An Architecture-Based Approach to Self-Adaptive Software

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

SELF-ADAPTIVE SOFTWARE REQUIRES HIGH DEPENDABILITY, ROBUSTNESS, ADAPTABILITY, AND AVAILABILITY. THIS ARTICLE DESCRIBES AN INFRASTRUCTURE SUPPORTING TWO SIMULTANEOUS PROCESSES IN SELF-ADAPTIVE SOFTWARE: SYSTEM EVOLUTION, THE CONSISTENT APPLICATION OF CHANGE OVER TIME, AND SYSTEM ADAPTATION, THE CYCLE OF DETECTING CHANGING CIRCUMSTANCES AND PLANNING AND DEPLOYING RESPONSIVE MODIFICATIONS.

CONSIDER THE FOLLOWING SCENARIO. A fleet of unmanned air vehicles undertakes a mission to disable an enemy airfield. Pre-mission intelligence indicates that the airfield is not defended, and mission planning proceeds accordingly. While the UAVs are en route to the target, new intelligence indicates that a mobile surface-to-air missile launcher now guards the airfield. 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. During the flight, specialized algorithms for detecting and recognizing SAM launchers automatically upload and are integrated into the SAM-suppression unit's software.

In this scenario, new software components are dynamically inserted into fielded, heterogeneous systems without requiring system restart, or indeed, any downtime. Mission replanning relies on analyses that include feedback from current performance. Furthermore, such replanning can take place autonomously, can involve multiple, distributed, cooperating planners, and where major changes are demanded and require human approval or guidance, can cooperate with mission analysts. Throughout, system integrity requires the assurance of consistency, correctness, and coordination of changes.

Other applications for fleets of UAVs might include environment and land-use monitoring, freeway-traffic management, fire fighting, 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 software architects can simply adapt the platform to the application at hand, and better yet, if the platform itself adapts 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, provided someone could change the software quickly enough and with sufficient assurance that the new system would perform as intended.

Software engineering aims for the systematic, principled design and deployment of applications that fulfill software's original promise—applications that retain full plasticity throughout their lifecycle and that are as easy to modify in the field as they are on the drawing board. Software engineers have pursued many techniques for achieving this goal: 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.

Self-adaptive software will provide the key. Many disciplines will contribute to its progress, but wholesale advances require a sys-

1094-7167/99/$10.00 © 1999 IEEE

tems perspective based on a broadly inclusive adaptation methodology that spans 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. This article examines the fundamental role of software architecture in self-adaptive systems and outlines technologies we have considered for supporting the methodology.

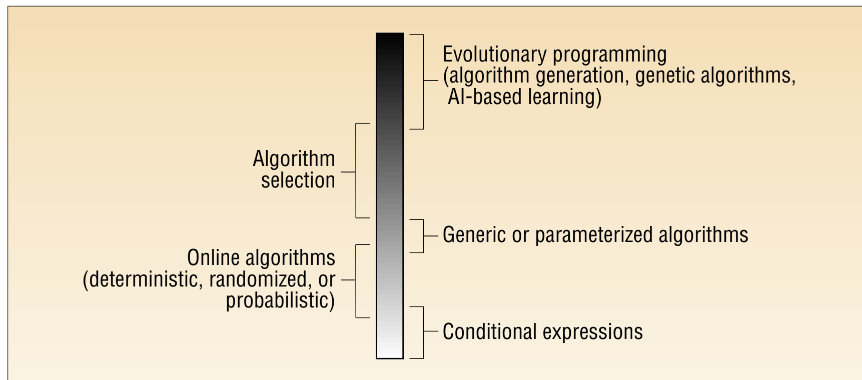


Figure 1. A spectrum of self-adaptability. Generally, approaches near the bottom select among predetermined alternatives, support localized change, and lack separation of concerns. Approaches near the top support unprecedented changes and provide a clearer separation of software-adaptation concerns.

What is self-adaptive software?

Self-adaptive software 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.

Application developers must answer several questions when developing a self-adaptive software system:

- Under what conditions does the system undergo adaptation? A system might, for example, modify itself 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 might 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 can 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 can be used, from continuous, precise, recent observations to sampled, approximate, historical observations.

What conditions? A fleet of UAVs might undergo adaptation under a variety of conditions. Mission replanning is a prime example because automated or human mission planners redirect the fleet in response to the changing battlefield. A mechanical failure of a UAV's generator might force the UAV to rely solely on battery power for its electronics, communications, and sensors. This in turn would require substantial adaptation to ensure sufficient electrical power for the mission's duration. A change in force composition (such as the loss of a fleet member to equipment failure) or the detection of an unanticipated threat might force rapid and substantial adaptation.

Open- or closed-adapted? A closed-adaptive UAV adapts in isolation, uninfluenced by the adaptations and behaviors of other fleet members. It has only a limited number of adaptive behaviors onboard, and no new behaviors can be introduced at runtime. Such a UAV might be capable of a limited number of evasive maneuvers in response to threats, for example, and its repertoire of evasions cannot be modified or expanded in flight. Conversely, an open-adaptive UAV accepts behaviors introduced from the outside, so an evasive maneuver known to one fleet member can be communicated to others while in flight.

Type of autonomy? Each UAV can be autonomous to a greater or lesser degree. For example, a UAV coping with an inflight subsystem failure might require that a human-in-the-loop direct, or at least approve, an adaptation. A sophisticated UAV with more onboard computing power might be highly autonomous, interacting with humans infrequently, if at all, over the course of its mission.

Frequencies? Adaptation is not without its cost, and even a useful or desirable adaptation might require more resources than the UAV can afford. For example, the UAV might be forced to permanently discard applications or system support for the sake of additional memory to accommodate an adaptation, or the adaptation might cut off future avenues of change. Implementing the adaptation might require processor cycles better used for other, more pressing concerns, or the adaptation, though desirable, might degrade the UAV's performance in other respects.

Cost-effectiveness? Adaptation frequency also matters. A UAV might be opportunistic, considering and implementing adaptations whenever it has spare processor cycles or additional communications bandwidth available. It might also adapt continuously, allocating an ongoing fixed percentage of its computing and communication resources to the adaptation process. Alternatively, adaptation might only be on demand as warranted by the UAV's condition and environmental state.

Information type and accuracy? Finally, the UAV might collect information from numerous sources on which to base its adaptation decisions. Information sources include

- real-time readings from internal sensors for monitoring subsystem health and status (such as battery voltage or fuel levels),
- telemetry from external sensors such as radar and magnetometers,
- sampled observations such as processor

load or radio signal strength over minutes, or historical data such as the movements of threat forces over hours.

Figure 1 illustrates the broad spectrum of self-adaptability. At one extreme, conditional expressions are a form of self-adaptation; the program evaluates an expression and alters its behavior based on the outcome. Although simplistic, conditional expressions are a common mechanism for implementing adaptive behavior. For example, a just-in-time compiler might invoke aggressive code-optimization techniques if a function is called frequently.

Online algorithms operate under the assumption that future events (inputs) are uncertain. Hence, they will occasionally perform an expensive operation to more efficiently respond to future operations.[1] Online algorithms are adaptive in that they leverage knowledge about the problem and the input domain to improve efficiency. A memory-cache-paging algorithm, for example, leverages the spatial and temporal locality of memory references in determining which cached page to evict when making room for a new page.

Generic and parameterized algorithms provide behaviors that are parameterized, usually through type instantiation or external inputs. Generic or polymorphic algorithms adapt by conforming to different data types. The C++ Standard Template Library, for example, provides generic iterator classes used to traverse a variety of data structures.

Algorithm selection uses properties of the operating environment to choose the most effective algorithm among a fixed set of available algorithms. Thus, a system that uses algorithm selection adapts to changes in its operating environment by switching among a set of algorithms. The Self dynamic optimizing compiler, for example, uses program-profiling data collected during program execution to select different code-optimization algorithms.[2]

At the other extreme, evolutionary programming and machine-learning techniques are adaptive in that they use properties of the operating environment and knowledge gained from previous attempts to generate new algorithms.[3]

Generally, approaches near the spectrum's bottom intertwine concerns regarding software adaptation and application-specific behavior. For example, a conditional expression combines the adaptation's specification with the application's specification. Consequently, understanding, analyzing, and modifying the two independently is arduous. Approaches near the top more clearly separate software-adaptation concerns and application-specific functionality. For example, algorithm generation separates the adaptation's specification from the produced algorithm. Separating the concerns of software adaptation from software function facilitates their independent analysis and evolution.

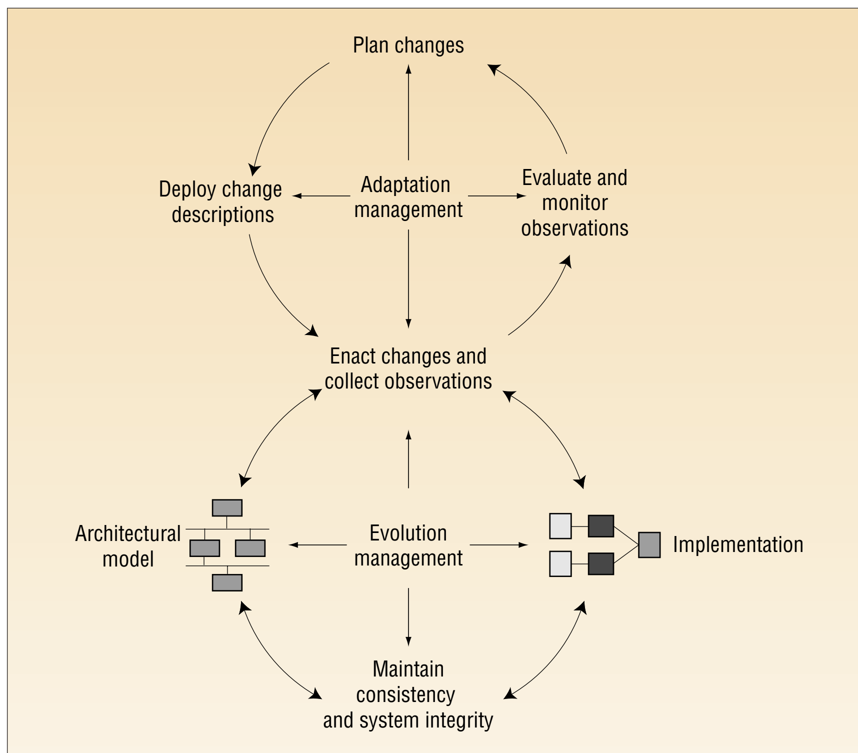


Figure 2. High-level processes in a comprehensive, general-purpose approach to self-adaptive software systems.

Software adaptation in-the-large

While technical advances in narrow areas of adaptation technology provide some benefit, the greatest benefit will accrue by developing a comprehensive *adaptation methodology* that spans adaptation-in-the-small to adaptation-in-the-large, and then develops the technology that supports the entire range of adaptations. Figure 2 illustrates just such a methodology that we are investigating.

The upper half of the diagram, labeled "adaptation management," describes the lifecycle of adaptive software systems. The lifecycle can have humans in the loop or be fully autonomous. "Evaluate and monitor observations" refers to all forms of evaluating and observing an application's execution, including, at a minimum, performance monitoring, safety inspections, and constraint verification. "Plan changes" refers to the task of accepting the evaluations, defining an appropriate adaptation, and constructing a blueprint for executing that adaptation. "Deploy change descriptions" is the coordinated conveyance of change descriptions, components, and possibly new observers or evaluators to the implementation platform in the field. Conversely, deployment might 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 this approach. Although 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 might outweigh those associated with shutting down and restarting a system.

Software adaptation is a complex process and is further complicated by change drivers ranging from purposeful adjustments in

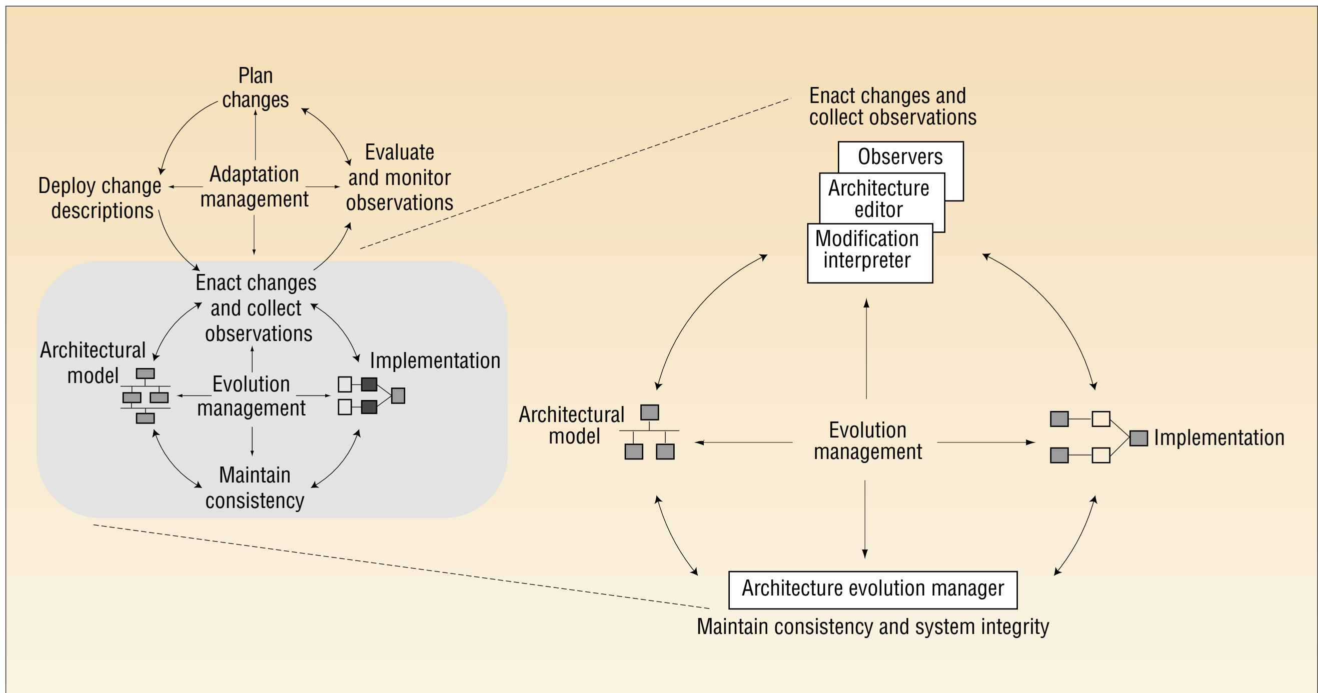


Figure 3. A high-level architecture diagram for the ArchStudio tool suite.

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 to support a full range of adaptation processes. The infrastructure relies on

- software agents that automate tasks within the process,
- explicit representations of software components, their interdependencies, and their environmental assumptions,
- explicit representations of the environments in the field where software is deployed, and
- 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, labeled "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 application's implementation, while ensuring that the model and the implementation are consistent with one another. Monitoring and evaluation services observe the application and its operating environment and feed information back to the diagram's upper half.

Software architectures view systems as networks of concurrent components bound together by connectors.[4] 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. Components are responsible for implementing application behavior and maintaining 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 lets a system's computation and communication relationships evolve independently of one another, including rearranging and replacing the components and connectors of an application while the application executes—a necessary, but insufficient, mechanism for self-adaptive software.

Evolution management

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. For example, an ill-considered change—such as the accidental removal of a critical navigation component—can compromise a UAV's safety, reliability, and correctness properties. Consequently, facilities for guiding and verifying modifications are an integral part of our architecture-centric approach. Figure 3 details our approach to evolution management, the process by which change is applied and controlled.[5] A variety of tools and adaptation mechanisms evolve an application by inspecting and changing its architectural model. Changes can include the addition, removal, or replacement of components and connectors, modifications to the configuration or parameters of components and connectors, and alterations in the component/connector network's topology. As we show next, our approach maintains system consistency and integrity by examining each change and vetoing any changes that render the system inconsistent or unsafe.

Dynamic software architectures. Supporting a broad class of adaptive changes at the architectural level requires that we not only change components on the fly but also their interconnections. However, simultaneously changing components, connectors, and top-

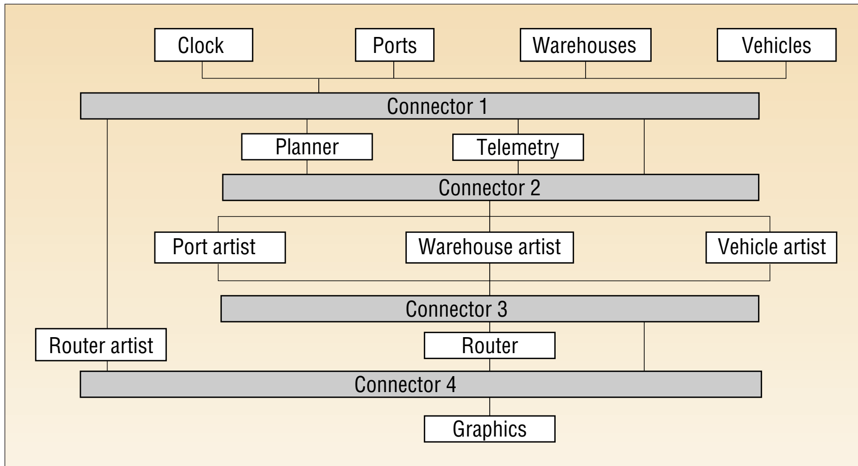


Figure 4. A C2-style architecture for a simple cargo-routing logistics system. Ports, vehicles, and warehouses are components that store application state. The telemetry component tracks en route cargo shipments. The port artist, vehicle artist, warehouse artist, and router artist components graphically depict the state of their respective counterparts. The planner component uses simple heuristics to suggest cargo routes, and the router component handles routing requests initiated by the end user. The graphics component renders the drawing notifications sent from the artists on the end-user's display.

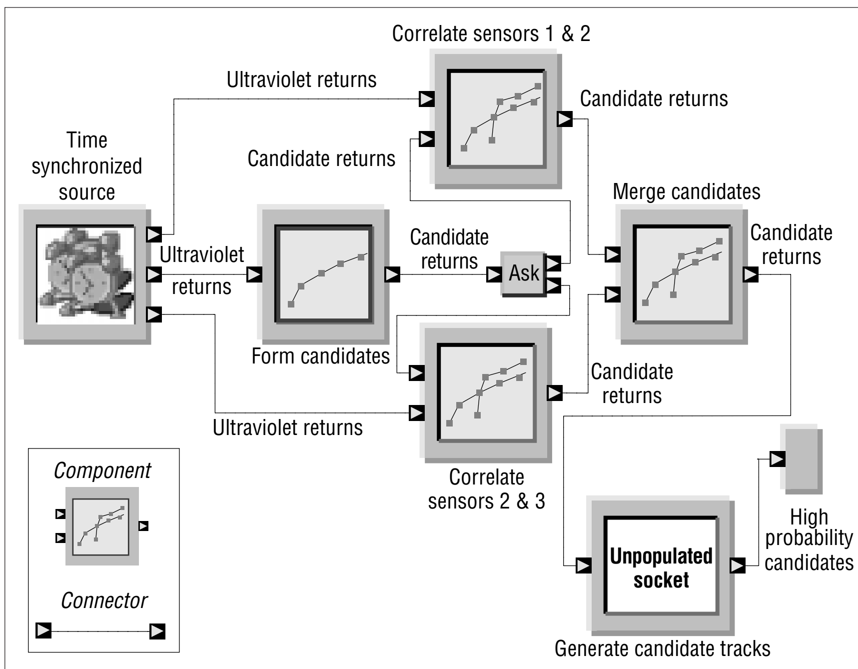


Figure 5. A portion of a Weaves architecture for a stereo-tracking system.

ology in a reliable manner requires distinctive mechanisms and architectural formalisms. Many systems are dynamic to some limited degree but few embrace dynamic change as a fundamental consideration.

There are two distinct approaches to dynamism at the architectural level: C2[6] and Weaves.[7] They have many features in common:

- both distinguish between components and connectors,
- neither places restrictions on the granularity of the components or their implementation language,
- both require that all communication between components occur by exchanging asynchronous messages (C2) or objects (Weaves), and
- components can encapsulate functionality of arbitrary complexity and exploit multiple threads of control.

However, C2 and Weaves take different approaches to system composition. C2 composes systems as a hierarchy of concurrent components bound together by connectors—message-routing devices—such that a component within the hierarchy can only be aware of components "above" it and is completely unaware of components residing at the same level or "beneath" it. Figure 4 shows an example C2-style architecture for a simple cargo-routing logistics system. A component explicitly utilizes the services of components above it by sending a request message. Communication with components below occurs implicitly; whenever a component changes its internal state, it announces the change by emitting a notification message, which describes the state change, to the connector below it. Connectors broadcast notification messages to every component and connector connected on its bottom side. Thus, notification messages provide an implicit invocation mechanism, allowing several components to react to a single component's state change. For example, the "Telemetry" component in Figure 4 is only aware of the "Clock," "Ports," "Vehicles," and "Warehouses" components. Furthermore, the C2-style components cannot assume that they will execute in the same address space as other components or share a common thread of control.

In contrast, Weaves is a dynamic, object-flow-centric architecture designed for applications characterized by continuous or intermittent voluminous data flows and real-time deadlines. Components in Weaves consume objects as inputs and produce objects as outputs ("object" is intended in the sense of C++, Smalltalk, or Java). Figure 5 depicts an example Weaves architecture for a portion of a stereo tracker. Weaves embraces a set of architectural principles known as the laws of blind communication:

- no component in a network knows the sources of its input objects or the destinations of its output objects;
- no network component knows the semantics of the connectors that delivered its input objects or transmitted its output objects; and
- no network component knows the loss of a connection.

These laws ensure that no component knows its location in the network, that every component is independent of the semantics of the connectors to which it is attached, and that any Weaves architecture can be edited, rewired, expanded, or contracted on the fly. Furthermore, Weaves permits connectors to be composed of other connectors and components, allowing connectors to be specially

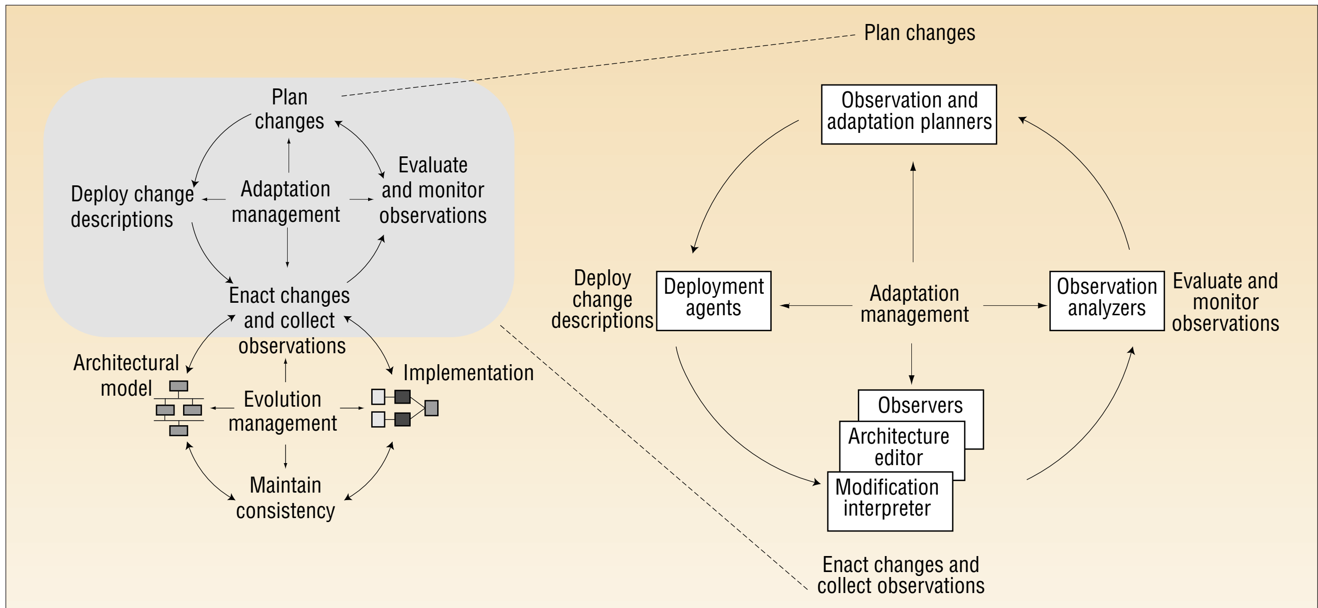


Figure 6. A high-level architecture diagram for adaptation evolution.

adapted to the characteristics of their operating environment with corresponding performance gains.

Several characteristics of C2 and Weaves facilitate runtime change. Because components communicate asynchronously, C2 and Weaves avoid several subtle complexities inherent in supporting runtime change in applications that utilize synchronous communication. While this restriction occasionally makes it more difficult to implement particular component interactions, because a component must continue to respond to service requests from other components while awaiting responses it has made of others, our experience demonstrates that its benefits for runtime change outweigh its costs.

The independence between hierarchical layers in a C2-style architecture further reduces component dependencies: a C2 component is unaware of components below itself, so it is oblivious to runtime changes that involve these components.. Conversely, a component can only be affected by runtime changes involving components strictly above itself. Because C2 components cannot assume that they will execute in the same address space as other components, complex component dependencies resulting from the use of pointer variables and global variables are avoided. Similarly, because components do not share a common thread of control, control dependencies are avoided. Taken together, C2's style rules ensure that each component is almost completely ignorant of the placement, function, and implementation of its fellow components.

Consequently, at runtime, C2 can add, delete, or rearrange components with remarkable ease and alacrity. In contrast, while Weaves supports like forms of component manipulation, it emphasizes the dynamic distribution, modification, and rearrangement of connectors. This lets developers optimize intercomponent communication while a Weaves architecture is executing, including wholesale movement of a subarchitecture 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 focuses on high-performance, flexible connectors. One research issue 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 architectures.

Maintaining consistency and system integrity. Ongoing adaptation continuously threatens system safety, reliability, and correctness. Therefore, facilities for guiding and checking modifications are an integral part of our adaptation infrastructure. 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 architectural 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.

To guard against untoward change, we propose an *architecture evolution manager (AEM)* that 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 can 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 this mapping to carry out modifications by mapping model

components and connectors into implementation artifacts and translating change operations into implementation actions.

Enacting changes. There are many possible sources of architectural change, including the application itself, external tools, and replanning agents. Software architects can use a visual, interactive, *architecture editor* to construct architectures and describe modifications. A variety of analysis tools can 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. The *modification interpreter* acts as a second, companion tool to interpret change scripts written in a change-description language to primitive actions supported by the AEM.

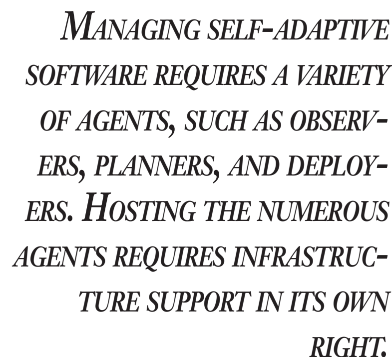
Adaptation management

A self-adaptive system observes its own behavior and analyzes these observations to determine appropriate adaptations. A 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, 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 coordinating physically distributed, logically decentralized adaptation tasks. For example, complex interdependencies might exist among changes such that the incorporation of one change could require the inclusion of several others for the change to work correctly in its environment. A standard locale helps ensure that such information is at hand. Additionally, an adaptation might require coordination among multiple sites when 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* for evaluating the behavior of the self-adaptive application and monitoring its operating environment, *planners* that utilize the observations to plan adaptive responses, and *deployers* to enact the responses within the application.

Hosting the numerous agents and supporting the various activities of adaptation management that result requires infrastructure support in its own right in the form of registries. Registries at each application site 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 provides 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. Interregistry communication takes 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.[8,9]



Collecting observations. Self-adaptive software requires large numbers and varieties of observations and measurements, ranging from event-generation within the application implementation to animations suitable for human observers. Furthermore, adjusting the number, extent, and detail of the observations and measurements must be possible as the application executes and evolves so as to reduce measurement overhead and avoid wasting communication bandwidth on unnecessary observations.

At a minimum, we 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.[10]

Detecting and noting single events is not enough because the occurrence of a pattern of events distributed in both time and place will trigger many adaptive strategies. 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*.[11] 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 Web-based technologies—HTTP and HTML—to provide a powerful and efficient infrastructure for remote observation of distributed applications.[12] Joist embeds a small Web server in the application's runtime environment, which then monitors the application and gathers information. This information is identified through a special URL namespace, and it is presented in HTML pages that the Joist server generates and communicates to any standard Web browser via HTTP.

New techniques must emerge for reducing the monitoring overhead. Weaves employs statistical monitoring techniques that lets observers trade accuracy in favor of reduced overhead. Using this approach, we can reduce the invasive effect of instrumentation on a running application to below the noise threshold while still obtaining useful

information. Furthermore, the instrumentation can stay permanently embedded within the application so that an observer can selectively measure only behaviors of interest without damaging the application. Application developers can use this technique in a variety of ways, including performance analysis and real-time animation of the behavior of running systems.

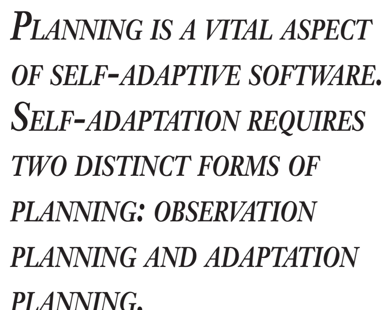
Evaluating and monitoring. Adaptive demands can arise from inconsistencies or suboptimal behavior within the system. In particular, inconsistencies can occur when some architectural element (ranging from a single component or connector to a subsystem, or the entire architecture) behaves in a manner inconsistent 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 requires a hybrid approach that combines both static and dynamic analysis. 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 an application's possible configurations where architectural changes are regarded as graph-rewrite operations. Analysis tools can determine if an invariant is preserved by all possible architectures or can return an example graph (architectural configuration) that violates the invariant.

Static analysis might be insufficient, in which case runtime checks are 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.

Planning changes. Planning is also a vital aspect of self-adaptive software. Self-adaptation requires two distinct forms of planning: 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. We can view this task 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. One possible approach relies on the use of predefined 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 might be found by selecting components for each socket from the set of eligible components. Using 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 preconditions and postconditions 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. We have already demonstrated this approach in another domain, the automatic generation of simulation scripts for tank training.[13]

Deploying change descriptions. Change agents propagate and move out among sites to carry out their tasks. Imagine a scenario in which a coordinated change is required at two separate sites. Agents responsible for each portion of the coordinated change dispatch 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 AEM, which translates the change transactions contained within the change descriptions into specific modifications of the system's implementation.

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 in support of self-adaptive software. ■

Acknowledgments

We thank David M. Hilbert and Jason E. Robbins for discussions that contributed to this work. Dennis Heimbinger and Alexander L. Wolf are sponsored by the Air Force Materiel Command, Rome Laboratory, and the Defense Advanced Research Projects Agency (DARPA) under contracts F30602-94-C-0253 and F30602-98-2-0163. Peyman Oreizy, Richard N. Taylor, Nenad Medvidovic, and David S. Rosenblum are sponsored by DARPA and the Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement

F30602-97-2-0021; by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant F49620-98-1-0061; and by the National Science Foundation under grant CCR-9701973. Alex Quilici was partially supported by DARPA contract N66001-96-C-8502. The US Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the Air Force Research Laboratory, the Air Force Office of Scientific Research, or the US Government.

References

1. S. Irani and A.R. Karlin, *On Online Computation: Approximation Algorithms for NP-Hard Problems*, Dorit Hochbaum, ed., PWS Publishing Company, Boston, 1996.

2. U. Holzle, *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*, PhD dissertation, Stanford Univ., Stanford, Calif., 1994.

3. W.M. Spears et al., "An Overview of Evolutionary Computation," *Proc. European Conf. Machine Learning*, Springer-Verlag, New York, 1993, pp. 442–459.

4. D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture," *Software Eng. Notes*, Vol. 17, No. 4, 1992, pp. 40–52.

5. P. Oreizy, N. Medvidovic, and R.N. Taylor, "Architecture-Based Runtime Software Evolution, *Proc. Int'l Conf. Software Eng.* (*ICSE '98*), 1998, pp. 117–186.

6. R.N. Taylor et al., "A Component- and Message-Based Architectural Style for GUI Software," *IEEE Trans. Software Eng.*, Vol. 22, No. 6, 1996, pp. 390–406.

7. M.M. Gorlick and R.R. Razouk, "Using Weaves for Software Construction and Analysis," *Proc. Int'l Conf. Software Eng. (ICSE '91)*, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 23–34**.**

8. R.S. Hall et al., "An Architecture for Post-Development Configuration Management in a Wide-Area Network, *Proc. 17th Int'l Conf. Distributed Computing Systems,* IEEE CS Press, 1997, pp. 269–278.

9. R. Hall, D. Heimbigner, and A.L. Wolf, "A Cooperative Approach to Support Software Deployment Using the Software Dock," *Proc. Int'l Conf. Software Eng.*, (*ICSE '99*), IEEE CS Press, 1999.

10. D.S. Rosenblum, "A Practical Approach to Programming with Assertions," *IEEE Trans. Software Eng.*, Vol. 21, No. 1, 1995, pp. 19–31.

11. D.M. Hilbert and D.F. Redmiles, "An Approach to Large-Scale Collection of Application Usage Data over the Internet," *Proc. Int'l Conf. Software Eng. (ICSE '98)*, IEEE CS Press, 1998, pp. 136–145.

12. M.M. Gorlick, "Distributed Debugging and Monitoring on $5 a Day," *Proc. California Software Symp.*, Univ. of California, Irvine, Calif., 1997, pp. 31–39.

13. D. Pautler, S. Woods, and A. Quilici, "Exploiting Domain-Specific Knowledge to Refine Simulation Specifications," *Proc. 12th Conf. Automated Software Eng.*, IEEE CS Press, 1997.

Peyman Oreizy is a PhD candidate at the University of California, Irvine. His research interests include software evolution, customization, and architectures. He received his BS and MS in computer science from UCI. He is a member of the IEEE Computer Society and the ACM. Contact him at UC Irvine, Information and Computer Science Bldg., Rm. 444, Irvine, CA 92697-3435; peymano@ics.uci.edu; www.ics.uci.edu/~peymano/.

Michael M. Gorlick is a research scientist at the Aerospace Corporation. His research interests include software architectures, large-scale system and software engineering, and wearable computers. He holds an MSc in computer science from the University of British Columbia, Canada. Contact him at the Aerospace Corp., Mail Station M1-102, PO Box 92957, Los Angeles, CA 90009; gorlick@aero.org.

Richard N. Taylor is a professor with the Department of Information and Computer Science, UCI, and is also the director of the Irvine Research Unit in Software (IRUS), an alliance between California industry and the university. His research interests are centered on software architectures, hypermedia and Web protocols, and workflow and process technologies. He received his PhD in computer science from the University of Colorado, Boulder. He is an ACM Fellow. Contact him at the Dept. of Information and Computer Science, UCI, Irvine, CA 92697-3425; taylor@ics.uci.edu; www.ics.uci.edu/~taylor.

Dennis M. Heimbigner is a research associate professor at the University of Colorado, Boulder. His research interests are in configuration management, paradigms for the engineering of distributed software, distributed computing models, software workflow, and federated databases. He received a BS in mathematics from the California Institute of Technology, and an MS and PhD in computer science from USC. He is a member of the IEEE and ACM, and is a principal investigator in the DARPA EDCS program. Contact him at the Dept. of Computer Science, Campus Box 430, Univ. of Colorado, Boulder, CO 80309-0430; dennis@cs.colorado.edu; www.cs.colorado.edu/~dennis.

Gregory Johnson is a member of the technical staff of Concept Shopping Inc. His interests include software-understanding tools and algorithm visualization. He received his doctorate in computer science from the University of Wisconsin-Madison. He is a member of the IEEE and the ACM. Contact him at 777 Silver Spur Rd., Ste. 229, Rolling Hills Estates, CA 90274; gregfjohnson@earthlink.net.

Nenad Medvidovic is an assistant professor in the Computer Science Department at the University of Southern California. He received his PhD from the Department of Information and Computer Science at UCI. He also received an MS in information and computer science from UCI, and a BS in computer science from Arizona State University. His research interests include software engineering, architectures, evolution, and reuse. Contact him at the Computer Science Dept., Henry Salvatori Computer Center, Rm. 338, Univ. of Southern California, Los Angeles, CA 90089-0781; neno@usc.edu; http://sunset.usc.edu/~neno.

Alex Quilici is an associate professor of electrical engineering at the University of Hawaii, Manoa. His research interests lie in applying AI techniques to software engineering, in particular in the areas of automated program understanding and the automated synthesis of component-based systems. He received his PhD in computer science from UCLA. He is a member of AAAI, the IEEE Computer Society, and the Cognitive Science Society. Contact him at the Univ. of Hawaii, Manoa, Dept. of Electrical Eng., 2540 Dole St., Holmes Bldg., Rm. 483, Honolulu, HI 96822; alex@wiliki.eng.hawaii.edu; www-ee.eng.hawaii.edu/~alex.

David S. Rosenblum is an associate professor in the Department of Information and Computer Science at UCI. His current research is centered on problems in the design and validation of large-scale distributed component-based software systems. He received a PhD from Stanford University. He is a senior member of the IEEE and a member of the ACM. Contact him at the Dept. of Information and Computer Science, UCI, Irvine, CA 92697-3425; dsr@ics.uci.edu; www.ics.uci.edu/~dsr.

Alexander L. Wolf is an associate professor in the Department of Computer Science at the University of Colorado at Boulder. His research interests are directed toward the discovery of principles and development of technologies for supporting the engineering of large, complex software systems. He received a BA in geology and computer science from Queens College, City University of New York, and his MS and PhD in computer science from the University of Massachusetts, Amherst. He is a member of the ACM and the IEEE Computer Society, and is Vice Chair of the ACM Special Interest Group in Software Engineering. Contact him at the Dept. of Computer Science, Campus Box 430, Univ. of Colorado, Boulder, CO 80309-0430; alw@cs.colorado.edu; www. cs.colorado.edu/users/alw.