Towards a Knowledge-Based Approach to Architectural Adaptation Management

John C. Georgas Richard N. Taylor

Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
+1 949 824 5160

{jgeorgas, taylor}@ics.uci.edu

ABSTRACT

Self-adaptive software continually evaluates and modifies its own behavior to meet changing demands. One of the key issues in constructing such software is that of planning when and what kinds of adaptations are appropriate. In this paper, we present an architecture-centric knowledge-based approach for specifying and enacting architectural adaptation policies as the main driver for self-adaptive behavior. Our work applies explicitly represented knowledge-based policies for the definition and enactment of software adaptation at the architectural level. A key benefit of our approach is the de-coupling of adaptation policy from system implementation as well as the independent and dynamic evolution of policies themselves. We elaborate our overall approach, present prototype tools and techniques for its support, and discuss future research directions.

Categories and Subject Descriptors

D.2.11 **Software Engineering**: Software Architectures – *languages*.

General Terms

Management, Design, Languages

Keywords

Architectural adaptation management, self-adaptive software.

1. INTRODUCTION

Many software systems need to be adaptive in response to a host of demands such as additional behavioral requirements, changing deployment platform conditions, or unexpected failures. More often than not these modifications are directed by the human custodians charged with the software's maintenance, and it may often take these operators anywhere from minutes to hours to perform needed adaptations. While for some systems this time scale is perfectly acceptable, other classes of systems operate under more stringent demands. Dynamic software radio systems, space exploration vessels, and distributed systems faced with unpredictable connectivity are just a few examples of systems requiring the capability to autonomously respond to situations dictating change.

Constructing such self-adaptive systems, however, is challenging. Adaptation-related concerns are often too deeply embedded into implementation code, or encoded as static, pre-planned responses to a set of situations predicted at design-time. To address these challenges, we present an approach which integrates insights from knowledge-based systems with high-level software architecture models.

An early formulation of architecture-based self-adaptive software is described in [10]; elaborating and expanding on that foundation, this paper describes a knowledge-based approach to reasoning over the space of possible adaptations in order to build self-adaptive systems. In this Knowledge-Based Architectural Adaptation Management (KBAAM) approach, reasoning and decision-making about the timing and nature of specific adaptations are grounded on knowledge-based adaptation policies. Both policies and relevant system knowledge are represented and explicitly modeled as first-class architectural elements and included in the architectural description of a self-adaptive system. These elements are then dynamically managed and reasoned over by an expert system, which communicates adaptations thought to be necessary to a framework responsible for their management and enactment.

The key elements of our approach are the treatment of both adaptation policies and related knowledge as first-class architectural entities decoupled from system implementation, and the integration of existing knowledge-based techniques for the representation and dynamic reification of these policies allowing for their dynamic and independent evolution.

2. BACKGROUND

The technical underpinnings of our approach can be found in the field of architecture-based runtime software evolution [9], which centers around the reliable runtime management of software dynamism based on architectural models. Such models usually describe systems in terms of components, connectors, and the interconnections between them [11]; in architecture-based software evolution, these models are explicitly bound to the running systems they describe. An Architectural Evolution Manager (AEM) establishes this binding and maintains consistency between runtime systems and their architectures. System change can then be expressed and enacted in terms of modifications to corresponding architectural models with the AEM ensuring that these changes are appropriately reflected on the running system.

An initial formulation of architecture-based self-adaptive software appears in [10], describing the management of such systems using two parallel and concurrent activities: evolution

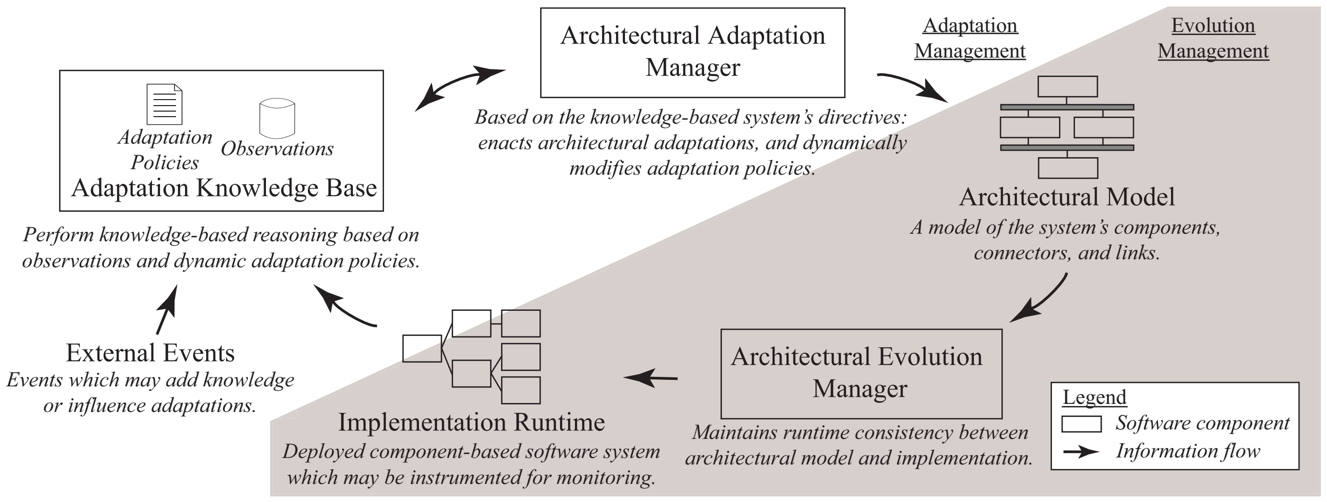
Architectural Adaptation Manager
Based on the knowledge-based system's directives: enacts architectural adaptations, and dynamically modifies adaptation policies.
Adaptation Management
Evolution Management
Adaptation Policies
Observations
Adaptation Knowledge Base
Perform knowledge-based reasoning based on observations and dynamic adaptation policies.
Architectural Model
A model of the system's components, connectors, and links.
External Events
Events which may add knowledge or influence adaptations.
Architectural Evolution Manager
Maintains runtime consistency between architectural model and implementation.
Implementation Runtime
Deployed component-based software system which may be instrumented for monitoring.
Legend
Software component
Information flow

Figure 1. An overview of the elements involved in our approach to knowledge-based self-adaptive software illustrating the interaction of evolution and adaptation management.

management and adaptation management. Evolution management focuses on maintaining runtime consistency between a dynamic architectural model and the actual system this model describes. Adaptation management evaluates events indicating that an adaptation is needed, decides on the specific content of changes, and finally enacts these modifications. Continuously applied, the combination of these two processes results in systems which modify their own behavior solely through the manipulation of architectural descriptions. Of these two activities, evolution management has been better developed and supported to date; our focus here is to better support adaptation management by elaborating on specific methodological and representational techniques for adaptation policies, presenting tools to support their management and evolution, and providing for self-adaptive capabilities through the integration of knowledge-based reasoning.

3. APPROACH

We adopt an architecture-centric approach to self-adaptive software and apply our methods to systems constructed using independent components interconnected through first-class connectors, both explicitly modeled using architectural descriptions. These architectural models are then used as the basis for the decomposition of adaptation into three elements: the kinds of knowledge and information which may indicate the need for adaptation, the architectural operations through which adaptations may be enacted, and the adaptation policies which map between the two. The key insights of this work are the treatment of adaptation policies as first-class architectural elements which can be dynamically evolved during runtime, and the leveraging of existing knowledge-based techniques for dynamic reasoning over the space of architectural knowledge.

A high-level overview of the elements involved in our approach appears in Figure 1. The central abstraction is the *architectural model* which defines the structure of a software system; these architectural models accompany the deployed system they describe at runtime. The *Architectural Evolution Manager* (AEM) maintains consistency between the architecture and its implementation. Events captured from the runtime system or those events generated by system monitoring facilities are collected in an *adaptation knowledge base*. Also collected are *external events* not originating from within the self-adaptive system; these events may encapsulate information about other collaborating software systems or remotely generated adaptation directives. The responsibility of the knowledge-base is to collect, maintain, and reason over these *observations* which encapsulate relevant knowledge about the self-adaptive system. This knowledge base underpins the reasoning process by applying a set of *architectural adaptation policies* over the existing body of observations and determining whether any adaptations are needed. Responses deemed necessary are expressed in terms of architectural modifications communicated to the *Architectural Adaptation Manager* (AAM). These changes may be of two types: modifications of the adaptation policies currently in use by the knowledge base, or structural changes to the system's architectural model. The cycle resumes with the AEM ensuring that enacted architectural changes are dynamically reflected in both the structure and operation of the running system.

3.1 Knowledge-Based Adaptation Policies

Architectural adaptation policies encapsulating the system's responses to prevalent conditions are explicitly modeled as first-class architectural elements and decoupled from the implementation of the self-adaptive system.

Information relevant to adaptation is represented as a collection of dynamically gathered *observations* which establish known facts about the system, while *adaptation policies* specify rules which may be triggered by these observations and determine necessary *adaptation responses* based on the asserted conditions. The general structure of an architectural adaptation policy appears below:

```
AdaptationPolicy id
  (Description desc)?
  (Observation id arg*)+
  (Response id arg*)+
```

Each policy has a unique identifier and may include a textual description. One or more observations are associated with the policy as triggering events, and one or more responses are also specified. Observations and responses are also uniquely identified so that each can be independently referenced and modified. Similar to traditional knowledge-based rules [5], when the entire set of observations is asserted, the adaptation responses specified will be triggered by the governing policy.

While comparable to conditional statements, a rule-based specification has significant advantages in terms of dynamism: the approach allows for individual policies to be dynamically added or removed from an architectural description. Furthermore, individual observations and responses within policies may also be independently modified. The difficulty in the prediction and management of policy interaction introduced by this dynamic behavior can be addressed by facilities external to the policy definition, as discussed in Section 3.2. Governing self-adaptive behavior using knowledge-based policies explicitly modeled as architectural elements couples these dynamic characteristics with a high-level of visibility for better understandability, potential for reuse, and a finer degree of control over adaptive behavior.

3.1.1 Observations

Architectural observations encapsulate knowledge relevant to the operation of a self-adaptive system and are the triggers of adaptation. An additional component being added into a software system, for example, would be represented as such an asserted observation.

In our approach, observations added into the knowledge-base are collected from two sources: the running self-adaptive system itself and external information communicated to the system. The former category of observations may originate in the system as a result of its normal operation or may be emitted by a monitoring infrastructure; our work does not explicitly depend on a particular monitoring approach and we consider a number of such infrastructures as complementary [3, 6]. The latter category of observations are the result of external events which may provide information about remotely located but collaborating software systems or specific adaptation directives provided by a human system operator.

An additional distinction is that knowledge which originates from the system itself and is encapsulated in observations may be of two kinds: structural information about the system's architecture, and semantically-dependent knowledge concerning its intended behavioral characteristics. While the latter kind of knowledge is closely tied to particular systems and would have to be specifically defined by the system's architect, the former may be expressed solely in architectural terms. For example, an observation stating that a particular component has failed carries no system-specific semantic information and would therefore be relevant and have meaning for all component-based systems.

Table 1. Architecture-based observations.

Observation	Description
ComponentFailure(C)	The indicated component or connector has failed.
ConnectorFailure(C)	
ComponentAdded(C)	The indicated component, connector, or link has been added to the architecture.
ConnectorAdded(C)	
LinkAdded(L)	
ComponentRemoved(C)	The indicated component, connector, or link has been removed from the architecture.
ConnectorRemoved(C)	
LinkRemoved(L)	
UnservicedRequest(R)	Request R is left unserviced.
IgnoredNotification(N)	Notification N is ignored.

Table 2. Architecture-based adaptation responses.

Response	Description
AddComponent(C)	Add the indicated component, connector, or link to the architecture.
AddConnector(C)	
AddLink(L)	
RemoveComponent(C)	Remove the indicated architectural element from the architecture.
RemoveConnector(C)	
RemoveLink(L)	
AddObservation(O)	Add the specified observation or adaptation policy to the knowledge base.
AddPolicy(P)	
RemoveObservation(O)	Remove the indicated observation or policy from the knowledge base.
RemovePolicy(P)	

The capability to relate information in this semantically-independent manner allows for the definition of generally applicable, architecture-centric observations that can be reused in multiple contexts and with many different heterogeneous systems. Table 1 presents the collection of such observations we use in our approach. The majority of these straightforwardly deal with information about structural changes of an architecture such as a component removal or the addition of a link. Two of these observations, however, denote potential problems in system composition; *UnservicedRequest* observations indicate that a particular request is not being serviced, while an *IgnoredNotification* specifies that a notification of a service having been completed is not being received by any component.

3.1.2 Adaptation Responses

Adaptation responses are the specific changes to be applied to an architecture as a result of the need for adaptation. These adaptation responses, similarly to the observations discussed in the previous section, may be expressed either in a semantically-dependent or -independent manner. Responses which depend on the semantics of a system would, by necessity, have to be specified by the system architect and included in the representational infrastructure of our approach. However, those responses not coupled with the self-adaptive system's functionality provide a reusable basis for the expression and enactment of changes strictly in terms of high-level architectural elements: components, connectors, links, and – in the case of KBAAM – adaptation policies. The responses presented in Table 2 direct two different classes of adaptations: structural modifications of an architecture, and changes to the policies governing adaptation. Both these classes of responses are independent of the specific functionality the system embodies. Architectures are modified through the addition and removal of structural elements, while the adaptive behavior of the system is modified through the addition and removal from the knowledge-base of both observations as well as policies (policies may also have the constituent elements modified, but these operations are omitted for brevity). The semantic independence of these operations provides a non-trivial basis for structural as well as policy modifications which is widely applicable.

3.2 Architectural Adaptation Management

In our approach, adaptation responses generated by the knowledge-based system are not directly applied; these changes are requested of, coordinated, and finally enacted by the *Architectural Adaptation Manager* (AAM). The AAM is responsible for the actual enactment of requested changes across potentially heterogeneous architectural representations as well as for providing a coordination point for the enactment of architectural constraint resolution and transaction management facilities.

Support for transactions as well as architectural constraints is essential in addressing the unpredictable and non-deterministic nature of knowledge-based systems. Facilities which preserve core parts of a self-healing system and enforce architectural invariants preserve system behavioral specifications in the face of potentially unpredictable adaptations while transaction management ensures that undesirable configurations may be recovered from gracefully.

3.3 Example

As an example of an adaptation policy, consider a node in a sensor network whose purpose is to collect measurement data from nearby sensors and to re-transmit this data using its low-gain antenna; the architecture of this node appears in Figure 2. Essentially, this node acts as longer-ranged data proxy for sensor nodes which have very limited transmission ranges. In this scenario, the useful operational life of the transmission node is limited by available battery power; though it is preferable for this node to continuously transmit data, a competing goal is the extension of its lifetime for as long as possible. The following policy is defined for the node:

```
AdaptationPolicy switch_to_burst
  Observation LowBattery
  Response AddComponent(B)
  Response RemoveLink(C1, T)
  Response AddLink(C1, B)
  Response AddLink(B, C2)
  Response AddLink(C2, T)
```

When the *LowBattery* (a semantically-significant observation defined for this system) observation is added into the adaptation knowledge-base, the specified responses will be enacted resulting in the addition of the *Data Buffer* component. This modification will result in the system reducing its duty-cycle by transmitting data only when the Data Buffer is full rather than continuously forwarding received data, therefore minimizing power-consuming radio transmissions. While this simple adaptive behavior could have been built into the system at design-time, using the architectural principles and adaptation mechanics of our approach means that such behavior can be added into the system during deployment with little foresight on the part of the architect at design-time.

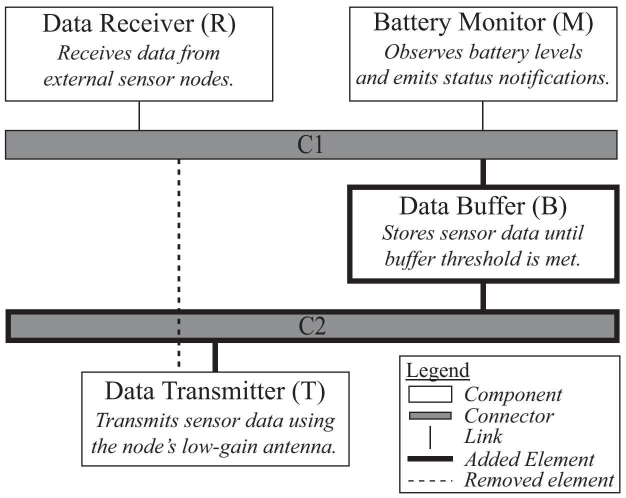
Data Receiver (R)
Receives data from external sensor nodes.
Battery Monitor (M)
Observes battery levels and emits status notifications.
C1
Data Buffer (B)
Stores sensor data until buffer threshold is met.
C2
Data Transmitter (T)
Transmits sensor data using the node's low-gain antenna.
Legend
Component
Connector
Link
Added Element
Removed element

Figure 2. The architecture of the transmitter node showing elements added (in bold lines) and removed (in dashed lines).

4. PROTOTYPE IMPLEMENTATION

To support experimentation with the methodology discussed in the previous sections, we have developed a prototype supporting infrastructure by integrating existing technologies for architectural modeling and evolution management with newly developed facilities for supporting a knowledge-based approach.

The representational basis for the architectural models used in KBAAM is xADL 2.0 [1]: a highly extensible, XML-based ADL. In addition to using the existing facilities of the language for the representation of architectures, we extended the base xADL schemas to define the structure of observations, responses, and adaptation policies. Despite the extensive namespace information associated with XML documents, the current schema extensions are rather lightweight at just over 100 lines of XML schema definitions. These schema definitions are further extensible so that architects can define their own system-specific observations and responses. We also leveraged the functionality of the ArchStudio 3 [7] environment for the runtime architecture-based evolution of software systems (the AEM, illustrated in Figure 1, is an existing component of the ArchStudio toolset).

To support the knowledge-based reasoning facilities of our approach, we implemented a knowledge-based expert system using the Java Expert System Shell (JESS) [2]. The expert system was constructed as an autonomous ArchStudio 3 component using the adaptation policies and observations specified in the system's xADL architectural description to dynamically instantiate, maintain, and reason over a running knowledge-base. Finally, we implemented a prototype of the AAM tool, also integrated into the ArchStudio 3 toolset, for the enactment of adaptation responses; rudimentary constraint resolution facilities were implemented using the existing Critic [12] infrastructure included in ArchStudio.

5. RELATED WORK

Garlan and Schmerl have also presented an architecture-based approach to developing self-adaptive systems [4]. In their work, architectural style specifications are used as the basis for deciding when to apply pre-specified programmatic modifications expressed using style-specific operations. Though KBAAM shares a similar overall approach, our work focuses on using dynamically maintained policies and system observations allowing for the evolution of adaptation policies at runtime.

The Chemical Abstract Machine (CHAM) model has also been presented as a basis for specifying autonomous software architectures. Inverardi and Wolf [8] describe how system components and the data elements they produce and require can be expressed as CHAM molecules while system configurations are represented as solutions. In their approach, system reconfigurations are driven by reaction rules that modify these abstract solutions. Despite their different formalizations, this work and KBAAM both express reconfigurations as an extensible, non-deterministic set of rules which modify the representation of a system. The CHAM approach, however, does not dynamically reconfigure its reaction rules during system operation, and its highly abstract nature makes mapping reconfigurations to running systems challenging.

Other researchers have also presented formal approaches for supporting software evolution. Wermelinger and Fiadeiro propose an algebraic approach to architectural reconfiguration using graph rewriting [13]; their approach is especially concerned with ensuring sane reconfigurations and system quiescence during such reconfigurations. While their discussion does not cover how to plan specific reconfigurations, which is the focus of KBAAM, we plan on investigating how their ideas on ensuring the sanity of architectural change integrate with our work.

6. CONCLUSIONS AND FUTURE WORK

One of the most challenging aspects of constructing self-adaptive software is determining the timing and the specifics of adaptations. Most current approaches to this challenge involve adaptation mechanisms which are static and embedded in the implementation details of the software system being adapted.

In this paper we present our research efforts towards developing systems exhibiting dynamic and independently evolvable adaptive behavior. KBAAM is an architecture-centric, knowledge-based approach to developing systems that are able to autonomously adapt in the face of change. Building on prior research performed in the context of runtime evolution management, the key features of our work are: the treatment of adaptation policies as explicitly defined architectural elements strictly decoupled from system implementations, the dynamic management and independent evolution of these adaptation policies, and the integration of existing knowledge-based techniques for the management and planning of adaptive behavior.

In the future, we plan on addressing areas where refinements and significant improvements can be made to the KBAAM approach and the prototype infrastructure realizing it. Short-term development plans include further work on the representational underpinnings of adaptation policies and their relation to structural architectural models in addition to improving the visibility of ongoing adaptations to the system architect through the collection and distribution of log information. In the long-term, we plan on investigating the applicability of our techniques in decentralized settings – where systems are composed of independent peers – and examining ways to achieve global adaptations through the coordination of localized per-peer changes in addition to evaluating the cognitive (how difficult our approach is to understand and use) and computational overhead involved in adopting our approach as well as its scalability characteristics.

7. ACKNOWLEDGEMENTS

The authors would like to thank Andre van der Hoek, Michael Gorlick, Eric Dashofy, and Kari Nies for their helpful comments and insights on this work. This work was supported in part by by NSF Grant CCF-0430066.

8. REFERENCES

[1] Dashofy, E.M., Hoek, A.v.d., and Taylor, R.N. 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.

[2] Friedman-Hill, E. *Jess in Action: Rule-Based Systems in Java.* Manning Publications Co., 2003.

[3] Garlan, D., Schmerl, B.R., and Chang, J. Using Gauges for Architecture-Based Monitoring and Adaptation. In *Proceedings of the The Working Conference on Complex and Dynamic System Architecture.* Brisbane, Australia, December, 2001.

[4] Garlan, D. and Schmerl, B. Model-based adaptation for self-healing systems. In *Proceedings of the First Workshop on Self-Healing Systems.* November, 2002.

[5] Hayes-Roth, F. The Knowledge Based Expert System: A Tutorial. *IEEE Computer.* 17(9), p. 11-28, 1984.

[6] Hilbert, D. and Redmiles, D. An Approach to Large-scale Collection of Application Usage Data over the Internet. In *Proceedings of the 20th International Conference on Software Engineering (ICSE '98).* p. 136-145, IEEE Computer Society Press. Kyoto, Japan, April 19-25, 1998.

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

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

[9] 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).* p. 177-186, IEEE Computer Society. Kyoto, Japan, April, 1998.

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

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

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

[13] Wermelinger, M. and Fiadeiro, J.L. Algebraic Software Architecture Reconfiguration. In *Proceedings of the 7th European Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* p. 393-409, Springer-Verlag. Toulouse, France, 1999.