Evolution styles: foundations and models for software architecture evolution

Jeffrey M. Barnes · David Garlan · Bradley Schmerl

Received: 15 March 2011 / Revised: 29 August 2012 / Accepted: 25 October 2012

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 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 planning and reasoning about architecture evolution. Our approach focuses on providing architects with the means to model prospective evolution paths and supporting analysis to select among these candidate paths. To demonstrate the usefulness of our approach, we show how it can be applied to an actual architecture evolution. In addition, we present some theoretical results about our evolution path constraint specification language.

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 be redesigned to accommodate them, and often this entails large-scale and systematic restructuring. In most cases such changes cannot be made overnight, so the architect must develop an evolution plan to change the architecture and implementation of a system through a series of phased releases, ultimately leading to the new target system.

Unfortunately, architects have little support 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 techniques 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 trade-offs 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?

This paper describes our approach to software architecture evolution planning. It is organized as follows. Section 2 describes the basic concepts underlying our approach. Section 3 describes more formally how we model evolution plans and how we define analyses of evolution plans. Section 4 proves some theoretical results about our language for specifying architecture evolution constraints. Section 5 presents a case study illustrating how these ideas can be applied to an actual software architecture evolution. Section 6

J. M. Barnes (✉) · D. Garlan · B. Schmerl
Institute for Software Research, Carnegie Mellon University,
Pittsburgh, PA, USA
e-mail: jmbarnes@cs.cmu.edu

D. Garlan
e-mail: garlan@cs.cmu.edu

B. Schmerl
e-mail: schmerl@cs.cmu.edu

discusses related work. Finally section 7 discusses future work, and section 8 concludes.

2 Our approach to architecture evolution

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;
- Trade-off analyses can be performed over alternative evolution paths to optimize expected value under uncertainty; and
- The evolution path concept is amenable to support by tools that architects can use to describe, analyze, track, and modify plans for architecture evolution.

A central idea behind our approach is the concept of an *evolution style*. An evolution style describes 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 correctness criteria and guidance (based on past experience in the domain) for an architect developing an evolution plan in that family. Moreover, we can support reasoning about the extent to which a specific path satisfies the quality and cost objectives in a particular business context.

To illustrate what we mean by an evolution style, consider the following typical scenarios: 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 [56]).

Each of these examples describes a class of evolutions that addresses 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 contain 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 a planned evolution 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 described in terms of architectural transformations that are selected from a predefined set of evolution operators. In this respect an evolution model is like a state machine for which an execution trace defines an evolution path.

An evolution style, then, provides the vocabulary of concepts necessary to define and analyze such an evolution model: the set of operators that are available to define the evolution transitions (which represent the evolution operations that can be carried out in the domain at hand), a set of evolution path constraints that define which paths are permissible in the evolution style (which capture things like ordering constraints or invariants that must hold for all nodes), and a set of evaluation functions that can be used to compare different evolution paths with respect to quality metrics (which are used to facilitate selection of an optimal path).

2.1 Example

To illustrate the concepts and benefits of our approach, consider the following fictitious, but representative, scenario: Company C runs an algorithmic-trading platform with an aging software architecture. Its clients, mostly fund managers, use the platform to research, develop, and execute high-frequency trading algorithms. Currently, these various features are accessed via separate web interfaces, which are showing their age. Input of trading algorithms to be executed is accomplished via one interface, retrieval and analysis of market data through another. A third interface allows clients to download a desktop analysis toolkit that they can use to backtest candidate trading algorithms. The interfaces are separated in this way for historical reasons, and while the separation makes maintenance easy, clients hate it; they would rather be able to research market history, backtest possible trading strategies, define algorithms, and activate algorithms for execution on one site. The current architecture has other problems too. First, while software maintenance is easy, maintaining the hardware is quite expensive. Indeed, many of the components of the system are hardwareintensive. Running the trading algorithms, for example, is quite processor-intensive, while storing the company's vast archive of market data requires a great deal of disk space. The hardware requires frequent upgrades; the company must have top-of-the-line computing hardware to keep up with its competitors. In addition, there are significant demand spikes, and the hardware the company has cannot always keep up. Recently, a hardware failure brought down one of the web dashboards for two full business days, enraging clients.

To address these concerns, Company C is considering migrating to a cloud-based architecture. In the cloud-computing model, computing resources are sold over the Internet as services. For example, rather than maintaining its own infrastructure, a company can pay a cloud service to provide the infrastructure for them. Concretely, what this means is that a cloud platform like Amazon Web Services will host Company C's software systems on its own infrastructure, providing whatever resources Company C needs (and is willing to pay for): data storage, computing capacity, bandwidth, and so on. But unlike a traditional hosting environment, cloud-computing resources are sold on demand (e.g., computing capacity is sold by the hour; storage is sold by the gigabyte-month) and are provisioned elastically (so a customer can have as much or as little of a service as needed).

By making use of such a platform, Company C could host its software in the cloud rather than maintaining infrastructure in-house—effectively outsourcing their hardware maintenance while retaining control of their software. Such a migration could solve many of the company's problems. Reliability would be assured by the cloud provider's service-level agreements. Hardware upgrades could be effected immediately on request. Resources could be increased on demand in response to usage spikes. Specialized hardware suitable for specific applications, such as high-CPU hardware for trading-algorithm execution or high-memory hardware for backtesting, is easy to provision. If the company's needs change in the future, its infrastructure can change with them; the company is not locked into the infrastructure that it owns. Finally, Company C could focus on its business of developing a great trading platform rather than the day-to-day problems of managing infrastructure. Also as part of the migration, the company plans to merge its multiple separate user interfaces to create one unified client experience.

The particulars of this example are contrived, but the scenario is a common one. Cloud computing is a hot topic in the electronic-trading community, and trading platforms are increasingly asking how they can use cloud architectures to improve reliability, increase scalability, and allow themselves to focus on their core expertise rather than the business of keeping hardware running [40]. Moreover, although we chose the algorithmic-trading domain for this example, largely the same concerns apply to a much broader category of systems. A great many organizations are migrating to the cloud, or contemplating migrations to the cloud, to address these same concerns.

Because of the system's complexity, the chief architect at C needs to develop a plan to carry out the evolution 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 migrating in-house ad hoc web applications to cloud-computing environments. Capitalizing on past experience in this area, the evolution style 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 some of the existing connections of the web application to continue to be present alongside the newly added connections to and within the cloud environment. Additionally, the style would identify a set of structure- and behavior-changing operations. Examples include the migration of data from an in-house database to a cloud data store, introduction of adapters as necessary to allow legacy subsystems to exist in the cloud, and reprovisioning of hardware within the hub as the company ramps up the nascent cloud architecture to full production capacity. Finally, the style 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; data should be migrated before applications; all the old componentry should continue to exist for at least a week after the new cloud environment becomes accessible to users, so that fallback to the old system is possible.

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 (namely the one just described). He would then start to define an evolution path. Likely the starting point for this would be characterization of the initial and target architectures. Existing tools make it relatively easy to specify these using standard architecture modeling and visualization techniques. Figure 1 illustrates a simplified version of the initial and target states. At this point the evolution tools would check that these two architectures satisfy the pre- and postconditions required by the style, 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 style-defined operators to the architecture to produce a first release—for example, by building a skeletal cloud application as an initial release. The tools would check that the release is well formed and that the path satisfies the constraints of the style, warning

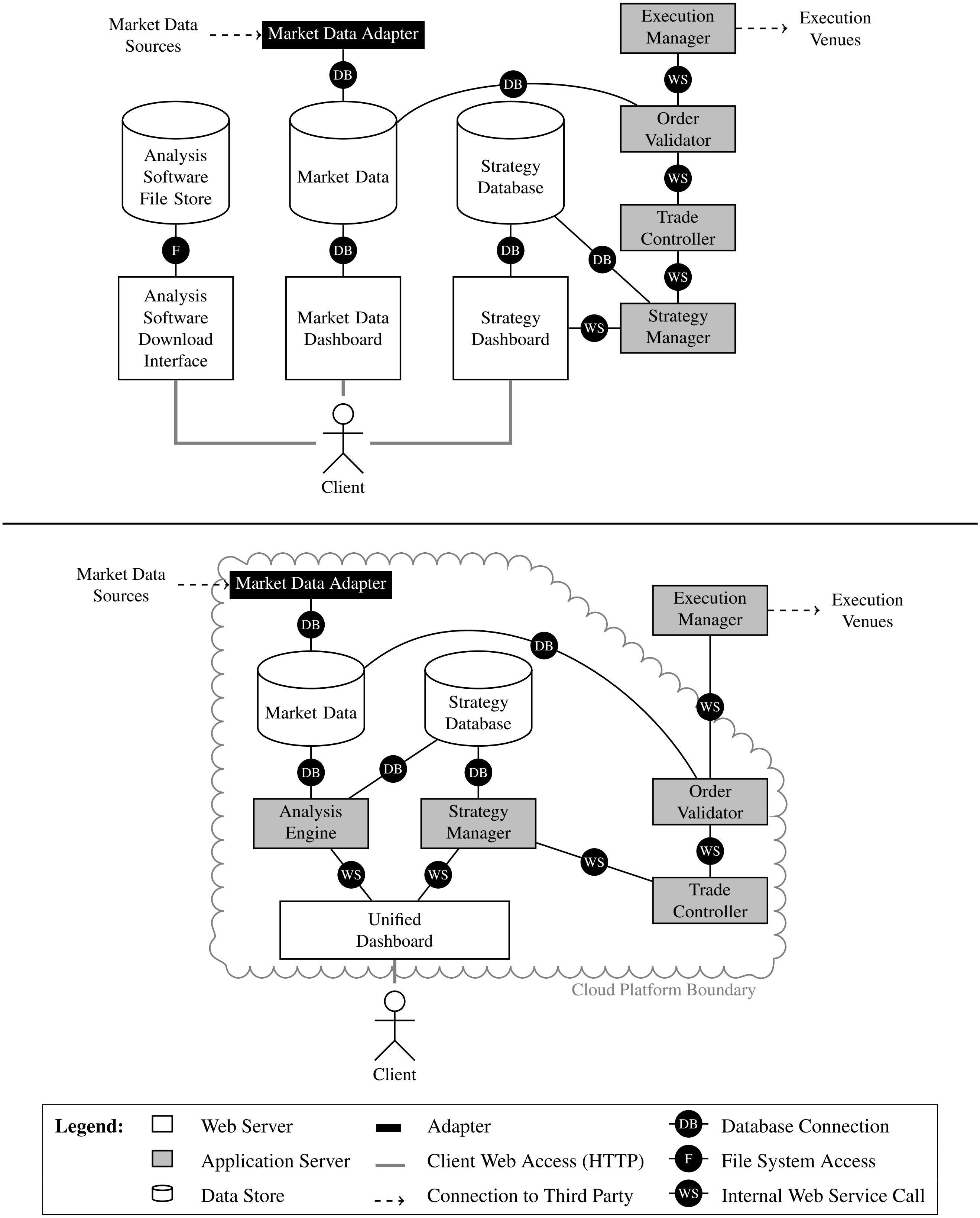


Fig. 1 Examples of component-and-connector views of architectural instances: initial architecture (top) and target architecture (bottom).

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 trade-offs, 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, define weighted utilities, and provide estimates of costs and durations (including uncertainties). With these annotations in hand, the tools calculate costs and utility, allowing the architect 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.

3 Modeling an evolution style

In the previous sections we outlined what we mean by *evolution paths* and *evolution styles* and provided an informal example illustrating how these concepts are useful in planning evolution in a particular domain. In this section, we describe the technical basis of our approach.

3.1 Specifying architectures

Our approach presumes that software architectures are documented in some formal modeling language, such as an architecture description language (ADL). Such languages provide a rigorous way of representing the structure of a software system. Different languages capture different concepts, but in general an architecture is represented as a graph, where the vertices represent the pieces of a software system (such as modules or components) and the edges describe how these pieces relate (e.g., edges may represent uses or attachments) [16,45,54,59]. In addition, there may be auxiliary elements like *ports* and *roles*, as well as *properties* that describe the characteristics of the architectural elements (often used to express things like reliability and protocols of interaction), and they may support expression of constraints on and analyses over architectures.

Our approach admits multiple *views* of an architecture at each stage of evolution. An architectural view is a particular representation of, or perspective on, a software architecture. Clements et al. [16] identify three basic types of views: module views (which document code structure), component-and-connector views (which document the structure of the running system), and allocation views (which document the deployment and execution context of the software). Documenting a software architecture completely requires the use of multiple views. Different projects require different views; the choice of views used for a software project should be guided by the kinds of analysis that are needed. Our approach allows architects to plan the evolution of a software architecture from multiple perspectives by representing multiple views of intermediate architectures. This makes possible constraints and analyses that make reference to multiple views. For example, “Component A shall not communicate with component B until database replicas are deployed to three separate geographic locations” is a constraint that requires examination of both a component-and-connector view and an allocation view. Architects should carefully select the views to include in the evolution plan based on the analyses they anticipate; representing additional views can add significant cost to the planning process, since each view must be documented for each candidate intermediate architecture. (Future work may reduce this burden; see section 7.)

Our approach is not tied to any particular modeling language (although any tool that implements our approach will be). In previous work [26,28], we have focused on Acme [27]. Many of the examples of architectural description in the present paper, including the examples in the case study in section 5, are based on UML 2 [49]. While UML lacks some of the more sophisticated architectural analysis features of Acme, it has the advantage of providing diagram types that support multiple views of an architecture, which allows us to demonstrate the aforementioned multiple-view feature of our approach. (Acme focuses on providing rich component-and-connector descriptions of architectures.)

3.2 Specifying families of architectures

To represent families of architectures we use the established notion of *architectural styles*. An architectural style is defined by specifying a vocabulary of architectural element types, together with a set of constraints that determine how instances of those types can be composed into systems [16, 59]. Some ADLs, such as Acme, provide native support for architectural styles. UML does not have any explicit notion of architectural styles, but they can be expressed in UML either through the use of UML *profiles* or in a makeshift fashion by defining a set of element types accompanied by constraints in UML’s constraint language, OCL. (Clements et al. [16] discuss in greater detail the use of UML for software architecture modeling.)

3.3 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 that can be used by constraints and analyses. An evolution style specifies the list of properties that are expected to be given values on the nodes and transition. For example, each node may need to specify whether it is intended to be a public release, or what the expected impact of the node on the market should be; transitions may provide information about the expected amount of time the transition will take, how many developers are required, or whether training will be needed. 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 has not yet involved training.

3.4 Relating architectures in a path

The nodes in an evolution path do not exist in isolation. The architectural snapshots that make up a path are, of course, closely related to each other. One way of understanding these relations is by means of the evolution operators that make up the transitions, as described in section 3.6. But sometimes it is also useful for constraints and analyses that operate over intermediate architectural representations to have direct access to the way those intermediate architectures relate. Given a node N_i and its successor N_{i+1}, typically most of the architectural elements in N_i and N_{i+1} will be identical. For almost every element in N_i, there is an identical counterpart in N_{i+1}, because most of the architecture did not change between steps i and $i + 1$. In analyzing the evolution, it is often helpful to know which component in N_i corresponds to which component in N_{i+1}. Of course, this information is normally self-evident from a glance at the architectural diagrams of the two nodes, but as we will see in examples later, it is often helpful to have easy analytical access to this information.

Therefore, we introduce a notion of *evolutionary identity*. Two model elements in different nodes along an evolution path are evolutionarily identical if they refer to the same architectural element at different stages of evolution. Labeling evolutionarily identical elements is not only useful, as it enriches our ability to express constraints and analyses over evolution paths, but also easy; a tool could automatically label evolutionarily identical elements by inferring the evolutionary identity relations from evolution transition metadata, so there need be no extra burden on the architect.

There are various ways in which we could model evolutionary identity, but the particular modeling mechanism makes little difference. One simple option is to define an *evId* (*evolutionary identity*) property on every architectural element of every intermediate architecture. Evolutionarily identical elements would be assigned the same *evId* (which could be as simple as a random numeric identifier). In UML, to add such a property to every element, we would define a profile such as the one in figure 2, which extends the UML metamodel by allowing an *evId* property to be set on any element.[1]

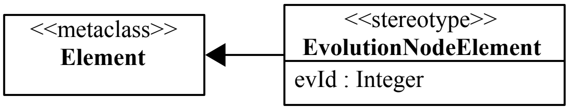


Fig. 2 UML profile view showing how an evolutionary-identity property can be defined so it can be set on any model element. In the UML metamodel, all diagram elements are specializations of *Element*, so it suffices to stereotype this single metaclass.

The same basic approach can be used to support other notions of relations among nodes. For example, if during an evolution a component c is split into two components c_1 and c_2, then neither c_1 nor c_2 can be said to be evolutionarily identical to c. But we still might be interested in noting that c_1 and c_2 *derive from* c. To support analyses that need to make use of this information, we would introduce an evolutionary relation indicating whether one architectural element derives from another.

3.5 Specifying and using path constraints

Path constraints are used to identify which evolution paths are permissible in an evolution style (or in a specific evolution). They can be used, for instance, to restrict releases to being in a particular family, define evolution dependencies (e.g., require certain architectural structures to be in place before other operations are performed), or preserve invariants across releases. For example, in the above scenario, path constraints might require that no connections to third-party services are disrupted in any release.

We use an augmented version of linear temporal logic (LTL) to specify these path constraints. Temporal logic is a

[1] In section 3.2, we mentioned that architectural styles can be captured by means of UML profiles. In this case, these architectural-style profiles should extend the profile of figure 2.
Incidentally, UML profiles could be used much more extensively than the way we present here. We could use them to define a much more elaborate base style that serves as a specialization of UML to formal software architecture modeling. That is, we could define a general profile for representing software architectures (or perhaps several profiles for different architectural views). However, the problem of defining a UML profile for software architecture is not particularly relevant to the topic at hand, is not necessary for the case study we present in section 5, and has already been explored in previous work, so we do not do so here.

natural choice, since 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.

We begin with the usual LTL operators, including:

- $\square$, *always*, to represent invariant properties of paths;
- $\lozenge$, *eventually*, to represent the existence in a path of an architecture with certain properties;
- $\mathcal{U}$, *until*, to represent properties that must remain true of a path until some other property becomes true; and
- $\bigcirc$, *next*, to represent properties that must be true in the next step of the path.

Ordinary LTL is sufficient to express many interesting properties. For example, suppose we want to specify (in the example in section 2.1) that the analysis software file store will not be removed until the analysis software download interface is removed first. We can represent this constraint as follows:

$$\begin{aligned}&\mathit{softwareFileStorePresent}(\mathit{system})\\ &\qquad\qquad \mathcal{U}\ \neg\mathit{softwareDownloadUiPresent}(\mathit{system})\end{aligned}$$

Here, *softwareFileStorePresent* and *softwareDownloadUiPresent* are predicates over systems, defined by the evolution style; *system* is a keyword that refers to the system associated with the current state. Note that each of the predicates is expressible with respect to a single state. For example, *softwareFileStorePresent* can be defined with an OCL constraint:

```
DataStore.allInstances()->exists(s | s.name =
        "AnalysisSoftwareFileStore")
```

Now consider a richer 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 way appropriate to the domain at hand). If we try to express the constraint in LTL, we quickly encounter a problem.

$$\square(\mathit{release} \rightarrow \square\mathit{hasAllFunc}(\mathit{system}, ?)) \tag{1}$$

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

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

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. Because of the finite nature of paths, it is possible to check whether a given evolution path satisfies a given set of evolution constraints. Thus verification of path constraints can be automated. In section 4, we will treat the path constraint language in detail, discussing a number of related logics and proving results about the language's tractability.

3.6 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 2.1 included operators to migrate a component to a cloud, to introduce a wrapper for a legacy component, and so on. These operators form the steps in an evolution path.

Concretely, an evolution operator comprises the following parts: (1) a description of the *structural changes* that the operator effects; (2) *preconditions* describing the conditions under which the operator may be applied; and (3) *additional information used to support analyses*—for example, information on the cost of carrying out the evolution step or the amount of time required. Thus, the definition of an evolution operator takes the form

```
operator operatorName(parameters) {
    transformations {
        // A description of the structural changes that
        // the operator effects
    }
    preconditions {
        // The conditions under which the operator
        // may be applied
    }
    analysis {
        // Additional information to support analysis
    }
}
```

We now describe each of these parts in turn.

The structural changes of the operator are defined by means of *elementary transformations*. An elementary transformation represents a basic structural change to an architectural model. Thus, elementary transformations are things like adding a component, deleting a port, renaming a connector, and modifying a property. For example, an evolution operator might be something like "Wrap a legacy component as a web service," which is a single operator from an architectural standpoint but actually requires a number of elementary transformations: introduce a new wrapper service, put the legacy component inside it, reconnect the ports appropriately, and so on. Since elementary transformations are basic changes to architectural *models*, the elementary transformations available depend on the modeling language in use. In the case of Acme, the elementary transformations will be things like adding a component, deleting a port, renaming a connector, and modifying a property. In the case of a UML deployment diagram, they will include deploying an artifact to a node and adding a communication path.

These examples are *abstract elementary transformations*. To make an abstract transformation like “delete a port” concrete, we have to specify which port we want to delete. “Delete port *foo*” is thus a *concrete elementary transformation*. Other transformations are more complex. For example, component creation requires us to specify not only the name of the component to be created, but also its type and properties. Formally, a concrete elementary transformation maps architectures to architectures; a concrete elementary transformation is a partial function over architectures.

Our operator specification language provides a simple means of expressing elementary transformations, which we illustrate by example:

```
create Device lb1:BladeServer
```

This is a concrete elementary transformation (based on the abstract transformation *create node*) for a UML deployment diagram. It creates a device node of type *BladeServer* whose name is *lb1*. The name *Device* refers to the UML metamodel. Any concept in the UML metamodel can be referenced similarly, and we likewise have syntax for establishing attributes, associations, and so on. Thus, UML models can be transformed in arbitrarily sophisticated ways.

Elementary transformations are composed to describe the structural changes that are effected by the operator. In addition to sequential composition of elementary transformations, our operator description language provides for simple control flow mechanisms such as loops. Here is an example of an operator that wraps a legacy component in a web service, this time in Acme rather than UML. It illustrates several of these concepts.

```
// This operator takes a legacy component, c, and
// wraps it in a new component of type WrapperService.
// The legacy component is moved inside a
// representation of the new wrapper component.
operator wrapLegacyComponent(c) {
    transformations {
        Component wrapper = create Component
                : WrapperService;
        for (Port p : c.ports) {
            Port pw = copy p to wrapper;
            for (Role r : p.attachments) {
                detach p from r;
                attach pw to r;
            }
        }
        Representation rep
                = create Representation of wrapper;
        move wrapper to rep;
        for (Port p : c.ports) {
            bind p to wrapper.ports[p.name];
        }
    }
    ...
}
```

UML has its own standard transformation language, QVT [48], which we could use to specify operators. Acme, however, like many ADLs, has no standard transformation language. While using QVT would be a perfectly workable approach for specifying operators when using UML, here we have consistently used the language described above, which has the advantage of being easily applicable to arbitrary modeling languages. In a pure UML context, QVT would be a reasonable choice. However, as these examples illustrate, our approach is not tied to a particular modeling language, so here we have opted for a modeling-language-neutral means of specifying operators.

As we pointed out in section 3.1, sometimes we want to describe evolution from multiple *views*, using more than one diagram type. In this case, an operator must include the structural changes for each view that is under consideration. The above example includes only one view; we will see an example with multiple views later.

In addition to transformations, an operator can include preconditions and analysis information. Preconditions are expressed as constraints over an architecture. An operator can be applied to an intermediate architecture only when that architecture meets the preconditions. Preconditions are expressed in the constraint language of the diagram language. When multiple views are under consideration, different preconditions can be specified for each view.

Analysis information is less structured than the *transformations* and *preconditions* section. The kinds of analysis information included with an operator are dependent on the analyses that the evolution style supports. To accommodate as much flexibility as possible, we allow the *analysis* block to contain arbitrary information in JavaScript Object Notation [17], which can be freely referenced in the evaluation functions described below.

3.7 Specifying and using evaluation functions

With the facets of evolution styles just described, an architect can define paths that are technically correct, taking into consideration operators and checking path constraints. However, the real benefit of defining paths for architecture evolution is to be able to compare them and decide which path is the best to take—where the meaning of *best* is dependent on the domain and the goals of the specific evolution at hand.

To enable this kind of optimization, we introduce *evaluation functions*, which provide a way of analyzing the utility of each evolution path under consideration. Conceptually, evaluation functions are similar to path constraints, but while a path constraint provides a hard, yes-or-no judgment about the *validity* of a path, an evaluation function provides a quantitative characterization of its *goodness*.

An evaluation function works by examining the properties assigned to the nodes and transitions in an evolution path (as described in section 3.3), as well as (possibly) the architectural content of the nodes; quantifying, weighting, and combining this information in some way; and producing a number that represents some quality of the path. Thus, there could be an evaluation function that estimates the total cost of an evolution path by analyzing relevant properties of the transitions that make up an evolution path. Another evaluation function could evaluate the architectural risk of a plan of evolution by analyzing the architectural changes between nodes in the evolution path.

Because evaluation functions are specific to an evolution style, they can take advantage of the specifics of the domain. Thus, a cost analysis for the service-oriented architecture domain could take advantage of expert knowledge about the costs of different service-oriented architecture operations, such as wrapping a legacy component in a service or introducing an enterprise service bus.

Ultimately, the goal of modeling paths of architecture evolution is path selection. To that end, a utility evaluation function is defined to assess the overall utility of each evolution path. Such a utility function will typically combine results of other, simpler utility functions to produce a measure of the overall goodness of an evolution path based on the evolution goals. In an evolution with stringent deadlines but ample resources, for example, a utility function would give great weight to estimated evolution duration and little weight to estimated cost. (Cost and time are just examples; there may be many other concerns, such as minimizing downtime during evolution or minimizing the number of releases.) Evaluation functions allow software architects to trade off various concerns intelligently.

4 Theoretical properties of our model

In section 5, we will present a case study of the approach we have just described, demonstrating its application to an example evolution. First, however, it is useful to conduct a *theoretical* analysis of some of the properties of our modeling apparatus. In this section, therefore, we evaluate our path constraint language by establishing a theoretical foundation for it and then using this foundation to evaluate its tractability (more precisely, the computational complexity of evaluating a path constraint).

We focus on our path constraint language in particular because it is the most theoretically interesting part of our modeling apparatus. Other parts are either trivial and uninteresting (e.g., operator preconditions and local judgments about architectural styles) or too general to say anything about (e.g., evaluation functions, which provide the evolution planner with what is essentially a general-purpose programming language, so the complexity of evaluation is entirely dependent on the complexity of the evaluation function that the planner chooses to write).

Sections 4.1 and 4.2 present a formal syntax and semantics, respectively, for our path constraint language. In section 4.3, we place this language in context by discussing a number of other, similar logics that have likewise been formed by supplementing LTL with a variable-binding operator, and we identify where the important differences lie. Section 4.4 contains the complexity result.

4.1 Syntax of our path constraint language

We introduced our path constraint language informally in section 3.5. As we said there, it is based on LTL, which has a very simple syntax:

$$\phi := p \mid \mathit{false} \mid \phi_1 \rightarrow \phi_2 \mid \bigcirc\phi \mid \phi_1 \,\mathcal{U}\, \phi_2$$

for propositional symbols p. Other connectives can be defined in terms of these, for example $\neg\phi := \phi \rightarrow \mathit{false}$ and $\Diamond\phi := \mathit{true}\,\mathcal{U}\,\phi$ and $\Box\phi := \neg\Diamond\neg\phi$.

Such a simple definition will not work for our path constraint language. We need to be concerned not only with ordinary propositions, but also with predicates and functions. A condition such as “The software architecture has at least one component” can be represented as a proposition p. But a more interesting condition such as “This software architecture has at least the same database components as the one in the previous state” expresses a relation over two different architectures. So in defining a syntax, we need to recognize that there are *atomic formulas* other than merely propositional symbols. In this respect, it is similar to first-order predicate logic (FOL), and so in formalizing the syntax for our path constraint language, we take a cue from FOL, giving separate, inductive definitions for *terms*, *atomic formulas*, and finally formulas:

Definition 1 (path constraint syntax) *Let V be a set of variables. For $n = 0, 1, 2, \ldots$, let F_n be a set of n-ary function* *symbols and let P_n be a set of n-ary predicate (relation) symbols. Together these sets form a* signature*, which we write as $\Sigma = (V, (F_n)_{n\in\mathbb{N}}, (P_n)_{n\in\mathbb{N}})$. (This is just as in FOL.)*

The terms π, atomic formulas α, and formulas ϕ are defined inductively:

$$\pi := x \mid f(\pi_1, \ldots, \pi_n)$$
$$\alpha := p(\pi_1, \ldots, \pi_n)$$
$$\phi := \alpha \mid \mathit{false} \mid \phi_1 \rightarrow \phi_2 \mid \bigcirc\phi \mid \phi_1 \,\mathcal{U}\, \phi_2 \mid \{x\}\phi$$

for $x \in V$, $f \in F_n$, and $p \in P_n$.

To make this discussion more concrete, let us analyze the example formulas from section 3.5 in terms of this syntax. In the first example,

$$\mathit{softwareFileStorePresent}(\mathit{system}) \;\; \mathcal{U} \neg \mathit{softwareDownloadUiPresent}(\mathit{system}),$$

softwareFileStorePresent and *softwareDownloadUiPresent* are unary predicates. The keyword *system* is a nullary function: it takes no arguments and behaves as a term. (This might be surprising, since in FOL we often refer to nullary functions as "constants," and *system* does not hold constant—on the contrary, it refers to something different in every state. But in a temporal context, *constant* is not a very good word for a nullary function, because a nullary function can refer to different states depending on the current state, just as a nullary predicate—a proposition—can have a different truth value from state to state. For a different approach, see half-order modal logic [35], where functions have a "rigid"—state-independent—interpretation and predicates have a "flexible" interpretation.) Similarly, in the formula

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

release is a proposition (nullary predicate) and *hasAllFunc* is a binary predicate.

4.2 Semantics of our path constraint language

We will now give a Kripke-style semantics for our path constraint language. Again we begin by recalling the semantics of LTL. There are various ways to formalize the semantics of LTL. In the following formalization, we identify a state with an interpretation of the propositional symbols. Alternatively we could externalize an interpretation function as a map from states to sets of propositions.

The Kripke semantics for LTL is as follows. Let P be a set of proposition symbols. Let σ be a sequence of states: $\sigma_1, \sigma_2, \ldots$, where $\sigma_i \subseteq P$ for each i. (Thus, each state comprises the set of proposition symbols that are interpreted to hold true in that state.) We write $\sigma, i \vDash \phi$ to say that σ satisfies the LTL(P) formula ϕ at a time $i > 0$. We define this satisfaction relation inductively:

- $\sigma, i \vDash p$ iff $p \in \sigma_i$ (i.e., iff the propositional letter p is true under the interpretation given by σ_i).
- $\sigma, i \vDash \mathit{false}$ never holds.
- $\sigma, i \vDash \phi \rightarrow \psi$ iff $\sigma, i \vDash \phi$ implies $\sigma, i \vDash \psi$.
- $\sigma, i \vDash \bigcirc\phi$ iff $\sigma, i + 1 \vDash \phi$.
- $\sigma, i \vDash \phi \,\mathcal{U}\, \psi$ iff there is some $j \geq i$ with $\sigma, j \vDash \psi$ such that $\sigma, k \vDash \phi$ whenever $i \leq k < j$.

There are a number of things we must change to obtain a semantics for our path constraint language. First, LTL normally models a sequence of infinitely many states, $\sigma_1, \sigma_2, \ldots$. We, however, are interested in expressing constraints over a finite sequence of states: the evolution path, which comprises finitely many intermediate software architectures. So we must restrict ourselves to a finite sequence of states.

The second change we need to make is to account for our additions to the syntax. Atomic terms are much richer than they are in LTL. Again we take a cue from FOL. In propositional logic, an interpretation is simply an assignment of truth or falsehood to each proposition symbol. But in FOL, an interpretation is a map that assigns a function to each function symbol and a relation to each predicate symbol. Similarly, in our path constraint semantics, a state now needs to be more than simply an identification of which propositional letters are true; it should be an interpretation function that maps the function and predicate symbols of the syntax to functions and relations. (In FOL, these are functions and relations on the domain of quantification; for us they are functions and relations on the temporal states.)

Finally, we need to express the semantics of our new variable-binding operator, which means we must keep track of what states the variables are binding to. After making all these changes, we obtain the following semantics.

Definition 2 (path constraint semantics) *Let*

$$\Sigma = (V, (F_n)_{n\in\mathbb{N}}, (P_n)_{n\in\mathbb{N}})$$

be a signature. Let σ be a sequence of states, $\sigma_1, \sigma_2, \ldots, \sigma_n$. As in LTL, we define a state *to be an interpretation, but now each state σ_i is a function that maps each function symbol to a function over states and each predicate symbol to a relation over states. That is,*

- *If $f : F_n$, then $\sigma_i(f) : S^n \rightarrow S$, where S is the set of states.*
- *If $p : P_n$, then $\sigma_i(p) \subseteq S^n$.*

A variable assignment *$s : V \rightarrow S$ is a function that maps variables to states. (This is needed to keep track of what free variables stand for.) We write $\sigma, i, s \vDash \phi$ to say that σ*

satisfies the path constraint ϕ at time $i \in \{1, \ldots, n\}$ under the assignment s. We define the denotation *of a term π in the structure σ at time i under assignment s, written $D_{\sigma,i,s}(\pi)$, by*

- $D_{\sigma,i,s}(x) := s(x)$
- $D_{\sigma,i,s}(f(\pi_1, \ldots, \pi_n)) := (\sigma_i(f))(D_{\sigma,i,s}(\pi_1), \ldots, D_{\sigma,i,s}(\pi_n))$

Finally, we define the satisfaction relation inductively:

- $\sigma, i, s \vDash p(\pi_1, \ldots, \pi_n)$ *iff* $(D_{\sigma,i,s}(\pi_1), \ldots, D_{\sigma,i,s}(\pi_n)) \in \sigma_i(p)$
- $\sigma, i, s \vDash$ *false never holds.*
- $\sigma, i, s \vDash \phi \to \psi$ *iff* $\sigma, i, s \vDash \phi$ *implies* $\sigma, i, s \vDash \psi$.
- $\sigma, i, s \vDash \bigcirc\phi$ *iff* $i = n$ *or* $\sigma, i + 1, s \vDash \phi$.[2]
- $\sigma, i, s \vDash \phi\, \mathcal{U}\, \psi$ *iff there is some* $j \in \{i, i + 1, \ldots, n\}$ *with* $\sigma, j, s \vDash \psi$ *such that* $\sigma, k, s \vDash \phi$ *whenever* $i \le k < j$.
- $\sigma, i, s \vDash \{x\}\phi$ *iff* $\sigma, i, s[x \mapsto \sigma_i] \vDash \phi$ *(where* $s[x \mapsto \sigma_i]$ *is the same assignment as s except with x now assigned to* σ_i*)*

If ϕ is a closed sentence (i.e., has no free variables) then the assignment s is irrelevant and we may simply write $\sigma, i \vDash \phi$.

In the end, we get something that looks more like the semantics of FOL than that of LTL. Indeed, as we will see in the next section, some authors who have introduced similar logics have referred to such a variable-binding construct as a special kind of quantifier; this formalization shows why this is appropriate.

4.3 Similar logics

It is important to understand how our path constraint language relates to the existing landscape of temporal logics, both to provide a context of related work and to clarify the significance of the results in section 4.4. Indeed, there are several logics that are similar to ours with respect to our introduction into LTL of variables that retain their values across states.

A logic that is very similar to ours is one developed by Richardson [55] to support lists in an object-oriented data model. Richardson's logic is essentially LTL with the addition of "rigid variables," which are semantically identical to our extension to LTL. Richardson's logic does not appear to have been studied for its theoretical properties; the paper has not been widely cited outside the database community. "Rigid variables" appear elsewhere in the literature as well, most famously Lamport's temporal logic of actions [39], although Lamport's rigid variables are somewhat different.

Another related logic has been developed to model real-time systems. A natural way to specify real-time systems is with LTL, but one problem that arises is the incorporation of hard real-time requirements. Alur and Henzinger [2] developed a logic that they called timed propositional temporal logic (TPTL), whose main feature was the introduction of *freeze quantifiers*, which bind a variable to a particular time so that it can be accessed later. These are similar to our rigid variables. There are a couple of differences, which we describe under the next subhead.

Yet another related logic is Goranko's temporal logic with reference pointers [33], which differs in a couple of ways. First, unlike Richardson, Alur and Henzinger, and us, who were devising specification languages for particular domains (object-oriented data models, real-time systems, and software architecture, respectively), Goranko is philosophically motivated. He notes that LTL lacks a way to refer to particular points in time—to express the concept "then." Unlike the other logics we have seen, which give explicit names to states, Goranko's logic simply uses the symbol ↓ to indicate a point that we might refer to later, then uses ↑ to refer to it (to say "then"). Syntactically, ↑ behaves like a propositional variable; semantically, ↑ is true if the current time is the same as the time of ↓. Goranko uses this to express things like "now will not occur again": $\downarrow\Box\neg\uparrow$.

A final related family of logics is hybrid logic [9, 10], where states can be referred to via labels called *nominals*. A nominal is an atomic symbol with the special property that it is true at exactly one state. We can also use a nominal a to build *satisfaction statements*, which have the form $@_a\phi$, which means "ϕ is true relative to the state characterized by a." Finally, often hybrid logics are supplemented with a ↓ binder, which binds a label to the current state, much like our rigid variables (or a named version of Goranko's ↓). The result is powerful. For example, we can now define *until* in terms of these hybrid logic constructs:

$$\phi \text{ until } \psi \quad := \quad \downarrow x(\Diamond\downarrow y(\psi \wedge @_x\Box(\Diamond y \to \phi)))$$

Hybrid logics are rather different from our logic, but the basic idea of named states, and in particular the ↓ binder, are closely related.

This is not an exhaustive list; Blackburn and Tzakova [10], for example, cite a few others, observing, "Labeling the here-and-now seems to be an important operation." Indeed, the idea seems to have been reinvented numerous times.

How our path constraint language differs. Our path constraint language fits comfortably within this family of related logics. Operators that bind variables are nothing new. However, there are some ways in which our path constraint language distinguishes itself semantically from its cousins. Although these distinctions are subtle, they turn out to have, in some cases, major theoretical consequences. Although

[2] This is "weak next," meaning that $\bigcirc\phi$ is interpreted to be true in the final state of a sequence. "Strong next" can be defined in terms of the weak next operator: $\bar{\bigcirc}\phi := \neg\bigcirc\neg\phi$.

the existing literature is rich, it is also somewhat patchy. There are interesting problems that have yet to be tackled. Notably, the question of the complexity of *model-checking a path*, which we discuss below and resolve in section 4.4, is a natural one that has been solved for LTL and a number of extensions to LTL [44] but not for LTL with a variable-binding operator. In the remainder of this section, we will discuss the differences between our path constraint language and other logics. We will focus on the two related logics that are the most mature and best-studied: TPTL and hybrid logic.

TPTL was invented by Alur and Henzinger to model real-time systems, but (likely due to their extensive theoretical characterization of their new "freeze quantifier" and their generalization of the idea beyond their domain) their work became quite influential outside of this field. The semantics of TPTL differs from our path constraint language in two important ways. First, TPTL, like LTL, assumes an infinite sequence of states; our logic assumes a finite sequence. Second, the variables that freeze quantifiers capture are times (natural numbers) rather than architectural models. All they do with the variables they freeze is compare them to other times with operators like ≤; we want to do architectural analysis. At first glance, these seem to be peripheral issues, but they turn out to be important. Indeed, Alur and Henzinger themselves showed that small changes to the language can substantially change its theoretical properties; for example, supplementing TPTL with addition over time renders the satisfiability problem highly undecidable.

Our results agree with this generalization. These seemingly subtle semantic changes have substantial ramifications on the theoretical properties of the language, and not always in the way that one would expect. For example, consider the problem of *model-checking a path*: evaluating whether a formula holds for a single path. Alur and Henzinger show that this problem is EXPSPACE-complete for TPTL, further noting that it would be undecidable if TPTL were modified to allow more powerful atomic propositions, such as addition over terms. Our path constraint language, on the other hand, goes so far as to allow atomic propositions made up of *arbitrary* predicates over arbitrary terms (just like FOL), but then we regain decidability by studying finite sequences rather than infinite sequences. Unsurprisingly, this completely changes the problem of model-checking a path. What is perhaps surprising is that the problem is still hard and interesting. The problem of model-checking a single, finite path might be naively thought to be rather trivial, but Markey and Schnoebelen [44] suggest that in fact the problem is of substantial theoretical interest. Our results support their contention. As we show in section 4.4, the problem of model-checking an evolution path constraint on a finite path is hard (PSPACE-complete). Not only that, but this complexity result does not lend itself to easy proof; a fairly sophisticated reduction strategy was necessary to prove PSPACE-hardness.

Much the same can be said of hybrid logic. Although hybrid logic *can* be used to express constraints over finite, linear paths, it does not seem to be done often, at least not often enough that anyone has bothered to study the theoretical properties of that case. And again, although naively we might assume this case to be trivial, boring, or a straightforward specialization of the more general case, in fact interesting and surprising results emerge from these restrictions.

Hybrid logic is quite different in character and focus from our constraint logic. In particular, although the general idea of *nominals* (propositions that are true in only one state and hence uniquely identify that state) is quite central to hybrid logic, the ↓ binder that corresponds to our variable-binding operator is not. Rather, ↓ was a late-breaking addition, imported into hybrid logic from Goranko's temporal logic of reference pointers [33]. This is not to say that the theory of ↓ in hybrid logic is underdeveloped—on the contrary, some remarkable results about it have been published—but the focus of hybrid logic is different from ours. More characteristic of hybrid logic than ↓ is the aforementioned *satisfaction operator*; it effectively jumps to a named state. Not having the satisfaction operator imposes some unexpected challenges. For example, as we note in section 4.4, having the satisfaction operator would make proving the PSPACE-hardness result dramatically less challenging.

Of course, nominals themselves are also somewhat different from bound variables in our language. A nominal is a proposition that is true in exactly one state; a bound variable is a *term* that directly captures a state object. This is an important semantic difference, although the effect is similar.

4.4 Computational complexity

The primary thing we want to do with path constraints, of course, is check whether a given path satisfies a given constraint. This can be easily stated as a model-checking problem. In general, model checking is the problem of checking whether a specific formula is true of a specific Kripke structure. More specifically, model checking is usually used to verify that a state transition system has some property. For some logics, such as CTL, this is easy. For others, such as LTL, it is hard—PSPACE-complete, in fact. That is, given a finite state transition system, determining whether an LTL formula is valid in that transition system is PSPACE-complete [60]. Moreover, the solution to the model-checking problem for LTL is intellectually rather challenging too, involving an intricate tableau construction. There is certainly not much hope that our variable-binding construct will make things any easier, especially in light of the result that model-checking TPTL is EXPSPACE-hard [2].

Fortunately, we are not terribly interested in this form of the problem. Instead, we are interested in model-checking a single, particular path—not verifying a formula over an entire state transition system. Our primary use case is telling software architects whether the paths that they have planned are admissible according to the constraints; we do not need to check all the paths in some transition system, nor even a great number of paths—just one, or a few, at a time. Likewise, all our paths are finite—and in fact rather short, since they are explicitly defined by humans.

Model-checking a single path is a much easier problem computationally, but one that has been recognized in recent years as theoretically interesting [44].[3] For pure, propositional LTL, model-checking a formula of length ℓ on a path of length m takes $O(\ell m)$. The algorithm for single-path model checking for LTL is the same as the familiar algorithm for model-checking CTL, since LTL and CTL coincide over individual paths [44]. This traditional algorithm [15] uses a dynamic-programming approach. To determine whether $\sigma, i \vDash \phi$ for some finite temporal sequence σ and formula ϕ, we list the subformulas of ϕ and solve $\sigma, i \vDash \phi_i$ for each subformula ϕ_i. We begin with the smallest subformulas (i.e., atomic formulas), which are immediately solvable, then inductively solve larger subformulas by using the solved subformulas that compose them.

Things become messier when we add the variable-binding operator, $\{x\}$. This simple dynamic-programming algorithm is no longer adequate, because we also need to keep track of variable valuations. For example, in the formula $\Box\{x\}\Box p(x)$, the truth of $p(x)$ depends not only on the state at which we are evaluating $p(x)$, but also on the value of x. To determine whether the formula holds on a path, we ultimately need to evaluate $p(x)$ at each state *and for each value of* x. The subformula $p(x)$ thus needs to be evaluated $O(m^2)$ times; in model-checking finite LTL paths, we never need to evaluate a subformula more than m times. So it is clear that model-checking our path constraint language will be harder than model-checking finite LTL paths. But how hard?

Consider a formula of the form

$$\Box\{x_1\}\Box\{x_2\}\cdots\Box\{x_k\}\Box p(x_1,\ldots,x_k)$$

The only apparent way to check this formula is to evaluate $p(x_1,\ldots,x_k)$ at each state and for each valuation $x_1 \le \cdots \le x_k$. There are $\binom{m+k-1}{k}$ such valuations, which is $\Theta(m^k)$ for fixed m. Note that k, the number of rigid variables, is asymptotically proportional to the length of the formula, ℓ. Thus, we have to evaluate $p(x_1,\ldots,x_k)$ under $\Theta(m^\ell)$ different valuations. This gives us a strong hint that we have departed the realm of polynomial-time algorithms.

However, we can model-check a path constraint in polynomial *space*, because we only need to work with one valuation at a time. In fact, even the naive recursive algorithm is polynomial-space:

Theorem 1 *There is a polynomial-space algorithm for model-checking a path constraint on a single path.*

Proof Let σ be a temporal sequence of finite length m. Let ϕ be a path constraint of length ℓ. We define a recursive algorithm to determine whether $\sigma, i \vDash \phi$:

```
CHECK($\sigma, i, \phi, s$)
    if $\phi$ is an atomic proposition
        evaluate $\phi$ at $i$ using assignment $s$
    if $\phi = false$
        return false
    if $\phi$ is of the form $\chi \rightarrow \psi$
        return CHECK($\sigma, i, \psi, s$) or not CHECK($\sigma, i, \chi, s$)
    if $\phi$ is of the form $\bigcirc\psi$
        return $i = m$ or CHECK($\sigma, i+1, \psi, s$)
    if $\phi$ is of the form $\chi\, \mathcal{U}\, \psi$
        for $j = i$ to $m$
            if CHECK($\sigma, j, \psi, s$)
                return true
            if not CHECK($\sigma, j, \chi, s$)
                return false
        return false
    if $\phi$ is of the form $\{x\}\psi$
        return CHECK($\sigma, i, \psi, s[x \mapsto \sigma_i]$)
```

To check whether $\sigma, i \vDash \phi$, we call CHECK($\sigma, i, \phi, \emptyset$).

Consider the space complexity of this algorithm. Since every recursive call is of a strict subformula, the stack depth of this algorithm never exceeds ℓ. Each execution of CHECK (excluding the recursion) uses only $O(1)$ space. Thus, the space complexity of the algorithm is $O(\ell)$. □

This shows that the model-checking problem for path constraints is in PSPACE. We will now prove that it is PSPACE-complete. The usual way to prove that a problem is PSPACE-hard is to prove that the quantified-Boolean-formula (QBF) problem reduces to it. The QBF problem is to determine whether a quantified Boolean formula such as

$$\exists x \forall y \exists z (z \wedge x \vee y) \tag{2}$$

is true. (We may assume the QBF is in prenex normal form.)

There is a fairly obvious transformation from QBF to our model-checking problem, but unfortunately this obvious reduction does not work. The obvious reduction is to change the $\forall$s into $\Box$s and the $\exists$s into $\Diamond$s, add variable bindings after

[3] In the program verification community, the problem of checking a finite, single trace is known as *runtime verification*, with the term *model checking* reserved for checking entire state structures [6].

each quantifier, then check the formula on a path of length 2, where the first state represents falsehood and the second represents truth. Thus formula (2) would become

$$\Diamond\{x\}\Box\{y\}\Diamond\{z\}(t(z) \wedge t(x) \vee t(y))$$

where t is a predicate that is false when its argument refers to state 1 and true for state 2. This *would* work if only □ meant "for all states" rather than "for all future states" and likewise for ◊. But since y cannot be a state previous to x and z cannot be previous to y, there are some assignments that will never arise from this path constraint, such as $\{x \mapsto 2, y \mapsto 2, z \mapsto 1\}$. Thus the path constraint is not equivalent to formula (2).

However, it is temptingly close, and it is easy to imagine language extensions that would make the proof work. For example, if we had an operator called @ that allowed us to jump to a previous state, we could translate formula (2) as

$$\{h\}\Diamond\{x\}@_0\Box\{y\}@_h\Diamond\{z\}(t(z) \wedge t(x) \vee t(y))$$

The @ operator exists in hybrid logic, and indeed precisely this approach was used to prove that model-checking a hybrid logic with @ and variable binders is PSPACE-hard [23].

Proving that model-checking a path constraint is PSPACE-hard is more challenging. Instead of a simple two-state model, we must build a $2k$-state model, where k is the number of quantifiers. The odd states will represent falsehood; the even, truth. The following proof provides the details.

Theorem 2 *The problem of model-checking a path constraint on a single path is PSPACE-complete.*

Proof By theorem 1, the problem is in PSPACE. To show it is PSPACE-hard, we exhibit a polynomial-time reduction of QBF. Let

$$Q_1 x_1 Q_2 x_2 \cdots Q_k x_k \phi(x_1, x_2, \ldots, x_k)$$

be a QBF in prenex normal form, so $Q_1, \ldots, Q_k$ are quantifiers and $\phi(x_1, \ldots, x_k)$ is an abbreviation for a propositional Boolean formula over the variables. We translate this QBF into a path constraint model-checking problem as follows. Define the signature by

$$\Sigma = ((x_1, \ldots, x_k), (\emptyset), (\{p_1, \ldots, p_k\}, \{t\}, \emptyset, \emptyset, \ldots))$$

Thus $x_1, \ldots, x_k$ will be variables, $p_1, \ldots, p_k$ will be nullary predicates, and t will be a unary predicate. Now let σ be a temporal sequence of $2k$ states, $\sigma_1, \ldots, \sigma_{2k}$. We define the interpretation of the nullary predicates at each state by

$$\sigma_i(p_j) = \begin{cases} \mathit{true} & \text{if } 2j-1 \leq i \leq 2j \\ \mathit{false} & \text{otherwise} \end{cases}$$

for all i, j. (A nullary relation can be simply identified with *true* or *false*.) We define the interpretation of the unary predicate t by

$$\sigma_i(t) = \{\sigma_2, \sigma_4, \sigma_6, \ldots, \sigma_{2k}\}$$

Figure 3 summarizes this model. Note that we can construct this model in linear time.

We now construct the path constraint that corresponds to the QBF. For the quantifier part, we translate:

- $\forall x_i$ into the string "$\Box(p_i \rightarrow \{x_i\}$"
- $\exists x_i$ into the string "$\Diamond(p_i \wedge \{x_i\}$"

The quantifier-free part we transcribe literally, except that each occurrence of x_i becomes $t(x_i)$. At the end of the formula, we add k closing parentheses. For example, formula (2) becomes

$$\Diamond(p \wedge \{x\}\Box(q \rightarrow \{y\}\Diamond(r \wedge \{z\}(t(z) \wedge t(x) \vee t(y)))))$$

This formula, too, can be constructed in linear time. (It is larger than the original QBF, but only by a constant factor.)

It remains to show that the constructed path constraint (call it ψ) is true iff the original QBF (call it χ) is true. Let χ_i denote χ with the first i quantifiers removed. Similarly let ψ_i denote ψ with the first i quantifier translations (and the last i parentheses) removed. We will show that for any i, for any $j < 2i$, and for any assignment s of the variables,

$$\sigma, j, s \vDash_{\mathrm{PCL}} \psi_i \quad \text{iff} \quad s^* \vDash_{\mathrm{FOL}} \chi_i$$

(We use $\vDash_{\mathrm{PCL}}$ and $\vDash_{\mathrm{FOL}}$ to indicate semantic consequence with respect to our path constraint language and FOL, respectively.) Note that we translate the path constraint language assignment $s : V \rightarrow S$ into the FOL assignment $s^* : V \rightarrow \{\mathit{true}, \mathit{false}\}$ given by

$$s^*(x_j) = \begin{cases} \mathit{true} & \text{if } s(x_j) \text{ is one of } \sigma_2, \sigma_4, \sigma_6, \ldots, \sigma_{2k} \\ \mathit{false} & \text{if not} \end{cases}$$

The proof is by induction on i. The base case is $i = k$, where all the quantifiers have been removed, so χ_i and ψ_i are both propositional formulas. The propositional connectives have the same semantics in our path constraint language and FOL, so it suffices to show that for any $j < 2i$ and for each h,

$$\sigma, j, s \vDash_{\mathrm{PCL}} t(x_h) \quad \text{iff} \quad s^* \vDash_{\mathrm{FOL}} x_h$$

By the definition of t and the semantics of our path constraint language, it suffices to show that

$$s(x_h) \in \{\sigma_2, \sigma_4, \sigma_6, \ldots, \sigma_{2k}\} \quad \text{iff} \quad s^* \vDash_{\mathrm{FOL}} x_h$$

This is immediate from the definition of s^*.

For the inductive step, suppose we have proven the result for $i + 1$, so we know that for any $j < 2i + 2$ and any

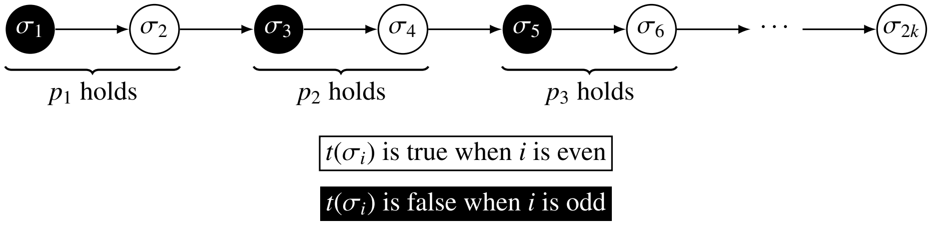


Fig. 3 A graphical representation of the path constraint language model that we construct in the reduction of a QBF.

assignment s, we have $\sigma, j, s \vDash_{\mathrm{PCL}} \psi_{i+1}$ iff $s^* \vDash_{\mathrm{FOL}} \chi_{i+1}$. We split by cases based on whether the ith quantifier is ∀ or ∃.

Case: ∀. We must show that for each $j < 2i$ and for any s,

$$\sigma, j, s \vDash_{\mathrm{PCL}} \Box(p_i \rightarrow \{x_i\}\psi_{i+1}) \quad \text{iff} \quad s^* \vDash_{\mathrm{FOL}} \forall x_i \chi_{i+1}$$

Whenever $j < 2i$, observe that

$$\begin{aligned} & \sigma, j, s \vDash_{\mathrm{PCL}} \Box(p_i \rightarrow \{x_i\}\psi_{i+1}) \\ \text{iff} \quad & \sigma, h, s \vDash_{\mathrm{PCL}} \{x_i\}\psi_{i+1} \text{ for all } h \geq j \text{ with } \sigma, h, s \vDash p_i \\ \text{iff} \quad & \sigma, h, s \vDash_{\mathrm{PCL}} \{x_i\}\psi_{i+1} \text{ for all } h \in \{2i-1, 2i\} \\ \text{iff} \quad & \sigma, h, s[x_i \mapsto \sigma_h] \vDash_{\mathrm{PCL}} \psi_{i+1} \text{ for all } h \in \{2i-1, 2i\} \end{aligned}$$

Similarly,

$$\begin{aligned} & s^* \vDash_{\mathrm{FOL}} \forall x_i \chi_{i+1} \\ \text{iff} \quad & s^*[x_i \mapsto h] \vDash_{\mathrm{FOL}} \chi_{i+1} \text{ for all } h \in \{\mathit{true}, \mathit{false}\} \end{aligned}$$

Thus, it suffices to show that

$$\begin{aligned} & \sigma, h, s[x_i \mapsto \sigma_h] \vDash_{\mathrm{PCL}} \psi_{i+1} \text{ for all } h \in \{2i-1, 2i\} \\ \text{iff} \quad & s^*[x_i \mapsto h] \vDash_{\mathrm{FOL}} \chi_{i+1} \text{ for all } h \in \{\mathit{true}, \mathit{false}\} \end{aligned}$$

For this, it suffices to show that

$$\begin{aligned} & \sigma, 2i-1, s[x_i \mapsto \sigma_{2i-1}] \vDash_{\mathrm{PCL}} \psi_{i+1} \\ \text{iff} \quad & s^*[x_i \mapsto \mathit{false}] \vDash_{\mathrm{FOL}} \chi_{i+1} \end{aligned}$$

and

$$\begin{aligned} & \sigma, 2i, s[x_i \mapsto \sigma_{2i}] \vDash_{\mathrm{PCL}} \psi_{i+1} \\ \text{iff} \quad & s^*[x_i \mapsto \mathit{true}] \vDash_{\mathrm{FOL}} \chi_{i+1} \end{aligned}$$

But each of these is simply an instance of the induction hypothesis, since by the definition of s^* we have

$$\begin{aligned} (s[x_i \mapsto \sigma_{2i-1}])^* &= s^*[x_i \mapsto \mathit{false}] \\ (s[x_i \mapsto \sigma_{2i}])^* &= s^*[x_i \mapsto \mathit{true}] \end{aligned}$$

Case: ∃. We must show that for each $j < 2i$ and for any s,

$$\sigma, j, s \vDash_{\mathrm{PCL}} \Diamond(p_i \wedge \{x_i\}\psi_{i+1}) \quad \text{iff} \quad s^* \vDash_{\mathrm{FOL}} \exists x_i \chi_{i+1}$$

By reasoning parallel to the ∀ case, it suffices to show that

$$\begin{aligned} & \sigma, h, s[x_i \mapsto \sigma_h] \vDash_{\mathrm{PCL}} \psi_{i+1} \text{ for some } h \in \{2i-1, 2i\} \\ \text{iff} \quad & s^*[x_i \mapsto h] \vDash_{\mathrm{FOL}} \chi_{i+1} \text{ for some } h \in \{\mathit{true}, \mathit{false}\} \end{aligned}$$

As in the previous case, this follows from the induction hypothesis.

This concludes the inductive proof that $\sigma, j, s \vDash_{\mathrm{PCL}} \psi_i$ iff $s^* \vDash_{\mathrm{FOL}} \chi_i$ for any i, any $j < 2i$, and any s. Choosing $i = 0$ and $j = 1$, we obtain $\sigma, 1 \vDash_{\mathrm{PCL}} \psi$ iff $\vDash_{\mathrm{FOL}} \chi$ as desired. □

Our QBF reduction is similar to that recently employed to prove PSPACE-completeness for the finitary model-checking problem for LTL augmented with finitely many registers, over deterministic one-counter automata [20].

Our PSPACE-completeness result is somewhat unfortunate, but perhaps to be expected, since, as we have seen, the variable-binding extension is a sort of quantifier, and a great many interesting decision problems for quantified logic are PSPACE-complete [25], including even the problem of checking first-order monadic logic over finite paths [44]. And of course hybrid logic with ↓ has the same problem as our path constraint language.

There are practical reasons not to be terribly concerned about the PSPACE-hardness of checking path constraints. First, actual path constraints are typically not terribly long. They are, after all, written by humans for the purpose of reasoning formally about constraints that arise naturally. Second, it is possible that the predicates that arise in this domain might lend themselves to specialized analyses with good performance. That is, the predicates we actually encounter are not arbitrary n-ary relations over the state space; they are simple tests like "has a database." Third, and most importantly, the constraints that arise are likely to be particularly well-behaved, so the worst-case running time will be quite rare in actuality. Of course, this is a claim that is likely to be made about any intractable problem, but at least in our case it can be formalized.

The complexity of the path-checking problem in our case arises, of course, from the variable-binding extension, but in particular it arises from *nesting* these binders. All of the intractable examples we have seen, such as the transformed formula in the QBF reduction, involve arbitrarily deeply nested variable bindings. The following theorem shows that model-checking a path constraint over a path is intractable only to the extent that variables are nested without bound.

Theorem 3 *Let σ be a temporal sequence of length m. Let ϕ be a path constraint of length ℓ. Let d be the maximum variable-nesting depth of ϕ (i.e., there is no point in ϕ at which more than d variables are bound at once). Then there is an algorithm that determines whether $\sigma, i \vDash \phi$ in $O(\ell m^{d+1})$.*

Proof Let $\phi_1, \ldots, \phi_n$ be the subformulas of ϕ, listed by nondecreasing length. We define a Boolean matrix $[t_{j,k}]_{m \times n}$. The follow recursive algorithm fills the kth column of the matrix under variable assignment s:

```
FillCol($k, s$)
    if $\phi_k$ is an atomic proposition
        for $j = 1$ to $m$
            $t_{j,k}$ := evaluate $\phi_k$ at $j$ using assignment $s$
    if $\phi_k = false$
        for $j = 1$ to $m$
            $t_{j,k} := false$
    if $\phi_k$ is of the form $\phi_g \rightarrow \phi_h$
        FillCol($g, s$)
        FillCol($h, s$)
        for $j = 1$ to $m$
            $t_{j,k} := t_{j,h}$ or not $t_{j,g}$
    if $\phi_k$ is of the form $\bigcirc\phi_h$
        FillCol($h, s$)
        for $j = 1$ to $m - 1$
            $t_{j,k} := t_{j+1,h}$
        $t_{m,k} := true$
    if $\phi_k$ is of the form $\phi_g \,\mathcal{U}\, \phi_h$
        FillCol($g, s$)
        FillCol($h, s$)
        $prev := false$
        for $j = m$ to $1$ step $-1$
            $prev := t_{j,h}$ or ($t_{j,g}$ and $prev$)
            $t_{j,k} := prev$
    if $\phi_k$ is of the form $\{x\}\phi_h$
        for $j = 1$ to $m$
            FillCol($h, s[x \rightarrow \sigma_j]$)
            $t_{j,k} := t_{j,h}$
```

To solve the model-checking problem, execute FillCol($n, \emptyset$) and read the answer from $t_{i,n}$. The correctness of the algorithm follows easily from the semantics of path constraints.

Note that in the case where there are no variable assignments, this algorithm reduces to the algorithm for LTL. When there are variable assignments, intermediate results in the matrix are overwritten; in an intermediate evaluation of a formula of the form $\{x\}\psi$, we use the columns to our left to first evaluate ψ under the assignment $x \mapsto 1$, then immediately overwrite them, using the same space to evaluate ψ under the assignment $x \mapsto 2$, and so on. More precisely, the kth column of the matrix is ultimately filled m^{d_k} times, where d_k is the variable-nesting depth of ϕ_k (i.e., the number of variables that are bound for the subformula ϕ_k). Thus no column is filled more than m^d times, where d is the maximum variable-nesting depth of ϕ. Filling a single column, not counting the time to fill its subformula columns to the left, takes $O(m)$ time. And there are n columns in total, which is $O(\ell)$. Therefore the total complexity of the algorithm is $O(\ell m^{d+1})$. □

If d is bounded (e.g., we never have constraints with a variable-nesting depth greater than 3) then we can model-check a path constraint over a path in polynomial time. (If $d = 0$, we get the same, linear performance as the LTL algorithm, as we would hope.) If d is unbounded, then the performance is exponential, $O(\ell m^{\ell})$, since the quantifier depth can approach the length of the formula. In practice, it seems unlikely that variable bindings will often be very deeply nested, since software architects are unlikely to be naturally interested in such convoluted constraints. However, although these theoretical results are illuminating, performance testing is needed to understand their practical significance. We leave this for future work.

5 Case study

In this section, we present a case study that illustrates the application of these ideas to an actual architecture evolution. The evolution in question was an artificial one, conducted in the laboratory for the purposes of this research. Nonetheless, we believe it provides a useful demonstration of how our approach may be applied in practice.

Section 5.1 introduces the case study and explains our reasons for our project selection. Section 5.2 describes the result of the evolution and shows how our model may be applied to analyze the evolution in retrospect. Section 5.3 discusses the limitations of the case study, and section 5.4 discusses the conclusions that may be drawn from it.

5.1 Case study description

Methodology. In this case study, we conducted a controlled evolution of a real software system. We began with an open-source software system and selected a target architectural style to which we would migrate the system. Then one of us (the first author) carried out the evolution unaided, without the use of the models we have described.

After the conclusion of the evolution, we analyzed it retrospectively. We mapped the course that the evolution had taken, using the model of section 3. We attempted to identify the constraints to which the evolution had been subject, and we formalized them in our constraint language and showed formally that the path adhered to them. Finally, we reflected on the question of whether the evolution might have gone more smoothly had we had access to an evolution style for the domain before the evolution began. We considered whether any automatable analyses could have been employed in the planning stage to make the evolution more successful.

The case study was paper-based rather than tool-based. There were two reasons for this. The first reason is practical: as we note in section 7, the tools we have developed so far do not support the full set of evolution modeling features we wished to evaluate in the case study. The second reason is that our goal for the case study was not to evaluate a tool, but rather to assess whether the modeling approach itself was suitable for representing an actual evolution.

Research questions. Because of its artificial nature and small size, this case study is able to address only limited research questions. In section 5.3, we will discuss these limitations in detail. Here we merely present the questions the case study sought to address:

1. *Can our model be applied to the analysis of a real evolution?* Previous studies in this area have tended to focus on fictitious examples, like our example in section 2.1. Although artificial, the present case study at least examines an evolution that was actually carried out rather than merely imagined; it represents a significant step toward real-world experimentation.
2. *Is our path constraint language expressive enough to capture the constraints that naturally arise in a real evolution?* We devised our path constraint language based on our intuition about what evolution constraints are likely to look like. But this intuition has never been tested. By seeing what constraints arise in this case study, we can achieve some level of validation of the practicality of the language.
3. *Is it likely that a model like ours, with appropriate tools to implement it, could be useful for planning evolutions?* Our approach in this case study is to perform the evolution unaided rather than with the use of our model. Nonetheless, we can make an attempt to get at this question by examining the mistakes and missteps that occur during the evolution, and asking whether they are the sort of difficulties that might plausibly be averted by the use of our model.

Project selection. We undertook an evolution of an existing small, stand-alone desktop application to the Amazon EC2 cloud-computing platform [3]. We picked these initial and target styles for several reasons. First, as noted in section 2.1, evolution to the cloud is a topic of considerable current interest. Not only does this give our case study greater practical relevance, it also means that there is a wide variety of existing resources that provide guidance on this style of evolution. Such resources can be a helpful source of evolution constraints and analyses, as we will see in section 5.2. Indeed, by using guidance from external sources to generate some of our constraints, we mitigate one threat to external validity: that we invented constraints that were particularly and unrealistically easy to formalize in our system.

Second, evolutions to the cloud are amenable to this kind of case study because they can be carried out at a large or small scale. An evolution of a large software system to the cloud can be a huge undertaking, requiring dozens (even hundreds) of engineers and months (even years) of time. On the other hand, a simple, small evolution of this kind can be done by a single person in a matter of weeks. Indeed, this is one of the attractions of cloud computing: software systems of modest size and need can migrate to the cloud to avoid the expense of dedicated hardware, so that they pay only for the resources they use. So cloud computing is a viable evolution style for both large and small systems.

Moreover, small and large cloud-based software systems are not fundamentally different. Usually, a large cloud-based system looks much like a scaled-up version of a small one. This is not generally true for other architectural styles, but it is a consequence of another important attraction of the cloud: scalability. Instead of buying new hardware, engineers simply order additional resources to scale a system in the cloud. Cloud-based systems are designed with this kind of horizontal scaling in mind. Small and large cloud-based systems alike are usually designed in this way; after all, many small systems hope to later become large systems when demand grows and more resources are available. Consequently, building a small cloud-based system can tell us something about the experience of building a large cloud-based system, to a greater degree than might be true of other architectural styles.

We picked GNU Chess [31] as our starting system. GNU Chess is a desktop chess engine with a venerable history. The first version dates back to 1984, although the latest version, GNU Chess 5, which was first released in 1999, shares no code with earlier versions. GNU Chess is quite well-known and has featured in a number of academic studies in various domains.

Although chess engines may seem an esoteric domain, they are quite suitable for this kind of evolution study. First, a chess engine is about the right size and complexity—small enough that a single person can understand and evolve the entire system in a reasonable amount of time, but large enough to have an identifiable architecture and to present evolution challenges. Second, as we will argue in section 5.3, chess engines are representative of a much broader class of systems amenable to cloud evolution. Indeed, as we will see, this case study bears interesting similarities to the more realistic scenario we described in section 2.1.

GNU Chess in particular is suitable because it is of an appropriate size (14,370 lines of C code), because it is open-source, and because it is neither artificially simple nor prohibitively complex. That is, it is a real software application that has emerged from a real-world development process and has enjoyed popular use, not a toy application introduced for the purposes of this study. At the same time, it is neither excessively complex, nor is it poorly structured spaghetti code; it has a discernible (albeit not explicitly documented) software architecture, which is a prerequisite of our approach to architecture evolution.

The latest version of GNU Chess available at the time we began this study was GNU Chess 5.0.7, released in 2003, so that is the version we used. (More precisely, we used gnuchess-5.07-7 [19], an unofficial release from the Debian project that fixed a number of problems in the aging official release, including some that prevented it from compiling on modern compilers.)

Evolution target. Our objective was to migrate GNU Chess to Amazon EC2, one of the best-known cloud computing platforms. The end goal was a web application, hosted in the cloud, that would allow users to play chess against a computer.

We chose Amazon EC2 over other cloud platforms for a couple of reasons. First, it is quite popular, boasting an impressive market share. Among other benefits, this means that there are ample existing resources on migration to EC2, some of which we thought might be helpful for generating evolution constraints and analyses. Second, EC2 is quite flexible, unlike some other cloud computing platforms that are tied to certain languages (e.g., Google App Engine, which supports only Python and Java [32]) or domains (e.g., Twilio, a cloud computing platform for telephony applications [66]). This was important, since we were dealing with a legacy application written in C.

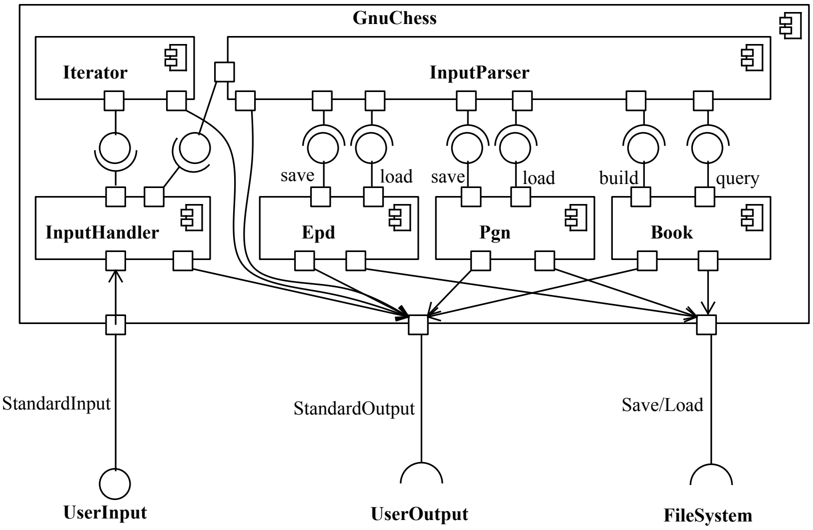
The key quality attribute on which we focused was scalability, which is, after all, one of the major appeals of cloud computing. We aimed to rearchitect the system in such a way that it could support arbitrarily many simultaneous users, simply by buying more resources and configuring the system accordingly. As we suggested in the last section, we planned to accomplish this by decomposing GNU Chess into separate components that could be deployed on separate machines to accommodate piecewise horizontal scaling.

Of course, this attribute of scalability, though it was critical to the planning process, would never actually be tested. In reality, our system will never have more than a couple of users at a time. That is, for the purposes of this case study, we *imagined* ourselves in the position of an organization with particular business constraints and business goals, one of which was the development of an online chess application that could be easily and economically scaled as it became more popular. In reality, of course, the application would never need to scale at all, let alone be economically feasible. For the number of users we would actually have, we would have been quite satisfactorily served by deploying an unmodified GNU Chess on a single host and building a web interface in front of it. But our architectural decisions were driven by this artificial scenario that we envisioned. In this way, we aimed to make the evolution simultaneously more realistic and less trivial. Of course, it would be better, from the standpoint of external validity, to be operating within a real business context rather than contriving one. This may be properly regarded as one of the limitations of our case study; we discuss this issue further in section 5.3.

5.2 Analysis

Figures 4 and 5 show high-level overviews of the architectures of the starting and final systems in the actual evolution. We considered the evolution from two architectural perspectives: a component-and-connector view, represented with the UML 2 component diagram type (in line with the conventions recommended by the Software Engineering Institute for representing component-and-connector views in UML [37]), and an allocation view, represented with the UML 2 deployment diagram type. We chose these two views for our case study because they are the most useful given the technical qualities of the evolution and the business context. Component-and-connector views are essential for understanding the run-time structure of a system, as well as its behavioral qualities, such as performance. A deployment view is useful because it pertains to a central goal of the evolution: the redeployment of the software to a different environment (servers in the cloud rather than a desktop).

The initial version of GNU Chess had a rather monolithic architecture (at least from the component-and-connector and deployment perspectives; a static code view would show a clearer structure). While we can identify some substructure with effort (some of the most important identifiable subcomponents are shown in figure 4), in essence the chess engine is one component, consisting of a single-threaded, single-process application running locally on a single machine. Only with difficulty can we identify any useful subcomponents. To put it another way, a component-and-connector



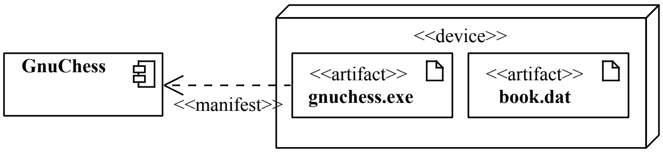


Fig. 4 Two views of the initial architecture in our case study evolution: a component-and-connector view depicted as a UML component diagram (top) and an allocation view depicted as a UML deployment diagram (bottom).

diagram is responsible for showing interactions among processing units, but there are not really any interacting processing units in GNU Chess. The only interaction is between the application and the user, who is not part of the software system (and occasionally between the application and the file system, when information is stored to or loaded from disk). So although we have made an effort to show some decomposition of the component-and-connector view by depicting the structure of the control flow, it is fair to regard the component-and-connector architecture of GNU Chess as rather monolithic.

Over the course of the evolution, we split the application into multiple separate parts that could be deployed separately (figure 5): an opening-book component responsible for looking up opening positions in a database, user interface components responsible for mediating between the chess engine and the user, and the core analysis component responsible for calculating moves on positions that are not in the opening database.

This kind of separation of concerns into separately deployable components is quite useful for evolution to the cloud, because it allows independent scaling of the different components. The different components will scale differently (and in a way that is to some degree unpredictable, since it may depend on the actual usage of the system), so it makes sense to separate them so that as the application grows, we can target additional resources specifically at the components that need them. In addition, different parts of the system have different requirements (e.g., some are processor-bound while others are disk-bound), so by decomposing the system into separately deployable parts, we can host them on different hardware, specially selected to be appropriate for each component.

In addition to splitting the system into separate components, of course, we also had to be concerned with how to hook these components together. We employed a diverse array of connector types. On the front end, the users interact with the web servers through standard web technologies (HTTP over TCP). The web servers then make requests of the analysis and opening-book servers using a custom application-level protocol built on TCP. Within the analysis servers, communication between components occurs using traditional means of interprocess communication. A dispatcher places requests in a POSIX message queue. Analysis engines dequeue these requests, handle them, and place the results in a named pipe owned by the dispatcher. The system is designed with scalability, performance, and cost in mind. More web servers and analysis servers can be added

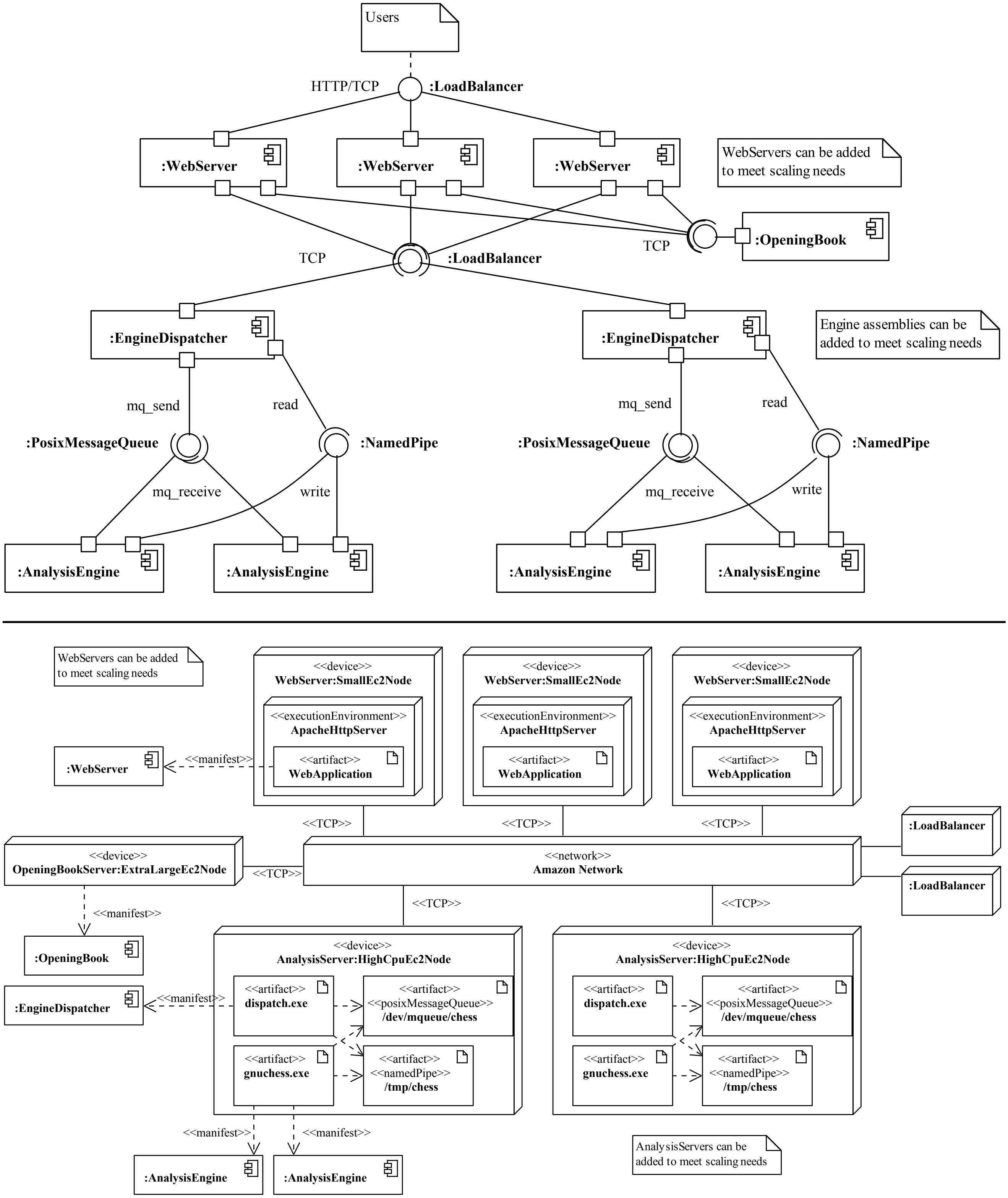

Users
HTTP/TCP
:LoadBalancer
:WebServer
:WebServer
:WebServer
WebServers can be added to meet scaling needs
:OpeningBook
TCP
TCP
:LoadBalancer
:EngineDispatcher
:EngineDispatcher
Engine assemblies can be added to meet scaling needs
mq_send
read
mq_send
read
:PosixMessageQueue
:NamedPipe
:PosixMessageQueue
:NamedPipe
mq_receive
write
mq_receive
write
:AnalysisEngine
:AnalysisEngine
:AnalysisEngine
:AnalysisEngine
WebServers can be added to meet scaling needs
<<device>> WebServer:SmallEc2Node
<<executionEnvironment>> ApacheHttpServer
<<artifact>> WebApplication
<<device>> WebServer:SmallEc2Node
<<executionEnvironment>> ApacheHttpServer
<<artifact>> WebApplication
<<device>> WebServer:SmallEc2Node
<<executionEnvironment>> ApacheHttpServer
<<artifact>> WebApplication
:WebServer
<<manifest>>
<<TCP>>
<<TCP>>
<<TCP>>
:LoadBalancer
:LoadBalancer
<<device>> OpeningBookServer:ExtraLargeEc2Node
<<TCP>>
<<network>> Amazon Network
<<manifest>>
<<TCP>>
<<TCP>>
:OpeningBook
<<device>> AnalysisServer:HighCpuEc2Node
<<artifact>> dispatch.exe
<<artifact>> <<posixMessageQueue>> /dev/mqueue/chess
<<artifact>> gnuchess.exe
<<artifact>> <<namedPipe>> /tmp/chess
:EngineDispatcher
<<manifest>>
<<device>> AnalysisServer:HighCpuEc2Node
<<artifact>> dispatch.exe
<<artifact>> <<posixMessageQueue>> /dev/mqueue/chess
<<artifact>> gnuchess.exe
<<artifact>> <<namedPipe>> /tmp/chess
<<manifest>>
<<manifest>>
:AnalysisEngine
:AnalysisEngine
AnalysisServers can be added to meet scaling needs

Fig. 5 A component diagram (top) and deployment diagram (bottom) of the final architecture in our case study evolution.

easily; traffic for these servers is mediated by load balancers responsible for routing requests to the different servers in a fair way. Each engine server runs as many AnalysisEngine instances as there are processing cores on the server; since the analysis engine is single-threaded, this is necessary to make full use of whatever computing resources we have purchased. (Amazon currently offers instances with up to eight virtual cores.)

Many of the constraints and planning considerations we naturally thought about during the evolution seem to fit well into our framework. Some of these constraints can be expressed with simple architectural-style constraints—constraints that can be evaluated with respect to each single intermediate architecture, without the need for notions of temporality. For example, consider the constraint: "Two components that are connected via a POSIX message queue must be on the same host." This can be expressed as a local constraint on a single state, which we express here in OCL:

```
context PosixMessageQueue inv:
let components = end.role.structuredClassifier in
let hosts =
  components.manifestation.client.deployment.location
in hosts->size() < 2
```

This says that for each PosixMessageQueue connector, if we look at the components it connects, and then look at all the artifacts that manifest these components, we will find that they are all on the same host. Note that this OCL constraint is at the UML metamodel level. To understand why this is necessary, bear in mind how our use case differs from the most common use of OCL. Usually, OCL is used with an existing, fixed UML diagram to specify additional constraints beyond those represented in the diagram. But we are using OCL to select from a space of many UML diagrams; therefore, our OCL constraints must operate on the UML metamodel rather than any particular UML model. Our use of OCL is similar to that of Cariou et al. [13], who use OCL to specify model transformation contracts. (By contrast, the constraint language in Acme is designed specifically to distinguish between allowed and disallowed architectures, so when we model evolutions using Acme rather than UML, we can use its constraint language in the usual way.)

Setting aside the specific mechanics of the OCL, the key point is that the constraint can be evaluated with respect to an individual intermediate architecture; that is, for each intermediate architecture, we can judge (without reference to other intermediate architecture or to temporality) whether its POSIX message queues are connected correctly. However, more complex constraints require temporality to express. For example, consider the constraint "The type of a connector can be changed at most twice." We might apply such a constraint to ensure that our evolution path does not involve a lot of repeated reworking of connectors. Changing the type of a connector (say, from a POSIX message queue to network-socket-based communication) is a sensible thing to do in an evolution, but we might decide that changing it more than twice, as in figure 6, indicates an excessive degree of reworking. Such a constraint cannot be represented as a local architectural constraint in OCL or Acme—it is a constraint on the entire evolution path, not an individual architecture—so we must use our temporal path constraint language to model it:

$$\Box\{s\}\,\Box\{t\}\,\Box\{u\}\,\Box\{v\}\,connTypeChangedAtMostTwice(s,t,u,v)$$

Of course, *connTypeChangedAtMostTwice* is not a keyword built into the path constraint language; it is a quaternary predicate over states that must be defined explicitly by the evolution style in the applicable architectural constraint language. In this case, we could define it in OCL by:

```
inv connTypeChangedAtMostTwice(s, t, u, v):
s::Connector.allInstances.forall(connS |
t::Connector.allInstances.forall(connT |
u::Connector.allInstances.forall(connU |
v::Connector.allInstances.forall(connV |
    let connectors = {connS,connT,connU,connV} in
        size(connectors.evId−>asSet()) = 1 implies
        size(connectors.type−>asSet()) < 4))))
```

In effect, this constraint says, "There is no connector whose type is different in all of s, t, u, v." The constraint is violated if there exist four connectors, one from each of the four input states, that are evolutionarily identical (i.e., they represent the same connector at different stages of evolution) but all have different types. (To determine whether the four connectors have the same evolutionary identity but different types, the constraint takes the set of four connectors, {connS, connT, connU, connV}, and counts the number of distinct evIds and the number of distinct types, to check whether these counts are 1 and 4, respectively.) Note that the definition of the constraint does not require temporal operators.

A good source for evolution constraints and analyses is existing documentation and advice on evolving to the target architectural style. Fortuitously, while we were in the midst of our case study, Amazon released a white paper with advice on "Migrating Your Existing Applications to the AWS Cloud" [67]. We mined this paper for evolution constraints that we could formalize. We found a number of them, although not all of them map very well to our case study because of its small size. The paper recommends a phased approach to cloud migration, with phases such as cloud assessment, proof of concept, data migration, application migration, and so on. In the recommended phase strategy, data migration

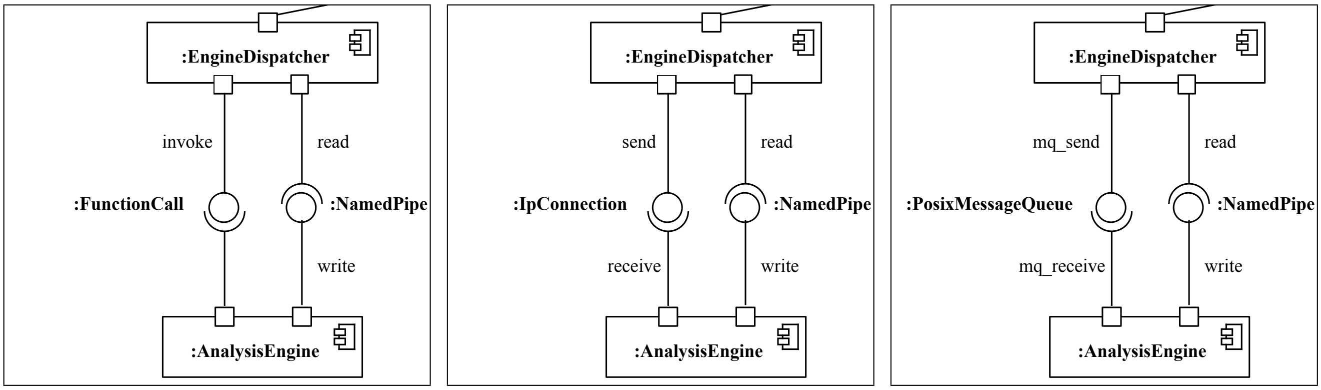


Fig. 6 This is a partial depiction of a portion of an alternate evolution path; each of the three frames shows a different stage of evolution. In this alternate path, the type of a connector is changed repeatedly, which may indicate an excessive degree of reworking. We can prohibit such paths by means of a path constraint.

(phase 3) precedes application migration (phase 4). Suppose that we wanted to make a constraint of this. (The paper itself notes that some companies prefer to vary the order of the phases, for example by performing application migration before data migration is complete. Nonetheless, suppose that we had decided that, for our purposes, it was indeed prudent to migrate all data before beginning application migration, and that we wanted to formalize this rule as an evolution constraint.) In our small case study, there is not much data to speak of—the only significant data store is the opening database. Nonetheless, if we did wish to formalize the constraint, we could do it easily:

$$\neg \mathit{hasMigratedApplication}(\mathit{system}) \;\; \mathcal{U} \; \mathit{allDataMigrated}(\mathit{system})$$

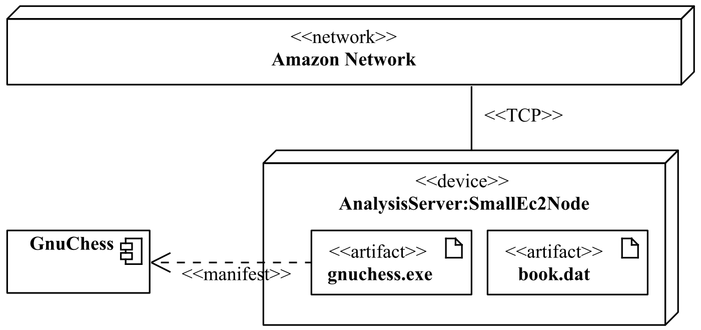
Once again, *hasMigratedApplication* and *allDataMigrated* are predicates that must be defined by the evolution style. So as not to belabor the point, we do not define them here, but the important point is that they can be defined in the architectural constraint language (in this case, OCL) with reference only to the arguments of the predicate—without use of temporal connectives.

We have now shown how a number of the constraints relevant to our case study can be formalized. We now consider operators, which are rather easier to come up with than constraints. In our case study, we found that the steps we carried out lent themselves naturally to expression as evolution operators. For example, one important operation that we carried out several times was moving a component to another host. We can represent this as an operator as follows:

```
operator migrateComponent(c, newHost) {
    transformations {
        UML2Deployment {
            let artifact = c.manifestation.client
            unlink artifact as deployedArtifact
                    from artifact.deployment
            link artifact as deployedArtifact
                    to newHost.deployment
        }
    }
    preconditions {
        UML2Component {
            c.ownedConnector->forAll(
                    type <> PosixMessageQueue
                    and type <> NamedPipe)
        }
        UML2Deployment {
            c.manifestation->notEmpty()
        }
    }
    analysis {
        "duration": {
            "amount": 10, "unit": "person-hour"
        }
    }
}
```

This is a good example of how an operator looks when multiple views are involved. The *transformations* and *preconditions* blocks each contain one subblock for each view (i.e., each model type). In this case, we have a component-and-connector view, depicted using a UML component diagram, and an allocation view, depicted by a UML deployment diagram, so the subblocks are labeled *UML2Component* and *UML2Deployment*. However, the *UML2Component* subblock of the *transformations* block was omitted because it was empty; component migration is visible only in the deployment view, not the component-and-connector view. The analysis information in this example is fictitious, intended simply to give an idea of what analysis information might look like. This analysis information would be used by an

[component-and-connector view identical to fig. 4]



```
operator setUpEc2Environment() {
  transformations {
    UML2Deployment {
      Node ec2Environment = create Node : Network;
      ec2Environment.name = "Amazon␣Network";
    }
  }
}
operator createSmallEc2Node(environment, name) {
  transformations {
    UML2Deployment {
      Device node = create Device : SmallEc2Node;
      node.name = name;
      Association a = create Association : TCP
          between node, environment;
    }
  }
}
operator deleteNode(n) {
  transformations {
    UML2Deployment {
      delete n;
    }
  }
}
```

Fig. 7 An example illustrating how an intermediate state can be defined by applying a sequence of evolution operators.

evaluation function for evolution duration and tells us how long component migration takes to carry out.

By composing such operators together, we can fully specify the architectural transformations making up the transitions of the evolution path, and thus fully define the intermediate states. Figure 7 provides an example of how an intermediate state (shown at left) can be defined by applying a sequence of operators (shown at right) to the initial architecture of the system (shown earlier in figure 4). In this example, the operators effect the migration of GNU Chess from the desktop environment where it initially resides to an Amazon EC2 node—the first step in the GNU Chess evolution. Only the deployment diagram changes, since at this stage GNU Chess is simply being migrated to a new deployment environment, without any architectural changes to the code. The component-and-connector diagram is therefore identical to that of the initial state.

The intermediate architecture in figure 7 can be obtained from the initial architecture in figure 4 by applying four operators (with the correct parameters and in an appropriate sequence): the three operators defined in the stubs in figure 7, along with the *migrateComponent* operator defined above. First, we apply the *setUpEc2Environment* operator to create the architectural representation of the EC2 environment. Second, we apply *createSmallEc2Node* to create a new EC2 node. Third, we invoke *migrateComponent* twice to migrate the two artifacts from the old desktop device to the new EC2 node. Lastly, we use *deleteNode* to remove the old desktop device from the model. In this way, we define the second state of the evolution path in terms of the first.

Finally, let us consider evaluation functions. There are a number of evaluation functions that would have been useful for planning an evolution such as the one we undertook in this case study. One of the most straightforward to implement is an analysis of the final running cost of the system—that is, the hourly cost that Amazon will charge us to run the system once the evolution is complete. This is a function of the number and types of hosts that are used in the final state, as well as some harder-to-determine figures like the amount of network traffic. This particular evaluation function requires information from only the final state; more sophisticated evaluation functions could perform arbitrarily complex reasoning involving all the states of an evolution path. For simplicity, we account for only the costs of running the instances, not other costs such as bandwidth, storage, and support. The language used here is JavaScript (more precisely, ECMAScript, fifth edition [21]).

```
function analyzeFinalHourlyRunningCost(states) {
    // The final state is all we are interested in
    var finalState = states.pop();

    // Get all the devices in the state by an OCL query
    // on the deployment view
    var devices = finalState.query(
        "Device.allInstances()", "UML2Deployment");

    // Calculate the total hourly cost
    var cost = 0;
    for (var device in devices) {
        switch (device.type) {
            case SmallEc2Instance:
                cost += .085; break;
            case LargeEc2Instance:
                cost += .34; break;
            case HighCpuEc2Instance:
                cost += .17; break;
            // Et cetera
        }
    }
    return cost;
}
```

Every evaluation function takes an argument, *states*, which is a JavaScript array whose elements are the intermediate evolution states that make up the path to be analyzed. Each element of this array is an object containing a queryable representation of the architectural model associated with the state that the object represents. This examples shows how these objects can be queried (with the *query* method) to access this information.

It seems clear that we can create an evolution style for migrations to Amazon EC2. But would such a style be useful? Would it have been useful for the evolution we carried out in our case study? There is no way we can give an absolute answer to this question objectively, but it does seem that a number of the questions and challenges that arose during the course of the evolution are of the sort that could plausibly be answered by such a style. For this case study, we deliberately chose starting and ending domains with which we were unfamiliar; that is, the author carrying out the evolution had no prior practical experience with desktop or network-based applications written in C, nor with developing cloud-based software. Evolution styles, like most forms of capturing architectural expertise, are likely to be most valuable in cases such as this, because their role is to provide expertise that the architects lack. If the architects are already experts in the domain, evolution styles will likely be of less help. Our experience in the case study seemed to support the idea that evolution styles might be able to compensate for lack of evolution domain expertise. Quite a bit of our design time was taken up in researching the architectural element types relevant to the domain, understanding how they could fit together, and working out the best way to evolve the system in the chosen direction. For example, we considered a number of possible ways to connect the pieces of the system together; to do so, we had to gain an understanding of the available connector types, such as network protocols and interprocess communication channels. We spent quite a bit of time simply understanding the basic constraints of these connector types. Which transport protocols (TCP, UDP, SCTP) support message-based communication? Does a named pipe admit multiple writers? These are the kinds of questions whose answers could be easily captured in a style.

Another example of architectural expertise that would have been useful to have is an understanding of how the system would scale. How many users can an extra-large high-CPU EC2 instance running our analysis engine support? What data must we gather to answer the question? Does it make more sense to rent a single extra-large high-CPU instance, or four medium high-CPU instances for the same price? These are questions which we lack the expertise to answer, but which an experienced cloud architect could address easily. They are quite well suited to formalization as evolution style analyses. An evaluation function could take as input information on the expected resource requirements of the software and then, based on the planned evolution path, estimate how many users we will be able to support at each stage of evolution. Or another analysis could take in a probability distribution expressing how many users we are likely to have in the future, and tell us how many on-demand instances we should run, and how many we should rent at the cheaper but more restrictive reserved-instance rate.

5.3 Limitations

Architecture evolution presents special validation challenges different from those in many other subfields of software engineering. In general, a good method for evaluating a software engineering model is to test it and see if it produces better results. For example, to evaluate a software evolution technique, the ideal approach is to carry out evolutions with and without the aid of the technique, and see if the use of the technique produces better results. However, architecture evolutions usually take place over long periods, often years, so the prospect of evaluating an architecture evolution model in this way is very different from that of evaluating, say, a new source control plug-in. Indeed, we expect that our architecture evolution work will be most useful for the largest evolutions—those which take years of work and hundreds of personnel. Our path-based approach is likely to be less appealing for small-scale evolutions, because the overhead may exceed the benefits gained. Thus, a fair experimental evaluation of our approach would require us to enlist a large organization to adopt our approach for a major evolution and observe the results over the subsequent years or even decades—and ideally to do this many times, to ensure that the result is robust. Obviously, this is impractical, so we must look to other evaluative methods and see what they can tell us about the usefulness, practicality, effectiveness, cost, and feasibility of our approach.

Given the limitations at hand, there are a number of approaches we can take to evaluation. In this paper, we have taken two separate approaches: a theoretical evaluation of the properties of part of our modeling apparatus (section 4) and a small, controlled case study. Although far more modest than the grand experiment we imagined in the previous paragraph, there is still a lot this case study can tell us—provided that we are cautious about interpreting the results and do not overreach. In this section, we consider the most significant limitations of the case study that might complicate its interpretation.

A significant problem in interpreting any case study is that of generalizability, or external validity. Unlike research based on quantitative methods, a case study does not attempt to study any sort of representative sample of a population; rather, the goal in a case study is to undertake a detailed examination of a single case, and draw broader inferences based on what is appropriate given the context of the case study. In a case study, the aim is analytic, rather than statistical, generalization.

Here, generalizability is of particular concern because the case study was an artificial one. In addition, the case study was small in size (in terms of both the size of the system and the size of the development team—one person) and relatively straightforward in concept. The external validity of the case study thus hinges in part on the question of how true to life it was—and, insofar as the case study differed from a real-world project, how relevant those differences are to the research questions the case study attempts to address.

One of the reasons we selected this particular example is that it bears some structural similarities to real-world evolutions, such as the kind of scenario we described in section 2.1. Like an algorithmic-trading platform, a chess engine has a variety of components with different requirements. Some parts are processing-intensive (like the main search algorithm responsible for finding good moves), some are disk-intensive (a six-piece endgame tablebase, for example, requires over a terabyte), some are memory-intensive (such as endgame tablebases and transposition tables), and some are not particularly demanding of computing resources but may have other special requirements (such as the user interface). As in the example of section 2.1, trading platforms are often similarly differentiated, with computation engines that require complex, real-time calculation; market data that may require vast amounts of storage; trade placement components that require fast and absolutely reliable network connections; and so on. This sort of differentiation of components is typical of a large class of software systems, and cloud computing makes particular technical sense for systems that have this characteristic. (As mentioned in the case study, cloud computing platforms allow deployment on different hardware depending on the needs of the application or component; for example, Amazon EC2 has “high-memory” and “high-CPU” instances, as well as instances that also have increased network performance for high-performance computing.) Thus, a migration of a chess engine to the cloud makes sense for some of the same technical reasons as a cloud migration of a quantitative-trading platform, and the evolution operators and constraints that arise are broadly analogous.

Aside from the structural similarities, the systems have some of the same quality attribute goals. Some of the reasons one might wish to migrate a chess engine to the cloud are the same reasons one might wish to migrate a trading platform: scalability, cost, agility, reliability, and ease of use.

Obviously, this is a rough analogy, and we must not carry it too far. There are great differences in scale and risk (if a chess engine miscalculates, the worst that can happen is checkmate—but a bad trade can lose millions of dollars). Nonetheless, we can expect this case study to give us insight into cloud evolutions beyond the narrow domain of the original system.

Another limitation of the case study, in addition to its small size and artificial purpose, is that the analysis was retrospective; that is, we carried out the evolution first and the analysis later. This is the inverse of the normal usage of our approach (in which an architect builds and analyzes a model in order to carry out the evolution as effectively as possible). This methodological choice was necessary to make the case study practical with the available resources, but we must keep it in mind when interpreting the case study.

Of course, while it is natural to focus on the question of *whether* a case study is generalizable, a more precise question is: *To what extent* (i.e., to what class of architecture evolutions) is the case study generalizable? The case study generalizes most easily to systems that are similar to the one in the case study. Thus, when attempting to generalize this case study to another situation, we can be the most confident when that situation is similar to this one—for example, another small-scale evolutions of a desktop application to a cloud-computing platform. The more dissimilar a situation is from this scenario, the harder it is to apply our results. For example, one important characteristic of the system in this case study is that its responsibilities could be easily separated into discrete components. Quite different would be a system with responsibilities delineated in some other manner (for example, a “spaghetti architecture” with no clear separation of responsibilities, or an aspect-oriented system that encapsulates cross-cutting concerns), so we cannot apply our results to such a scenario.

5.4 Results

In section 5.1, we presented three research questions in which we were primarily interested. We now return to them and discuss the extent to which we have answered them, bearing in mind the limitations discussed in section 5.3.

1. *Can our model be applied to the analysis of a real evolution?* This case study was *real* only in a very limited sense; it was never intended to result in a running system that would have real users. Nonetheless, it was an evolution that was really carried out, as opposed to one that was merely imagined; in this respect it is considerably more real than much of the previous work in this area.
   In this case study, we did not encounter any serious limitations of our approach that would establish a negative answer to this research question. That is, our model turned out to be quite adequate for modeling the evolution we carried out in this case study, and we have no reason to believe that less artificial examples would present any additional difficulties. This answer is not quite as good as applying our approach to a real evolution of a real system with real users, but it is pretty good; our case study was designed to be realistic if not entirely real, and if there were to be applicability problems with our approach, it seems likely that they would have appeared here.
2. *Is our path constraint language expressive enough to capture the constraints that naturally arise in a real evolution?* Again with a caveat about the interpretation of the word *real*, we can give a qualified affirmative answer. We were able to model the constraints that naturally arose during our case study with relative ease. We do not anticipate any obstacles that would prevent this result from continuing to hold in a real-world evolution, but we cannot know the outcome of such an application with certainty.
3. *Is it likely that a model like ours, with appropriate tools to implement it, could be useful for planning evolutions?* This question is a little more elusive than the other two, because we did not use the model to carry out the evolution. Instead, we have to look back at the unaided evolution in retrospect and ask whether we would have had an easier time if we had used our model (assuming we had the tools necessary to apply it effectively). In section 5.2, we argued that the architectural expertise that could be captured in an evolution style likely would have been useful in this evolution. If anything, we would expect this kind of expertise to be even more useful in a larger, less artificial evolution with more uncertainties.

Thus, despite the limitations inherent to this kind of case study, there is some information it can give us about the applicability of our approach. Based on the results of the case study, it seems that our model of architecture evolution may be useful in practice.

6 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 [53], indications of maintainability such as coupling and cohesion [4,69], code refactoring [50], and many others [30]. 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 [14], and design patterns that support maintainability [24].

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. 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 [8] 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 [1,34], 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 [11] 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 [22]. 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 [34,62,68]. For example, Wermelinger uses category theory to describe how transformations can occur in software architecture [68]. 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 [34] 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 [61] 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 architecture evolution operators, they differ from our approach in that 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 (like our path constraints), they do not address issues of evolution quality (like our evaluation functions).

In recent years, Tamzalit, Le Goaer, and others have investigated recurring patterns of architecture evolution, primarily with respect to component-based architectures [41,42,51,63,64]. 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 trade-off analysis for architectural evolution. The work of Kazman et al. [38] applies architectural analysis and trade-off 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. Ozkaya et al. [52] propose to use techniques from option 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.

Brown et al. [12] present an approach to iterative release planning based on analysis and selection of development paths, where each development path consists of a sequence of releases. Their analysis is based on measurement of architectural design dependencies as represented by design structure matrices. Like us, they approach the problem of software architecture evolution from the perspective of analyzing and selecting among candidate paths. However, they do not support definition of evolution constraints, nor do they attempt to capture domain-specific evolution expertise.

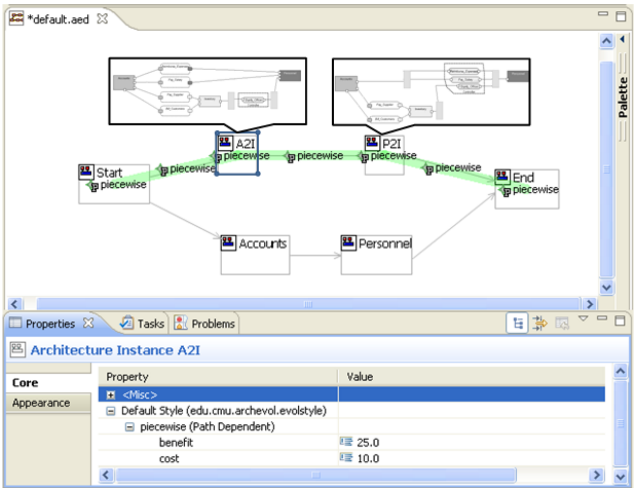
There is also work that addresses architecture evolution in the context of a specific style, such as Darwin [43] and C2 [65]. Like our approach, this work can take advantage of domain-specific classes of systems, and thereby achieve analytic leverage, as well as tool support for evolution. However, these approaches are limited to a particular architectural style.

7 Future work

Our ongoing work is devoted to elaborating the definition of evolution styles by enhancing the concepts of evolution operators and evolution analyses. We would like to explore other ways of specifying evolution operators declaratively, perhaps in the style of graph grammars [68] or rewrite rules [36]. Furthermore, we plan to develop and explore better ways to analyze evolution paths, perhaps considering approaches from economic option theory. Another class of analyses that we envision is based on the idea of *technical debt* introduced by Cunningham [18]. Technical debt is often advanced as a concept but seldom treated formally. We believe it is amenable to formal analysis at the architectural level and that such analysis could be useful in planning evolutions (e.g., analyzing trade-offs between taking on technical debt, gaining a short-term advantage, and avoiding technical debt, obviating the need to "repay" it later). Our model provides a sound foundation for a wide range of such path analysis techniques.

In addition, we would like to relax some of the restrictions that our current approach imposes, particularly the assumption that the target architecture is known in advance. In reality, there may be multiple candidate target architectures. Indeed the ultimate architecture may be unknowable at the outset of the evolution, because future developments could depend on contingencies that may or may not occur during the evolution. We believe that our model could be expanded to account for such uncertainties.

Another area that merits exploration is the use of planning to generate candidate paths automatically. Since an evolution path is simply a composition of well-defined evolution operators, and since our path constraints provide an automatically checkable notion of path correctness, it may be possible to



Architecture Instance A2I

Property	Value
<Misc>	
Default Style (edu.cmu.archevol.evolstyle)	
piecewise (Path Dependent)	
benefit	25.0
cost	10.0

Fig. 8 The Ævol workbench.

use a planning approach similar to those discussed in [29] to generate alternative paths automatically.

A final promising topic for future research is linking architecture evolution to the code level. Of course, there exists a great deal of research on linking software architecture to code in general (e.g., [46,58]), but architecture evolution presents special challenges and opportunities. A worthy goal would be to tie architectural operators to code transformations.

7.1 Tool support

To support our research, we have developed two partial prototype implementations based on the software architecture evolution concepts we have presented here. We described them in two previous conference papers [5,28]. The first tool, Ævol [28], was a plug-in framework that we developed in 2009 to support basic evolution planning and analysis. Architects can define an evolution graph in Ævol and link its nodes to system architecture descriptions that are represented in AcmeStudio [57], an editor for Acme.

Figure 8 is a screenshot of 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 semitransparent, thick lines in the diagram. 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.

Although Ævol provided a vehicle to explore several of the key concepts we have discussed in this paper, including definition of candidate evolution paths and evaluation of path constraints, it lacked other key features such as support for multiple architectural views and a robust mechanism for defining evolution operators.

In more recent work, the first author carried out a case study at NASA's Jet Propulsion Laboratory, in which he extended a commercial, off-the-shelf UML modeling tool, MagicDraw [47], to model an architecture evolution. This work is described in a 2012 conference paper [5]. This prototype supports a number of new features, such as describing an architecture evolution in terms of architectural transformations between states and documenting multiple architectural views. However, it is missing support for some of the key concepts of our approach, including path constraints and evaluation functions.

Neither of these implementations has reached an adequate stage of maturity to support an architect in carrying out a real evolution. Thus, tool development is a priority for us as we continue our research. We are currently engaged in a follow-on project to extend the MagicDraw-based tool to add support for some of the missing features and to improve the usability of the tool so that it could be used by practitioners.

8 Conclusion

In this paper we outlined foundations for 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, allowing for support by tools.

We evaluated this approach for software architecture evolution modeling by two very different methods. First, we formally evaluated the computational complexity of model-checking evolution path constraints. This theoretical study showed that our approach to verifying path validity is computationally feasible (as long as the variable-nesting depth of path constraints is suitably limited). Second, we demonstrated the applicability of our approach on a small case study that exemplifies the kinds of concerns that arise in real-world evolutions. These two kinds of evaluation complement each other to provide evidence for the practicality of our approach. However, future work would be useful to understand more fully the domains and situations for which this kind of architecture evolution modeling technique is most useful.

Acknowledgements We are grateful to the Software Engineering Institute, IBM, and NASA's Jet Propulsion Laboratory for providing support for this work. We would especially like to thank Ipek Ozkaya, Sridhar Iyengar, S Sivakumar, and Brian Giovannoni. Finally, we would like to thank the anonymous reviewers for their constructive suggestions, which improved the manuscript greatly.

References

1. Abi-Antoun, M., Aldrich, J., Nahas, N., Schmerl, B., Garlan, D.: Differencing and merging of architectural views. In: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE 2006), pp. 47–58. IEEE, Los Alamitos (2006)
2. Alur, R., Henzinger, T.A.: A really temporal logic. In: Proceedings of the 30th Annual Symposium on Foundations of Computer Science, pp. 164–169. IEEE (1989)
3. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/
4. Baldwin, C.Y., Clark, K.B.: Design Rules, vol. 1. MIT, Cambridge (1999)
5. Barnes, J.M.: NASA's Advanced Multimission Operations System: A case study in software architecture evolution. In: Proceedings of the 8th International ACM SIGSOFT Conference on the Quality of Software Architectures (QoSA'12), pp. 3–12. ACM (2012)
6. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM T. Softw. Eng. & Meth. **20**(4) (2011)
7. Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P., McKenzie, P.: Systems and Software Verification: Model-Checking Techniques and Tools. Springer, Berlin (2001)
8. Berliner, B.: CVS II: Parallelizing software development. In: Proceedings of the Winter 1990 USENIX Conference, pp. 341–352. USENIX, Berkeley (1990)
9. Blackburn, P.: Internalizing labelled deduction. J. Logic & Comput. **10**(1), 137–168 (2000)
10. Blackburn, P., Tzakova, M.: Hybrid languages and temporal logic. Log. J. IGPL **7**(1), 27–54 (1999)
11. Boehm, B.W.: Software Engineering Economics. Prentice Hall, Upper Saddle River (1981)
12. Brown, N., Nord, R.L., Ozkaya, I., Pais, M.: Analysis and management of architectural dependencies in iterative release planning. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'11), pp. 103–112. IEEE (2011)
13. Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: OCL for the specification of model transformation contracts. In: Proceedings of the Workshop on OCL and Model Driven Engineering. University of Kent, Canterbury (2004). http://www.cs.kent.ac.uk/projects/ocl/oclmdewsuml04/papers/2-cariou_marvie_seinturier_duchien.pdf
14. Chaki, S., Sharygina, N., Sinha, N.: Verification of evolving software. In: Proceedings of the Workshop on Specification and Verification of Component Based Systems (SAVCBS 2004), pp. 55–61. Iowa State University, Ames (2004)
15. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM T. Progr. Lang. & Sys. **8**(2), 244–263 (1986)
16. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., , Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond. Addison–Wesley, Upper Saddle River, 2nd edn. (2011)
17. Crockford, D.: The application/json media type for JavaScript Object Notation (JSON). RFC 4627, IETF (2006). http://www.ietf.org/rfc/rfc4627
18. Cunningham, W.: The WyCash portfolio management system. In: Addendum to the Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92), pp. 29–30. ACM, New York (1992)
19. Debian: Package gnuchess (5.07-7) (2009). http://packages.debian.org/stable/gnuchess
20. Demri, S., Lazić, R., Sangnier, A.: Model checking memoryful linear-time logics over one-counter automata. Theor. Comput. Sci. **411**(22–24), 2298–2316 (2010)
21. Ecma International: Standard ECMA-262: ECMAScript Language Specification, 5th edn. (2009). http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf
22. Erder, M., Pureur, P.: Transitional architectures for enterprise evolution. IT Pro **8**(3), 10–17 (2006)
23. Franceschet, M., de Rijke, M.: Model checking hybrid logics (with an application to semistructured data). J. Appl. Logic **4**(3), 279–304 (2006)
24. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison–Wesley, Boston (1994)
25. Garey, M.R., Johnson, D.S.: Computers and Intractability. Freeman, San Francisco (1979)
26. Garlan, D., Barnes, J.M., Schmerl, B., Celiku, O.: Evolution styles: Foundations and tool support for software architecture evolution. In: Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA 2009), pp. 131–140. IEEE (2009)
27. Garlan, D., Monroe, R., Wile, D.: Acme: An architecture description interchange language. In: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '97), pp. 169–183. ACM, New York (1997)
28. Garlan, D., Schmerl, B.: Ævol: A tool for defining and planning architecture evolution. In: Proceedings of the International Conference on Software Engineering (ICSE 2009), pp. 591–594. IEEE, Piscataway (2009)
29. Gerevini, A.E., Haslum, P., Long, D., Saetti, A., Dimopoulos, Y.: Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. Artif. Intell. **173**(5–6), 619–668 (2009)
30. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice Hall, Upper Saddle River (1991)
31. GNU Chess. http://www.gnu.org/software/chess/
32. Google App Engine. http://code.google.com/appengine/
33. Goranko, V.: Temporal logic with reference pointers. In: D.M. Gabbay, H.J. Ohlbach (eds.) Proceedings of the International Conference on Temporal Logic (ICTL '94), LNCS, vol. 827, pp. 133–148. Springer, Berlin (1994)
34. Grunske, L.: Formalizing architectural refactorings as graph transformation systems. In: Proceedings of the International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2005), pp. 324–329. IEEE, Los Alamitos (2005)

35. Henzinger, T.A.: Half-order modal logic: How to prove real-time properties. In: Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC'90), pp. 281–296. ACM (1990)
36. Inverardi, P., Wolf, A.L.: Formal specification and analysis of software architectures using the chemical abstract machine model. IEEE T. Software Eng. **21**(4), 373–386 (1995)
37. Ivers, J., Clements, P., Garlan, D., Nord, R., Schmerl, B., Silva, J.R.O.: Documenting component and connector views with UML 2.0. Tech. Rep. CMU/SEI-2004-TR-008, Software Engineering Institute, Pittsburgh (2004)
38. Kazman, R., Bass, L., Klein, M.: The essential components of software architecture design and analysis. J. Syst. & Software **79**(8), 1207–1216 (2006)
39. Lamport, L.: The temporal logic of actions. ACM T. Progr. Lang. & Sys. **16**(3), 872–923 (1994)
40. Laursen, E.: High-end trading strategists see cost savings in cloud computing. Inst. Investor (Jan. 2011). http://www.institutionalinvestor.com/Popups/PrintArticle.aspx?ArticleID=2750046
41. Le Goaer, O.: Styles d'Évolution dans les Architectures Logicielles. Ph.D. thesis, LINA, Nantes (2009)
42. Le Goaer, O., Tamzalit, D., Oussalah, M.: Evolution Shelf: Reusing evolution expertise within component-based software architectures. In: Proceedings of the IEEE International Computer Software and Applications Conference (COMPSAC 2008), pp. 311–318. IEEE, Los Alamitos (2008)
43. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying distributed software architectures. In: W. Schäfer, P. Botella (eds.) Proceedings of the European Software Engineering Conference (ESEC '95), LNCS, vol. 989, pp. 137–153. Springer, Berlin (1995)
44. Markey, N., Schnoebelen, P.: Model checking a path. In: R. Amadio, D. Lugiez (eds.) Proceedings of the International Conference on Concurrency Theory (CONCUR 2003), LNCS, vol. 2761, pp. 251–265. Springer, Berlin (2003)
45. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE T. Software Eng. **26**(1), 70–93 (2000)
46. Murphy, G.C., Notkin, D., Sullivan, K.: Software reflexion models: Bridging the gap between source and high-level models. In: Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '95), pp. 18–28. ACM, New York (1995)
47. No Magic, Inc.: MagicDraw. http://magicdraw.com/
48. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT). http://www.omg.org/spec/QVT/
49. OMG: Unified Modeling Language (UML). http://www.omg.org/spec/UML/
50. Opdyke, W.F., Johnson, R.E.: Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In: Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA 1990), pp. 145–160. Marist College, Poughkeepsie (1990)
51. Oussalah, M., Sadou, N., Tamzalit, D.: SAEV: A model to face evolution problem in software architecture. In: L. Duchien, M. D'Hondt, T. Mens (eds.) Proceedings of the International ERCIM Workshop on Software Evolution 2006, pp. 137–146. USTL, Lille (2006)
52. Ozkaya, I., Kazman, R., Klein, M.: Quality-attribute-based economic valuation of architectural patterns. Tech. Rep. CMU/SEI-2007-TR-003, Software Engineering Institute, Pittsburgh (2007)
53. Parnas, D.L.: Information distribution aspects of design methodology. In: Proceedings of IFIP Congress '71, pp. 339–344. North-Holland, Amsterdam (1972)
54. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Eng. Notes **17**(4), 40–42 (1992)
55. Richardson, J.: Supporting lists in a data model (a timely approach). In: Proceedings of the International Conference on Very Large Data Bases (VLDB'92), pp. 127–138. Morgan Kaufmann, San Mateo (1992)
56. Rushby, J.: Bus architectures for safety-critical embedded systems. In: T.A. Henzinger, C.M. Kirsch (eds.) Proceedings of the International Workshop on Embedded Software (EMSOFT 2001), LNCS, vol. 2211, pp. 306–323. Springer (2001)
57. Schmerl, B., Garlan, D.: AcmeStudio: Supporting style-centered architecture development. In: Proceedings of the International Conference on Software Engineering (ICSE 2004), pp. 704–05. IEEE, Los Alamitos (2004)
58. Sefika, M., Sane, A., Campbell, R.H.: Monitoring compliance of a software system with its high-level design models. In: Proceedings of the International Conference on Software Engineering (ICSE 1996), pp. 387–396. IEEE, Los Alamitos (1996)
59. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Upper Saddle River (1996)
60. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. J. ACM **32**(3), 733–749 (1985)
61. Spitznagel, B., Garlan, D.: A compositional approach for constructing connectors. In: R. Kazman, P. Kruchten, C. Verhoef, H. van Vliet (eds.) Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001), pp. 148–157. IEEE, Los Alamitos (2001)
62. Spitznagel, B., Garlan, D.: A compositional formalization of connector wrappers. In: Proceedings of the International Conference on Software Engineering (ICSE 2003), pp. 374–384. IEEE, Los Alamitos (2003)
63. Tamzalit, D., Oussalah, M., Le Goaer, O., Seriai, A.D.: Updating software architectures: A style-based approach. In: H.R. Arabnia, H. Reza (eds.) Proceedings of the International Conference on Software Engineering Research & Practice (SERP'06), pp. 336–342. CSREA, Las Vegas (2006)
64. Tamzalit, D., Sadou, N., Oussalah, M.: Evolution problem within component-based software architecture. In: Proceedings of the 18th International Conference on Software Engineering & Knowledge Engineering (SEKE 2006), pp. 296–301. Knowledge Systems Institute, Skokie (2006)
65. Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead Jr., E.J., Robbins, J.E., Nies, K.A., Oriezy, P., Dubrow, D.L.: A component- and message-based architectural style for GUI software. IEEE T. Software Eng. **22**(6), 390–406 (1996)
66. Twilio. http://www.twilio.com/
67. Varia, J.: Migrating Your Existing Applications to the AWS Cloud. Amazon Web Services (2010). http://media.amazonwebservices.com/CloudMigration-main.pdf
68. Wermelinger, M., Fiadeiro, J.L.: A graph transformation approach to software architecture reconfiguration. Sci. Comput. Program. **44**(2), 133–155 (2002)
69. Yourdon, E., Constantine, L.L.: Structured Design. Prentice Hall, Upper Saddle River (1979)