Towards Architecture-based Self-Healing Systems

Eric M. Dashofy
edashofy@ics.uci.edu

André van der Hoek
andre@ics.uci.edu

Richard N. Taylor
taylor@ics.uci.edu

Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425, U.S.A.
+1 949 824 4101

ABSTRACT

Our approach to creating self-healing systems is based on software architecture, where repairs are done at the level of a software system's components and connectors. In our approach, event-based software architectures are targeted because they offer significant benefits for run-time adaptation. Before an automated planning agent can decide how to repair a self-healing system, a significant infrastructure must be in place to support making the planned repair. Specifically, the self-healing system must be built using a framework that allows for run-time adaptation, there must be a language in which to express the repair plan, and there must be a reconfiguration agent that can execute the repair plan once it is created. In this paper, we present tools and methods that implement these infrastructure elements in the context of an overall architecture-based vision for building self-healing systems. The paper concludes with a gap analysis of our current infrastructure vs. the overall vision, and our plans for fulfilling that vision.

1. INTRODUCTION

Two distinct elements are required for the development of self-healing systems. First, an automated or semi-automated agent must be present to make the decision of when and how to effect a repair on a system. Second, an infrastructure for actually executing the repair strategy must be available to that agent. In this paper, we present a long-term vision and approach for developing self-healing systems, but we focus primarily on an infrastructure we have developed to support the creation and execution of repair strategies on software systems.

Our primary strategy for effecting repairs in running software systems is architecture-based evolution. We believe that software change at the level of its architecture—that is, in terms of its components and connectors, is the approach that offers the most flexibility in the types of repairs that can be performed in a system. Component boundaries are, ideally, the most loosely coupled connection points in a software system, making them the most flexible points of reconfiguration.

Our strategy is applied to event-based architectures. In such architectures, all communication across component boundaries is via independent messages, or *events*, and there is no assumption of shared memory between components. Additionally, all events go through first-class connectors, which allow for further flexibility by reducing direct inter-component dependencies. Because components are not allowed to directly point to one another, they are essentially ignorant of their location in the architectural topology. This allows components to be inserted, removed, and replaced without explicit changes to component code. For these reasons, event-based architectures offer a significant degree of *loose coupling* and *autonomy of components* that we feel is necessary for software repair "without foresight"—that is, without the types of repairs that can be performed being explicitly coded into the individual components. Past research has revealed that event-based wrappers can be developed for a large variety of off-the-shelf components [7], even if they were not originally developed to use event-based communication.

The ability to dynamically repair a system at runtime based on its architecture requires several capabilities:

1. The ability to describe the current architecture of the system;
2. The ability to express an arbitrary change to that architecture that will serve as a repair plan;
3. The ability to analyze the result of the repair to gain confidence that the change is valid; and
4. The ability to execute the repair plan on a running system without restarting the system.

To date, we have developed technologies and methods that address each of these areas. In our approach, architectures are represented in xADL 2.0 [3,15], an extensible, XML-based architecture description language that provides facilities for mapping architecture descriptions to running systems. Changes to software architectures are represented as architectural differences, also expressed in a subset of the xADL 2.0 language. These differences can describe arbitrary changes to a software architecture in xADL 2.0, and are automatically generated by an architectural differencing engine. These differences serve as architectural patches in our approach. The results of applying such a patch can be analyzed before applying them to a running system using design critics. We have already developed a set of critics for analyzing basic architectural properties, but we also provide a framework in which new critics for specific domains or concerns can be developed. Finally, we have developed a significant infrastructure for building and evolving event-based software architectures that supports architectural reconfiguration, and a software component called an *architecture evolution manager* that can instantiate and update a running system whenever its architecture description changes.

Tool integration is especially important in the context of self-healing systems since no human can be involved in manually transforming tool outputs or invoking tools. Recognizing this, all our tools for managing architectural repairs and changes are integrated in an architecture-based software

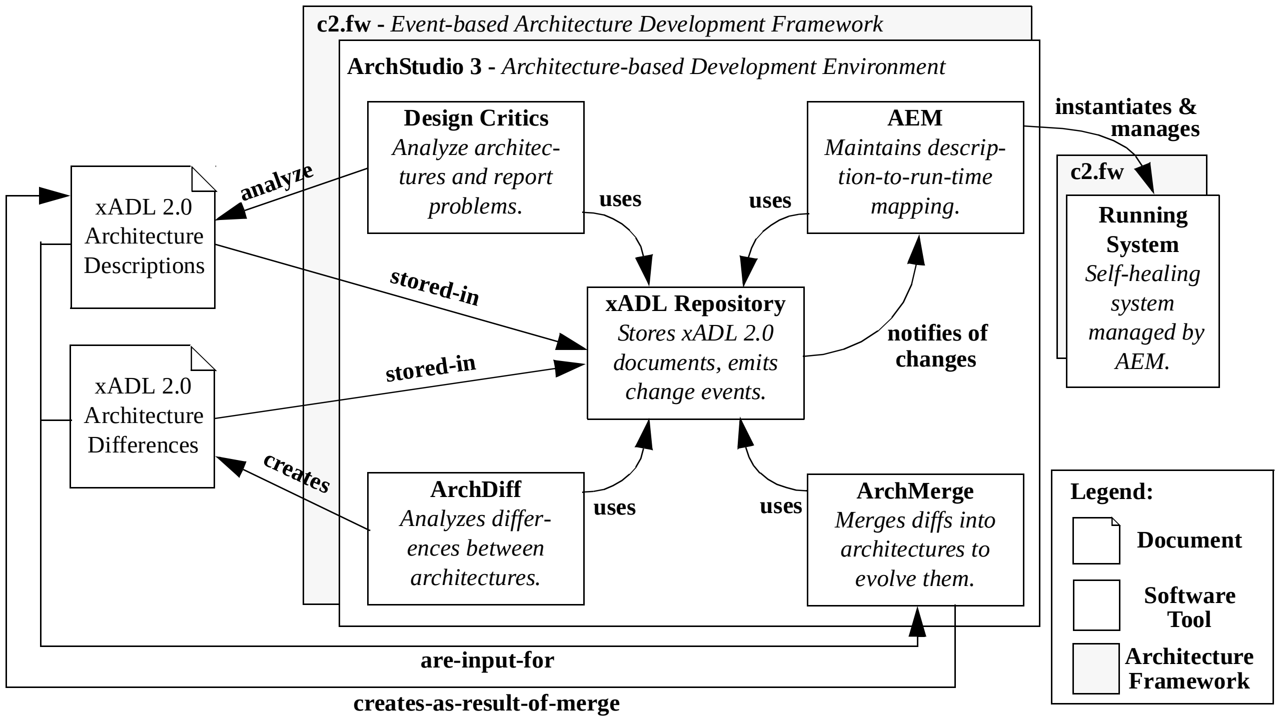
c2.fw - Event-based Architecture Development Framework
ArchStudio 3 - Architecture-based Development Environment
Design Critics
Analyze architectures and report problems.
AEM
Maintains description-to-run-time mapping.
instantiates & manages
c2.fw
Running System
Self-healing system managed by AEM.
xADL 2.0 Architecture Descriptions
analyze
uses
uses
stored-in
xADL Repository
Stores xADL 2.0 documents, emits change events.
notifies of changes
xADL 2.0 Architecture Differences
stored-in
creates
ArchDiff
Analyzes differences between architectures.
uses
uses
ArchMerge
Merges diffs into architectures to evolve them.
are-input-for
creates-as-result-of-merge
Legend:
Document
Software Tool
Architecture Framework

Fig. 1. Tools and documents found in our approach to self-healing systems and their relationships to each other.

development and evolution environment called ArchStudio 3 [1]. This integration allows our tools to interoperate with each other and use each others' outputs to analyze or evolve the system, which is necessary for complete automation of the repair process. For instance, the output of an analysis tool can be used to verify whether an architecture after repair is valid before the architecture evolution manager performs the changes to the running system.

2. BACKGROUND

The vision for the approach we advocate for self-healing systems is detailed in Oreizy et. al., [8]. This vision can be summarized as self-adaptation based on event-based software architectures, using a deployed architecture description as the basis for reflection. While the overall approach is maintained, we have more deeply explored many aspects of it. For instance, we have defined a format for expressing architecture-based change and tools to evolve architectures based on that format, we have integrated analysis tools into the approach, and we have refined the process of evolving a system. In doing so, we have brought new technologies and methods to aspects of the approach that will allow us, in the near future, to more accurately fulfill this vision.

Due to space constraints and scope, it is impossible to detail all these technologies fully in this paper; however, this section highlights the technologies that we and others have developed that contribute to our overall approach.

xADL 2.0 [3,15] is an extensible, XML-based architecture description language that is used in our approach to model software architectures and architectural changes. Previous efforts have used static description languages to describe systems; any additions or changes to the notation required costly tool rewrites. xADL 2.0, along with its tool-set that supports its extensibility, allows us to experiment with new constructs more easily, to better support different domains and types of self-healing systems.

ArchStudio 3 [1] is an architecture-centric development and evolution environment that is an outgrowth of the ArchStudio 2.0 project [5]. It incorporates xADL 2.0 as its internal representation for architectures under development, and the xADL 2.0 tools are components in the environment. ArchStudio 3 is built with and can be evolved using the same techniques described in the Approach section.

Our approach to expressing and performing architectural changes, architectural "differencing and merging," is described in detail in [13]. Our approach to performing architecture-based analysis is based on design critics, which are described in [9].

3. APPROACH

Our work to date on the creation of architecture-based self-healing systems has focused on describing and executing planned repairs. Fig. 1 shows the relationships between the various tools and documents in our approach; these are discussed in detail throughout this section.

3.1. Describing System Architectures

In our approach, the architectures of the software systems under consideration are described in xADL 2.0, an extensible, XML-based architecture description language. A unique aspect of our approach is that software architecture descriptions are an integral part of the deployed software system they describe. The system is instantiated from the software architecture description meaning that components are located and loaded based on implementation information contained in the architecture description. Links among components and connectors are created based on the architecture description, rather than by individual components or a hard-coded bootstrap loader.

Using an extensible architecture description language is important for our approach to have wide applicability. Our use of an extensible ADL allows us to tailor the language to support specific types of repairs, domains, or implementation platforms, middleware, and languages. In contrast to much previous research done in the architecture community, which focused on detailed formal specification of architecture behaviors, we have found that basic architectural repair can be done with only a modicum of information about the location and format of component and connector implementations. For instance, xADL 2.0 includes a Java implementation mapping that specifies only the location and name of the main and auxiliary Java class files for a component. The mapping also includes support for component initialization parameters, allowing slightly different behaviors for components without having to maintain parallel versions. We have found that this level of specification is sufficient for many types of software repairs. Retaining small implementation mappings eases the job of the architecture specifier and makes the approach more practical. For domains that require more information about a system's behavior or run-time state to perform a repair, appropriate specifications can be added to the ADL in whatever formalism is natural for the domain.

3.2. Describing Changes to an Architecture

To effect a repair on a running software system, the changes to the system that will occur because of the repair must be specified and machine-readable. An architectural repair can be expressed in terms of an architectural difference. An architectural difference, or 'diff,' is a document that describes the difference between two software architectures specified in an ADL (in our case, xADL 2.0). Architectural diffs are similar to textual diffs (as would be generated by the 'diff' program on UNIX). In the context of self-healing systems, architectural diffs describe the difference between the software system's architecture before a repair and the architecture after the repair.

Using the xADL 2.0 extensibility mechanism and tools, we defined a new schema for describing architectural diffs. The schema itself is fairly simple, consisting of two sequences: one of additions and one of removals. Elements in the diff include components, connectors, links, component types, connector types, and interface types. To complement this schema, we built an architectural differencing engine called ArchDiff. ArchDiff takes, as input, two xADL 2.0 architecture descriptions. In this context, the first describes the current architecture, and the second describes the architecture after a proposed repair. The "after" description can be generated by a tool or by hand. ArchDiff outputs an architectural diff conformant to the xADL 2.0 diff schema. Differences are primarily determined by the presence of new elements or the absence of old elements in the "after" description, but changes to existing elements are also detected by comparing the contents of element descriptions. A change to an element is treated as a replacement, and results in the diff containing both a "remove" for the old version of the element and an "add" for the new version.

ArchDiff and the diff schema take advantage of xADL 2.0's extensibility mechanism. If the definition of one of the existing elements in the diff (component, connector, etc.) changes or is extended, neither the ArchDiff tool nor the diff schema need to be changed. Changes to the tools and the schema are only necessary if new first-class constructs are added to the architecture description language.

In complement to the ArchDiff differencing engine, we have built an architecture merging engine called ArchMerge. ArchMerge can "apply" a diff to a base architecture, transforming it into a target architecture. If the base architecture is the same as the "before" architecture used to create the diff, then the output will be the "after" architecture used as the second argument to ArchDiff. So, if we express differencing and merging as functions:

d := diff(arch1, arch2)
merge(arch1, d) == arch2

It is possible, however, to merge a patch into a new architecture, different than the "before" architecture that it was originally created from. So:

d := diff(arch1, arch2)
merge(arch3, d) == ???

Whether this merge will be successful depends on how similar *arch3* is to *arch1*. If the diff indicates the addition of a link with an endpoint on a component called *comp1*, and *comp1* is not present in *arch3*, then this will result in a dangling (broken) link. As explained in the next section, design critics can determine whether the application of a patch to an architecture description will create a valid result or not.

3.3. Analyzing the Result of an Architectural Change

Depending on the nature of the system being repaired, varying levels of analysis may need to be performed to determine whether a particular repair is a valid one. Some kinds of analysis are fairly straightforward and universal. For instance, a broken link (an architectural connection with one endpoint missing) would be considered invalid in almost every situation. Some kinds of analysis depend on constraints at the level of an architectural style. For instance, a particular style may prohibit components from being connected directly without an intervening connector, or may prohibit cycles of connection. Specific applications may have yet-more specific constraints. For instance, there may be only one database component permitted in a particular system. Undoubtedly, many systems will want to delay a repair until there is confidence that the result of a repair will not violate these constraints.

In the general case, it is unlikely that any single analysis tool or constraint check will be able to determine whether an architectural repair is valid or not. However, developing a monolithic analyzer for every self-healing application is not practical, since many types of analysis will be shared by different systems (broken link analysis will be useful to almost any system, no-directly-connected-components analysis will be useful to any system whose architectural style prohibits such connections).

To support reuse of existing analysis tools and rapid development and integration of new tools, we have developed a framework supporting the use of *design critics* for architecture analysis. This framework is a part of the ArchStudio 3 tool suite. Each design critic monitors open architecture descriptions in the architecture for changes. When a change is made, the design critics check the description for any problematic issues, which they then report to a central issue database. If a change was made in a part of the architecture description irrelevant to a particular critic, that critic can choose not to perform any analysis at all. ArchStudio 3 ships with a set of base classes for critics that make it relatively easy to implement new critics, since these base classes perform most of the boilerplate duties of a critic (monitoring for architecture description change events, for example).

Critics collaborate entirely through the central issue repository, using it as a ‘blackboard’ for issue data. This makes it easy to create ‘composite’ critics whose analysis is partially or wholly based on the output of other critics. For instance, we have created an AEM Critic that determines whether an architecture description contains enough data for the Architecture Evolution Manager to instantiate and manage the architecture. It bases its decision entirely on the output of other critics that each verify one sub-constraint of that high-level goal, such as ‘no broken links’ and ‘each component and connector must be mapped to an implementation.’

Our approach can be applied to self-healing systems by using critics to perform a “what-if” analysis on the affect of a possible repair. A copy is made of the architecture description of a system about to undergo repair. ArchMerge merges the proposed repair diff into the copied description. Seeing this change, the design critics will go to work determining if the new description is valid. If all appropriate critics finish their analysis and have not reported any issues with the changed architecture description, then a repair planning agent can have confidence that making the same change to the running system will be successful. In this case, the repair diff would be applied, using ArchMerge, to the description of the *running* system, and the repair would be performed as a result. If, however, issues are reported, then replanning can be performed, using the critic-identified issues as data to avoid similar problems in the replanning.

3.4. Executing a Repair Plan

Once a repair plan is chosen, expressed, and analyzed, it must be executed. Any architectural changes that have to be made must be supported by the underlying infrastructure. That is, an infrastructure that does not support the addition or removal of components in a software system is unsuitable for this approach. In our work so far, evolution has been done on a development framework for event-based software architectures called c2.fw [2]. c2.fw is a Java library that eases development of event-based software systems in Java. c2.fw supports event based architectures with arbitrary topologies, although it provides additional classes to specifically support the C2-style [12] topology. It provides a set of useful base and boilerplate classes for components, connectors, links, etc., as well as a runtime infrastructure that allows a running application to change its topology. It supports the ability to control components’ and connectors’ access to threads, so it can start, stop, suspend, and resume individual components and connectors as necessary. While our work to date has focused on c2.fw as the underlying architecture framework, many similar frameworks have been developed [12] that support other languages (C++, Ada) and specific platforms (UNIX, PocketPC).

In our approach, changes to a software architecture are made as a result of a change to the architecture model, rather than the model simply being a passive reflection of the running system. Inside the ArchStudio 3 environment, the Architecture Evolution Manager (AEM) component starts running systems by using the data in xADL 2.0 architecture descriptions. When an architecture is started, the architecture description is considered by the AEM to be “bound” to the running system. Whenever a change is made to the architecture description, ArchStudio’s internal repository for xADL 2.0 files emits an event describing the change, which AEM listens for. These changes are expressed at a lower level than an architectural diff, in terms of changes in individual elements and attributes in the architecture description, so as to capture every type of change that can happen to a xADL 2.0 document. ArchStudio’s internal repository supports transactional changes, so sets of changes (as occur in an architecture merge operation) are performed atomically and reported in a single event.

When AEM receives a change notification on a bound architecture description, it examines the notification to determine if the structure of the architecture has changed. If it has, it makes appropriate calls to the c2.fw run-time infrastructure to change the running system. Currently, the evolution policy supported by AEM is fairly straightforward. The steps in this policy are:

1. Components and connectors about to be removed are given an opportunity to execute cleanup code (if necessary), send data about their state to another component for a replacement to retrieve later, or send other final messages.
2. Components and connectors bordering the affected area(s) in the architectural topology are suspended, preventing messages from being sent to components or connectors while they are being removed or added.
3. Components, connectors, and links are removed and added as defined in the change. Because affected areas are suspended, the ordering of additions and removals is irrelevant, but we plan to investigate change-ordering issues more thoroughly in the future to see if this holds in all cases.
4. Components and connectors bordering the affected area(s) are resumed and new components/connectors are given an opportunity to send out initial messages.

While this policy is clearly not sufficient for every single type of repair, we have found that it is sufficient for many applications, such as those without strict timing constraints. In giving affected components and connectors the opportunity to gracefully suspend, shut down, and start up, architectural repairs can be made much safer than simply performing changes immediately.

4. RELATED WORK

In addition to the work that is mentioned in Section 2, several other papers and approaches have influenced our overall approach and technologies.

Schmerl and Garlan [11] have also developed an architecture-based approach to self-healing systems. Their approach focuses more on detecting when to make a particular repair, and choosing that repair, based on architectural styles. In their approach, an architectural style is a set of formally specified constraints over an architecture; a constraint violation is cause for inducing a repair. The primary difference between their approach and ours is that, in their approach, repair operations are pre-specified in the system code. Our approach does not require that the system make any assumptions about what sorts of repairs can/will be made in advance, therefore opening up the possibility of making repairs that were planned after the system was developed and deployed.

Kramer and Magee [6] have examined the problem of reliably and safely evolving running software systems, and the interaction between the evolution agent and the component behavior in the running software system. In particular, their concept of *quiescence* and the conditions under which a system can be safely evolved are important in the context of self-healing systems. Our tools thus far implement a simple notion of quiescence by suspending all bordering elements before an

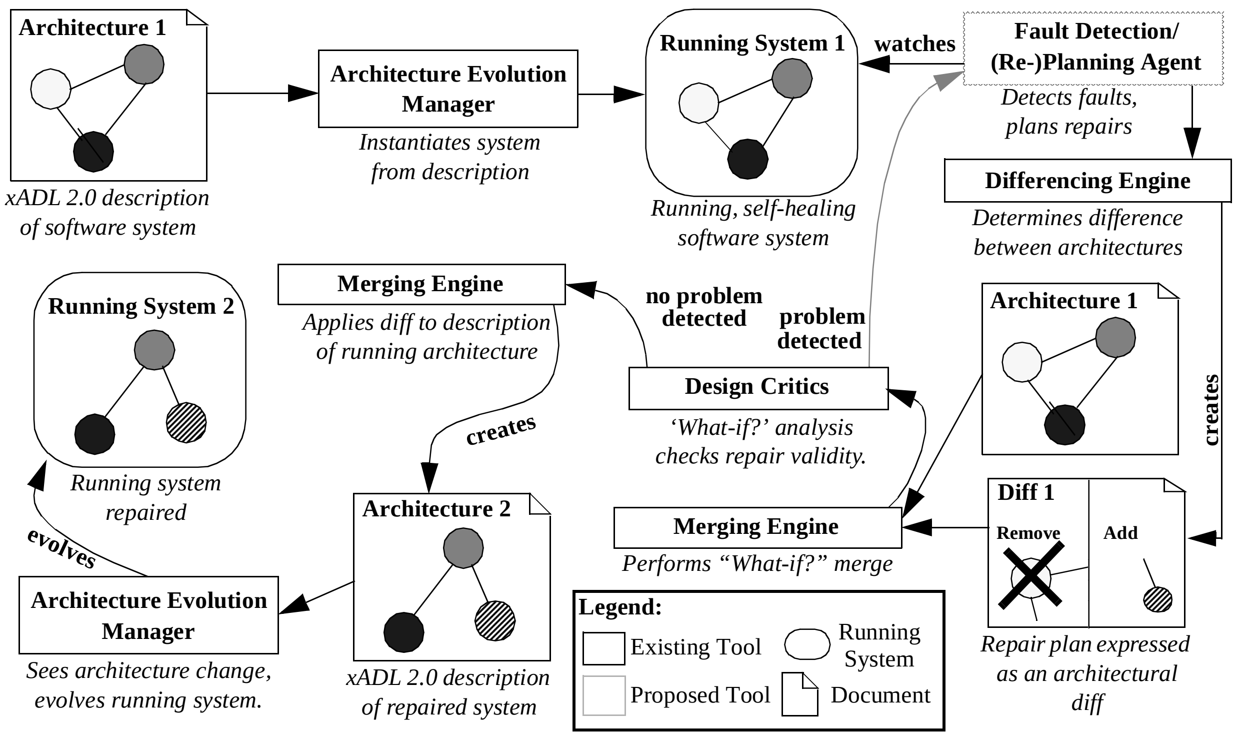
Architecture 1
xADL 2.0 description of software system
Architecture Evolution Manager
Instantiates system from description
Running System 1
Running, self-healing software system
watches
Fault Detection/ (Re-)Planning Agent
Detects faults, plans repairs
Differencing Engine
Determines difference between architectures
creates
Architecture 1
Diff 1
Remove
Add
Repair plan expressed as an architectural diff
Merging Engine
Performs "What-if?" merge
Design Critics
'What-if?' analysis checks repair validity.
no problem detected
problem detected
Merging Engine
Applies diff to description of running architecture
creates
Architecture 2
xADL 2.0 description of repaired system
Architecture Evolution Manager
Sees architecture change, evolves running system.
evolves
Running System 2
Running system repaired
Legend:
Existing Tool
Proposed Tool
Running System
Document

Fig. 2. Diagram showing the **process steps** of our approach, indicating completed tools and tools that we propose to build.

evolution takes place, but we fully plan to experiment with integrating these ideas into our tool set in the near future.

Formal approaches can inform our work as well. In particular, Wermelinger [14] describes an extension to Kramer and Magee's notion of quiescence that attempts to minimize disruption when evolving a system's architecture. As in our method, Wermelinger's approach "freezes," or quiesces, components by suspending the connectors around the affected area. The work also addresses hierarchically constructed systems, something which we will consider in the future.

The CHAM approach [4,14] offers a formal way to think about architectural reconfiguration in terms of 'molecules' and 'reactions.' In this case, components (or collections of components) can be represented as atoms, connected into molecules. Reconfigurations are specified as reactions. Provided with, an appropriate set of reaction rules, it is possible to determine whether a certain reconfiguration is valid, since all valid reconfigurations would be specified with in the reaction rules. Use of the CHAM approach and associated tools may make it possible to prove properties of systems before and after reconfiguration as well. Challenges in integrating the CHAM work with our approach include determining whether the CHAM representation of molecules can be mapped onto some extension of xADL 2.0 (or the language as it exists now), and determining how to best implement the reconfigurations based on reactions. However, this approach does offer several interesting directions with regard to implementing planning agents (i.e. using reaction rules as the basis for choosing or constructing repair plans), an area of future work for us.

Rutherford et. al. [10] describe an approach to system evolution that uses Enterprise JavaBeans as the underlying component model. Their approach shares some similarities with ours, notably, the use of XML-based reconfiguration scripts driving an infrastructure that supports a basic reconfiguration API. However, their work does not maintain an architecture model of the system as the basis for reconfiguration, and does not yet present a strategy for analyzing the results of a change.

5. CONCLUSIONS AND FUTURE WORK

Our long-term approach can be considered a refinement of the approach described in [8], exploring tools and techniques that provide more flexibility or reliability in system reconfiguration in general than have previously been used. Our implementation strategy has been bottom-up; that is, starting with the ability to execute and specify architecture changes before building automated tools for making decisions about when and how to repair a system. The ability to specify and execute a change is a prerequisite for being able to plan and initiate changes; without the language for specifying a change or the infrastructure to enact it, a planning phase will have no output target. Fig. 2 diagrammatically summarizes the steps of our approach.

To date, we have created a significant infrastructure to support these activities. Software architectures and their changes are described in xADL 2.0, an extensible architecture description language. Software systems are instantiated and managed using a flexible framework for building event-based systems, c2.fw. The ArchStudio 3 development environment, also built atop c2.fw, maintains and manages the mapping between architecture descriptions and running systems, and also hosts design critics, which can be used to analyze architecture descriptions or the impact of a change before it is made.

In the future, we plan to continue extending this infrastructure to better fit a complete approach for self-healing systems. In the short-term, we plan on improving the reliability and safety of repairs by integrating Kramer, Magee, and Wermelinger's work on quiescence into the Architecture Evolution Manager. We also plan on creating new design critics that can check more sophisticated constraints on architecture descriptions and change descriptions.

In the medium-term, we plan to extend our tool-set to fully support distributed applications. Repairing distributed systems is more complex than repairing single process systems if realistic assumptions about the distributed environment are made. Many previous approaches to evolving distributed systems assume that the architecture description is available to every process in the system, that unplanned failures will not occur, or that there is a single, distinguished, central evolution agent that is assumed not to fail. Outside the context of these simplifying assumptions, distributed system repair requires better methods and tools for fault tolerance, both in the individual system components and in the infrastructure itself. We also plan to investigate how we can leverage common middleware packages to support a larger variety of legacy systems with our approach. For instance, message-oriented middleware supports many of the operations required by our infrastructure (independent components communicating via messages, with no assumption of shared memory, etc.) We aim to find out whether different types of middleware can augment or be used in place of a framework like c2.fw.

In the long-term, we plan to complete our vision of architecture-based self-healing systems by investigating the detection of faults in various classes of systems and automatic planning of architecture-based repairs. We believe that there is a strong potential for product-family architecture work to contribute to the repair planning phase. Product family architectures describe a set of valid, closely-related architectures, and therefore can serve as potential target architectures for repair. For instance, a product family might define one member of a system in “normal” operating mode, and one in “degraded” mode (for cases when a component fails or resources are constrained). Rather than having to invent a repair from first principles, a planning agent could use the “degraded” mode family member as a potential pre-verified target of a repair strategy. We also plan to investigate other methods of repair planning, possibly using artificial intelligence techniques.

6. ACKNOWLEDGEMENTS[1]

The authors would like to thank Chris van der Westhuizen and Ping Chen for their contributions to architectural differencing and merging.

7. REFERENCES

[1] ArchStudio 3. URL: http://www.isr.uci.edu/projects/archstudio/

[2] ArchStudio 3 Foundations - c2.fw. URL: http://www.isr.uci.edu/projects/archstudio/c2fw.html

[3] Dashofy, E.M., van der Hoek, A., and Taylor, R.N., “An Infrastructure for the Rapid Development of XML-based Architecture Description Languages.” In *Proc. of the 24th International Conference on Software Engineering (ICSE2002)*, Orlando, Florida, May 2002.

[4] Inverardi, Paola and Wolf, Alexander. “Formal specification and analysis of software architectures using the chemical abstract machine.” In *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[5] Khare, R., Guntersdorfer, M., Oreizy, P., Medvidovic, N., and Taylor, R.N. “xADL: Enabling Architecture-Centric Tool Integration With XML.” In *Proceedings of the 34th Hawaii International Conference on System Sciences (HICSS-34)*, Maui, Hawaii, January 3-6, 2001.

[6] Kramer, J. and Magee, J. "The Evolving Philosophers Problem: Dynamic Change Management." In *IEEE Transactions. on Software Engineering*, SE-16, 11 (1990), 1293-1306.

[7] Medvidovic, N., Oreizy, P., and Taylor, R.N. "Reuse of Off-the-Shelf Components in C2-Style Architectures" In *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, Boston, MA, May 17-23, 1997, pp. 692-700.

[8] Oriezy, P., Gorlick, M.M., Taylor, R.N., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., and Wolf, A. An Architecture-Based Approach to Self-Adaptive Software. In *IEEE Intelligent Systems* 14(3):54-62, May/June 1999.

[9] Robbins, J., Hilbert, D., and Redmiles, D. “Using Critics to Analyze Evolving Architectures.” In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, 1996.

[10] Rutherford, M.J., Anderson, K., Carzaniga, A., Heimbigner, D., and Wolf, A.L. “Reconfiguration in the Enterprise JavaBean Component Model” In *Proceedings of the IFIP/ACM Working Conference on Component Deployment*, Berlin, 2002, pp. 67-81.

[11] B. Schmerl and D. Garlan. “Exploiting Architectural Design Knowledge to Support Self-repairing Systems.” In *Proceedings of the Fourteenth International Conference on Software Engineering and Knowledge Engineering*, Ischia, Italy, July 15-19, 2002.

[12] Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead, E. J., Robbins, J., Nies, K., Oreizy, P., and Dubrow, D. "A Component- and Message-Based Architectural Style for GUI Software" In *IEEE Transactions on Software Engineering*, June 1996.

[13] van der Westhuizen, C. and van der Hoek, A. “Understanding and Propagating Architectural Change.” In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA 3)*, Montreal, Canada, August 2002.

[14] Wermelinger, M. Specification of Software Architecture Reconfiguration. Ph.D. Thesis, September 1999.

[15] xADL 2.0 Homepage. URL: http://www.isr.uci.edu/projects/xarchuci/

1.Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. Effort also partially funded by the National Science Foundation under grant number CCR-0093489. 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 (DARPA), the Air Force Laboratory, or the U.S. Government.