In *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture 2009 & European Conference on Software Architecture 2009,* Sept. 14-17, 2009, Cambridge, UK.

Evolution Styles: Foundations and Tool Support for Software Architecture Evolution

David Garlan, Jeffrey M. Barnes, Bradley Schmerl, Orieta Celiku
Carnegie Mellon University, Pittsburgh, PA 15213
{garlan,jmbarnes,schmerl,orietac}@cs.cmu.edu

Abstract

As new market opportunities, technologies, platforms, and frameworks become available, systems require large-scale and systematic architectural restructuring to accommodate them. Today's architects have few tools and techniques to help them plan this architecture evolution. In particular, they have little assistance in planning alternative evolution paths, trading off various aspects of the different paths, or knowing best practices for particular domains. In this paper we describe an approach for assisting architects in developing and reasoning about architectural evolution paths. The key insight of our approach is that, architecturally, many system evolutions follow certain common patterns – or evolution styles. We define what we mean by an evolution style, and show how it can be used to provide automated assistance for expressing architectural evolution, and for reasoning about both the correctness and quality of evolution paths.

1. Introduction

Architecture evolution is a central feature of virtually all software systems. As new market opportunities, technologies, platforms, and frameworks become available, systems must change their organizational structures to accommodate them, in many cases requiring large-scale and systematic restructuring. In most cases such changes cannot be made overnight, and hence the architect must develop an evolution plan to change the architecture (and implementation) of a system through a series of phased releases, eventually leading to a new target system.

Unfortunately, architects have few tools to help them plan and execute such evolutionary paths. While considerable research has gone into software maintenance and evolution, dating from the beginning of software engineering, there has been relatively little work focusing specifically on foundations and tools to support architecture evolution. Architecture evolution is an essential complement to software evolution because it permits planning and system restructuring at a high level of abstraction where quality and business tradeoffs can be understood and analyzed.

In particular, architects have almost no assistance in reasoning about questions such as: How should we stage the evolution to achieve business goals in the presence of limited development resources? How can we assure ourselves that intermediate releases do not break existing functionality? How can we reduce risk in incorporating new technologies and infrastructure required by the target architecture? How can we make principled tradeoffs between time and development effort? What kinds of changes can be made independently, and which require coordinated system-wide modifications? How can we represent and communicate an evolution plan within an organization?

Such questions require new foundations that permit architects to reason about and plan large-scale system-wide changes at an architectural level of abstraction. In this paper, we describe an approach that allows one to precisely express and reason about architecture evolution. We support the expression and checking of correctness conditions (e.g., to guarantee that a proposed path satisfies certain sequencing constraints), that intermediate states of a system evolution do not introduce anomalous behavior, and that the proposed path will lead to a system with desired architectural properties. Moreover, our approach allows an architect not only to reason about "correct" evolution, but also to make tradeoffs to achieve business goals, such as minimizing the time to reach the target architecture and the costs involved in doing so. Finally, we describe a tool to automate these analyses.

As we will see, the key insight behind our approach is that at an architectural level of abstraction many system evolutions follow certain common patterns, dictated by the style of architecture that their origin and target architectures conform to. By taking advantage of regularity in the space of common architectural evolutions we can provide automated assistance for capturing and reusing knowledge about architectural evolution. Specifically, we refer to collections of related paths as *evolution styles*. Evolution styles can be de-

fined, reasoned about, analyzed, applied to the evolution of specific systems, and supported by tools. By capturing such styles we not only raise the level of abstraction for representing specific evolution paths, but also provide the opportunity for reuse, path analysis, decision automation, tradeoff analysis, and formal guarantees of correctness.

This paper is organized as follows: in Section 2, we discuss existing work in architecture evolution and related areas. Section 3 describes our general approach to architecture evolution, and Section 4 presents an example to illustrate these notions. In Section 5, we describe with more formality the foundations of our model of architecture evolution. Finally, we discuss the tool that we have developed to support evolution in Section 6, and in Section 7 we conclude with a short discussion.

2. Related Work

Today's approaches to addressing problems of architecture evolution fall into four categories. The first is support for *software* evolution. Since the early days of software engineering there has been concern for the maintainability of software, leading to concepts such as criteria for code modularization [28], indications of maintainability such as coupling and cohesion [2], [39], code refactoring [25], and many others [16]. These techniques, which focus on the code structures of a system, have led to numerous advances, such as language support for modularization and encapsulation, analysis of module compatibility and substitutability [6], and design patterns that support maintainability [12].

While such advances have been critical to the progress of software engineering, they generally do not treat large-scale reorganization based on architectural abstractions. Working primarily in the domain of code units, they do not capture the essential, high-level, runtime structures that are necessary to reason about the architecture of a complex software system. Also, the techniques are typically general-purpose, focusing on general properties of modularity such as coupling and cohesion. In contrast, our work focuses on the reuse of specifications and analyses for domain-specific evolution at an architectural level of abstraction.

The second related area of research and development is tool support for versioning and project planning. Version control systems such as CVS [4] allow different versions of artifacts to be compared and reviewed. In these tools, the primary managed artifact is source code rather than architectural models. Consequently these tools do not support comparison or reasoning about different versions of the architecture. More recent software architecture research has investigated architectural versioning [1], [18], but these tools and techniques do not provide any reasoning framework other than comparison. In particular, they are silent with respect to what might constitute a correct or optimal evolution path.

In the domain of project planning, traditional project management approaches and software development planning approaches such as COCOMO [5] provide ways to plan and analyze software development. Unfortunately, because they focus primarily on the end state of a maintenance or development effort, they do not provide ways to directly plan and reason about sequences of developments, nor do they have any way to state and enforce constraints on a system's architectural structure. Advice on how to organize architecture evolution steps into waves and plateaus is given in [11]. The advice is pragmatic in nature, suggesting that introducing major infrastructure changes (waves) should be followed by periods of relative stability so that new infrastructure changes can be properly adjusted to (plateaus).

The third related area is formal approaches to architecture transformation. A number of researchers have proposed formal models that can capture structural and behavioral transformation [17], [35], [40]. For example, Wermelinger uses category theory to describe how transformations can occur in software architecture [40]. His approach separates computations of a system from its configuration, allowing the introduction of a "dynamic configuration step" that produces a derivation from one architecture to another. Architecture in this sense is defined by the space of all possible configurations that can result from a certain starting configuration, and then applying a sequence of transformations. Grunske [17] shows how to map architectural specifications to hypergraphs and uses these to define architectural refactorings that can be applied automatically. These refactorings are shown to preserve architectural behavior. Spitznagel [34] focuses on the transformation of architectural connectors as a way to augment the communication paths between components.

There has been some research into abstracting code to an architectural level, transforming the architecture, and then regenerating code [10], [19]. These approaches focus on methods for retrieving architectures from legacy systems and reflecting (single-step) architectural changes back into the system, and do not deal with how to specify and reason about entire architectural transformations or a sequence of architectural releases. In essence, they provide a complementary bridge between code-related software evolution and refactoring, and the area of architectural transformation.

While such formal approaches lay a foundation for architecture evolution operators, they differ from our

approach in that they are not amenable to specialization for specific classes of transformations and systematic reuse. Moreover, while they can provide some support for characterizing forms of evolution correctness, they do not address issues of evolution quality, nor do they address planning evolutions in multiple large steps.

Recently Tamzalit and others have begun to investigate recurring patterns of architecture evolution, primarily with respect to component-based architectures [26], [36], [37]. They use the term *evolution style* to denote a pattern for updating a component-based architecture. They provide a formal approach based on a three-tiered conceptual framework. Like us, they attempt to capture recurring and reusable patterns of architecture evolution. However, they do not explicitly characterize or reason about the space of architecture paths, or apply utility-oriented evaluation to selecting appropriate paths.

The fourth related area is tradeoff analysis for architectural evolution. The work of Kazman et al. [22] applies architectural analysis and tradeoff techniques to incrementally improve architectures through the application of *tactics*. Their approach, however, has not been used for planning architecture evolution, which looks at large-scale, system-wide evolution over a long period of time. The work in [27] proposes to use techniques from options theory to determine investments in introducing flexibility into a system. This work is similar to ours in that it provides some basis for analyzing architectural quality, but differs in that it does not consider correct architectural transformations or reuse through evolution styles.

One important subset of this work, which does focus on architectural evolution for specific classes of systems, addresses architecture evolution in the context of a specific style, such Darwin [23] and C2 [38]. Like the work proposed here, these approaches can take advantage of domain-specific classes of systems, and thereby achieve analytic leverage, as well as tool support for evolution. However, these approaches are limited to a particular architectural style.

3. Approach

The basis for our approach to architecture evolution centers on the concept of *evolution paths*:

- Evolution paths can be represented and analyzed as first-class entities;
- Classes of domain-specific evolution paths can be formally specified, thereby supporting reuse, correctness checking, and quality analysis;
- Tradeoff analyses can be performed over alternative evolution paths to optimize expected value under uncertainty; and
- Tools can support the description, analysis, tracking, and modification of architecture evolution for a particular system through a widely used integrated development environment framework.

The principal idea behind our approach is the concept of an *evolution style*. An evolution style defines a family of domain-specific architecture evolution paths that share common properties and satisfy a common set of constraints. The key insight is that by capturing evolution paths for specialized families we can define constraints that each path in that family must obey, thereby providing guidance (based on past experience) and correctness criteria (based on formal constraints) for an architect developing a particular evolution plan in that family. Moreover, we can support reasoning about the extent to which a specific path satisfies the quality/cost objectives in a particular business context.

To illustrate what we mean by an evolution style, consider the following typical scenarios of evolving an architecture from an ad hoc peer-to-peer assemblage of legacy subsystems to a hub-and-spoke architecture that leverages commercial middleware for coordinating the subsystems; from a traditional thin-client/mainframe system to a four-tiered web services architecture; from a web services architecture based on J2EE to a service-oriented architecture based on BEA's WebLogic product family; from a control system based on CAN-bus integration to one that supports a more reliable protocol (e.g., FlexRay [31]).

Each of these examples has the property that they refer to a class of evolutions addressing a recurring, domain-specific architectural evolution problem. (Indeed, such evolutions are the core concern of an important business segment represented by well-paid consultants who specialize in assisting companies with their specific evolution problems.) Each of them has identifiable starting and ending conditions (namely, that the initial and final system contain certain architectural structures). Each embodies certain constraints – for example, that no essential service should become unavailable during the evolution. Finally, although they share many commonalities, the specific details of how those evolutions should be carried out may well be influenced by concerns such as the time it takes to do the transformation, the available resources to carry it out, etc. We can take advantage of these characteristics of system evolution.

Summarized briefly, we can model an evolution style formally as a set of finite evolution paths, where each path defines a sequence of architectures in which

the first element in the path is the architecture of the current system, and the final element is a desired target architecture. Links between successive nodes in a path are associated with transitions that are composed using a set of evolution operators for that style. In this respect an evolution style is like a state machine for which an execution trace defines an evolution path.

The evolution style may further constrain the space of paths in its family by specifying path constraints. Path constraints embody things like ordering constraints or invariants that must hold for all nodes or all releases. We can then talk about whether a given path is *correct* with respect to an evolution style – meaning that the path is an element of the family circumscribed by that evolution style. To complete the picture, we will introduce the notion of an evaluation function that allows us to compare different paths with respect to quality metrics. Intuitively, an evaluation function determines the expected utility (in a probabilistic sense) of a given path with respect to business and management priorities, relative to a space of properties of interest (e.g., time, resources, risk, downtime, etc.) and in the presence of uncertainty.

4. Example

To illustrate the concepts and benefits of our approach, consider the following simple, but representative, scenario: Company C delivers a set of services using software and data that is spread out over a loose collection of relatively independent legacy IT subsystems that have accrued over time. Because of historical independent development and acquisition, different subsystems are based on different platforms. For example, one subsystem might be used to manage personnel, based on a PeopleSoft platform; another subsystem manages inventory using SAP; yet another manages accounts using Oracle Applications. In cases where delivered services need to access multiple subsystems, the IT division of C has created ad hoc, hand-coded bridging elements. For example, there may be connections between the personnel management and the accounts system to print paychecks, and between inventory and accounts to pay suppliers and bill vendors.

This form of system is common in today's IT world,

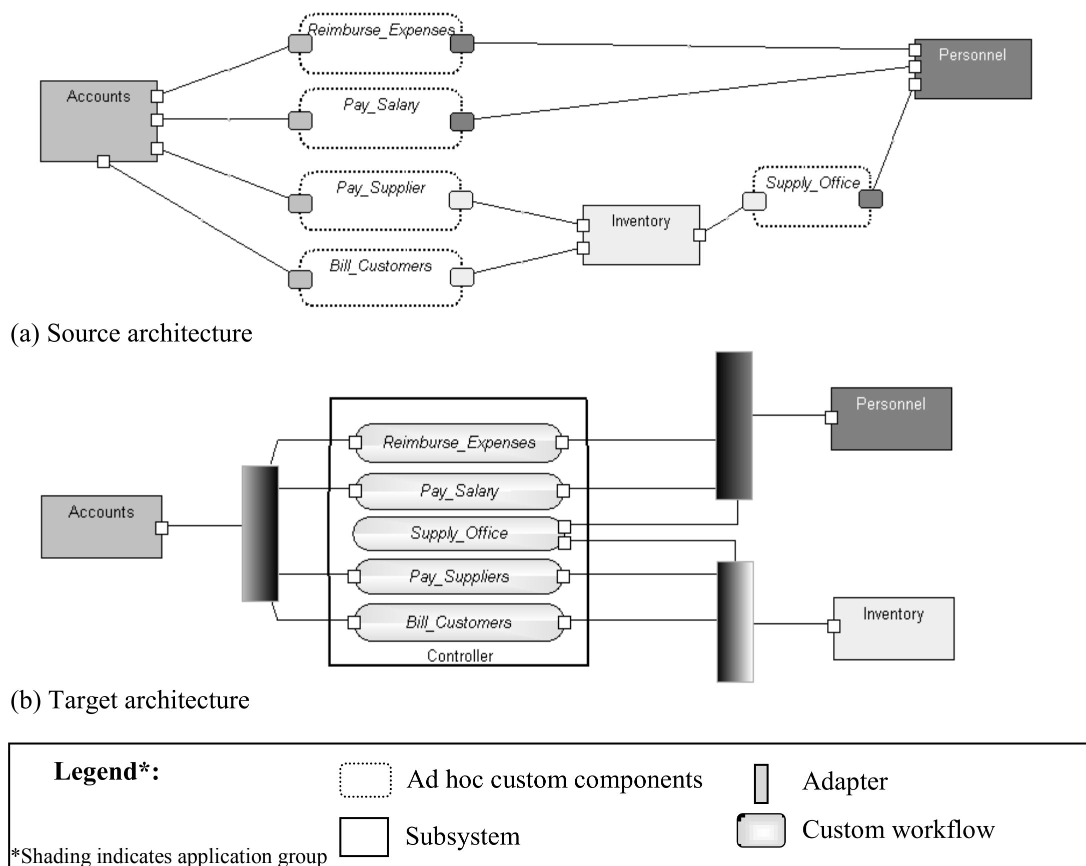

Figure 1. Examples of architectural instances.

and represents an example of an ad hoc peer-to-peer architecture. Each of the software systems has its own language (e.g., SAP has BAPI, Oracle Applications has its own SQL-like language, etc.), and these applications and the workflow between them are commonly cobbled together to suit the IT business as needed. Such a system is difficult to maintain and evolve, both because the integration code was not developed with future evolution in mind, and because new technological domains such as online services were not anticipated when the code was developed.

As the company evolves to meet business needs, it can no longer easily change the integrated functionality or add new integrated functionalities in a timely fashion. The company's IT department decides to evolve the system to more centralized and uniform control using an off-the-shelf integration/coordination technology – specifically, IBM's Message Queue Series Workflow. By using a unified language to specify integrated workflow, and adapters that map the workflow onto existing subsystems' languages and schema, the number of ad hoc connections between the subsystems is reduced, improving maintainability and extensibility.

Figure 1 illustrates a drastically simplified version of the initial and target architectures. The components *Accounts*, *Personnel*, and *Inventory* represent large subsystems that have been acquired over time. In the source architecture, these subsystems have been integrated as needed, in an ad hoc fashion, using custom components to implement the logic for each pairwise integration. These components are ad hoc in the sense that they are written using the specific interfaces of each of the subsystems with which they integrate. In the target architecture, these components have been replaced by business logic in a standard, common business language, and adapters have been introduced to map the specific interfaces of the subsystems to standard interfaces that can be used for the business logic. The advantages of the target architecture are obvious: the integrating business logic is easier to maintain and evolve because the language used is common regardless of the subsystems that are used, and the system is now more flexible to changes in the subsystem. For example, if the *Accounts* subsystem were replaced, only the adapter would need to change; in the source architecture, such a situation would involve rewriting the integrating business logic components that interacted with it.

Because of the system's complexity and importance to C's business operations, the chief architect at C needs to plan an evolution path to do this in a set of staged releases. Let us see how this might be accomplished using the concept of evolution styles.

The evolution style for this problem is one that is specialized to the problem of transitioning systems from an ad hoc peer-to-peer architecture to a hub-and-spoke architecture, in which the core functionality offered by subsystems remains unchanged, but the coordination of the parts is changed. Capitalizing on past experience in this area, the evolution style, which we will call PP2HS, would identify the essential characteristics of the initial and target architecture styles. It would also characterize the style of architectures for intermediate releases: in this case, a mixture of the initial and target structures, allowing both peer-to-peer connections as well as hub-and-spoke. Additionally, PP2HS would identify a set of structure- and behavior-changing operations. Examples include the introduction of the central hub infrastructure as a new kind of component (in the mixed intermediate style), addition of adapters to allow legacy subsystems to talk to the hub, and migration of bridging component functionality into the hub. Finally, PP2HS would specify a set of path constraints. These would capture the correctness conditions for a valid evolution path. Specifically they would express things like: in every release all existing functionality must continue to be available, before adapters are introduced the hub components must be incorporated into the system, when a coordinated service is transitioned to the hub, all subsystems that are used by that service must have adapters.

How would this be used by the chief architect at C? Using his tools for architecture evolution the architect would first select the appropriate evolution style (here, PP2HS). He would then start to define an evolution path. Likely the starting point for this would be the characterization of the initial and target architectures. Existing tools make it relatively easy to specify these using standard architecture modeling and visualization techniques. At this point the evolution tools would check that these two architectures satisfy the pre- and post-conditions required by PP2HS, perhaps noting situations in which the target architecture is missing certain required structures, or is otherwise malformed with respect to the target style.

The architect now starts filling in intermediate stages. Again using the tools, he applies a series of operators of PP2HS to the architecture to produce a first release – for example, by adding the hub and the adapter for one subsystem as an initial release. The tools would check that the release is well formed, and that the path satisfies the constraints of PP2HS, warning the architect when it identifies divergences. This process repeats until the architect has fully specified a set of releases and transitions to arrive at the target architecture.

Along the way, however, the architect also needs to make decisions about various tradeoffs, for example, reconciling available resources (e.g., programmers) with the effort and time needed to create each release.

To do this the architect uses one of several parameterized evaluation functions for this evolution style. The evaluation functions require the architect to select dimensions of concern, provide weighted utilities, and estimates of costs and durations (including uncertainties).With these annotations in hand the tools calculate for the architect costs and utility, allowing him to explore alternative scenarios. Over time, as the evolution proceeds, the architect will update the values, and perform recalculations, perhaps leading to revisions of the remaining releases on the path.

5. Evolution Styles

In the previous sections we outlined informally what we mean by *evolution style* and provided an example illustrating how particular aspects of the evolution style are useful in planning evolution in a particular domain. In this section, we describe the technical basis of our approach to evolution styles – specifically, how we represent architectures and evolution path constraints.

(a) Specifying Architectures

We specify architectures using the Acme architecture description language (ADL) [13]. As in most modern ADLs, including UML 2.0, an architecture is represented as a graph in which the nodes represent *components* and the edges represent *connectors* [9], [24], [29], [33]. Components correspond to the major run-time computational elements and data stores of a system, while connectors define the pathways of interaction between them. Interfaces of components are termed *ports.* Architectures may be defined hierarchically: elements may be elaborated as sub-architectures.

Augmenting architectural structure, we allow architecture elements (components, connectors, ports)[1] to be annotated with properties that provide more-detailed semantics. While the list of properties will vary from architecture to architecture, typically they are used to represent things like reliability (for components), protocols of interaction (for connectors), or signatures of required and provided services (for ports).

(b) Specifying Sets of Architectures

To represent *sets* of architectures we use the established notion of architectural styles as embodied in ADLs such as Acme. Specifically, an architectural style is defined by specifying a vocabulary of architectural structures as a set of component, connector, and port types, together with a set of constraints that determine how instances of those types can be composed into systems.[2] Constraints may also refer to properties of the elements. In Acme, constraints are specified in a first-order predicate logic, similar to UML's OCL, but augmented with architecture functions, such as retrieving the components connected to another one, returning the set of ports of a component, etc. (For constraint language details see [13].)

Referring back to the example of Section 4, there would likely be three relevant styles: the peer-to-peer style of the initial system, the hub-and-spoke style of the target system, and the combination style for intermediate releases. Component types in the hub-and-spoke style include things like the controller and various adapters. Connector types include the standard adapter-hub communication protocol, as well as the specialized connectors that link adapters to subsystems. Constraints specify that all service-delivering subsystems must be connected to the hub (possibly via an adapter).

(c) Specifying Evolution Path Properties

We allow nodes and transitions in an evolution path to be annotated with an extensible list of properties. These properties provide information so that constraints and analyses can be performed. An evolution style specifies the list of properties that may be set on the nodes and transitions. For example, an evolution style may stipulate that each node must specify whether it is intended to be a public release, or what the expected impact of the node on the market is; transitions might provide information about the expected amount of time the transition will take, how many developers are required, whether training will be needed, etc. Each path may require different values for the same properties on a node. For example, the expected time to take one step of an evolution may be different in a path where a previous step involved programmer training than in a path that hasn't yet involved training.

(d) Specifying and Using Path Constraints

Path constraints are used to identify the set of evolution paths allowed by the evolution style. In particular, they can be used to restrict releases to being in a particular architectural style, make sure that certain dependencies in evolution are reflected (e.g., requiring certain architectural structures to be in place before other operations are performed), or require preservation of invariants across all releases. For example, in the above scenario, path constraints might require that no externally accessible services are removed in any release.

[1] We use the term "architecture element" to refer generally to components, connectors, and ports.

[2] This is similar to defining a profile in UML.

We use an augmented version of linear temporal logic (LTL) to specify these path constraints. Temporal logic is a natural choice, since our underlying model of evolution styles is an augmented state machine. In particular, evolution spaces give rise to standard Kripke structures [3] in a direct way, where the node labels represent architectural properties expressed as predicates that hold for a given architecture in an evolution path. Therefore, temporal formulas over evolution spaces can be interpreted in a straightforward manner.

We begin with the usual LTL operators, including:

- $\Box$ – *always*, to represent invariant properties of paths;
- $\Diamond$ – *eventually*, to represent the existence in a path of an architecture with certain properties;
- **U** – *until*, to represent properties that must remain true of a path until some other property becomes true; and
- $\circ$ – *next*, to represent properties that must be true in the next node of the path.

Ordinary LTL is sufficient to express many interesting properties. For example, suppose we want to specify (in the example in Section 4) that the billing component will not be removed until a controller is introduced. We might represent this constraint as follows:

$$billingComponentPresent(system)\ \mathbf{U}\ controllerPresent(system)$$

Here, *billingComponentPresent* and *controllerPresent* are predicates over systems, defined by the evolution style; *system* is a keyword that refers to the system architecture associated with the current state. Note that each of the predicates is expressible with respect to a single state. We represent these predicates using the first-order predicate logic constraint language in Armani.

Now consider a richer path constraint. Suppose we want to specify that all the functionality that is present at a release point remains present throughout the evolution (where "functionality" is formalized in some sensible way – e.g., by looking at all components of type *FunctionalModule*). If we try to express the constraint in LTL, we quickly encounter a problem.

$$\Box(release \rightarrow \Box hasAllFunc(system, ?)) \qquad (1)$$

The problem is that, to express this constraint, we need to refer back to a previous state, namely the previous release. That is, we want to replace the question mark in equation (1) with a reference to the previous release state. We thus introduce the *rigid-variable operator*, which allows us to refer directly to states that we have already "seen." In our notation, equation (1) would be correctly rendered as

$$\Box(\{s\}\ release \rightarrow \Box hasAllFunc(system, s.system))$$

The braces are our rigid-variable operator. When we encounter them, they "save" the current state to the rigid variable *s* so that we can refer back to it as such in a subsequent step. Our approach to rigid variables is similar to that of [30], which uses them in the context of object-oriented data models.

In the interest of space, we will not give a full formal syntax or semantics for our constraint language here, but it is worth briefly noting that the rigid-variable operator is a simple addition to the conventional Kripke-style semantics of ordinary LTL (e.g., definition 3.13 in [20]). We simply add another conjunct to the satisfaction relation:

$$\pi \vDash \{s\}\phi \text{ iff } \pi \vDash \phi[s \leftarrow \pi^1],$$

where $\phi[s \leftarrow \pi^1]$ denotes π^1 substituted for s in ϕ.

Because of the finite nature of paths, it becomes possible to check whether a given evolution path satisfies a given set of evolution constraints. Thus tools can check the correctness of path constraints.

(e) Specifying Evolution Operators

An evolution style comes with a set of operators that are specific to that style. For example, the evolution style for the example in Section 5 included operators to add an adapter to a system and connect it to a subsystem, to add a new hub to the system, and to migrate service functionality into the hub.

Currently we define architectural operators in an imperative manner using the Stitch language, developed as part of the Rainbow project [7], [8]. Specifically, an operator is defined using a set of primitive architecture operators and standard programming control constructs (conditionals, loops, etc.). Primitive operators include adding, removing, or replacing architectural elements, attaching connectors to component ports, encapsulating a part of an architecture as a higher-level component, and changing the value of a property.

(f) Specifying and Using Evaluation Functions

Given the facets of evolution styles just described, it is possible for an architect to define paths that are technically "correct," in the sense that they are created by using valid operators, and satisfy all path constraints of the style. However, an important additional benefit of defining paths for architecture evolution is to be able to compare them and decide which path is the best to adopt.

To enable this, we introduce the concept of *evaluation functions* into evolution styles. The purpose of an evaluation function is to help the architect determine whether a path satisfies business and management goals. In general, evaluation functions will depend on

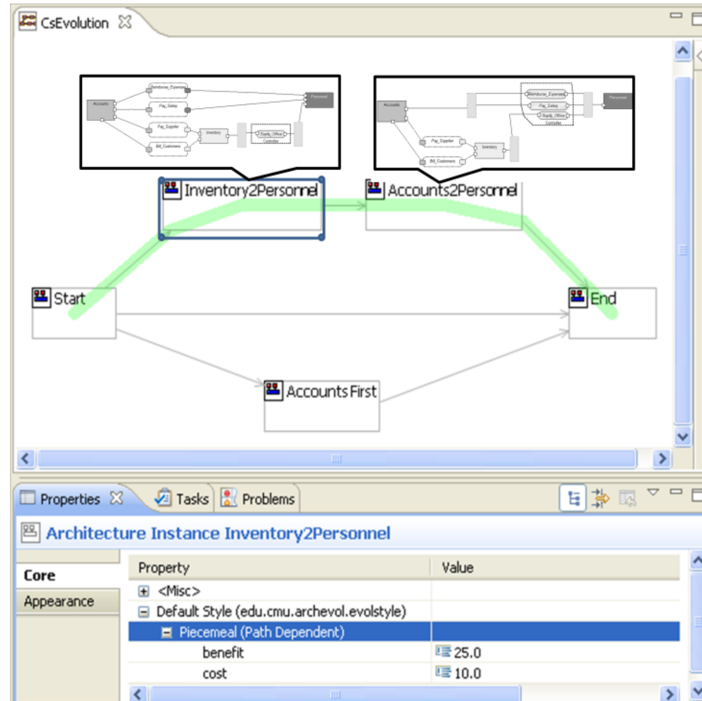

Figure 2. The Ævol workbench.

attributes specific to a particular business context: (a) the qualities of concern (cost, functionality, time, etc.) and their relative priorities, and (b) constraints on resources (number of personnel, time to deliver a release, etc.).

Currently we support the specification of the following kinds of auxiliary information that can be associated with nodes and transitions on paths:

(a) A vector of *quality* attributes that can be associated with releases, together with a utility function that determines the value of a certain release based on its associated quality attributes.

(b) A vector of *cost* attributes that can be associated with operations, together with a cost function that calculates the aggregate cost of a sequence of operations.

(c) A set of constraints on costs and qualities that determine the business context.

As illustrated earlier, the primary use of such analysis is to provide feedback to an architect about the costs and quality of a given evolution path, allowing the architect to explore the consequences of different decisions about the path. For example, the architect may decide to use a few releases with major changes, requiring the investment of substantial resources to achieve this, but reducing the time to reach the target architecture. Alternatively, if time is not a constraint, and cost is a constrained resource, the architect may decide to stretch the evolution out over a larger number of releases. The use of an evaluation function, based on the attributes listed above, permits such tradeoff analyses.

To enable analysis, we allow the properties and constraints required by the analysis to be specified as properties and path constraints of the evolution style. Using the annotated evolution graph as a model we provide tool support to plug different analysis implementations into the tool to conduct the analysis. We also provide plug-in support for reporting and comparing analyses of different paths.

Analyses can vary considerably in their level of detail, and the specific algorithms used. For example, as illustrated below, a very simple analysis might simply add up costs and benefits. More sophisticated analyses can used multiple cost and benefit dimensions, uncertainties, and complex evaluation functions, based, for example on options theory [27].

6. Tool Support

We have developed a tool that functions as a platform for exploring the approach to architecture evolution described above. The tool, called Ævol [14], is a plug-in framework that supports different forms of analysis and planning to be implemented and tested within the environment. Architects define an evolution graph in Ævol and link nodes to architectures of systems that are represented in AcmeStudio [32], an editor for Acme [13].

Figure 2 shows Ævol displaying an evolution graph. Nodes are linked to architectural instances, which can be opened in AcmeStudio. Associated with each node and transition in the graph is a set of properties. The selected element's properties are shown in the Properties view at the bottom of the figure. This view displays the instances that the node is linked to, in addition to properties required for analysis (in the example in the figure, simply *cost* and *benefit*). Paths are represented as semi-transparent, thick lines in the diagram. (Only one such path is highlighted in the figure.) The callouts show thumbnail sketches of the architectures that are attached to the intermediate steps on the path. Once the properties on each path are filled in, it is possible to run an analysis to compute overall utility of a path and then to compare utilities of different paths.

Ævol is written in Java as a plug-in to the Eclipse framework using Eclipse's Graphical Modeling Framework. It is also a plug-in to AcmeStudio architecture development environment (itself an Eclipse plug-in) to link evolution path nodes with architectural instances for each step in the evolution. Analyses are written as Java plug-ins using APIs provided by Ævol.

7. Conclusion and On-going Work

In this paper we outlined what we feel to be foundations for specifying and reasoning about and supporting architectural evolution. The key idea is to focus on evolution paths, with the goal of choosing an optimal path to achieve business objectives of an organization. Optimality is achieved by adopting a utility-theoretic approach, allowing us to tailor the analysis to the context. Additionally, we characterize recurring patterns as a set of related paths, which we term evolution styles. Such styles can be formally characterized, and supported by tools.

Our ongoing work in this area is devoted to elaborating the definition of evolution styles by enhancing the concepts of evolution operators and evolution analyses. We plan to explore other, more declarative, ways of specifying evolution operators, perhaps in the style of graph grammars used in [40] or rewrite rules as in [21]. Furthermore, we would like to develop and explore better ways to analyze evolution paths, perhaps considering approaches from various economic theories. We believe that we have developed a sound foundation for a wide range of path analysis techniques.

We are also actively enhancing support for evolution styles – specifically through an evolution style editor, new ways to visualize evolution paths and analyses, better support for constraint specification and checking, and a catalog of common evolution styles constraints, or the use of operators. This is an area of active development.

One other area that we plan to explore is the use of planning to automatically generate possible paths. Given our use of temporal logic expressions to define correct paths, and the use of operators that may be used to construct paths, it may be possible to use a planning approach similar to those discussed in [15] to automatically generate alternative paths.

Acknowledgements

This work is supported in part by the National Science Foundation under Grant No. 0615305 and by the Office of Naval Research (ONR), United States Navy, N000140811223, as part of the HSCB project under OSD. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring offices.

8. References

[1] M. Abi-Antoun, J. Aldrich, N. Nahas, B. Schmerl, and D. Garlan, "Differencing and merging of architectural views," in *Proc. ASE'06*, Tokyo, Japan, Sept. 18–22, 2006, pp. 47–58.

[2] C. Baldwin and K. Clark. *Design Rules: The Power of Modularity*, vol. 1. Cambridge, MA: MIT Press, 1999.

[3] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie. *Systems and Software Verification: Model-Checking Techniques and Tools*. Berlin, Germany: Springer, 2001.

[4] B. Berliner, "CVS II: Parallelizing software development," in *Proc. USENIX Winter '90*, Washington, DC, Jan. 22–26, 1990, pp. 341–352.

[5] B. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981.

[6] S. Chaki, N. Sharygina, and N. Sinha, "Verification of evolving software," in *Proc. SAVCBS '04*, Newport Beach, CA: Oct. 31–Nov. 5, 2004, pp. 55–61.

[7] S.-W. Cheng. "Rainbow: Cost-effective software architecture-based self-adaptation," Ph.D. dissertation, Carnegie Mellon Univ., Pittsburgh, PA, May 2008. Institute for Software Research Tech. Rep. CMU-ISR-08-113.

[8] S.-W. Cheng, D. Garlan, and B. Schmerl. "Architecture-based self-adaptation in the presence of multiple objectives," in *Proc. SEAMS'06*, Shanghai, China: May 21–22, 2006, pp. 2–8.

[9] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. *Documenting Software Architectures: Views and Beyond*. Boston, MA: Addison–Wesley, 2002.

[10] R. Correia, C. Matos, M. El-Ramly, R. Heckel, G. Koutsoukus, and L. Andrade. Software Engineering at the Architectural Level: Transformation of Legacy Systems. Technical Report, Department of Computer Science, University of Leicester, U.K., 2002.

[11] M. Erder and P. Pureur, "Transitional architectures for enterprise evolution," *IT Professional*, vol. 8, no. 3, pp. 10–17, 2006.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison–Wesley, 1994.

[13] D. Garlan, R. Monroe, and D. Wile, "Acme: An architecture description interchange language," in *Proc. CASCON'97*, Toronto, ON, Nov. 10–13, 1997, pp. 169–183.

[14] D. Garlan and B. Schmerl, "Ævol: A tool for defining and planning architecture evolution," in *Proc. ICSE'09*, Vancouver, BC, May 16–24, 2009.

[15] A. E. Gerevini, P. Haslum, D. Long, A. Saetti, and Y. Dimopoulis. "Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners." *Artif. Intell.*, vol. 173, nos. 5–6, pp. 619–668, 2009.

[16] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Englewood Cliffs, NJ: Prentice Hall, 1991.

[17] L. Grunske, "Formalizing architectural refactorings as graph transformation systems," in *Proc. SNPD/SAWN '05*, Towson, MD, May 23–25, 2005, pp. 324–329.

[18] A. van der Hoek, D. Heimbigner, and A. L. Wolf, "Versioned software architecture," in *Proc. ISAW3*, Orlando, FL, Nov. 1–2, 1998, pp. 73–76.

[19] S. Hunold, M. Korch, B. Krellner, T. Rauber, T. Rechel, and G. Runger. Transformation of Legacy Software into Client/Server Applications through Pattern-based Rearchitecturing," in *Proc. the 32nd IEEE International Computer Software and Applications Conference* (COMPSAC), 2008.

[20] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*, Cambridge, England: Cambridge UP, 2000.

[21] P. Inverardi and A. Wolf, "Formal specification and analysis of software architectures using the chemical abstract machine model," *IEEE T. Software Eng.*, vol. 21, no. 4, pp. 373–386, 1995.

[22] R. Kazman, L. Bass, and M. Klein, "The essential components of software architecture design and analysis," *J. Syst. & Software*, vol. 79, no. 8, pp. 1207–1216, 2006.

[23] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures," in *Proc. ESEC'95* (ser. Lect. Notes Comput Sc., vol. 989), Sitges, Spain, Sept. 25–28, 1995, pp. 137–153.

[24] N. Medvidovic and R. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE T. Software Eng.*, vol. 26, no. 1, pp. 70–93, 2000.

[25] W. F. Opdyke and R. E. Johnson, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in *Proc. SOOPPA'90*, Poughkeepsie, NY, Sept. 14–15, 1990, pp. 145–160.

[26] M. Oussalah, N. Sadou, and D. Tamzalit, "SAEV: A model to face evolution problem in software architecture," in *Proc. ERCIM Workshop on Software Evolution*, Lille, France, Apr. 6–7, 2006, pp. 137–146.

[27] I. Ozkaya, R. Kazman, and M. Klein, "Quality-attribute-based economic valuation of architectural patterns," Software Engineering Institute, Pittsburgh, PA, Tech. Rep. CMU/SEI-2007-TR-003, May 2007.

[28] D. L. Parnas, "Information distribution aspects of design methodology," in *Proc. IFIP Congress '71*, Ljubljana, Slovenia, Aug. 23–28, 1971, pp. 339–344.

[29] D. Perry and A. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Eng. Notes*, vol. 17, no. 4, pp. 40–42, 1992.

[30] J. Richardson, "Supporting lists in a data model (a timely approach)," in *Proc. VLDB'92*, Vancouver, BC, Aug. 23–27, 1992, pp. 127–138.

[31] J. Rushby, "Bus architectures for safety-critical embedded systems," in *Proc. EMSOFT'01* (ser. Lect. Notes Comput. Sc., vol. 2211), Tahoe City, CA, Oct. 8–10, 2001, pp. 306–323.

[32] B. Schmerl and D. Garlan. AcmeStudio: Supporting style-centered architecture development. *Proc. ICSE'04*, Edinburgh, Scotland, May 23–28, 2004, pp. 704–05.

[33] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice Hall, 1996.

[34] B. Spitznagel and D. Garlan, "A compositional approach for constructing connectors," in *Proc. WICSA '01*, Amsterdam, The Netherlands, Aug. 28–31, 2001, pp. 148–157.

[35] B. Spitznagel and D. Garlan, "A compositional formalization of connector wrappers," in *Proc. ICSE'03*, Portland, OR, May 3–10, 2003, pp. 374–384.

[36] D. Tamzalit, N. Sadou, and M. Oussalah, "Evolution problem within component-based software architecture," in *Proc. SEKE'06*, San Francisco, CA, Jul. 5–7, 2006, pp. 296–301.

[37] D. Tamzalit, M. Oussalah, O. Le Goaer, and A. Seriai, "Updating software architectures: a style-based approach," in *Proc. SERP'06*, Las Vegas, NV, Jun. 26–29, 2006, pp. 336–342.

[38] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oriezy, and D. L. Dubrow, "A component- and message-based architectural style for GUI software," *IEEE T. Software Eng.*, vol. 22, no. 6, pp. 390–406, 1996.

[39] E. Yourdan and L. Constantine, *Structured Design*. Englewood Cliffs, NJ: Prentice Hall, 1978.

[40] M. Wermelinger and J. L. Fiadeiro, "A graph transformation approach to software architecture reconfiguration," *Sci. Comput. Program.*, vol. 44, no. 2, pp. 133–155, 2002.