

Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach

Marija Rakic and Nenad Medvidovic
Computer Science Department
Henry Salvatori Computer Center 300
University of Southern California
Los Angeles, CA 90089-0781, USA
Phone: 1-213-740-6504; 1-213-740-5579
{marija, neno}@usc.edu

ABSTRACT

The promise of component-based software development is that larger, more complex systems can be built reasonably quickly and reliably from pre-fabricated (“off-the-shelf”) building blocks. Additionally, such systems can be upgraded incrementally, simply by replacing individual components with their new versions. However, practice has shown that, while it may improve certain aspects of an existing component, a new component version frequently introduces unforeseen problems. These problems include less efficient utilization of system resources, errors in the newly introduced functionality, and even new errors in the functionality carried over from the old version. This paper presents an approach intended to alleviate such problems. Our approach is based on explicit software architectures and leverages flexible software connectors in ensuring that component versions can be added and removed in the deployed, running system. Our connectors, called multi-versioning connectors, also unintrusively collect and compare the execution statistics of the running component versions (e.g., execution time and results of invocations). We illustrate our approach with the help of an example application.

1. INTRODUCTION

Component-based software development and software reuse have become areas of increasing interest to software researchers and practitioners. A number of component-based software technologies have emerged over the past decade, potentially enabling development of entire software systems from off-the-shelf (OTS), pre-fabricated modules. At the same time, a number of well-documented problems can arise when attempting software reuse [4,5,12,16,18,24]:

- OTS components may adhere to different, incompatible implementation substrates (i.e., middleware);
- they may be implemented in different programming languages;
- they may be poorly documented;
- the granularity of OTS components may be too coarse or too fine for the needs of a given system;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SSR '01, May 18-20, 2001, Toronto, Ontario, Canada.
Copyright 2001 ACM 1-58113-358-8 /01/0005...\$5.00.

- the OTS components may not provide the exact functionality required;
- different OTS components may assume different types of interaction within the system (e.g., procedure calls vs. asynchronous event notifications);
- specialization and integration of OTS components is frequently unpredictably complex; and
- the costs associated with locating, understanding, and evaluating a component for reuse may be higher than engineering the component from scratch.

A specific instance of software reuse, and one of particular interest to us in this paper, deals with *incorporating into a system a new version of an already existing component* [10]. Several of the reuse challenges outlined above may not be as pertinent in the context of component upgrades. Typically, vendors try to maintain the old component version’s key properties (e.g., granularity, implementation language, and interaction paradigm), while enhancing its functionality (by adding new features) and/or reliability (by fixing known bugs). However, component upgrades raise another set of questions, including whether the new version correctly preserves the functionality carried over from the old version, whether the new version introduces new errors, whether there is any performance discrepancy between the old and new versions, and so forth. Depending on the kinds of problems a new component version introduces and the remedies it provides for the old version’s problems, this scenario can force an engineer to make some interesting choices:

- replace the old version of the component with the new version in the system;
- retain the old version (e.g., in case the new version introduces too many problems); and
- maintain multiple (i.e., both the old and new) versions of a single component in the system.

Prior to making one of these choices, the engineer must somehow assess the new component in the context of the environment within which the old component is running. A simplistic solution to this task is to try to replicate the execution environment and the running system and test the new component within the replicated system and environment; the results of testing are then compared with those produced by the old component. However, this solution has several drawbacks. First, it may be difficult to faithfully replicate the execution environment, especially in heterogeneous, highly distributed systems. Second, multiple versions of a given system

may be deployed in the field [13]; each such system version may employ different versions of (different) components, rendering the “off-line” testing approach impractical. Third, assuming that the new component version passes a “quality threshold” such that the engineer decides to add it to the deployed system (either as a replacement for the old version or in addition to the old version), the above strategy does not make any provisions for deploying such a component without disrupting the operation of the running system(s).

In this paper, we present an approach that addresses these three challenges. This work was inspired by Cook and Dage’s technique for reliable upgrade of software at the level of individual procedures [10]. Our approach is predicated upon minimizing the interdependencies of *coarse-grained* components, such that new component versions may be added to a (running) system more easily. In particular, we employ explicit *software connectors* as component interaction facilities. Software connectors have been extensively studied and used in architecture-based software development [1,21,23,25]. We have leveraged the connectors used in our previous work [18,26] to develop special-purpose connectors specifically intended to support software component upgrades. These connectors allow multiple versions of a component to execute “side-by-side” in the *deployed* system with a minimal effect on the rest of the system. Furthermore, the connectors allow dynamic addition and removal of different component versions, as well as reverting the system and its constituent components to any previous state during their execution (“architectural undo”). Finally, the connectors are tailored to monitor the execution of the multiple component versions and perform comparisons of their performance (i.e., execution speed), reliability (i.e., number of failures), and correctness (i.e., ability to produce expected results). Such comparisons serve to increase the reuser’s confidence in a new component version. Coupled with our previous work on component reuse [18,20], the approach discussed in this paper allows us to effectively support upgrades of components, regardless of their implementation language, concurrency and distribution assumptions, or employed interaction mechanisms.

The remainder of the paper is organized as follows. Section 2 presents a brief overview of software architecture, which serves as the basis of this work. Section 3 details our approach, while Section 4 describes its current implementation. Section 5 discusses related work. Finally, we present several open issues in Section 6 and our conclusions in Section 7.

2. BACKGROUND

Our support for reliable component upgrades leverages explicit *software architectural* models. As software systems have grown more complex, their design and specification in terms of coarse-grain building blocks has become a necessity. The field of software architecture addresses this issue and provides high-level abstractions for representing the structure, behavior, and key properties of a software system. Software architectures involve (1) descriptions of the elements from which systems are built, (2) interactions among those elements, (3) patterns that guide their composition, and (4) constraints on these patterns [23]. In general, a particular system is defined in terms of a collection of *components*, their interconnections (*configuration*), and interactions among them (*connectors*).

Components are architectural entities that perform some computation and/or store data. The granularity of components may vary from a single procedure to an entire application. Connectors are architectural elements that represent interactions among components and rules that govern those interactions [21]. A connector can represent a simple interaction mechanism such as a procedure call or shared variable access, but can also represent complex and semantically rich interactions such as client-server protocols, database access protocols, asynchronous event multicast, security protocols, and so forth. Explicit connectors have proven to be very useful in architecture-based development since they enable separation of computation from interaction in a system and minimize component interdependencies.

Another key architectural concept is *architectural style*. An architectural style defines a *vocabulary* of component and connector types and a set of *constraints* on how instances of these types can be combined in a system or family of systems [25]. When designing a software system, selection of an appropriate architectural style becomes a key determinant of the system’s success. Styles also influence architectural evolution by restricting the possible changes an architect is allowed to make. Examples of styles include pipe and filter, layered, blackboard, client-server [25], GenVoca [3], and C2 [26].

3. APPROACH

Architecture-based software development has shown great potential to aid reuse since it enables a separation of components (data and processing elements) from connectors (interaction elements) [18]. We leverage this key property of architectures in our current focus on supporting component upgrades. Additionally, we make the following key observations about the problem space we are addressing:

- real, OTS components can be of arbitrary granularity;
- an OTS component can have complex internal state that both influences the outcome of a given operation and changes as a result of that operation (e.g., the result of a stack *Pop* depends on the value of the stack’s top element; a *Pop* also changes the state of the stack);
- a component can provide *non-terminal* operations;¹
- components can assume various interaction protocols; and
- the data structures affected and possibly returned by the different components’ operations may be arbitrarily complex.

When performing an upgrade of an existing component, one can encounter three possible relations between the old and new versions of the component:

- (1) the new version provides the same functionality, but better performance, security, reliability, and so forth;
- (2) the new version also provides additional functionality, but the basic approach for providing that functionality is unchanged; and

¹ By *non terminal operation* we mean that the component will need to interact with other components in order to perform a requested operation. For example, operation *a* in component *A* may need to invoke operation *b* in component *B* to complete its task; in turn, *b* may need to invoke other components’ operations *c* and *d*; and so forth.

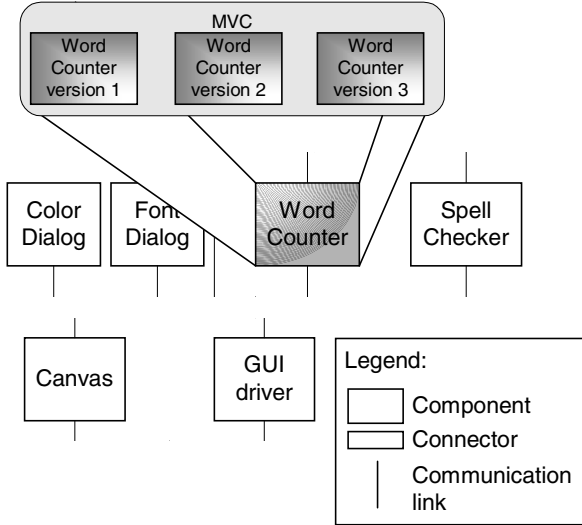


Figure 1. Multi-versioning connector.

- (3) the new version’s functionality has some overlap with the old version’s, but the implementation approach is completely different (e.g., a novel algorithm or more efficient data structure).

Our approach supports upgrades in all three cases provided that certain assumptions hold (discussed in Section 6).

We illustrate our approach with the help of an example. Figure 1 shows the partial architecture of a word processor. In this architecture, a *Word Counter* component provides several operations as illustrated in Figure 2. Let us assume that, after this application is deployed, we obtain two new versions of *Word Counter* that are claimed to be improvements over the old version. What we would like to do is assess both of the new versions before deciding which one to deploy in our system.

Figure 1 depicts the essence of our approach: a component (*Word Counter*) is replaced by a set of its versions encapsulated in a wrapper. The wrapper serves as a connector between the encapsulated component versions and the rest of the system [21]. We say that *WordCounter* is *multi-versioned* and call the wrapper *multi-versioning connector (MVC)*.² The MVC is responsible for hiding from the rest of the system the fact that a given component exists in multiple versions. The role of the MVC is to relay to all component versions each invocation that it receives from the rest of the system, and to propagate the generated result(s) to the rest of the system. Each component version may produce some result in response to an invocation. The MVC allows a system’s architect to specify the component authority [10] for different operations. A component designated as *authoritative* will be considered nominally correct with respect to a given operation. The MVC will propagate only the results from an authoritative version to the rest of the

² Note that this usage of the *MVC* acronym is entirely unrelated to the Model-View-Controller approach for constructing Smalltalk applications [15].

system. At the same time, the MVC logs the results of all the multi-versioned components’ invocations and compares them to the results produced by the authoritative version.

We are allowing authority specification to be at the level of the entire invocation domain (e.g., for each invocation, the entire component version v1 will be considered nominally correct). Similarly to Cook and Dage [10] we are also supporting authority specification at the level of individual operations (e.g., component version v1 is authoritative for *countWords*, while v2 is authoritative for *countLines*). For each possible invocation, we are assuming that there is going to be exactly one component designated as authoritative.

```

Component: WordCounter
Operations:
  countWords(text):int;
  countSentences(text):int;
  countLines(text, pageWidth):int

```

Figure 2. Operations for the *WordCounter* component.

We illustrate the notion of component authority with a simple example: let us assume that the MVC encapsulates two versions, v1 and v2, of the *Word Counter* component. The two versions provide document statistics incrementally, such that the number of words in a document is continuously tracked and stored in an external component (e.g., *Active Document Repository* shown in Figure 3). Any new changes to the document (e.g., added or deleted words) are based on the “running total.” Clearly, if both v1 and v2 were allowed to update the word count without interference, the result would not be correct: for example, a newly added word would be counted twice. However, if only one of the versions, designated as authoritative for the *countWords* operation, is allowed to request updates to the *Repository*, the result will be correct. It is important to note that the MVC can still assess the correctness of the non-authoritative version by comparing its results to those returned by the authoritative version, without propagating those results.

In addition to this “basic” functionality of insulating multi-versioned components from the rest of a system, and vice versa, the MVC provides several additional capabilities. The MVC allows component authority for a given operation to be changed at any point in time. It also allows insertion of a new component version into the system during runtime without removing the old version. The MVC can monitor the execution of the multiple component versions and perform comparisons of their performance (i.e., execution speed), correctness (whether they are producing the same results as the authoritative version) and reliability (number of failures). Furthermore, the MVC logs the execution history as a sequence of invocations of the multi-versioned component. In case a failure has occurred, this information can be used to determine which sequence of invocations has led to the failure. The MVC also periodically records the state of each component version. The execution history and the state “snapshots” can be used to roll back the execution of a multi-versioned component to any point in the past; this capability is further discussed in Section 4.

MVC’s monitoring mechanism (logging of component invocations,

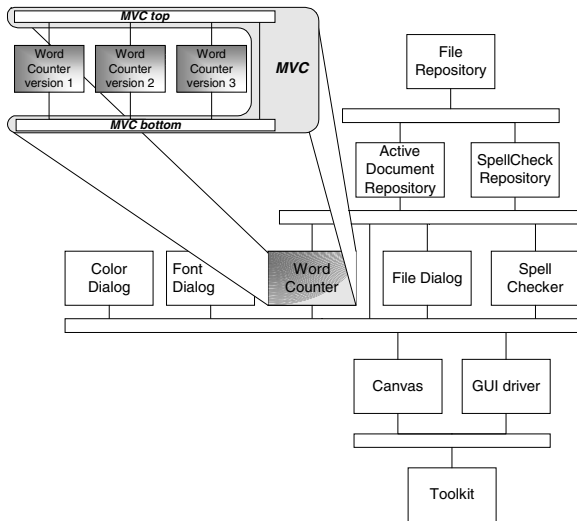


Figure 3. Architecture of the word processor application. The internal structure of the MVC is highlighted.

comparisons of their results, and component state “snapshots”) can help an engineer make one of the following choices:

- replace the old version of the component with the new one,
- retain the old version (in case the new version introduces too many problems), or
- maintain both the old and new versions of the component.

In the first two cases, the MVC can be removed from the system to reduce the overhead introduced by its insertion.³ In the last case, the MVC will be retained and used to accomplish the desired functionality of a single conceptual component by simultaneously employing multiple implemented versions. In that case, the monitoring can be disabled to minimize the overhead.

To date, we have primarily focused on the use of our approach in the case of component upgrades. However, the same infrastructure can be used when an entirely new component needs to be inserted into a system. The wrapped component can be inserted at the desired location in the system’s architecture. The component’s behavior can then be assessed with minimal disturbance to the rest of the system, since the MVC will be configured to intercept and “swallow” all the invocations the component tries to make. Once the new component is assessed in the context of the deployed system and it is established that the component produces satisfactory results, the wrapper around it (i.e., the MVC) may be removed.

4. IMPLEMENTATION

Our implementation of the multi-versioning connector (MVC) wrapper leverages the *C2 architectural style* [26]. The key elements of C2 are *components* and *connectors*. In particular, the style mandates that components cannot directly interact with other components, but must do so via connectors. Both components and

connectors have a defined top and bottom. The top (bottom) of a component can be attached to the bottom (top) of a single connector; there is no bound on the number of components or connectors that may be attached to a connector. All communication in C2 style applications is solely achieved by exchanging messages. A component sends a message to the connector attached on its top or bottom; the connector relays the message to its destination. The connector may implement complex message filtering and routing policies (e.g., broadcast, multicast, unicast). Messages consist of a name and set of typed parameters. There are two types of messages: notifications and requests. Notifications are sent down through a C2 architecture, while requests are sent up. C2 messages are predominantly sent and received asynchronously [26]. However, in the course of developing the MVC, we have also implemented synchronous message passing for the purpose of supporting a wider range of OTS components.

C2 is accompanied by a large suite of tools for architecture modeling, analysis, evolution, implementation, and deployment [18,19,22]. Particularly relevant to this paper is the infrastructure for implementing and deploying C2-style architectures: it is this infrastructure that we have leveraged in the implementation of the MVC and the deployment of the multi-versioned components to the running systems. C2’s implementation and deployment infrastructure is a framework of abstract classes for C2 concepts such as architectures, components, connectors, and messages [18]. The framework implements component interconnection and message passing protocols, and supports a variety of deployment configurations for a given architecture: the entire resulting system may execute in a single thread of control, or each component may run in its own thread of control or operating system process, possibly distributed across multiple locations. This lightweight implementation framework is extensible, and supports composition of arbitrarily complex components and connectors. Furthermore, the framework’s explicit, flexible connectors with polymorphic interfaces have been used as the foundation of our support for architecture-based runtime evolution of applications [22]. The framework has been implemented in Java, C++, and Ada.

As in the preceding section, we illustrate the properties of MVC’s implementation via the word processor example (recall Figure 1). The complete C2-style architecture of the word processor application is shown in Figure 3. The word processor has the basic functionality for editing text and manipulating its appearance by changing the font and color. The *Word Counter* component is responsible for collecting various statistics, such as numbers of words, lines, and sentences (recall Figure 2). *Word Counter* exists in three versions, which we have deployed with the running application. In our example, version 1 of *Word Counter* counts the words properly; version 2 does not handle correctly multiple spaces between words; finally, version 3 does not implement the *countWords* operation. Initially, version 2 is designated as authoritative for the *countWords* operation. For simplicity, in the ensuing discussion we assume that the entire application will be deployed on a single machine. However, it is important to note that this is not a requirement imposed by our implementation of the MVC.

We have directly leveraged the Java version of C2’s implementation infrastructure to construct the MVC. In particular, we have implemented two special-purpose, reusable software connectors, called *MVC-Top* and *MVC-Bottom*. As shown in Figure 3, these two

³ Insertion of component wrappers will inevitably introduce certain performance overhead [19].

collaborating connectors encapsulate multiple versions of a component, allowing their parallel execution and monitoring as discussed in Section 3. The intrinsic support of the C2 connectors for runtime addition and removal of components [22] is leveraged in the context of the MVC to add and remove component versions during runtime.

When a message is sent to a multi-versioned component (e.g., *Word Counter* in Figure 3) from any component below MVC-Bottom or above MVC-Top, the corresponding connector invokes within each component version the operation that is responsible for processing that message. Even though the operation is invoked on all the installed versions, only the messages generated by the authoritative version are propagated by the two MVC connectors to the rest of the system. In our example, whenever a *countWords* request message is sent from the *GUI Driver* component, MVC-Bottom will return to *GUI Driver* only the result produced by (the authoritative) version 2; the results produced by versions 1 and 3 are compared with those of version 2 and logged, but are not propagated to the rest of the system.

The user interface (UI) of our implementation of the MVC is shown in the bottom window of Figure 5. This window is separate from the application UI, shown in the top window. The MVC window shows the list of component versions in the upper left frame. The table in

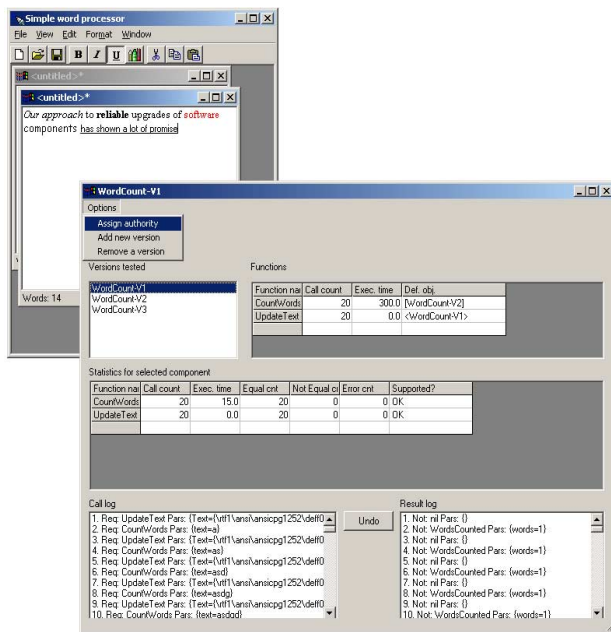


Figure 5. MVC monitoring window (bottom) and screenshot of the word processor application (top).

the upper right frame shows current authority specification and the total (cumulative) execution time, in milliseconds, for each invoked operation of a selected component version (in this case, version 1 of *Word Counter*).

The table in the middle frame displays the execution statistics for the selected component version. For each operation, the table shows

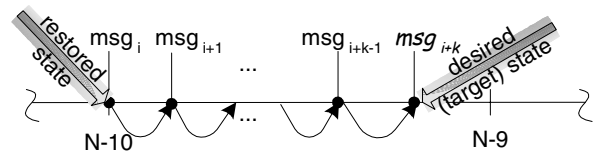


Figure 4. Restoring a component's state to a point between two recorded states.

the number of times the operation has been invoked, the average execution time for that operation (-1 if the operation is not implemented by the component version), and the number of times the operation produced identical (and different) results as the authoritative version. The table also displays the number of times an error (e.g., an exception or a failure) occurred during the execution of the operation, and whether the invoked operation is implemented by the component version.

The bottom two frames in the MVC window display the call and result logs as a sequences of generated messages. Using these logs, the *Undo* button can revert the states of a given set of multi-versioned components to any point in the past. This capability is achieved by taking “snapshots” and storing the multi-versioned components’ states at regular intervals⁴ and by logging each message sent to a multi-versioned component. The undo operation determines whether an architectural change (i.e., addition or removal of a component version) occurred between the application’s past state to which we want to revert and its current state. The MVC restores the application’s previous architecture if necessary (e.g., by removing the added component versions) and restores the desired states of all component versions. The state of a given component is restored to any point in the past in a six-step process:

- (1) *Select a Past State:* The engineer selects the desired (target) state to which she wants to revert by selecting a particular message in the call log (the bottom left frame of the MVC window in Figure 4). As depicted in Figure 5, the selected message corresponds to a point (msg_{i+k}) in the multi-versioned component’s execution. Since the component states are recorded by the MVC only at discrete intervals, note that there may be no recorded state corresponding to the exact point in time signified by the selected message. This issue is addressed in step 4 below.
- (2) *Insulate the Multi-Versioned Component:* MVC-Top and MVC-Bottom continue logging all messages arriving from outside components, but temporarily stop relaying them to the multi-versioned components.
- (3) *Remove the Existing Component Versions:* Each component version (e.g., the three versions of *Word Counter*) is disconnected from MVC-Top and MVC-

⁴ To implement the “snapshots” we have experimented both with Java’s serialization mechanism and cloning (deep copy) of objects in memory. Even though we consider this issue to be outside the scope of the paper, we should note that the current version of the MVC implements deep copy of objects as it proved much more efficient.

Bottom (using the implementation and deployment framework’s *unweld* operation [22]) and unloaded from the application.

- (4) *Roll Back the Clock in Large Increments*: The desired past, recorded states of all component versions, selected in step 1 above, are retrieved, loaded, and attached to MVC-Top and MVC-Bottom (using the implementation framework’s *weld* operation). Since the component states are recorded only at *discrete* intervals, the state restored in this step should be the nearest recorded state *preceding* the desired state. For example, Figure 5 shows that the desired state is between the states recorded at times $N-9$ and $N-10$; in that case, the retrieved and restored state should be the one recorded at time $N-10$.
- (5) *Advance the Clock in Small Increments*: If the component state restored in step 4 above precedes the target state, as shown in the example in Figure 5, the recorded message log is used by MVC-Top and MVC-Bottom to send messages to the reinserted component and thus rapidly advance its state to the *exact* desired point in the (more recent) past.
- (6) *Re-Join the Application*: Finally, MVC-Top and MVC-Bottom reopen the communication between the multi-versioned component and the rest of the application.

Our implementation of the MVC also allows the user to change the authority of a given operation at runtime (signified by the highlighted menu option in the bottom window of Figure 4). This change may be reflected in the running application, since different component versions may provide different implementations of the operation. If, in our example, we change the authority of the *countWords* operation from version 2 to version 1, the user-perceptible behavior of the application will change in the case of consecutive spaces between words: version 1 (correctly) ignores consecutive spaces and the word count shown in the status bar of Figure 4’s top window will decrease.

Finally, we leverage the rules of the C2 style to enable the use of the MVC in different locations in a *single* architecture. C2’s rules of composition and reliance on asynchronous message-based interaction directly support *simultaneous* upgrades of multiple components. In the example architecture shown in Figure 3, the *Active Document Repository* component may be multi-versioned at the same time as the *Word Counter* component.

5. RELATED WORK

In addition to software architectures, discussed in Section 2, this section outlines several other research areas and approaches from which our work has drawn inspiration.

Cook and Dage [10] have developed an approach to reliable software component upgrades that has directly influenced ours. Their component upgrade framework, HERCULES, treats only individual procedures as components, allows multiple such procedures to be executed simultaneously, and provides a means for comparing their execution results. Unlike our approach, HERCULES elaborates on the relationships among different versions (e.g., it supports explicit version trees). HERCULES allows the component authority to be specified at the level of a *subdomain*, an explicitly

constrained subset of a procedure’s domain.⁵ Given that our components are coarser-grained and potentially much more heterogeneous, in our work to date specifying subdomains has shown to be of little practical use. Cook and Dage also propose voting as a mechanism for resolving situations in which there is no component version designated as authoritative. We are also considering such an approach. Finally, unlike our MVC, HERCULES does not provide any support for inserting and removing component versions at system runtime, or reverting a multi-versioned component to its past execution state.

The field of configuration management (CM) [7, 9] deals with capturing the evolution of individual components and entire systems at the source code level. CM provides means for storing and relating different versions of a component. The goals of CM are to properly facilitate simultaneous changes to a component and to track the evolution over time of each component, as well as of entire systems. CM systems typically do not ensure the preservation of desired functionality across different versions of the same component. We have thus recently integrated our implementation of MVC into an architecture-based CM system called Mae [13].

Recording the states of components in a distributed system is a difficult problem. The work of Chandy and Lamport [8] introduces an algorithm by which a set of cooperating operating system processes can determine and record the global state of a distributed system during execution. Similarly to our components, their approach assumes that the processes in a distributed system communicate by sending and receiving messages. Their approach also requires that each process be capable of recording its own state and the messages it sends and receives, and that any process in the system can send its state to the “main” process performing the system “snapshot.”

Simultaneously using multiple (“N”) versions of independently developed components has been investigated as a way of increasing the reliability of a software system [2,6,14]. Similarly to our approach, all components process the same requests and data during system execution, and voting mechanisms are used to determine the relative correctness and reliability of the different versions. In addition, our approach also assesses component performance and provides support for manipulating the deployed systems at runtime.

Finally, in our previous work we have performed extensive studies of C2’s ability to support reuse of arbitrarily complex, heterogeneous OTS components and connectors [11,17,18, 20]. In the process, we have developed a set of simple heuristics for reusing different classes of OTS components and connectors. These heuristics have been implemented in our architecture modeling, analysis, implementation, deployment, and evolution tools [13,19,22]. We consider the work described in this paper to be a complement to our previous work on reuse.

6. OPEN ISSUES

Our approach to reliable upgrades of software components has shown a lot of promise to date and appears better suited to supporting realistic, large-scale OTS components than related approaches (e.g., [10]). At the same time, we consider this research

⁵ A domain comprises the combined value spaces of a procedure’s input parameters.

a work in progress. Our future efforts will address several unresolved issues and eliminate the simplifying assumptions we have made thus far. We discuss a number of these below.

We currently assume that different implementations of a multi-versioned component will be similar enough to allow comparisons of the results returned by the versions' operations. The comparisons are also predicated upon the existence of an "equal" method for the returned (arbitrarily complex) data structures. This is clearly inadequate in the case where fundamental differences exist between the different versions' implementations. In such a situation, a combination of the operations' results, different versions' internal states, and the message patterns they produce in the process of performing an operation may be needed to properly compare the components.

Furthermore, if an invoked operation of a multi-versioned component is non-terminal (recall Footnote 1), the MVC must decide which version's invocations to propagate. Note that this is different than propagating an operation's results: it may simply be that version 1 of component A needs to invoke an operation in component B, while version 2 needs to invoke an operation in component C. We are currently supporting this type of authority specification by assuming that the component designated as authoritative for performing a given non-terminal operation will also be authoritative for invoking any other operation. However, to properly address this situation, both invocations (by versions 1 and 2) should be propagated. This is, in turn, likely to affect the application's functionality, so we clearly must develop a different solution to deal with this issue.

In our work to date, we have placed the burden of making the final decision regarding a component upgrade on the human engineer. The monitoring of the multi-versioned components can be used to aid automated decision-making. In this scenario, the engineer would just specify a set of criteria for removing a version (e.g., number of errors produced by a version over a given period of time) or for keeping only one version (e.g., if the version performs without any errors over a specific period of time). We can extend with minimal effort the current implementation of the MVC to support such automated decision making. One aspect of our future research will be to develop component upgrade heuristics that can then be used in automating this task.

In addition to these issues, we also intend to learn more about the implications of the current design and implementation of the MVC. For example, we need to perform measurements of the overhead, both in terms of execution speed and application size, introduced by the MVC. We also intend to assess the dependency of our current implementation of the MVC on C2 and its connectors. One pertinent question is whether the relatively simple topological rules of the C2 style have simplified certain aspects of this task that may not be as easy in a more general case.

7. CONCLUSIONS

In the rapidly growing world of software development, releases of new components and component versions happen with increasing frequency. In order to effectively leverage the resulting large numbers of new components, increasing the confidence of the component "consumers" in the key properties of those components (e.g., correctness, reliability, safety, efficiency, and so on) becomes a critical issue. Our approach tries to maximize the benefits of the

availability of many components, while, at the same time, minimizing their potential drawbacks (e.g., fear of unknown bugs, difficulties with deploying the components in the running systems, and so on).

This paper presented an architecture-based approach to deploying and increasing the confidence in new versions of an OTS component. The essence of our approach is the use of explicit software connectors. We have built special purpose connectors (MVC) that allow multiple versions of a component to execute in parallel in the running system. The MVC also monitors component execution with minimal intrusion on the rest of the application. Our reliance on explicit multi-versioning connectors allows us to experiment with supporting some less common, but quite likely component upgrade and reuse situations. For example, we can utilize the behaviors of more than one component version in the same application, at the same time. While this is still work in progress and there are many questions still left to be answered, we believe the approach described in this paper to be very promising and intend to further pursue it in our future work.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9985441. Effort also sponsored by the Defense Advanced Research Projects Agency, Rome Laboratory, Air Force Materiel Command, USAF under agreement number F30602-00-2-0615. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. 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.

REFERENCES

1. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, pp. 213–249, July 1997.
2. A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, 11(12):1491-1501, 1985.
3. Batory D., and O'Malley S., The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4), October 1992.
4. T. J. Biggerstaff. The Library Scaling Problem and the Limits of Concrete Component Reuse. *IEEE International Conference on Software Reuse*, November 1994.
5. T. J. Biggerstaff and A. J. Perlis. Software Reusability, volumes I and II. ACM Press/Addison Wesley, 1989.
6. S. Brilliant, J. Knight, and N. Leveson. Analysis of Faults in an N-Version Software Experiment. *IEEE Transactions on Software Engineering*, 16(2): 238-247, 1990.
7. Burrows C., and Wesley I., Ovum Evaluates Configuration Management, Burlington, Massachusetts: Ovum Ltd., 1998.

8. K. M. Chandy and L. Lamport, Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Transactions on Computer Systems*, 3(1):63-75, February 1985.
9. Conradi R., and Westfechtel B., Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2): p. 232-282, 1998.
10. J. E. Cook and J. A. Dage, Highly Reliable Upgrading of Components. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE '99)*, pages 203-212, Los Angeles, CA, May 1999.
11. E. M. Dashofy, N. Medvidovic, and R. N. Taylor. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pp. 3-12, Los Angeles, CA, May 16-22, 1999.
12. D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch, or, Why It's Hard to Build Systems out of Existing Parts. In *Proceedings of the 17th International Conference on Software Engineering (ICSE 17)*, Seattle, WA, April 1995.
13. A. van der Hoek, M. Rakic, R. Roshandel, and N. Medvidovic. Taming Architectural Evolution. Submitted for publication. Available as Technical Report USC-CSE-00-523, Center for Software Engineering, University of Southern California, August 2000.
14. J. Knight and N. Leveson. An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming. *IEEE Transactions on Software Engineering*, 12(1):96-109, 1986.
15. G. E. Krasner and S. T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26-49, August/September 1988.
16. C. W. Krueger. Software Reuse. *ACM Computing Surveys*, pages 131-183, June 1992.
17. N. Medvidovic, R. F. Gamble, and D. S. Rosenblum. Towards Software Multioperability: Bridging Heterogeneous Software Interoperability Platforms. In *Proceedings of the Fourth International Software Architecture Workshop (ISAW-4)*, Limerick, Ireland, June 4-5, 2000.
18. N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability (SSR '97)*, pages 190-198, Boston, MA, May 17-19, 1997. Also in *Proceedings of the 1997 International Conference on Software Engineering (ICSE '97)*, pages 692-700, Boston, MA, May 17-23, 1997.
19. N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 44-53, Los Angeles, CA, May 1999.
20. N. Medvidovic and R. N. Taylor. Exploiting Architectural Style to Develop a Family of Applications. *IEEE Proceedings Software Engineering*, vol. 144, no. 5-6, pages 237-248 (October-December 1997).
21. N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 178-187, Limerick, Ireland, June 4-11, 2000.
22. Oreizy P., Medvidovic N., and Taylor R. N., Architecture-Based Runtime Software Evolution in *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)*, pp. 177-186, Kyoto, Japan, April 1998.
23. D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, pages 40-52, October 1992.
24. M. Shaw. Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging. In *Proceedings of IEEE Symposium on Software Reusability*, April 1995.
25. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
26. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, pages 390-406, June 1996.