Evolution Styles - Formal foundations and tool support for software architecture evolution

David Garlan

June 2008
CMU-CS-08-142

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

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, requiring large-scale and systematic restructuring. Today architects have few tools to help them plan and execute such evolutionary paths. In particular, they 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 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 an evolution plan be represented and communicated within an organization? In this report we outline first steps towards a formal basis for assisting architects in developing and reasoning about architectural evolution paths. The key insight behind the approach is that at an architectural level of abstraction many system evolutions follow certain common patterns – or *evolution styles*. By taking advantage of regularity in the space of common architectural evolutions, and by making the notion of evolutions styles a first-class entity that can be formally defined, we can provide automated assistance for expressing architecture evolution, and for reasoning about both the correctness and quality of evolution paths.

This research has benefited from a discussion of Architecture Evolution with the members of the ABLE research group (Jeff Barnes, Roni Burd, Orieta Celiku, Owen Cheng, George Fairbanks, Jung Soo Kim, Vahe Poladian, and Bradley Schmerl); members of the SEI SAT initiative who participated in a joint reading group on this topic (Ipek Ozkaya and Mark Klein). I would also like to acknowledge Sungwon Kang and Dalila Tamzalit, who have sparked many useful insights about the topic of Architectural Evolution.

Keywords: Software Architecture, Architecture Evolution

Table of Contents

Table of Figures

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 trade-offs 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?

We argue that such questions require new foundations that permit architects to reason about and plan large-scale system-wide changes at an architectural level of abstraction. Ideally these foundations would allow one to represent architecture evolution paths as first-class entities that can be expressed precisely and reasoned about. They should 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, they should allow an architect to reason not only about "correct" evolution, but also make tradeoffs to maximize business goals, such as the time to reach the target architecture and the costs involved in doing so. Finally, there should be practical tool support to automate these analyses.

In this report we consider the first steps towards this vision by outlining a formal basis for assisting architects in developing and reasoning about architectural evolution paths. The key insight behind the 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 expressing architecture evolution, and for reasoning about both the correctness and quality of evolution paths. We refer to collections of related paths as *evolution style*s. Evolution styles can be defined, reasoned about, analyzed, applied to the evolution of specific systems, and supported by tools. By capturing such families 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.

2 The Problem

In today's commercial environments architecture evolution is a fact of life. Two primary forces contribute to this.

First is the desire to incorporate new technologies, infrastructures and capabilities into existing systems. For example, many IT-based companies have gone through evolutions that take them from thin-client, mainframe-based systems, to three- or four-tiered architectures [10]. The move to a tiered architecture allows them to factor their application software into more maintainable layers (such as a business tier or web presentation tier), scale their processing (for example, through replication of data stores at the data layer or multiple servers at the business logic tier), and take advantage of common services (such as security) provided by commercial infrastructure. A similar transformation is now taking place for companies that move from traditional N-tiered systems to service-oriented architectures.

Second is the prevalence of mergers and acquisitions. Increasingly companies are opting to buy new systems as a way to expand their business. In many cases the architectures of existing and newly acquired systems will not be the same. While ad hoc integration solutions may work initially, as the number of subsystems increases, more radical and systematic restructuring will be needed. Therefore architectural redesign is necessary to achieve a combined system that functions as a single system.

In both cases abandoning the existing legacy system(s) and reengineering a new target system from scratch is not a viable option. Instead, one must create an evolution plan that achieves a desired architecture through a series of incremental milestones. Such evolutions require careful planning, Indeed, because in most cases intermediate releases require a mixture of old and new technologies and architectures, there may be considerable risk of inconsistency, performance degradation, loss of functionality, and potential downtime during the transition process. Moreover, there may be many ways in which one might carry out a plan: for example, some taking longer time but with lower risk and cost, others with more aggressive schedules that have a higher competitive pay-off.

3 Existing Approaches

Today's approaches to solving such problems 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 [32], indications of maintainability such as coupling and cohesion [5][45] code refactoring [29], reverse engineering, regression testing, and many others [20]. These techniques focus on the code structures of a system, and have led to numerous advances, such as programming language support for modularization and encapsulation, analysis of module compatibility and substitutability [11], and design patterns that support maintainability [15].

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 run-time structures that are necessary to reason about the architecture of a complex software system. Moreover, the techniques are typically (a) general-purpose, focusing on general properties of modularity (such as coupling and cohesion), or (b) oriented towards low-level code structures, such as class organizations in object-oriented programs. In contrast, as we will see, our proposed work focuses on the reuse of specifications and analyses for domain-specific evolution at an architectural level of abstraction.

The second closely-related area of research and development is *tool support for project management and planning*. For example, modern version control systems such as RCS [44], CVS [8], and Subversion [6] allow different versions of artifacts to be compared and reviewed. In most of these tools, the primary managed artifact is linear source code, rather than architectural structures. 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][22] 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 evolution path or a path that optimizes business goals.

In the domain of project planning, traditional project management approaches and software development planning approaches such as COCOMO [9] 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 correctness constraints on any states of a system's architectural structure. Advice on how to organize architecture evolution steps into waves and plateaus is given in [14]. 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 [21][40][46]. For example, Wermelinger uses category theory to describe how transformations can occur in software architecture [46]. 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 the next. Architecture in this sense is defined by the space of all possible configurations that can result from a certain starting configuration. Grunske [21] 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 in [39] focuses on the transformation of architectural connectors as a way to augment the communication paths between components.

While such formal approaches lay a foundation for generic forms of architecture operators, essential for reasoning about architectural evolution, unlike the work proposed here, they are not amenable to specialization for specific classes of transformation and systematic reuse. Moreover, while they can provide some support for characterizing forms of evolution *correctness*, but they do not address issues of evolution *quality*.

Recently Tamzalit and others have begun to investigate recurring patterns of architecture evolution, primarily with respect to component-based architectures [41][42][30]. They use the term evolution style to refer patterns for updating a component-based architecture. They provide a formal approach based on a three tiered conceptual framework. Like the work proposed here, they attempt to capture recurring and reusable patterns of architecture evolution. However, unlike the work that we propose, 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 in the area of *tradeoff analysis for architectural evolution.* The work of Kazman et. al. [24] applies existing architectural analysis and trade-off techniques to improve architectures. The improvements are incremental, and take into consideration only known attributes. The approach has not been considered for architecture evolution that looks at large scale, system-wide evolution, over a long period of time, where there is naturally some uncertainty in the attributes over that period of time. The work in [31] proposes to use option-based techniques from economic option theory to characterize uncertainty and options available in evolution, and identifies several techniques can then be used to calculate the points in time where introducing changes would be cost-effective in a business sense. 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 *does* focus on architectural evolution for specific classes of systems. Typically this work addresses architecture evolution in the context of a specific style, such Darwin [26] and C2 [43]. 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, unlike our proposed work these approaches are limited to systems constructed in the particular architectural style that they support.

4 Approach

We propose a formal basis for software architecture evolution that has the following key properties

- 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 [35]).

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 such evolutions.) Each of them has identifiable starting and ending conditions (namely, that the initial and final system have certain architectural structures). Each embodies certain constraints – for example, that the set of essential services should not 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 (possibly infinite) 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 selected from 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.

Additionally, however, each path in an evolution style is associated with a set of *releases,* modeled as a subset of the nodes in the path. Intuitively, releases represent points in the path at which the system will be deployed. 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 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 or to search for optimal paths in the evolution style. 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 features (e.g., time, resources, risk, downtime, etc.) and in the presence of uncertainty.

5 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, and based on a PeopleSoft platform; another subsystem manages inventory using SAP, and 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 pay checks, and between the inventory and accounts to pay suppliers and bill vendors.

This form of system is common in today's IT world, 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 the existing subsystems' languages and schema, the number of ad hoc connections between the subsystems is reduced, improving maintainability and extensibility.

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

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 families. It would also characterize the family 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 family), 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 essential 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

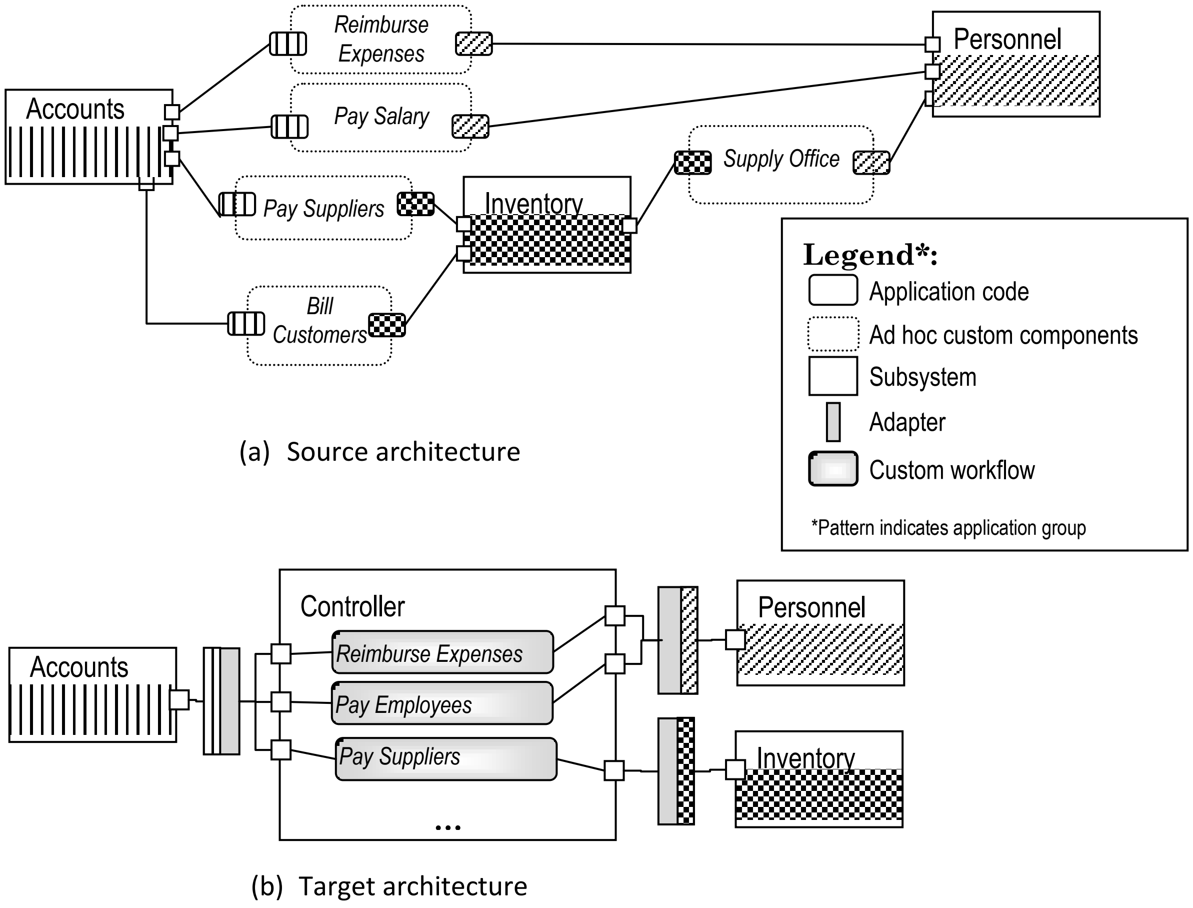

Figure 1. Examples of architectural instances.

for this would be the characterization of the initial and target architectures. The tools make it relatively easy to specify these using standard architecture modeling and visualization techniques. For example, Figure 1 illustrates a drastically simplified version of the initial and target states. 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 family.

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 made decisions about various tradeoffs, for example, reconciling the 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 in those values).With these annotations in hand the tools calculate for the architect costs and utility, allowing him to explore alternative scenarios using the tools. 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.

6 Details of the Research Plan

We now describe a formal framework on which our approach is based, followed by a more detailed description of the specific research approach and problems that must be solved.

6.1 A Formal Model for Evolution Styles

Formally, we define an *evolution space* to be a 5-tuple (A, I, F, Ops, δ) where

- A is a set of architectures,
- $I \subseteq A$ is a set of *initial* architectures,
- $F \subseteq A$ is a set of *target* architectures,
- Ops is a set of *operator names*, and
- $\delta \subseteq A \times Ops \times A$ is a relation that describes how architectures are transformed according to the operators in Ops.

We then define the *behavior* of an evolution space $S = (A, I, F, Ops, \delta)$, denoted $Beh(S)$, as all traces $<a_0, op_1, a_1, op_2, \ldots op_n, a_n>$ such that

$a_i \in A$, $op_i \in Ops$, $a_0 \in I$, $a_n \in F$, and $(a_i, op_{i+1}, a_{i+1}) \in \delta$.

An *evolution path* over an evolution space (A, I, F, Ops, δ) is a pair (t, R) such that

$t \in Beh(S)$,

$\boldsymbol{R} \subseteq \boldsymbol{Range(t)} \downarrow \boldsymbol{A}$ is a subset of the range of $\boldsymbol{t}$ restricted to architectures, and if $\boldsymbol{t = <a_0, op_1, a_1, op_2, \ldots op_n, a_n>}$ then $\boldsymbol{a_0, a_n} \in \boldsymbol{R}$.

The $\boldsymbol{R}$ in the above definition refers to the set of release points in the path. We denote the set of all evolution paths over an evolution space $\boldsymbol{S}$ by $\boldsymbol{EP(S)}$.

An *evolution style* over an evolution space $\boldsymbol{S}$ is a subset of $\boldsymbol{EP(S)}$. That is, an evolution style constrains the collection of paths to a subset of those otherwise permitted by simple application of operators.

An *evolution evaluation function* over an evolution style $\boldsymbol{E}$ is a mapping $\boldsymbol{EP(E)} \rightarrow \boldsymbol{R_{\geq 0}}$. An evolution function represents a quality metric that allows us to compare any two evolution paths within an evolution style. Note that the evaluation function is independent of the evolution style reflecting the principle that different organizations will assign different evaluations to paths in the style, depending on their business goals.

Finally, we define the set of *release paths* to be the set of equivalence classes of paths under the following relation: two evolution paths $\boldsymbol{(t_1, R)}$ and $\boldsymbol{(t_2, R)}$ are *release equivalent* if and only if $\boldsymbol{t_1} \downarrow \boldsymbol{R} = \boldsymbol{t_2} \downarrow \boldsymbol{R}$. That is to say, the sequence of appearances of the release points is the same in both traces. Thus, release paths are a kind of abstraction over evolution paths, focusing only on the sequence of releases in a path, and not the order of (micro) steps that lead from one release to another. In practice, we believe that many analyses will be in terms of release paths, rather than the more detailed evolution paths.

6.2 Specifying Evolution Styles

While the formal model outlined above provides a good mathematical framework for understanding evolution styles and analyses of paths, it is not by itself very practical. To make the concepts useful in practice we need specification languages (and tools) that allow architects to easily define and analyze evolution styles, paths, etc.

A significant component of our on-going research in this area is the definition of these specification languages. In particular, based on the formal model above, we are developing ways to characterize (a) architectures, (b) sets of architectures, (c) evolution operators, (d) path constraints, (e) releases, and (f) evaluation functions. We outline how we are addressing each of these, indicating the specific research issues that we are currently pursuing.

(a) Specifying Architectures:

For this aspect of the research we specify architectures in a relatively standard way using the Acme architecture description language (ADL) [19]. Like 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* [13][27][33][37]. 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 at a more detailed level.

Augmenting architectural structure, we allow the elements of an architecture (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 performance properties (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 families as embodied in ADLs, such as Acme [17][19]. Specifically, an architectural family 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. 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 set of components connected to another one, returning the set of ports of a component, etc. (For details of the constraint language see [19].)

Referring back to the example of Section 5, there would likely be three relevant families: the peer-to-peer family of the initial system, the hub-and-spoke family of the target system, and the combination family for intermediate releases. Component types in the hub-and-spoke family would include things like the hub and various adapters. Connector types would include the standard adapter-hub communication protocol, as well as the specialized connectors that link adapters to subsystems. Constraints might specify that all service-delivering subsystems must be connected to the hub (possibly via an adapter).

(c) Specifying Evolution Operators:

As in the formal model, 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.

Defining a usable evolution operator specification language is a challenging research problem. As a starting point we are building on existing work on the definition of architectural operators from a related project focused on dynamic adaptation [12]. Specifically, an operator is defined in an imperative way 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.

Research questions that we hope to answer in this research focus on elaboration of the imperative language, and exploration of other more-declarative ways of specifying evolution operators, perhaps in the style of graph grammars used in [46] or rewrite rules as in [23].

(d) Specifying Releases

[1] In the remainder of this proposal we use the term "architecture element" to refer generally to components, connectors, and ports.

Given an evolution path, specifying a release is a relatively straightforward matter of identifying which architectures in the sequence are intended for deployment.

In the context of practical evolution tools, we expect that the notion of releases will take center stage. In particular, as illustrated in the above scenario, we envision that architects will typically plan their evolutions by specifying release paths (i.e., sequences of releases), which, as noted above, provide a simplified view and abstract over the details of the specific low-level operations that were used to get from one release to another.

(e) 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 family, making sure that certain dependencies in evolution are reflected (e.g., requiring certain architectural structures to be in place before other operations are performed), or preserving invariants across all releases. For example, in the above scenario, path constraints might require that no externally accessible services are removed in any release.

As a starting point, we are investigating the use of a subset of temporal logic. This choice is a natural one given the fact that our underlying model of architectural styles is an augmented state machine. In particular, evolution spaces give rise to standard Kripke structures [7] in a direct way, where the node labels represent architectural properties expressed as predicates that hold for a given architecture in a behavioral path. Therefore, temporal formulas over evolution spaces can be interpreted in a straightforward manner.

Among the operators that we expect to include are the standard temporal modalities:

- $\Box$ – *always*, to represent properties of paths that are invariant.
- $\Diamond$ – *eventually*, to represent the existence of an architecture in a path that has certain properties
- **U** – *until*, to represent properties that must remain true of a path until some other property becomes true.

Given a set of path constraints and a proposed evolution path (or release path), because of the finite nature of paths, it becomes possible to check whether that path satisfies the constraints. Thus tools can check the correctness of path constraints.

Research questions that we hope to answer for this component of the research include: What constraint operators are most useful? Are the standard temporal operators adequate to express realistic constraints? What properties of a system are important to represent in path constraints? In practice is it sufficient to specify constraints over release paths, or do we also need to support specifications of full evolution paths?

(f) Specifying and Using Evaluation Functions

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 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 (numbers of personnel to do the work, time to get out a new release, etc.).

An important component of our research is to iudentify practical ways to specify evaluation functions in terms of these attributes. As a starting point, we are investigating specification of the following kinds of auxiliary information:

(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.

Each of the values in these three categories is represented stochastically, incorporating a measure of the uncertainty in the values. This is important because cost and benefit valuations are rarely precise. Given these, we believe we will be able to calculate the overall expected utility of a given path relative to business constraints.

As illustrated earlier, the primary use of such analysis will ultimately be to provide feedback to an architect about 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 will permit such tradeoff analyses.

Important research questions to answer include: What is the best way to represent such vectors, functions and uncertainties? What algorithms should be used to determine expected utility? Is it possible to abstract over operator costs to release transition costs, thereby simplifying the job of specify?

However, in addition to giving feedback on specific evolution paths we anticipate that it will also be possible to use this same information to automatically generate candidate paths for the architect. To do this we are exploring two techniques.

The first is the use of Markov Decision Processes (MDPs). An MDP is a mathematical framework that can be used to model problems in which outcomes are partly under the control of a decision maker and partly decided probabilistically. Formally, an MDP is a discrete-time stochastic control process defined by a 4-tuple **(*S, A,* P.(·,·), R(·))**, where:

- ***S*** is a set of possible states,
- ***A*** is a set of actions,
- **P.(·,·)** is a transition probability matrix, and
- **R(·)** is an immediate reward function.

The state of the decision maker changes over time, in part due to the actions he chooses. In each state, **s**, there are a number of actions **a** for the decision maker to choose from. The destination state **s'** is determined randomly according to the transition probability matrix $\mathbf{P.(\cdot,\cdot)}$. Formally, $\mathbf{P_a(s, s') = Pr(s_{t+1} = s' \mid s_t = s, a_t = a)}$, where **t** is the current time, i.e. $\mathbf{P_a(s, s')}$ is the probability of going into state **s'** when the decision maker chooses action **a** while in state **s** at time **t**. When the decision maker is in state **s**, an immediate reward equal to **R(s)** is earned. Earned rewards accrue over time.

The objective of the decision maker is to accumulate the maximum reward possible over time. This is accomplished by solving for a policy $\boldsymbol{\pi}$ that provides the optimal action to choose in each state, regardless of prior history of states. There are a number of known solutions to finding an optimal policy. These solutions use iterative methods or dynamic programming (also known as backwards induction).

Because the architectural evolution of a system requires the architect to make choices at each evolutionary step, and the outcome of a step exhibits some degree of uncertainty, we believe that architectural evolution may be characterized as an MDP, which can be solved for the optimal policy for architectural evolution for that system, which specifies the appropriate transformation decision to make in every evolutionary state.

The second approach that we are exploring is the use of a new technique called Simple Temporal Problems with Preferences and Uncertainties (STPPU) to provide a planning schedule in the presence of desired quality attributes and uncertain timing constraints [34]. An STPPU is a combination of Simple Temporal Networks with Uncertainty and Simple Temporal Network with Preferences, each of which are extensions of simple temporal networks. Variables in the network are time points, and edges are constraints between the time points. An STPPU partitions time points into those that can be controlled, and those that are fixed by the environment. An STPPU is represented using the tuple $\mathbf{(V_x, V_o, E_c, E_u, EQ_c)}$, where:

- $\mathbf{V_x}$ is a set of time points that can be controlled. For example, the starting and ending points of particular steps in the evolution;
- $\mathbf{V_o}$ is a set of time points that represent observable events over which there is no control. For example, in architectural evolution these could represent business restrictions such as the time that the evolution should complete, the time period for which contractors are available, or the required release dates;
- $\mathbf{E_c}$ is the set of controllable edges, representing a simple temporal constraint between two timepoints, such that $\mathbf{lb_{ij} \leq V_j - V_i \leq ub_{ij}}$, $\forall \mathbf{i,j} \in \mathbf{E_c}, \mathbf{V_j} \in \mathbf{V_x}$, and $v_i \in \mathbf{V_x} \cup \mathbf{V_o}$, and **lb** and **ub** are lower and upper bounds;
- $\mathbf{E_u}$ is the set of contingent edges with constraints set by the environment, as above, except $\forall \mathbf{i,j} \in \mathbf{E_u}, \mathbf{V_i} \in \mathbf{V_x} \cup \mathbf{V_o}, \mathbf{V_j} \in \mathbf{V_o}$;
- $\mathbf{EQ_c}$ is the set of quality profiles associated with controllable edges. The quality profiles could be desired architectural qualities, or desired cost and personnel requirements. The quality profiles are assumed to be piece-wise linear and convex.

The STPPU can then be used to find an execution strategy (i.e., path) that works in all realizations, is optimal in each of them, and where certain qualities (expressed as preferences) are met. Algorithms for finding paths tractably in polynomial time are given in [34].

7 Summary

In this report we have outlined what we feel to be promising first steps to a formal basis for 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 on-going work in this area is devoted to finding ways to make this formal basis practical through notations and tools that allow evolution planners to specify the relevant constraints of their domain, invoke analyses specific to their context, and plan effectively for a series of phased releases of a system.

Acknowledgements

This research has benefited from a discussion of Architecture Evolution with the members of the ABLE research group (Jeff Barnes, Roni Burd, Orieta Celiku, Owen Cheng, George Fairbanks, Jung Soo Kim, Vahe Poladian, and Bradley Schmerl); members of the SEI SAT initiative who participated in a joint reading group on this topic (Ipek Ozkaya and Mark Klein). I would also like to acknowledge Sungwon Kang and Dalila Tamzalit, who have sparked many useful insights about the topic of Architectural Evolution.

References

[1] M. Abi-Antoun, J. Aldrich, N. Nahas, B. Schmerl, D. Garlan. Differencing and Merging Architectural Views. Accepted for publication in the Automated Software Engineering Journal, Springer, 2008.

[2] R. Allen, D. Garlan. A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology, **6**(3):213-249, July 1997.

[3] R. Allen, R. Douence, D. Garlan. Specifying and Analyzing Dynamic Software Architectures. Fundamental Approaches to Software Engineering, in Lecture Notes in Computer Science **1382** 1998.

[4] R. Allen, D. Garlan, J. Ivers. Formal Modeling and Analysis of the HLA Component Integration Standard. In Proc. the Sixth International Symposium on the Foundations of Software Engineering (FSE-6),.1998.

[5] C. Baldwin, K. Clark. Design Rules: The Power of Modularity, Volume I. MIT Press, Cambridge MA, 1999.

[6] B. Behlendorf, C.M. Pilato, G. Stein, K. Hancock, and B. Collins-Sussman, "Subversion Project Homepage," http://subversion.tigris.org, 2003.

[7] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schoebelen, P. McKenzie. Systems and Software Verification: Model-checking Techniques and Tools. Springer, 2001.

[8] B. Berliner. CVS II: Parallelizing software development. In Proc. the 1990 USENIX Conference, Jan. 22-26, 1990, Washington, D.C.

[9] B. Boehm. Software Engineering Economics. Englewood Cliffs, NJ. Prentice-Hall, 1981.

[10] B. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture – A System of Patterns, Volume I. Wiley, 1996.

[11] S. Chaki, N. Sharygina, and N. Sinha. Verification of Evolving Software. In Proceedings of the 3rd International Workshop on Specification and Verification of Component-based Systems (SAVCBS) 2004.

[12] S.-W. Cheng, D. Garlan, B. Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In Proc. the ICSE 2006 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Shaghai, China, 2006.

[13] P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. Documenting Software Architectures: Views and Beyond. Addison Wesley, 2002.

[14] M. Erder, P. Pureur. Transitional Architectures for Enterprise Evolution. IT Professional, **8**(3):10-17, May/June, 2006.

[15] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley 1994.

[16] D. Garlan, M. Shaw, C. Okasaki, C. Scott, R. Swonger. Experience with a course in on Architectures for Software Systems. In Lecture Notes in Computer Science 640:23-43, 1992.

[17] D. Garlan, R. Allen, J. Ockerbloom. Exploiting style in architectural design environments. In Proc. SIGSOFT'94: The 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 179-185, Dec. 1994.

[18] D. Garlan, R. Allen, J. Ockerbloom. Architectural Mismatch, or Why it's hard to build systems out of existing parts. Proc. the 17th International Conference on Software Engineering, 1995.

[19] D. Garlan, R. Monroe, D. Wile. Acme: An architecture description interchange language. In Proc. CASCON'97, pp. 169-183, Nov. 1997.

[20] C. Ghezzi, M. Jazayeri, D. Mandrioli. Fundamentals of Software Engineering. Prentice Hall 1991.

[21] L. Grunske. Formalizing Architectural Refactorins as Graph Transformation Systems. Proc. the 6th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Conference on Self-Assembling Wireless Networks (SNPD/SAWN'05). Towson, MD, 23-25 May, 2005.

[22] A. van der Hoek, D.M. Heimbigner, and A.L. Wolf. Versioned Software Architecture. In Proc. the Third International Software Architecture Workshop, pp. 73-76, Orlando, FL, Nov. 1998.

[23] P. Inverardi, A. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. IEEE Transactions on Software Engineering, Special Issue on Software Architecture, **21**(4):373-386, April, 1995.

[24] R. Kazman, L. Bass, M. Klein. The essential components of software architecture design and analysis. The Journal of Systems and Software **79**, pp. 1207-1216, 2006.

[25] T. LaToza, D. Garlan, J. Herbsleb, B. Myers. Program Comprehension as Fact Finding. In Proc. the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007), pp. 361-370, Dubrovnik, Croatia, Sept. 2007.

[26] J. Magee, N. Dulay, S. Eisenbach, J. Kramer. Specifying distributed software architectures. Proc. the 5th European Software Engineering Conference (ESEC'95) 1995.

[27] N. Medvidovic, R. Taylor. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering **26**(1):70-93, Jan. 2000.

[28] R. Monroe, D. Garlan. Style-based reuse for software architectures. In Proc. the Fourth International Conference on Software Reuse, April, 1996.

[29] W. Opdyke, R. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. Proc. the Symposium of Object Oriented Programming Emphasizing Practical Applications (SOOP-PA), 1990.

[30] M. Oussalah, N. Sadou, D. Tamzalit. SAEV: a Model to Face Evolution Problem in Software Architecture. Proceedings of the International ERCIM Workshop on Software Evolution, April 2006.

[31] I. Ozkaya, R. Kazman, M. Klein. Quality-Attribute-Based Economic Valuation of Architectural Patterns. Software Engineering Institute Technical Report CMU/SEI-2007-TR-003, 2007.

[32] D.L. Parnas. Information Distribution Aspects of Design Methodology. Proc. the IFIP Conference '71, 1971, Booklet TA-3, pp. 26-30.

[33] D. Perry, A. Wolf. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes, **17**(4):40-42, 1992.

[34] F. Rossi, K. Venable, N. Yorke-Smith. Uncertainty in soft temporal constraint problems: a general framework and controllability algorithms for the fuzzy case. JAIR 27:617-674, 2006.

[35] J. Rushby. "Bus architectures for safety-critical embedded systems." In Lecture Notes in Computer Science **2211**, 2001.

[36] M. Shaw, D. Garlan. Formulations and Formalisms in Software Architecture. *Computer Science Today: Recent Trends and Developments*, *1000,* 1995.

[37] M. Shaw, D. Garlan. Software Architectures: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.

[38] B. Spitznagel. Compositional Transformation of Software Connectors. PhD Thesis, School of Computer Science, Carnegie Mellon University Technical Report CMU-CS-04-128, May 2004.

[39] B. Spitznagel, D. Garlan. A Compositional Approach for Constructing Connectors. *Working IEEE/IFIP Conference on Software Architecture*. 2001.

[40] B. Spitznagel, D. Garlan. A Compositional Formalization of Connector Wrappers. In Proc. the 2003 International Conference on Software Engineering (ICSE'03), 2003.

[41] D. Tamzalit, N. Sadou and M. Oussalah. Evolution problem within Component-Based Software Architecture. Proceedings of the 2006 International Conference on Software Engineering and Knowledge Engineering (SEKE'06). July 2006.

[42] D. Tamzalit, M. Oussalah, O. Le Goaer, A. Seriai. Updating Software Architectures: a style-based approach. International Conference on Software Engineering Research and Practice, July 2006.

[43] R. Taylor, N. Medvidovic, K. Anderson, E. Whitehead, E. Robbins, K. Nies, P. Oriezy, D. Dubrow. A component- and message-based architectural style for GUI software. IEEE Transactions on Software Engineering **22**(6), 1996.

[44] W.F. Tichy. RCS: a system for version control. Software: Practice and Experience, 15(7):637-54, 1985.

[45] E. Yourdan, L. Constantine. Structured Design. Englewood Cliff, NJ: Prentice-Hall, 1978.

[46] M. Wermelinger, J.L. Fiaderob. A graph transformation approach to software architecture reconfiguration. Science of Computer Programming 44:133-155, 2002.