Architectural Runtime Configuration Management in Support of Dependable Self-Adaptive Software

John C. Georgas
jgeorgas@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 5160

ABSTRACT

The dynamic nature of some self-adaptive software systems can result in potentially unpredictable adaptations, which may be detrimental to overall system dependability by diminishing trust in the adaptation process. This paper describes our initial work with architectural runtime configuration management in order to improve dependability and overall system usefulness by maintaining a record of reconfigurations and providing support for architectural recovery operations. Our approach—fully decoupled from self-adaptive systems themselves and the adaptation management processes governing their changes—provides for better adaptation visibility and self-adaptive process dependability. We elaborate on the vision for our overall approach, present early implementation and testing results from prototyping efforts, and discuss our future plans.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*languages*; D.2.7 [**Software Engineering**]: Distribution and Maintenance—*version control*

General Terms

Design, Management, Reliability

Keywords

Architectural runtime configuration management, self-adaptive software, dependability

1. INTRODUCTION

Architecture-based self-adaptive systems modify their architectural composition in order to dynamically change their behavior or improve their operation in the face of dynamic deployment conditions. For some self-adaptive systems, these architectural adaptations are explicitly specified during system design; therefore, possible adaptations and resulting architectural configurations which may be exhibited at runtime can be easily predicted. Other self-adaptive systems, however, are dynamic in nature and allow for changes in the manner in which they adapt throughout their deployment lifetime. As a result of this dynamism, some architectural configurations reached during adaptation may be undesirable and harmful to overall system goals.

In general, dependability can be defined as the extent to which a software system can be trusted to deliver exactly its intended service [1]. In the domain of self-adaptive system management, dependability can be seen as the extent to which a system can be trusted to appropriately and correctly adapt, considering both the circumstances triggering adaptation as well as the adaptation's end-goals. Dynamic, policy-based self-adaptive systems give architects fine-grained control over a system's adaptive behavior during runtime; faulty policy design or interpolicy conflicts may contribute to the potential of undesirable or harmful architectural modifications. The dynamic and malleable nature of these systems complicates the task of assuring their correctness, as appropriate adaptive behavior has to be ensured over a collection of dynamically changing policies. In addition, little to no visibility of system adaptations and their governing processes—especially in the case of modifications which do not result in immediately apparent behavioral changes—diminish user trust in this class of self-adaptive systems. Focusing on adaptation fault recovery rather than prevention, removal, or forecasting, we identify certain fundamental facilities which are needed in support of dependability for self-adaptive systems: monitoring and recording of adaptations, enhancing adaptation visibility, and supporting architectural recovery operations.

This paper describes our vision of the *Architectural Runtime Configuration Management* (ARCM) system in support of dependable self-adaptive software. Our vision with this work is to support adaptation process dependability through: (1) the monitoring and recording of changes in architectural configurations during system runtime, (2) the presentation and graph-based visualization to architects of configuration changes over the entire system lifetime, and (3) the provision of recovery facilities to explicitly reverse harmful or undesirable configurations. Using these facilities, a system architect may easily access a history of self-adaptive behavior expressed in terms of overall system configurations reached after enacted adaptations, as well as manually modify architectural configurations through rollback and rollforward operations. While still at an early stage of development, we also present our prototype ARCM tool support deployed in conjunction with the ArchStudio 3.0 architecture-based development environment [6].

A key feature of ARCM is the system's architecture-centric focus: the problem of enhancing the dependability of self-adaptive software is tackled solely through explicit architectural models. In a fully decoupled and modularly applied manner—independent of the adaptation management process and the self-adaptive application itself—ARCM promotes the visibility of self-adaptive behavior and provides facilities which can be used to recover from undesirable states.

The remainder of this paper is organized as follows: Section 2 offers background information on architecture-based self-adaptive systems, which contextualizes this work. Section 3 outlines our basic vision of architectural configuration management and how it promotes dependability, while Section 4 presents our proof-of-concept prototype tool. Work related to adaptation management appears in Section 5, and concluding remarks appear in Section 6.

2. BACKGROUND AND RELATED WORK

Architecture-based self-adaptive software [10] is underpinned by runtime software evolution using explicit architectural models [11] and approaches the problem of adaptation management by using the system's architectural model as the central artifact. These architectural models—describing software systems in terms of functional *components*, *connectors* encapsulating communication, and *links* between these elements [12]—are the focus of the self-adaptive process in terms of both decision-making and actual change enactment. The ARCM approach also adopts this architecture-centric view, by monitoring the system's dynamically changing architectural model and expressing the system's adaptation history in terms of architectural configurations exhibited over time.

Framing the ARCM approach is our ongoing work on the development of self-adaptive software using knowledge-based reasoning in the context of the KBAAM system [5]. Self-adaptive behavior in this work is expressed and managed using a collection of rule-based policy specifications which may be dynamically modified during system runtime, while reasoning over this collection of policies is supported through knowledge-based expert systems. The high level of dynamism which this approach engenders introduces a degree of unpredictability to the self-adaptive system's overall behavior, which may be detrimental to dependability. Our work on ARCM, discussed in this paper, is one attempt to address this unpredictable and dynamic nature of the self-adaptive systems, which are a main focus of our research.

The approach underlying ARCM is based on the recognition that our problem resembles the issue of multiple developers making changes to a common code base, changes which must be historically tracked in order to allow for the return to previous versions [14, 2]. By replacing developers with adaptation policies and the code base with the architecture of the running application, the problem is transposed into the domain of self-adaptive software. Our solution, therefore, borrows techniques from the field of configuration management for capturing, storing, and manipulating multiple versions of an artifact. Specifically, each transactional adaptation is captured as a change, and changes are related to each other in a version graph. This change graph is based on software architectural configurations and expressed in xADL [3], much as laid out in [13].

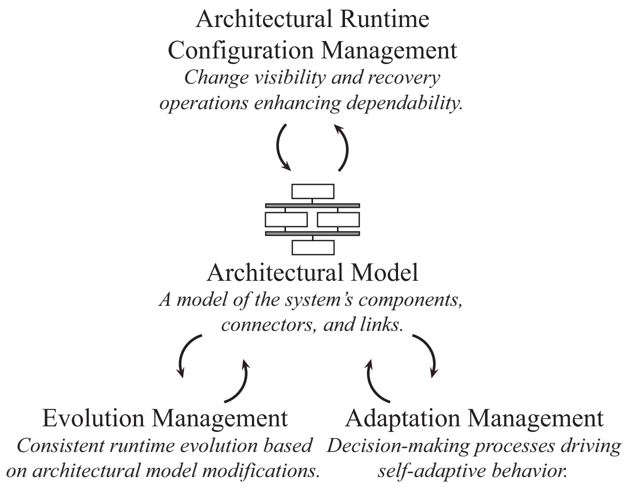
Architectural Runtime Configuration Management
Change visibility and recovery operations enhancing dependability.
Architectural Model
A model of the system's components, connectors, and links.
Evolution Management
Consistent runtime evolution based on architectural model modifications.
Adaptation Management
Decision-making processes driving self-adaptive behavior.

Figure 1: An illustration of architectural runtime configuration management as an independent process, providing architecture-centric adaptation visualization and recovery operations.

3. VISION

Our vision of architectural runtime configuration management through the ARCM system is aimed at increasing the dependability of architecture-based self-adaptive systems in a system-independent manner. Based on our definition of self-adaptive system dependability as the extent to which a system can be trusted to correctly adapt, there are two aspects of dependability which we aim to address: increasing trust in the adaptation management process through enhanced visibility of architectural modifications, and providing for fault recovery operations through which adaptation correctness can be partially enforced. Figure 1 outlines the placement of ARCM's functionality in relation to other high-level processes involved in managing self-adaptive software systems.

3.1 Adaptation Visibility

Architecture-based self-adaptive software pursues behavioral modifications through architectural changes: adaptation goals are achieved through appropriate changes to the software system's runtime architecture and we assume that all system adaptations are expressed as architectural modifications. In this class of systems, the primary means through which users and architects perceive architectural adaptations are the resulting changes in system behavior, which give little insight on the cause of these changes and the specific architectural modifications which brought them about. In the case of adaptations which modify system functionality that is not readily and immediately noticeable, adaptations become completely invisible. The opaque nature of most adaptation processes and potential lack of

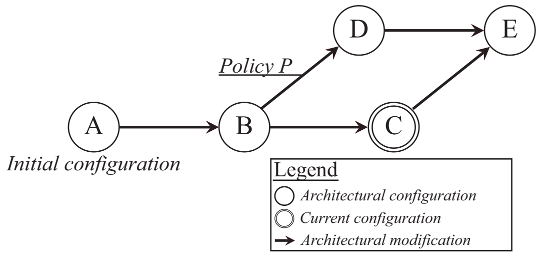

Figure 2: An illustration of an ARCM configuration version graph encompassing architectural configurations, adaptations linking configurations, and indications of policies which cause adaptations.

visibility of their outcomes result in diminished trust and confidence in their correct operation.

While the focus of architecture-based approaches on explicit architectural models increases adaptation visibility over other approaches relying on artifacts at a lower level of abstraction (such as source code or dynamic libraries), adaptations are still primarily perceived by both users and architects when system behavior changes. Even when directly observing an architectural model under adaptation, it is difficult to discern small- and medium-grained architectural changes and recognize their impacts on system functionality for any but the most basic of systems.

Our vision for ARCM aims at addressing this concern through increased adaptation awareness and visualization: an explicit configuration version graph records and displays—during system runtime—the changes that a self-adaptive system undergoes. As adaptations are enacted, ARCM maintains a record of these architectural modifications throughout the system's lifetime, along with the causes which triggered them and the resulting architectural configurations, and graphically displays this information to interested system stakeholders (most notably, system users or software architects managing the system at runtime). We believe that providing this "window" enhances the visibility of both adaptation processes and the architectural modifications they enact, therefore resulting in increased confidence and trust in the adaptation process.

As illustrated in the example of Figure 2, the adaptations of a system are represented as a directed graph indicating architectural configurations connected by the modifications which cause them. An ARCM configuration version graph, then, is defined as an ordered pair $G = (N, E)$, where N is a set of nodes representing architectural configurations and E is a set of unidirectional edges connecting nodes such that $E = \{(x, y)|x, y \in N\}$. Cycles—directed paths beginning and ending on the same node—in G are allowed in order to represent adaptations which transition a system to some existing configuration, but loops—edges which have the same head and tail—are disallowed since they would represent modifications which resulted in no configuration changes. Only a single unique edge is allowed between nodes, as distinct architectural modifications may not result in identical configurations, although anti-parallel edges—edges connecting the same nodes but with opposite directions—may be included in order to represent adaptation operations which return the system to its immediately previous configuration.

ARCM further annotates nodes in N to indicate additional information. *Initial* indicates that the configuration is the first one recorded by the ARCM system, but may not necessarily correspond to the system's pre-adaptation configuration in the case where ARCM monitoring was initiated after adaptations had already taken place, and *current* indicates that the configuration is the one currently exhibited by the system. Edges in E are annotated with information as to the cause of the specific modification represented by the edge; in the particular case of our work with KBAAM, this is expressed in terms of the rule-based policy which triggered the change.

The boundary of individual modifications is dependent on the self-adaptation management process, which is external to the ARCM system. The configuration version graph records both atomic and transactional modifications; for example, an atomic modification may be caused by a simple adaptation involving the addition of a single component, while a transactional modification may include additions and removals of multiple components and connectors.

Using a configuration graph, such as that found in Figure 2, promotes adaptation visibility by providing a runtime artifact explicitly targeted toward illustrating and recording adaptations. Given that ARCM graphs do not contain nodes with duplicate architectural configurations and that adaptations are intuitively and graphically represented in a graph-based structure, we posit that employing such an artifact is a smaller, more usable way in which to maintain cognitive control over a dynamically adapting system; though such a configuration graph used with systems that change frequently may also drastically scale upwards in terms of nodes and edges, it will still be more tractable and useful than only a view of the current architecture. The configuration graph also provides an adaptation history for the system: both intermediate configurations which came between the initial system state and the current configuration, as well as configurations which were manually rejected, are included and explicitly marked.

3.2 Recovery Operations

Some self-adaptive systems may only modify themselves toward specific configurations carefully and explicitly defined during design. However, allowing the rules by which the self-adaptive process is governed to change during runtime introduces the possibility of undesirable modifications, which indicates a need for facilities supporting recovery. It is important to note that the determination of whether a configuration is undesirable is architect- or user-determined, at this point in ARCM's development. While both the maintenance of the ARCM version graph and the implementation of recovery operations are not coupled to a particular method or criteria for detecting the merit of architectural configurations, our work to this point relies on human judgment and intervention for the determination of whether a configuration must be recovered from.

Addressing this recovery aspect of dependability, ARCM provides for *rollback* and *rollforward* operations. The details of each are straightforward: rollback reverts from the current configuration to a previous one, while rollforward transitions the system to subsequent configurations in the version graph (in Figure 2, for example, a rollforward operation from node C would transition the system into the configuration indicated by node E). For both rollback and rollforward operations, if a graph node n has an in- or out-degree, respectively, of greater than one, then the selection of the operation's destination node is user-determined.

At the most basic level, these operations provide a system architect with facilities for explicit intervention into the adaptation process; while partially short-circuiting self-adaptation, these operations are a necessary facility in accounting for dynamically changing self-adaptive behavior. Additionally, these facilities enable system architects to manually guide adaptations informed by specific insights gained through observation of adaptive behavior; for example, when a combination of adaptation policies leads to an insecure architectural configuration, architects may use these recovery operations to revert to a safer alternative and manually modify the system's architecture to that end.

Furthermore, invocations of these operations have the potential of contributing a degree of reflection to the self-adaptive process by being interpreted as a tangible and explicit measure of adaptation desirability. A capability which would be fully realized when integrated with the decision-making processes guiding adaptation, invocations of these operations can guide future architectural modifications by informing these management processes of explicit user preference. For example, if a particular adaptation policy causes an architectural reconfiguration which is subsequently reverted by the architect through a rollback operation, then the triggering policy could be automatically disabled: the manual intervention of the architect is an indication that the policy is behaving incorrectly given the prevalent conditions. Other, more complex responses could include modifications of the body of policies governing self-adaptive behavior such as policy additions and modifications. While the current status of our work does not yet include these "feedback" capabilities, they are a key motivating factor for ARCM's future development.

4. ARCM PROTOTYPE

Early development toward the ARCM vision described in the previous section led to the creation of a prototype, proof-of-concept tool which implements basic runtime architectural configuration management functionality as part of the ArchStudio environment. Though we believe that the configuration management methods we describe in this paper are applicable across a wide variety of architectures, our work to this point has focused on component-based systems using event-driven communication facilitated through explicit, reified connectors.

4.1 ArchStudio

The prototype ARCM component is fully integrated into the ArchStudio 3.0 architecture-based development environment [6] as an independent component. The ArchStudio toolset includes a variety of independently developed tools for the description, maintenance, and runtime management of software systems using explicit architectural models; the most important of these pre-existing tools upon which ARCM depends on are the *Architectural Evolution Manager*, *ArchDiff*, and *ArchMerge*.

The Architectural Evolution Manager (AEM) is a software component responsible for maintaining consistency between an adapting architectural model and its runtime implementation. The AEM instantiates and performs runtime maintenance of architecture-based systems: an explicit architectural model is used as a guide for the instantiation and connectivity of runtime objects implementing system functionality. Architectural modifications to systems are expressed as changes to the architectural descriptions maintained by the AEM, and as changes are made to these architectural models, the AEM ensures that these changes are reflected on the runtime system implementation. The AEM is ultimately responsible for architectural evolution concerns relevant to ARCM's recovery operations, such as component state restoration during rollback and rollforward operations or ensuring system quiescence [8] before these operations are applied. Though fundamental to evolution management and the enactment of adaptive behavior, these issues are outside the scope of our work with ARCM.

ArchDiff is responsible for performing differencing between two architectural configurations and generating *diff* descriptions encapsulating the structural changes that one configuration must undergo in order to be identical to the second. These XML-based descriptions take the form of modification directives for the addition and removal of architectural elements. The following simple *diff* description (with XML namespace information omitted for brevity), for example, is composed of the removal of a connector and the addition of a component with the indicated type:

```
<diff>
  <diffPart><remove id="BusConnector"/></diffPart>
  <diffPart><add>
    <component id="GameLogic">
      <type id="GameLogic_type"/>
    </component>
  </add></diffPart>
</diff>
```

ArgeMerge modifies architectural models by performing runtime merging of *diff* files with target architectural descriptions. Using the contents of the ArchDiff-generated *diff* file, ArchMerge applies the modifications indicated; it is then the responsibility of the AEM to reflect these changes on the running system to which the target architectural description corresponds. More detail on ArchDiff and ArchMerge and architectural *diff* files can be found in [15].

4.2 ARCM Driver

The *ARCM Driver*—a prototype component implementing our vision of architectural runtime configuration management—monitors software systems maintained by the AEM at runtime, and is informed of when these systems undergo adaptation through the detection of notifications to that effect emitted by the AEM. It then records both the architectural configuration before the change and the configuration reached after the adaptation is enacted as well as leveraging the differencing capabilities of ArchDiff to record the differences between these two architectures. Recording the *diff* information during the graph building phase is a significant convenience as this data is readily available and does not have to be generated during the more computationally expensive recovery operation phase, thus amortizing costs between these two phases. This information forms the core of the ARCM configuration version graph and is graphically presented to the system architect at runtime. For recovery operations, ARCM uses the *diff* information stored in the version graph to determine what changes need to be made in order to transition from one configuration to another as

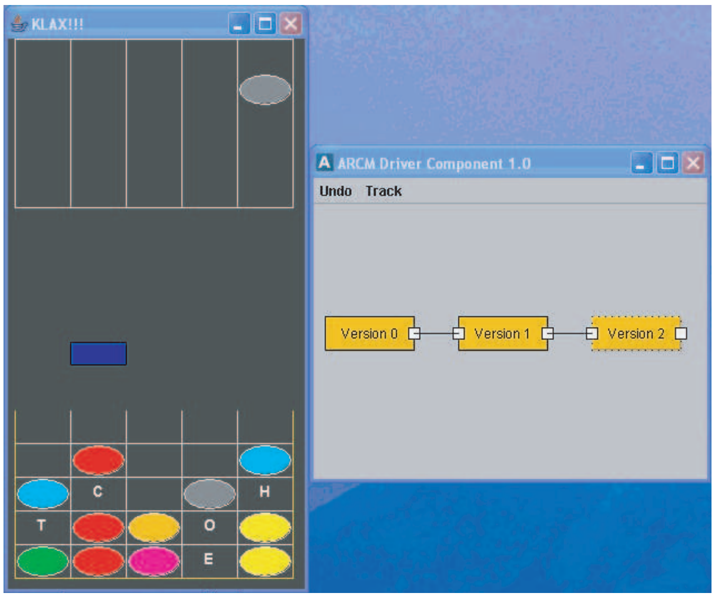

Figure 3: A screenshot of our prototype tool being used with our implementation of the KLAX game.

a result of a rollback or a rollforward operation; ARCM also invokes the capabilities of ArchMerge to perform these changes at runtime—changes which are then reflected in the running system implementation by the AEM.

We have experimented with using the tool on various systems exhibiting adaptive behavior, including the ArchStudio environment itself. The screenshot found in Figure 3 shows the component in use during an adaptation of the KLAX game: in this simple example, the KLAX game is adapted at runtime from a shape- to a word-matching game, and then reverted back to its original configuration (both shapes and letters can be seen as this adaptation takes place). Both these modification are recorded and illustrated by the ARCM prototype in a three-node runtime configuration graph. While the core functionality of the vision we outline in Section 3 is present in this early prototype, only the structure of the configuration graph is currently illustrated (information such as the specifics of configurations, differences between configurations, and adaptation causes are not yet visually presented at this point).

The ARCM prototype is at an early stage of work and under continual refinement: the initial prototype was developed in an eight week period by a single undergraduate researcher with the authors' guidance. While incomplete, the tool performs the basic facilities we envision for the ARCM system: runtime architectural modification detection, persistent configuration and change recording, graph-based presentation, and manual recovery operations. In addition to a significant number of enhancements and feature additions, short-range development plans will address the tool's primary shortcomings: the detection of identical configurations and non-replication of version graph nodes, for which we intend on leveraging the differencing facilities already discussed.

5. RELATED WORK

The development of self-adaptive software systems is an active area of research in software engineering. While the development of ARCM (as discussed in Section 2) has, so far, been framed by a specific approach to self-adaptation, there are a variety of related research efforts.

Garlan and Schmerl use architectural style as the central decision-making element in their work on architecture-based self-adaptive systems [4]. In their approach, the existence of style-specific conditions trigger the application of pre-specified, style-specific architectural modification operations which drive self-adaptive behavior. The Chemical Abstract Machine (CHAM) model of software architectures [7] is used by Wermelinger [16] as the basis for self-adaptive software. In this work, an evolution CHAM is used as a reconfiguration manager driving adaptations of a software system expressed in terms of abstract *molecules* and *solutions*. Expressly targeted toward distributed systems with unpredictable participants, Magee and Kramer [9] present an approach to self-adaptive systems based on global architectural constraints; based on these constraints, these self-adaptive systems automatically self-organize as components enter and leave the system.

The ARCM approach discussed in this paper is highly complementary to the representative approaches addressed here. We have striven to develop ARCM so that its recording, visualization, and recovery facilities can be modularly applied in tandem with any self-adaptive management process that uses and maintains explicit architectural models.

6. CONCLUSION

Enabling the runtime modification of adaptation policies results in potentially unpredictable self-adaptive systems for which reaching undesirable configurations during adaptation becomes a real possibility. The ARCM system discussed in this paper represents our initial efforts to improve the dependability of such systems by developing facilities for adaptation recording, enhancing configuration visibility over the entire system lifetime, and providing user-driven support for architectural recovery from undesirable configurations.

We believe our approach offers several contributions in the domain of self-adaptive software by providing a mechanism for the maintenance of a system's adaptation history and the visual presentation of that history in an easily understood configuration version graph form. Dependability characteristics of self-adaptive systems are enhanced by increasing user and architect confidence in the self-adaptive process through improved adaptation visibility as well as by providing recovery operations for direct intervention into the self-adaptive process. Furthermore, our approach is modular both in concept and implementation: the ARCM capabilities and tool support are decoupled from the functionality of the systems they are applied to as well as the processes managing self-adaptation. Provided that explicit architectural models are used for the maintenance of systems and adaptations are enacted through the modification of these models, system architects may adopt our techniques regardless of specific system functionality or self-adaptation management processes.

Our future plans for the ARCM system include significant enhancements to our prototype tool in order to support version graph multiple branching and refined visualizations. Additionally, we plan on investigating facilities for the automated detection of faulty configurations by establishing architectural support for evaluation criteria to be used in determining the desirability of configurations. Finally, we intend to examine a closer integration between the work we describe here and adaptation policy specifications in order to leverage the information stored in the version graph and use ARCM as part of a reflection layer for self-adaptive behavior management processes.

7. ACKNOWLEDGMENTS

The authors would like to thank Matthew Wastrodowski for his work on developing prototype tool support, Eric M. Dashofy for his development of the AEM and evolution management support, and the anonymous reviewers for their helpful comments which helped strengthen this paper. This work sponsored in part by NSF Grants CCF-0430066, CCR-0093489, and IIS-0205724.

8. REFERENCES

[1] A. Avizienis, J.-C. Laprie, and B. Randell. Fundamental Concepts of Dependability. Technical Report 010028, University of California, Los Angeles, April 2001.

[2] R. Conradi and B. Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[3] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. A Highly-Extensible, XML-Based Architecture Description Language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*, Amsterdam, The Netherlands, August 28-31 2001.

[4] D. Garlan and B. Schmerl. Model-based Adaptation for Self-Healing Systems. In *Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)*, November 2002.

[5] J. C. Georgas and R. N. Taylor. Towards a Knowledge-Based Approach to Architectural Adaptation Management. In *Proceedings of ACM SIGSOFT Workshop on Self-Managed Systems (WOSS 2004)*, Newport Beach, CA, October 2004.

[6] Institute for Software Research, University of California, Irvine. *ArchStudio, An Architecture-based Development Environment.* http://www.isr.uci.edu/projects/archstudio/.

[7] P. Inverardi and A. L. Wolf. Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[8] J. Kramer and J. Magee. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, 11(4):424–436, 1985.

[9] J. Magee and J. Kramer. Self Organising Software Architectures. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 35–38, San Francisco, CA, 1996.

[10] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An Architecture-based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, May-June 1999.

[11] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 177–186, Kyoto, Japan, April 1998. IEEE Computer Society.

[12] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[13] R. Roshandel, A. van der Hoek, M. Mikic-Rakic, and N. Medvidovic. Mae—A System Model and Environment for Managing Architectural Evolution. *ACM Transactions on Software Engineering and Methodology*, 13(2):240–276, 2004.

[14] W. F. Tichy. RCS—A System for Version Control. *Software—Practice and Experience*, 15(7):637–654, 1985.

[15] C. van der Westhuizen and A. van der Hoek. Understanding and Propagating Architectural Changes. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA 3)*, pages 95–109, Montreal, Canada, August 2002.

[16] M. Wermelinger. Towards a Chemical Model for Software Architecture Reconfiguration. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 111–118, Annapolis, Maryland, May 1998.