of these is planned for the future. If a change violates the C2-

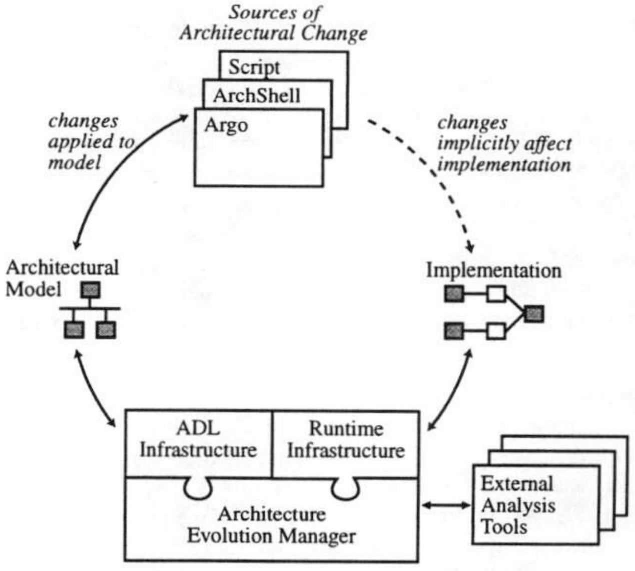
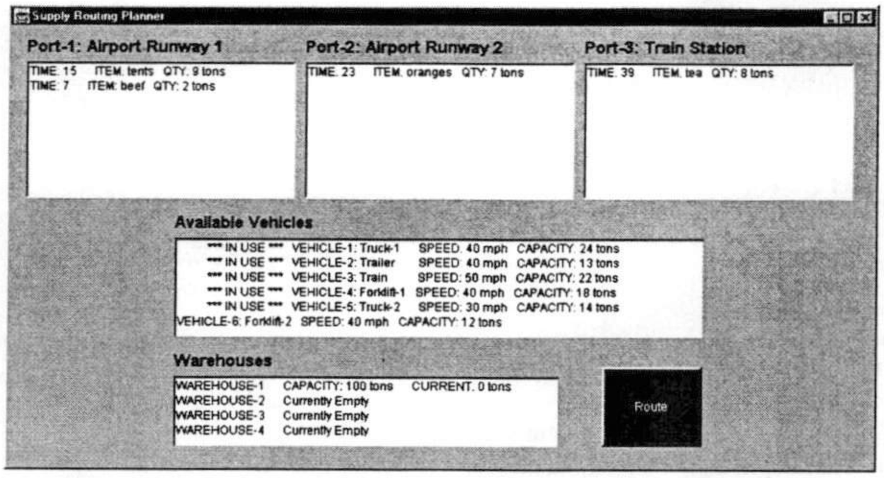


Figure 2. High-level architectural diagram for the ArchStudio tool suite.



```
Supply Routing Planner

Port-1: Airport Runway 1        Port-2: Airport Runway 2        Port-3: Train Station
TIME: 15   ITEM: tents  QTY: 9 tons    TIME: 23   ITEM: oranges  QTY: 7 tons    TIME: 39   ITEM: tea  QTY: 8 tons
TIME: 7    ITEM: beef   QTY: 2 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
```

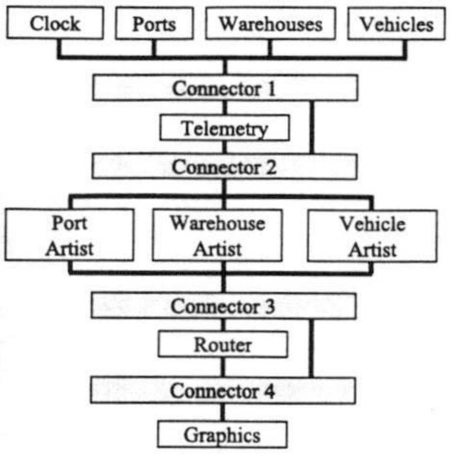


Figure 3. (a) On the left, the cargo routing system's user interface. (b) On the right, the architecture of the cargo routing system in the C2-style.

style rules, the AEM prevents 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.

Argo [30] provides a graphical depiction of the architectural model that may be directly manipulated by the architect. 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 [24] 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. The second command causes ArchShell to read and execute commands from an ASCII text file.

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

Argo and ArchShell are interactive tools meant for use by software *architects* to describe architectures and architectural modifications. The *Extension Wizard,* in contrast, provides 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. Modification scripts are written by system architects, and 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 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 needed by the change. 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 contained within, and executes it. Our approach to end-user system extension is similar to that of Hall et al. [15].

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 user interface for the cargo routing system. 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 cargo'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*, *Warehouses*, and *Vehicles* 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 a

```
> add comp
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.

particular warehouse. The *Port Artist*, *Vehicle Artist*, and *Warehouse Artist* components are responsible for depicting the state of their respective ADTs. The *Router* component records the port, vehicle, and warehouse last selected by the end-user and notifies the telemetry component when the end-user presses the "Route" button. The *Graphics* component renders the graphical drawing requests sent from the artists using the Java AWT graphics package.

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

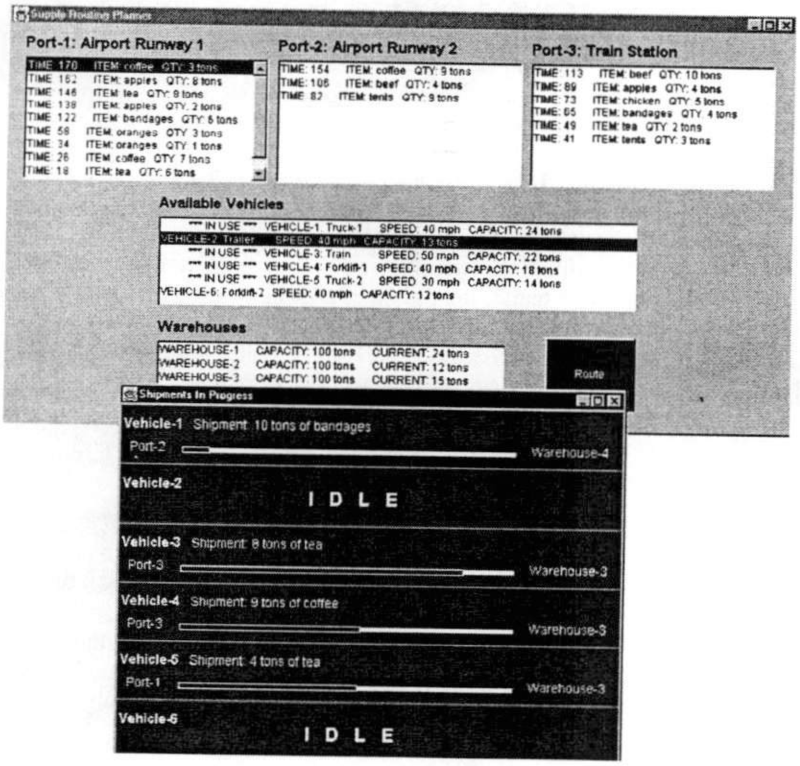
Adding the new visualization involves adding a *Router Artist* component to the architecture. The new router artist is added between Connector 1 and Connector 4 since it uses notification messages provided by the *Port*, *Warehouse*, and *Vehicle* ADTs and utilizes the *Graphics* component for drawing graphics. The architect uses ArchShell to add the component using the "add comp" 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).

```
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.

Adding the automated planner involves adding a *Planner* component to the architecture. The new planner component is added below Connector 1 since 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 to install the change for end-users. 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. Operations may fail if the architectural elements on which the change relies have been previously altered by other architectural modifications.

Figure 6(a) and (b) depict the updated user interface and



```
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: 108   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 Vehicles
    *** IN USE ***  VEHICLE-1: Truck-1     SPEED: 40 mph  CAPACITY: 24 tons
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: 24 tons
WAREHOUSE-2   CAPACITY: 100 tons   CURRENT: 12 tons
WAREHOUSE-3   CAPACITY: 100 tons   CURRENT: 15 tons

Route

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
```

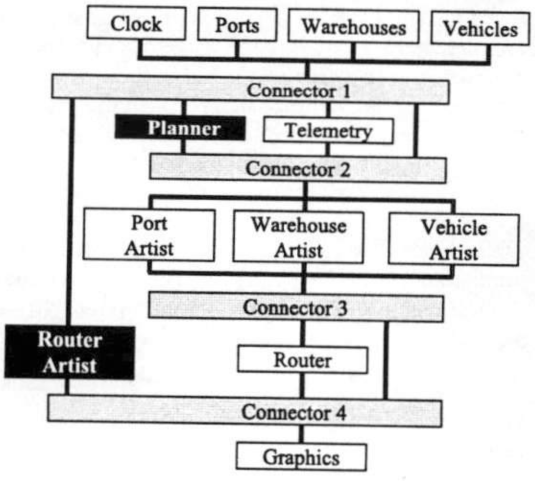


Figure 6. (a) On the left, the cargo routing system user interface after the addition of the new router artist and planner components. (b) On the right, the updated cargo system architecture highlighting new components.

architecture of the system after both modifications have been made.

The cargo routing system consists of approximately 190 kilobytes of compiled Java code, which includes 100 kilobytes for the Java-C2 class framework. 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 initial prototype has facilitated 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 on using language independent facilities, such as those provided by CORBA and COM.

For simplicity, we assume 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 of significant complexity [10, 22].

The runtime 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, the runtime infrastructure currently only checks constraints derived from the C2-style. Integrating a general constraint checking system is a topic of future work.

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 a static description of a system, but provide no facilities for specifying runtime architectural changes. Although a few ADLs, such as Darwin [19], Rapide [18], and LILEANNA [36], 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. It is important to note that 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. 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 [20], 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 [19] specifications. Others advocate behavioral constraints on components and their interactions [18]. 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 modifications, as opposed to brittle, "one-of-a-kind" patches. A principled approach to runtime change can reduce the risks and costs designers have traditionally associated with such change. Basing runtime evolution on architecture can benefit from a component-based development philosophy, explicit representation of system structure, and separation of computation from communication. This paper has outlined an approach to architecture-based runtime evolution. Our approach is characterized by (1) an explicit architectural model deployed with the system, (2) explicit software connectors that manage the scope and policy of runtime changes, and (3) a tool suite that effectively supports such changes.

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 effecting runtime architectural changes to implemented systems. We have been able to confirm the central role of connectors in supporting runtime change and identify 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 the types of runtime change.

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. 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.
15. 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.
16. C. R. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. Ph.D. Thesis. University of Maryland, Computer Science Department, 1993.
17. J. Kammer, J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, vol 16, no 11, November 1990.
18. D. Luckham, J. Vera. An event-based architectural definition language. *IEEE Transactions on Software Engineering,* pp 717-734, September 1995.
19. J. Magee, J. Kramer. Dynamic structure in software architectures. *Fourth SIGSOFT Symposium on the Foundations of Software Engineering*, San Francisco, October 1996.
20. N. Medvidovic. ADLs and dynamic architecture changes. *Second International Software Architecture Workshop (ISAW-2)*, San Francisco, October 1996.
21. N. Medvidovic, P. Oreizy, R. N. Taylor. Reuse of off-the-shelf components in C2-style architectures. *Symposium on Software Reusability*, Boston, May 1997.
22. M. Moriconi, X. Qian, R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*. pp 356-372, April 1995.
23. Object Management Group. *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, July 1996. http://www.omg.org/corba/corbiiop.htm
24. 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.
25. D. E. Perry, A. L. Wolf, Foundations for the study of software architecture. *Software Engineering Notes*, vol 17, no 4, October 1992.
26. 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.
27. J. Purtilo. MINION: An environment to organize mathematical problem solving. *Proceedings of the 1989 International Symposium on Symbolic and Algebraic Computation*, July 1989.
28. J. Purtilo. The Polylith software bus. *ACM Transactions on Programming Languages and Systems*. vol 16, no 1, Jan. 1994.
29. S. P. Reiss. Connecting tools using message passing in the FIELD environment. *IEEE Software*. vol 7, no 4, pp 57-67, July 1990.
30. 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.
31. L. Sha, R. Rajkumar, M. Gagliardi. Evolving dependable real-time systems. *IEEE Aerospace Applications Conference*. New York, NY, pp 335-346, 1996.
32. 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.
33. M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.
34. 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.
35. 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.
36. W. Tracz. Parameterized programming in LILEANNA. *Proceedings of ACM Symposium on Applied Computing SAC'93,* February 1993.