Self-Adaptive Software

Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor,
Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic,
Alex Quilici, David S. Rosenblum, Alexander L. Wolf

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

Technical Report 98-27

August 1, 1998

Abstract

An increasing number of mission-critical software systems require dependability, robustness, adaptability, and continuous availability. Self-adaptive software strives to fulfill these requirements. Self-adaptive software requires the simultaneous management of two related processes—system evolution, the consistent and reasoned application of change over time, and system adaptation, the cycle of detecting changing circumstances, planning responsive modifications, and deploying those modifications to field sites. We indicate how these two processes can be combined and coordinated in an infrastructure that supports ongoing continuous system change without requiring any downtime, rebuilds, or system restarts—all the while guaranteeing system integrity. Central to our methodology is the dominant role of software architectures in planning, coordinating, monitoring, evaluating, and implementing software adaptation. We argue that explicit system structure and representation, the separation of architecture from implementation, and the separation of communication from computation are scalable abstractions suitable for domains and adaptations of all sizes.

Self-Adaptive Software

Peyman Oreizy[1], Michael M. Gorlick[2], Richard N. Taylor[3], Dennis Heimbigner[4], Gregory Johnson[5], Nenad Medvidovic[6], Alex Quilici[7], David S. Rosenblum[8], and Alexander L. Wolf[9]

ABSTRACT

An increasing number of mission-critical software systems require dependability, robustness, adaptability, and continuous availability. Self-adaptive software strives to fulfill these requirements. Self-adaptive software requires the simultaneous management of two related processes—*system evolution*, the consistent and reasoned application of change over time, and *system adaptation*, the cycle of detecting changing circumstances, planning responsive modifications, and deploying those modifications to field sites. We indicate how these two processes can be combined and coordinated in an infrastructure that supports ongoing continuous system change without requiring any downtime, rebuilds, or system restarts—all the while guaranteeing system integrity. Central to our methodology is the dominant role of software architectures in planning, coordinating, monitoring, evaluating, and implementing software adaptation. We argue that explicit system structure and representation, the separation of architecture from implementation, and the separation of communication from computation are scalable abstractions suitable for domains and adaptations of all sizes.

1 INTRODUCTION

Consider the following scenario. A fleet of unmanned air vehicles (UAVs) is sent to disable an enemy airfield. Pre-mission intelligence indicated that the airfield is not defended and the mission is planned accordingly. While en route to the target new intelligence indicates that the airfield is now being guarded by a surface-to-air missile (SAM) site. The UAVs autonomously replan their mission, dividing into two groups—a SAM suppression unit and an airfield suppression unit—and proceed to accomplish their objectives. Specialized algorithms for the detection and recognition of SAM launchers are automatically uploaded and integrated into the software of the SAM suppression unit without requiring any downtime, rebuilds, or system restarts, yet guaranteeing ongoing system integrity.

One can imagine other applications for fleets of UAVs, including environment and land use monitoring, freeway traffic management, firefighting, airborne cellular telephone relay stations, and damage surveys in times of natural disaster. How wasteful to construct afresh a specific software platform for each new UAV application. Far better if the platform is just adapted to the application at hand, and most desirous of all, if the platform is adapted on demand even while serving some other purpose. For example, an airborne sensor platform designed for environmental and land use monitoring could prove useful for damage surveys following an earthquake or hurricane if only we could change the software quickly enough and with sufficient assurance that the new system would perform as intended.

The holy grail of software engineering is the systematic, principled design and deployment of applications that fulfill the original promise of *soft*ware—applications that retain full plasticity throughout the lifecycle and that are as easy to modify in the field as they are on the drawing board. Many techniques for achieving this goal have been

1. University of California at Irvine, peymano@ics.uci.edu
2. The Aerospace Corporation, gorlick@rush.aero.org
3. University of California at Irvine, taylor@ics.uci.edu
4. University of Colorado, dennis@cs.colorado.edu
5. Northrop Corporation, johnson@charming.nrtc.northrop.com
6. University of California at Irvine, neno@ics.uci.edu
7. University of Hawaii at Manoa, alex@wiliki.eng.hawaii.edu
8. University of California at Irvine, dsr@ics.uci.edu
9. University of Colorado, alw@cs.colorado.edu

pursued—specification languages, high-level programming languages, and object-oriented analysis and design, to name just a few. However, while each contributes to the goal, the sum total still falls short.

What are some of the distinguishing characteristics of the scenario given above? New software components are dynamically inserted into fielded, heterogeneous systems without requiring system restart, or indeed, any downtime at all. Mission replanning is based on analyses that include feedback from current performance. Furthermore, such replanning may take place autonomously, may involve multiple, distributed, cooperating planners, and in the case where major changes are demanded and require human approval or guidance, may take place cooperatively with mission analysts. Throughout, consistency, correctness, and coordination of changes must be ensured to guarantee system integrity.

While many disciplines will contribute to the progress of self-adaptive software, wholesale advances are possible only if we adopt a systems perspective based upon a broadly-inclusive adaptation methodology spanning a wide range of adaptive behaviors. Central to our view is the dominant role of software architecture in planning, coordinating, monitoring, evaluating, and implementing seamless adaptation. In this article we present a definition of self-adaptive software and outline our adaptation methodology. We also examine the fundamental role of software architectures in self-adaptive systems and outline some the technologies that we have considered to support the methodology. Finally, we assess the hazards of self-adaptive software and review some of the research questions that must be resolved before self-adaptive systems are widely deployed.

2 WHAT IS SELF-ADAPTIVE SOFTWARE?

We define self-adaptive software as a system that modifies its own behavior in response to changes in its operating environment. By operating environment we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation.

Several questions must be answered when developing a self-adaptive software system:

- Under what conditions does the system undergo adaptation? A system may, for example, modify itself in order to improve system response time, recover from a subsystem failure, or incorporate additional behavior during runtime.
- Should the system be open-adaptive or closed-adaptive? A system is open-adaptive if new application behaviors and adaptation plans can be introduced during runtime. A systems is closed-adaptive if it is self-contained and not able to support the addition of new behaviors.
- What type of autonomy must be supported? A wide range of autonomy may be needed, from fully automatic, self-contained adaptation to human-in-the-loop.
- Under what circumstances is adaptation cost-effective? The benefits gained from a change must outweigh the costs associated with making the change. Costs include the performance and memory overhead of monitoring system behavior, determining if a change would improve the system, and paying the associated costs of updating the system configuration.
- How often is adaptation considered? A wide range of policies may be used, from opportunistic, continuous adaptation to lazy, as-needed adaptation.
- What kind of information must be collected to make adaptation decisions? How accurate and current must the information be? A wide range of strategies may be used, from continuous, precise, recent observations to sampled, approximate, historical observations.

Our definition covers a broad spectrum of self-adaptability (see Figure 1). At the bottom of the spectrum, one could argue that the use of conditional expressions constitutes a form of self-adaptation—the program evaluates an expression and alters its own behavior based on the outcome. Although simplistic, conditional expressions are a common mechanism for implementing adaptive behavior. For example, a just-in-time compiler may utilize aggressive optimization techniques only for frequently invoked functions.

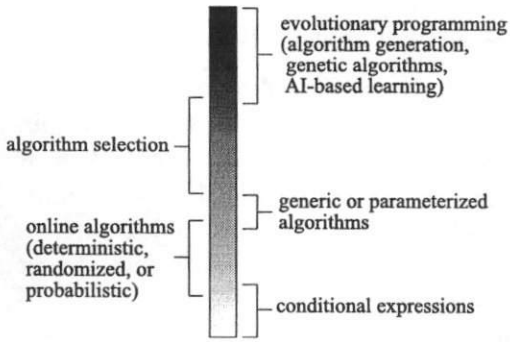

Figure 1. A spectrum of self-adaptability. Generally, approaches near the bottom select among predetermined alternatives, support localized change, and lack separation of concerns. Approaches near the top support unprecedented changes and provide a clearer separation among adaptation issues.

Online algorithms operate under the assumption that future events (e.g., inputs) are uncertain. Hence, online algorithms will occasionally perform an expensive operation in order to more efficiently respond to future operations (see [6] for a general discussion of online algorithms). Online algorithms are adaptive in that they leverage knowledge about the problem and input domains in order to modify their behavior to execute as efficiently as possible. An memory cache paging algorithm, for example, leverages spacial and temporal locality of memory references when determining which cached page to evict in order to make room for a newly requested page.

Generic and parameterized algorithms provide parameterized behaviors either through type instantiation or tunable parameters. Generic algorithms are adaptive in that they conform to different data types. The C++ Standard Template Library (STL), for example, provides generic iterator classes that may be used to traverse many different data structures.

Algorithm selection uses properties of the operating environment to select the most effective algorithm from a given collection of algorithms. Systems based on the algorithm selection are adaptive in that they switch between different algorithms based on the operating environment. For example, the Self dynamic optimizing compiler [5] uses program profiling data generated during program execution to guide the selection of different optimization algorithms.

At the top of the spectrum, evolutionary programming represents problem-solving techniques that simulate evolutionary mechanisms found in biological systems (such as mutation, natural selection, and learning) to solve computational problems [11]. Systems based on evolutionary programming techniques are adaptive in that they use properties of the operating environment and knowledge gained from previous attempts to generate new algorithms.

Generally, approaches near the bottom of the spectrum intertwine concerns regarding software adaptation and application-specific behavior—application functionality and adaptation strategy are inextricably tied to one another—making independent modification unrealistic. Approaches near the top of the spectrum make a clearer separation between software adaptation concerns and application-specific functionality. As a result, different strategies for monitoring, evaluating, and enacting adaptation are possible.

3 SOFTWARE ADAPTATION IN-THE-LARGE

While technical advances in narrow areas of adaptation technology benefit some, the greatest benefit will accrue by developing a comprehensive *adaptation methodology* spanning adaptation-in-the-small to adaptation-in-the-large, and then developing the technology that supports the entire range of adaptations. We illustrate such a methodology in Figure 2.

The upper half of the diagram, *adaptation management*, describes the lifecycle of adaptive software systems. The lifecycle may have humans-in-the-loop or be fully autonomous. *Evaluation and monitoring* refers to all forms of evaluation and observation of the execution of an application including, at a minimum, performance monitoring,

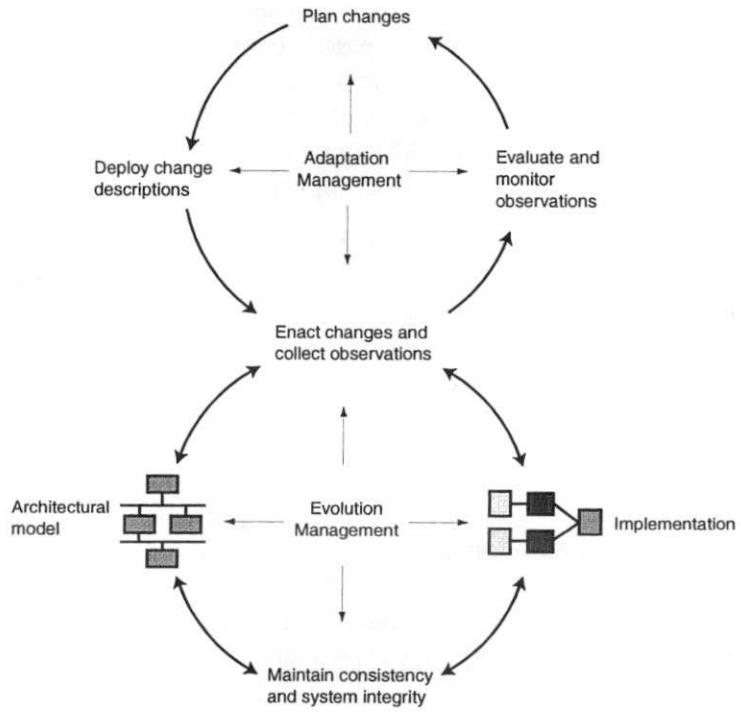

Figure 2. High-level processes in a comprehensive, general-purpose approach to self-adaptive software systems.

safety inspections, and constraint verification. *Planning* refers to the task of accepting the evaluations, defining an appropriate adaptation, and constructing a blueprint for carrying out that adaptation. *Deployment* is the coordinated conveyance of change descriptions, components, and possibly new observers or evaluators to the implementation platform in the field. Conversely, deployment may also extract data, and possibly components, from the running application and convey them to some other point for analysis and optimization.

Adaptation management and consistency maintenance play key roles in our approach. While mechanisms for runtime software change are available in operating systems (for example, dynamic link libraries in Unix and Microsoft Windows), component object models, and programming languages, these facilities all share a major shortcoming—they do not ensure the consistency, correctness, or other desired properties of runtime change. Change management is a critical aspect of runtime system evolution that: identifies: what must be changed; provides the context for reasoning about, specifying, and implementing change; and controls change to preserve system integrity. Without change management, the risks engendered by runtime modifications may outweigh those associated with shutting down and restarting a system.

Even disciplined adaptation is a complex process. It is further complicated by change drivers ranging from purposeful adjustments in fielded systems to unanticipated perturbations in the operational environment. The changes themselves encompass everything from a simple replacement of an isolated component to wholesale reconfigurations that are pervasive and physically distributed. Our approach addresses these demanding and unprecedented requirements by managing adaptation using a flexible infrastructure supporting the full range of adaptation processes. Our infrastructure relies upon: (1) proactive and reactive agents (i.e., independent "loci of initiative") that automate tasks within the process; (2) explicit representations of software components, their interdependencies, and their environmental assumptions; (3) explicit representations of the environments in the field where software is deployed; and (4) wide-area messaging and event services that connect adaptation managers to adaptive systems to permit coordinated and coherent adaptation in physically distributed, logically decentralized environments.

The lower half of Figure 2, *evolution management,* focuses on the mechanisms employed to change the application software. Our approach is architecture-based: changes are formulated in, and reasoned over, an explicit *architectural model* residing on the *implementation* platform. Changes to the architectural model are reflected in modifications to the implementation of the application while ensuring that the model and the implementation are consistent with one another. Monitoring and evaluation services observe the system and its operating environment and feed information back to the upper half of the diagram.

Software architectures [8,10] are a foundation for systematic runtime software evolution and change management and provide a cornerstone of our approach. An architectural perspective shifts focus away from source code to coarse-grained components and their interconnections. Designers can abstract away obscuring details and concentrate on the big picture: the system structure, the interactions among components, the assignment of components to processing elements, and runtime change. Architectures provide three key enablers for effective change management: (1) the separation of structure from function enables dynamic manipulation of the structure's topology independent of manipulation of the functional behavior of its components; (2) the separation of architecture from implementation allows architectural changes to be analyzed before they are applied; and (3) the separation of computation from communication allows designers to focus on changes to the functionality (components) or to the communication mechanisms and protocols (connectors). Connectors mediate and govern interactions among components—by separating computation from communication they minimize component interdependencies and aid system understanding, analysis, and evolution. Connectors are a distinctive aspect of architecture-centric dynamic change and play a key role in our approach.

4 EVOLUTION MANAGEMENT

Dynamic architectures view systems as networks of concurrent components bound together by connectors. Components are responsible for implementing application behavior and maintain state information. Connectors are transport and routing services for messages or objects. Components do not know or care how their inputs and outputs are delivered or transmitted or even what their sources or destinations might be. On the other hand, connectors know exactly who is talking to whom and how—but are ignorant of the computations of the components they serve. Strictly separating computation from communication makes it possible to evolve the computation and communication relationships of a system independently of one another, including rearranging and replacing the components and connectors of an application while the application is executing—a necessary, but insufficient, mechanism for self-adaptive software.

It is not enough that we can rearrange and replace portions of an application while it is executing. Self-adaptive systems present a unique set of challenges with respect to safety, reliability, and correctness. Consequently, facilities for guiding and checking modifications are an integral part of architecture-centric change. Evolution management, the process by which change is applied and controlled, is shown in Figure 3. A variety of tools and adaptation mechanisms evolve an application by inspecting and changing its architectural model. Changes may include the addition, removal, or replacement of components and connectors, changes to the configuration or parameters of components and connectors, and alterations in the topology of the component/connector network. Evolution management maintains system consistency and integrity by examining each change and vetoing any changes that render the system inconsistent or unsafe. We briefly examine possible mechanisms by which evolution management will maintain integrity and enact changes.

4.1 Maintain Consistency and Integrity

As an application adapts and evolves we face the problem of preserving an accurate and consistent model of the application architecture and its constituent parts --- the components and the connectors. We must also maintain a strict correspondence between the architecture model and the executing implementation. To deal with these problems we deploy, as an integral part of the application, an architectural model that describes the interconnections among components and connectors and their mappings to implementation artifacts. The mapping permits changes, given in terms of the architectural model, to effect corresponding changes in the actual implementation.

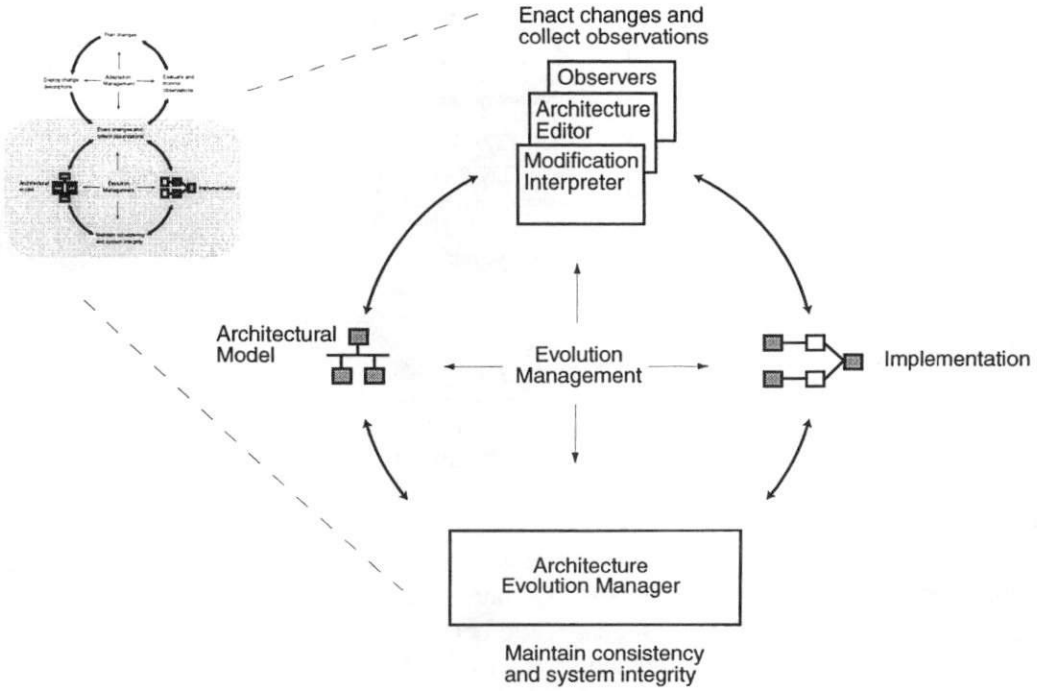

Figure 3. A high-level architecture diagram for the ArchStudio tool suite.

To guard against untoward change an *Architecture Evolution Manager* (AEM) mediates all change operations directed toward the architectural model. A change is expressed either as a single basic operation or as a *change transaction* composed of two or more basic operations. All changes are atomic; that is, they either complete without error or leave the application untouched. A change transaction includes operations for forcing components and connectors into safe or halt states; adding, removing, and replacing components and connectors; and changing the architectural topology.

The AEM maintains the consistency between the architectural model and the implementation as changes are applied, reifies changes in the architectural model to the implementation, and prevents changes from violating architectural constraints. For example, it can enforce the generic constraint that all components must be connected to at least one connector but not more than two. The AEM is also tailored by application- and domain-dependent *change policies* that dictate the forms of acceptable change. Within the UAV domain the AEM may require that the UAV system contain at least one navigation component. The AEM, which maintains the mapping between the architectural model and the implementation, uses it to carry out modifications by mapping model components and connectors into implementation artifacts, and translating change operations into implementation actions. Self-adaptive systems present a unique set of challenges with respect to safety, reliability, and correctness, and therefore facilities for guiding and checking modifications must be provided as an integral part of the support for architecture-centric change.

4.2 Enact Changes

There are many possible sources of architectural change—including the application itself, external tools, and replanning agents. A visual, interactive, *architecture editor* may be used by software architects to construct architectures and describe modifications. A second, companion tool is a *modification interpreter*. Its inputs are expressed in a *change description language*. The interpreter translates a change description into the primitive actions supported by the AEM. A variety of analysis tools may accompany the editor; for example, a design wizard that critiques an architecture as a designer constructs it, or application- and domain-dependent design wizards that, by exploiting specialized knowledge, can prevent semantic errors or ensure a minimum level of performance or safety.

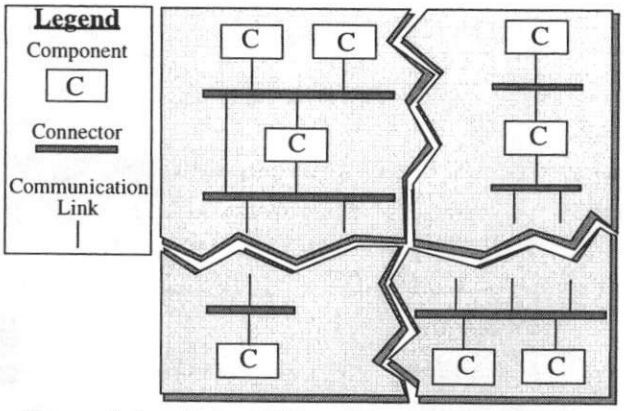

Figure 4. An abstract C2 architecture. Jagged lines represent portions of the architecture not shown.

4.3 Dynamic Software Architectures

C2 and Weaves are two related approaches to the problem of implementing dynamic architectures. Both distinguish between components and connectors, neither C2 nor Weaves places restrictions on the granularity of the components or their implementation language, and both require that all communication between components occur by exchanging asynchronous messages (C2) or objects (Weaves). Components may encapsulate functionality of arbitrary complexity and exploit multiple threads of control.

4.3.1 The C2 Architectural Style

The C2-style [12] embraces a particular architectural style based on the principle of limited visibility or *substrate independence*: a component within a hierarchy can only be aware of components "above" it and is completely unaware of components residing "beneath' it. The C2-style is illustrated in Figure 4. All components and connectors have a "top" and a "bottom." The top of a component may be connected to the bottom of at most one connector. Conversely, the bottom of a component may be connected to the top of at most one connector. A component explicitly utilizes the services of components "above" it by sending request messages to the connector above it directed to a specific component. The connector transmits the request message to the named component. The request message may transit multiple connectors before arriving at its destination.

Communication with components below occurs implicitly: whenever a component changes its internal state, it announces the change by emitting a notification message describing the state change to the connector below it. The connector broadcasts these notification messages to every component (and connector) connected on its bottom side, propagating downward to all components below. Thus, notification messages provide an implicit invocation mechanism, allowing several components to react to a component's state change. The directed communication from bottom to top and the broadcast communication from top to bottom is a critical element of the C2-style.

4.3.2 Weaves

Weaves [2] are a dynamic, object-flow-centric architecture designed for applications characterized by continuous or intermittent high-bandwidth data flows. Components in Weaves consume objects as inputs and produce objects as outputs (object is intended in the sense of C++, Smalltalk, or Java). An example Weaves architecture for a portion of a stereo tracker is depicted in Figure 5. Weaves embraces a set of architectural principles known as the *laws of blind communication*:

- No component in a network is aware of the sources of its input objects or the destinations of its output objects
- No component in a network is aware of the semantics of the connectors that delivered its input objects or transmitted its output objects
- No component in a network is aware of the loss of a connection

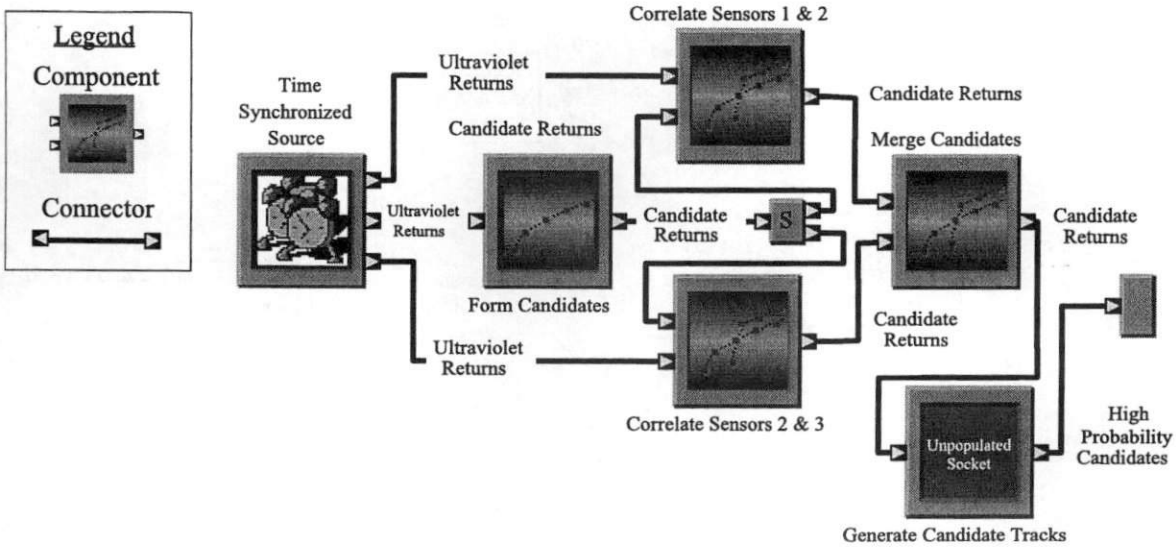

Figure 5. A portion of a Weaves architecture for a stereo tracking system.

These laws ensure that no weave component is aware of its location in the network, that every component is independent of the semantics of the connectors to which it is attached, and that any weave can be edited, rewired, expanded, or contracted on the fly.

Weaves permit connectors to be composed of other connectors and components, allowing weave connectors to be specially adapted to the characteristics of their operating environment with corresponding performance gains.

Weaves were originally developed for processing satellite telemetry, an application domain characterized by voluminous, continuous streams of data and real-time deadlines. Additional applications include satellite ground station architectures, targeting and tracking, and satellite attitude control.

4.3.3 Comparison of C2 and Weaves

Although they have much in common, C2 and Weaves emphasize different aspects of the problem of dynamic architectures. C2 permits a degree of dynamic *component flexibility* unmatched by any other architectural medium. The connector semantics of C2 and its accompanying style rules ensure that one component is almost completely ignorant of the placement, function, and implementation of its fellow components. Consequently, at runtime, components may be added, deleted, or rearranged with remarkable ease and alacrity.

In contrast, Weaves, while it supports like forms of component manipulation, emphasizes the dynamic distribution, modification, and rearrangement of *connectors*. This allows developers to optimize inter-component communication while a weave is executing, including wholesale movement of a (sub)weave from one host to another along with dramatic changes in the semantics and implementations of its connectors. In short, C2 has been optimized for flexible components, while Weaves focus on high performance, flexible connectors. One of the research issues we face is blending these two approaches to dynamic architectures into a single, cohesive whole. One possible approach is to treat Weaves as an implementation substrate for C2 and "compile" C2-style architectures into lower level, but more efficient, weaves.

5 ADAPTATION MANAGEMENT

A self-adaptive system observes its own behavior and analyzes these observations to determine appropriate adaptations. Companion to the process of evolution management is the process of adaptation management, illustrated in Figure 6. Adaptation management monitors and evaluates the application and its operating environment, plans adaptations (i.e., changes), and deploys change descriptions to the running application.

Viable self-adaptive systems require long-term continuity in the face of dynamic change, in other words, both a standard locale for the information and tasks required to carry out the function of adaptation, and a focal point for

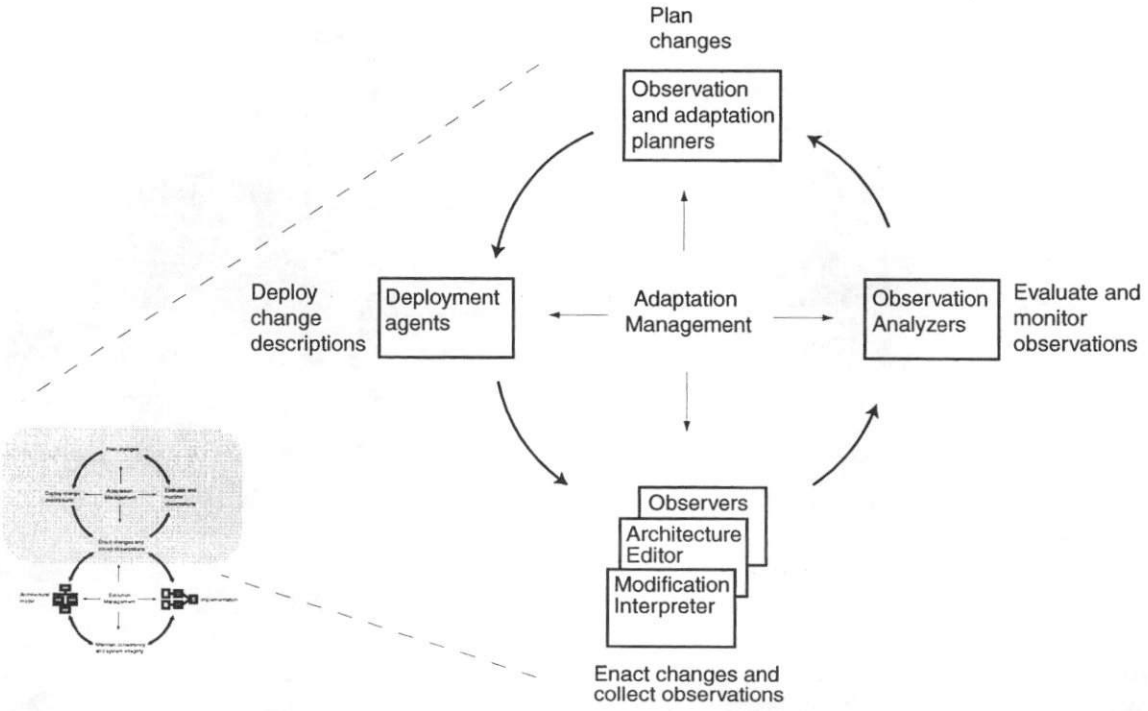

Figure 6. A high-level architecture diagram for adaptation evolution.

coordinating physically distributed, logically decentralized adaptation tasks. For example, complex interdependencies may exist among changes such that the incorporation of one change may require the inclusion of several others for the change to work correctly in its environment and a standard locale helps ensure that such information is at hand. Additionally, an adaptation may require coordination among multiple sites in the case where the application is physically distributed and adaptation requires changes at several sites simultaneously.

Additionally, managing self-adaptive software requires a variety of "agents" such as *observers*, which evaluate the behavior of the self-adaptive application and monitor its operating environment, *planners* that utilize the observations to plan adaptive responses, and *deployers* which enact the responses within the application.

These requirements suggest that a "distributed registry" will be needed to host the numerous agents and support the various activities of adaptation management: monitoring and evaluating the adaptive application, planning changes, deploying change descriptions to the application, and enacting the changes. Registries at each application site would contain resource descriptions, configurations, and other declarative information relevant to the site and the adaptive application. Registries elsewhere might be dedicated to overseeing and coordinating the activities of the individual application site registries. Each registry would provide a standard interface by which disparate agents and interests can query and manipulate the contents of the registry, which acts as a "blackboard" for exchanging information. Inter-registry communication could take many forms, ranging from directed updates to wide-area messaging and event notification. One promising starting point is the Software Dock, an infrastructure element for the distributed configuration and deployment of software systems now under development at the University of Colorado, Boulder [1].

5.1 Collect Observations

Large numbers and varieties of observations and measurements are required for self-adaptive software ranging from event generation within the application implementation to animations suitable for human observers. Furthermore, it must be possible to adjust the number, extent, and detail of the observations and measurements as the application executes and evolves so as to reduce measurement overhead to a minimum and avoid wasting communication bandwidth on unnecessary observations.

At a minimum, we will require embedded assertions (inline observers) within the application itself for notification of exceptional events such as resource shortages or the violation of low-level constraints. Additional required capabilities include dynamic control and alteration of the scope of the assertions, insertion and removal of assertions while the application is executing, language-independent assertions, and architecture-sensitive assertions. One potential candidate for this facility is APP, a tool that supports the automated checking of logical assertions expressed in first-order predicate logic [9].

Detecting and noting single events is not enough since many adaptive strategies will be triggered by the occurrence of a pattern of events distributed in both time and place. One approach is to model application behavior abstractly in terms of patterns of events. In this way, the architect's expectations are expressed as an *expectation agent* [4]. The expectation agent responds to the occurrence of event patterns, including generating a higher-level abstract event for the benefit of other (expectation) agents. The expectation agent is a formal specification that, depending upon its complexity, can be translated into an observer embedded within the application or implemented as an agent that eavesdrops on the activity of the local registry. In addition, we must monitor events that occur outside of the application—such as the quality or availability of a network connection—as well as adaptation events that arise as a consequence of dynamic architectural change.

We must also make provision for observation by human observers in cooperation with automated agents. One appealing technology is Joist, which exploits standard World Wide Web-based technologies—HTTP and HTML—to provide a powerful and efficient infrastructure for remote observation of distributed applications [3]. Joist embeds a small Web server in the runtime environment of the application, which then monitors the application and gathers information. This information is identified through a special Uniform Resource Locator (URL) namespace and it is presented in HTML pages that are generated by the Joist server and communicated to any standard Web browser via HTTP.

New techniques must be developed to reduce the overhead of monitoring. Weaves employ statistical monitoring techniques that allow observers to trade accuracy in favor of reduced overhead. Using this approach the invasive effect of instrumentation on a running application can be reduced to below the noise threshold while still obtaining useful information. Furthermore, the instrumentation can be left permanently embedded within the application and an observer can selectively measure only behaviors of interest without adversely impacting the application. This technique can be used in a variety of ways, including performance analysis and real-time animation of the behavior of running systems.

5.2 Evaluate and Monitor

Adaptive demands can arise from inconsistencies or suboptimal behavior within the system. In particular, inconsistencies can occur when some architectural element (ranging from a single component or connector to a subsystem, or the entire architecture) behaves in a manner inconsistent with the behavior required of it, or when an element's assumptions about its operating environment become invalid. Maintaining consistency in these situations requires monitoring and evaluating representative behaviors of the running system and comparing them to an explicit formulation of behavioral requirements or environmental assumptions.

Successful consistency management will require a hybrid approach that combines both static and dynamic analysis. We outline a few of the many possible techniques here. One promising form of static analysis exploits attributed graph grammars. Recall that dynamic architectures can be characterized as graphs of components and connectors. Attributed graph grammars can represent the set of all possible configurations of an application where architectural changes are regarded as graph rewrite operations. Analysis tools can determine if an invariant is preserved by all possible architectures or return an example graph (architectural configuration) that violates the invariant.

Static analysis may be insufficient in which case runtime checks will be employed to detect inconsistencies. *Observers* inspect both the application and the environment in which the application operates and evaluate their observations for consistency with relevant *annotations* obtained from the registries. Observers are generated and launched automatically based on the constraints and properties extracted from annotations pertaining to the element under observation. Observers post observed inconsistencies, aggregated observations, and analyses to the registry.

5.3 Plan Changes

Planning is also a vital aspect of self adaptive software. Two distinct forms of planning are required for self-adaptation: *observation planning* and *adaptation planning*. Observation planning determines which observations are necessary for deciding when and where adaptations are required. The observation planner takes into account environmental assumptions, expected behaviors, the availability of observers and observations, and their costs. This task can be viewed as a classic planning problem where the goals are information needs, the operators are the observers, the preconditions are required event types, the postconditions are observer-generated event types, and the operators have observation and notification costs. This type of planning is well within the range of today's planning technology.

Adaptation planning determines exactly which adaptations to make and when. The adaptation planner must take into account the purpose of components, their environmental assumptions, and known properties of the environment. We are interested in applications (such as UAVs) where adaptations must be planned in minutes, not hours. Our approach to this problem relies on the use of pre-defined *solution frameworks* that, by limiting the range and variation of possible adaptations, drastically reduce the computation required for planning.

A solution framework is a partially-instantiated hierarchical solution architecture consisting of connectors, sockets (placeholders for components), and an equivalence class of candidate components for each socket (where components can themselves be solution frameworks). Given such a framework, an initial solution architecture may be found by selecting components for each socket from the set of eligible components. Given this as a starting point the system can undergo *incremental adaptation* by choosing alternatives for problematic components whose environmental assumptions no longer hold in the observed environment.

A more general approach explicitly represents the pre- and post-conditions of each component that could affect, or be affected by, other components; represents each socket in terms of one or more required postconditions; and treats framework instantiation as a planning problem. This approach requires a partial domain model and additional computation but no longer requires that candidate components form an equivalence class. It has already been demonstrated in another domain, the automatic generation of simulation scripts for tank training.

5.4 Deploy Change Descriptions

Change agents propagate and move out among sites to carry out their tasks. Imagine a scenario where a coordinated change is required at two separate sites. Agents responsible for each portion of the coordinated change are dispatched from a third site (which oversees the other two) taking with them the change descriptions to be installed. Included in the change descriptions are any new required components or connectors and their affiliated annotations. These agents, once situated at the registries of their respective sites, will interact with the local *modification interpreter,* which translates the change transactions contained within the change descriptions into the primitive actions supported by the site's architecture evolution manager.

6 CONCLUSIONS

Although each individual aspect of our approach has been the focus of much research, integrating these aspects into a comprehensive self-adaptive software methodology is unprecedented. In the near future, we hope to complete an initial integration of our dynamic architecture technology, event-based monitoring and evaluation technology, and software deployment technology.

7 ACKNOWLEDGEMENTS

We would like to thank David M. Hilbert and Jason E. Robbins for discussions and feedback on this work.

8 REFERENCES

1. R. S. Hall, D. Heimbigner, A. van der Hoek, A. L. Wolf. An architecture for post-development configuration management in a wide-area network. *17th International Conference on Distributed Computing Systems*, Baltimore, Maryland, May 1997.
2. M. M. Gorlick, R. R. Razouk. Using Weaves for software construction and analysis. In the *13th Proceedings of the International Conference on Software Engineering (ICSE'91),* May 1991.
3. M. M. Gorlick. Distributed debugging and monitoring on $5 a day. In the *Proceedings of the 1997 California*

Software Symposium, pp. 31-39. November 1997.
4. D. M. Hilbert, D. F. Redmiles. An approach to large-scale collection of application usage data over the internet. In the *Proceedings of the International Conference on Software Engineering 1998 (ICSE'98).* Kyoto, Japan, April 1998.
5. U. Holzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming,* Dissertation. Stanford University. August 1994.
6. S. Irani, Anna R. Karlin. On Online Computation. *Approximation Algorithms for NP-hard Problems*, edited by Dorit Hochbaum. PWS Publishing Company, Boston, MA. July 1996.
7. P. Oreizy, N. Medvidovic, R. N. Taylor. Architecture-based runtime software evolution. Proceedings of the *International Conference on Software Engineering 1998 (ICSE'98),* Kyoto, Japan. April 1998.
8. D. E. Perry, A. L. Wolf, Foundations for the study of software architecture. *Software Engineering Notes*, 17(4), 1992.
9. D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering,* 21(1):19-31, January 1995.
10. M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.
11. W.M. Spears, K.A. DeJong, T. Baeck, D. Fogel, H. de Garis. An Overview of Evolutionary Computation, Proceedings of the European Conference on Machine Learning, 1993. pp. 442-459. Springer, New York, NY, USA.
12. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, D. L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering,* 22(6), 1996.