Automated Planning for Software Architecture Evolution

Jeffrey M. Barnes, Ashutosh Pandey, and David Garlan
Institute for Software Research
Carnegie Mellon University, Pittsburgh, PA
jmbarnes@cs.cmu.edu · ashutosp@cs.cmu.edu · garlan@cs.cmu.edu

***Abstract*—In previous research, we have developed a theoretical framework to help software architects make better decisions when planning software evolution. Our approach is based on representation and analysis of candidate evolution paths—sequences of transitional architectures leading from the current system to a desired target architecture. One problem with this kind of approach is that it imposes a heavy burden on the software architect, who must explicitly define and model these candidate paths. In this paper, we show how automated planning techniques can be used to support automatic generation of evolution paths, relieving this burden on the architect. We illustrate our approach by applying it to a data migration scenario, showing how this architecture evolution problem can be translated into a planning problem and solved using existing automated planning tools.**

I. Introduction

Software architecture—the discipline of designing the high-level structure of a software system—is today widely recognized as an essential element of software engineering. However, one topic that today's approaches to software architecture do not adequately address is *software architecture evolution*. Architectural change occurs in virtually all software systems of significant size and longevity. As systems age, they often require redesign in order to accommodate new requirements, support new technologies, or respond to changing market conditions. At present, however, software architects have few tools to help them plan and carry out such evolution.

In our previous research, we have developed an approach to support architects in reasoning about evolution [1], [2], [3], [4]. In our model, the architect considers a set of candidate *evolution paths*—sequences of transitional architectures leading from the current state to a desired target architecture—and a tool helps the architect to select which path best meets the evolution goals.

A significant limitation of this approach, and other similar approaches that have been proposed [5], [6], [7], is that they impose a substantial burden on the architect. The architect must explicitly define the candidate evolution paths and specify the architectural transformations within each such path. In this way, the architect fully defines the *evolution space* of possible evolutions under consideration, permitting various kinds of automated analysis. However, in a scenario with many candidate evolution paths, and numerous transitions within each path, this can be an onerous task.

A better approach would be to generate these evolution paths automatically. Rather than fully specifying the evolution space, the architect could simply define the initial and target architectures; then a tool could select architectural transformations from a predefined library of domain-relevant *evolution operators* and apply them in sequence to generate candidate paths from the initial architecture to the target architecture.

While this would alleviate the burden on the architect, it introduces a new difficulty: determining how to compose these operators together so as to generate the target architecture from the initial architecture. (Given n operators, each with m parameters ranging over a domain of d architectural elements, there are $(nd^m)^l$ evolution paths of length l. Clearly an undirected brute-force search for an optimal path would be unwise.) This problem is very much akin to the *planning problem* in artificial intelligence [8]: given a description of the state of the world, a goal, and a set of actions, how can we generate a *plan*—a sequence of actions leading from the initial state to the goal?

In this paper, we describe our attempt to apply existing approaches and tools from automated planning to the architecture evolution path generation problem. Adapting these existing approaches to software architecture evolution is a difficult problem, as it requires consideration of a number of concepts—architectural changes, technical and business constraints, rich temporal relationships among events, trade-offs among evolution concerns—that do not translate easily into the planning domain.

The paper is organized as follows. Section II presents necessary background on architecture evolution and automated planning. Sections III through V present our main contributions:

- a systematic approach for translating architecture evolution problems into automated planning problems (Section III);
- an application of the approach to a scenario based on a real-world evolution problem, which we use to evaluate the practicality and efficacy of the approach (Section IV); and
- a discussion of the fundamental challenges involved in applying automated planning technology to software architecture evolution (Section V).

Finally, Section VI reviews related work, and Section VII concludes with a discussion of future work.

II. Background

A. Software Architecture

Software architecture is the subdiscipline of software engineering that pertains to the overall structure of a software system. Software architects represent software systems in terms of the high-level elements of which they are made. At a basic level,

978-1-4799-0215-6/13 © 2013 IEEE

ASE 2013, Palo Alto, USA

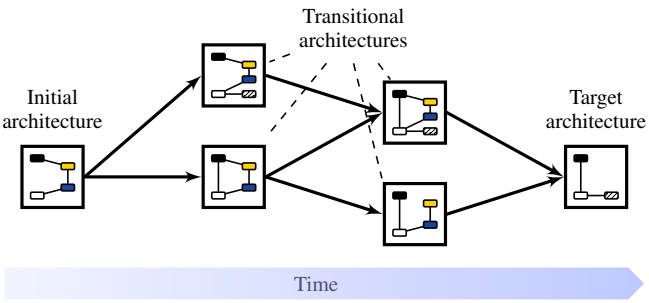

Fig. 1. A depiction of an evolution graph. Each node is a complete architectural representation of the system. Edges represent possible evolutionary transitions. The architect's task is to select the optimal path through the graph.

a software architecture can be thought of as an arrangement of *components* (the computational elements and data stores of a system) and *connectors* (interaction pathways among components) [9]. Although software architecture is a relatively young field, it has grown rapidly in importance and influence. Today, software architecture is practiced in some form at nearly all real-world software organizations of significant size.

B. Software Architecture Evolution

The problem of understanding software architecture evolution, however, has just begun to be explored. In recent years, we and other researchers have been working to develop techniques and tools for understanding and modeling software architecture evolution [1], [2], [5], [6], [7]. In this section, we describe the approach that we have developed in our own research [1], [2]. However, many other approaches share conceptual similarities with our own, and so the general principles that we describe in the following sections are, by and large, applicable to these other approaches as well. We discuss this further in Section VI-A.

Our approach is based on considering possible *evolution paths* from the initial architecture of the system (as it exists at the outset of the evolution) to the target architecture (the desired architecture that the system should have when the evolution is complete). Each such evolution path can be represented as a sequence of *transitional architectures* leading from the initial architecture to the target architecture. We can represent and relate these evolution paths within an *evolution graph* whose nodes represent transitional architectures and whose edges represent the possible transitions among them (Fig. 1). These transitions, in turn, may be understood as sequences of *evolution operators*—reusable architectural transformations such as *add adapter* or *migrate database*.

Once the evolution graph is defined, the next step is to apply analyses to select the optimal path—the one that best meets the evolution goals, while adhering to any relevant technical and business constraints, and subject to concerns such as cost and duration. To support the architect in selecting a path, we provide two kinds of analysis: *evolution path constraints*, which define which paths are legal or permissible, and *path evaluation functions*, which provide quantitative assessments of qualities such as duration and cost. Operators and analyses are generally specific to particular domains of evolution; for example, an evolution of a desktop application to a cloud-computing platform will have different operators and analyses than an evolution of a thin-client/mainframe system to a tiered web services architecture.

Further details are given elsewhere [2]. Here it suffices to observe that while the analysis step (i.e., the execution of constraints and evaluation functions) is easily automatable, the definition of the evolution graph (i.e., the definition of evolution paths in terms of evolution operators and transitional architectures) is a manual and time-intensive process. This limits the practical applicability of this kind of approach, since in many cases it may be difficult to justify the time and effort necessary to model the evolution paths under consideration.

C. Automated Planning

Given a set of states S, a set of actions $A : S \to S$, an initial state $s_0 \in S$, and a set of goal states $S_g \subseteq S$, the *planning problem* is the task of finding a sequence of actions that, when applied to s_0, yield one of the goal states.[1] The planning problem has broad applications, from robotics to business management to natural language generation, and has received a great deal of attention from artificial-intelligence researchers. A variety of approaches and tools for solving planning problems have been developed over the last several decades.

To solve a planning problem, a planner must receive a specification of the problem in a standard format. A number of specification languages for planning problems have been devised, but by far the most popular—the lingua franca of automated planning—is the Planning Domain Description Language. PDDL was first introduced in 1998 [10] and soon became a de facto standard in the planning literature, facilitating reuse of research and allowing easy comparison of planners, systems, and models [11]. These qualities, along with its feature set, make PDDL a good choice for our work.

PDDL has undergone several revisions. The version that we adopt in this paper is PDDL2.1 [11], introduced in 2002, which greatly enhanced the language's expressivity by introducing:

- *numeric fluents*, which provided full support for modeling numerically valued resources such as fuel and distance;
- *durative actions*, which greatly enriched the temporal expressiveness of the language; and
- *plan metrics*, which allowed specification of a metric with respect to which a plan should be optimized (e.g., minimize fuel consumption).

All three of these are extremely useful for modeling architecture evolution problems (as we will see later). Most of PDDL2.1 is now reasonably well supported by the leading planners. There have subsequently been further additions to the language, such as the introduction of derived predicates in PDDL2.2 [12] and constraints and preferences in PDDL3 [13]. While these features would certainly be useful to us, they are not as broadly supported by planners, so we chose to target PDDL2.1.

[1]This is a very abstract formulation of the planning problem. For a discussion of alternative definitions, including some that are more computationally oriented, see Ghallab et al. [8].

D. Structure of a PDDL Specification

A PDDL specification comprises two parts, which appear in separate files: a *domain description* (consisting chiefly of a description of possible actions that characterize domain behaviors) and a *problem description* (consisting of the description of specific objects, initial conditions, and goals that characterize a problem instance). Thus, a domain description can be shared across multiple planning problems in the same domain. Both the domain file and the problem file are expressed in a Lisp-like syntax, as a list of parenthesized declarations.

In PDDL2.1, a domain file can declare:

- A set of *types* to which objects may belong. Each type may optionally declare a supertype. If a type does not declare a supertype, it is deemed to be a subtype of the built-in type Object; all types are ultimately subtypes of Object (perhaps indirectly). A type is simply a name; it does not define a set of properties or methods. Rather, predicates, functions, and actions can specify the types that they govern.
- A set of *predicates* over objects.
- A set of *functions* that map $\mathsf{Object}^n \to \mathbb{R}$.
- A set of *action* schemata, each comprising a list of parameters, the conditions under which the action may be taken, and the effects of the action. A *durative action* additionally specifies its duration.

A problem file declares:

- A list of *objects*.
- The *initial conditions*, consisting of truth assignments for predicates and numeric value assignments for functions.
- The *goals*, which are defined in first-order predicate logic.
- A *metric* to be minimized or maximized.

A planner takes a domain description and problem description as input and produces a *plan* as output—a timed list of actions (with parameters specified) that achieves the specified goals.

III. Approach

The problem of generating an evolution path from an initial architecture to a target architecture can be framed as a planning problem in the sense of Section II-C as follows:

- S, the set of states, is defined to be the set of legal software architectures.
- A, the set of actions, is defined to be the set of evolution operators.
- s_0, the initial state, is defined to be the initial architecture.
- S_g, the set of goal states, is defined to be the singleton set consisting of the target architecture of the system.

With the problem framed in this manner, we can apply automated planning tools to the task of generating evolution paths.

In the remainder of this section, we will describe an approach for translating an architecture evolution problem into a planning problem expressed in PDDL. (A summary appears in Table I.) In Section IV, we will make this discussion concrete by showing how we applied it to a specific architecture evolution problem and used off-the-shelf planners to generate evolution paths.

TABLE I
Summary of Our Approach for Translating Elements of an Architecture Evolution Problem into PDDL

Evolution Element	PDDL Translation
Transitional architecture	State
Architectural element type	Object type
Architectural element	Object
Relationship among architectural elements	Predicate
Evolution operator	Action
Parameter	Action parameter
Precondition	Action condition
Architectural transformation	Action effect
Property	Action duration, or action effect modifying function value
Evolution path	Plan
Initial architecture	Initial state
Target architecture	Goal state
Path constraint	PDDL3 constraint, or action condition supported by predicates to track the state
Path evaluation function	Metric

A. Representing the Initial and Target Architectures

The first step of modeling an architecture evolution problem is to specify the initial and target architectures. As noted in Section II-A, a software architecture is conventionally conceived as an arrangement of *components* and *connectors*. (Of course, this is a simplification. Architectural specifications may be enriched in various ways, for example by adding further elements such as ports and roles, or by decomposing architectural elements to reveal their substructure. We will see an example of architectural decomposition in Section IV.) These components and connectors are often expressed in terms of *component types* (such as *WebService* or *Database*) and *connector types* (such as *EventBus* or *HttpConnection*).

PDDL's type system, though simple, is quite adequate for our needs. We can define component and connector types as types in the PDDL domain description, then define the components and connectors themselves as PDDL objects of the defined types. Finally, the relationships among the components and connectors can be expressed using predicates, which are defined in the domain description and assigned truth values in the problem description. Fig. 2 shows a simple example.

The specification of the initial architecture will appear within the **:init** block, which defines the initial conditions, and the specification of the target architecture will appear within the **:goal** block, which defines the goals.

B. Representing Evolution Operators

An evolution operator, of course, corresponds to a PDDL action. But how can we actually capture an evolution operator as an action using the specification facilities that PDDL provides?

In our model of architecture evolution, an operator comprises:

- A set of *parameters*. For example, a *wrap legacy component* operator will take as a parameter the component to wrap. In PDDL, an action likewise specifies its parameters.
- A description of the *architectural transformations* that the operator effects. These are expressed as a sequence of elementary architectural changes such as *delete component*

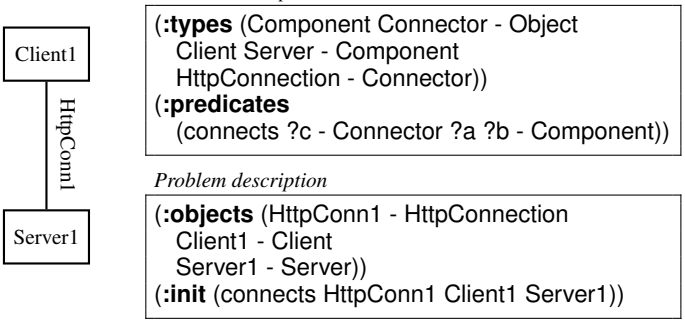

Fig. 2. An extremely simple software architecture and its PDDL representation.

or *attach connector*. In PDDL, we can represent these transformations via the action's effects.

- A description of the operator's *preconditions*. These map into PDDL in a straightforward manner; in PDDL, any action may declare its preconditions in terms of predicates and functions over the action parameters.[2]
- A list of *properties* of the operator, used to support evaluation functions. Examples of properties are the time needed to carry out the operator, the cost of doing so, and the operator's effects on system performance. In PDDL, the duration property is given special prominence due to its importance in temporal reasoning; a durative action must specify its duration. As for the other properties, these are best captured via PDDL effects. For example, if an evolution operator has a cost property (indicating that it costs $1,000), we can define a cost function in the PDDL specification, then add an **increase** cost 1000 effect to the action.

One subtlety worth noting is that PDDL does not permit actions to create new objects (nor destroy existing ones). This is significant because many evolutions entail the creation of new architectural elements, or the decommissioning of existing ones. As a result, in an evolution that may involve creation of new elements, we must declare some *potential* objects that do not exist in the initial architecture but may be used to stand in for elements created during the evolution. In this case, we can define an isReal predicate that is false for such potential objects and becomes true when an action creates a new architectural element out of a potential object. Such approaches have substantial limitations and are rather cumbersome, and Frank et al. [14] identify this as an important limitation of PDDL. (A related point is that a PDDL specification can have only finitely many objects, while the set of software architectures reachable via a set of evolution operations may be infinite in general.)

C. Representing Path Constraints

Path constraints are perhaps the most challenging element of an architecture evolution problem to translate into PDDL. In our previous work [2], we have represented path constraints using an extension of linear temporal logic (LTL). Temporal logic provides a natural way of representing a wide variety of path constraints. For example, a constraint such as "the legacy bus must not be removed until the new enterprise service bus is installed" can be quite simply represented in LTL by the formula

$$legacyBusPresent\ \mathcal{U}\ esbInstalled,$$

where *legacyBusPresent* and *esbInstalled* are predicates over architectural models. Unfortunately PDDL2.1 does not have any means to define constraints using temporal formulas.

One way of addressing this would be to develop a way of translating temporal formulas into PDDL directly. Indeed, there is previous work in this direction; Cresswell & Coddington [15] present a means of compiling an LTL goal formula into PDDL. They use a two-step process; first they generate a finite-state machine that accepts traces of the LTL formula, then they encode this automaton as a collection of facts in PDDL and modify the actions to track the current state. This process is conceptually complex and encumbers the specification with numerous state variables. Therefore, we leave to future work the challenge of extending this compilation process to the augmented version of LTL that we use to capture path constraints.

To avoid such conceptual complexities here, we take a pragmatic approach: we characterize certain restricted classes of path constraints (with an eye toward the kinds of constraints that will arise in the example of Section IV) and show how they can be easily represented using the existing facilities of PDDL.

Constraints that must hold throughout an evolution. The simplest possible kind of constraint is one that must hold continuously through the entire duration of the evolution (e.g., a system must always be protected by a firewall, or a trusted component may never connect directly to an untrusted one). In LTL, such a constraint takes the form $\Box\phi$ for some propositional formula ϕ. Despite their simplicity, these constraints are quite common. Such a constraint amounts to an architectural constraint that persists through an evolution. In PDDL, we can model such a constraint easily (if verbosely) as a condition on every action.

Ordering constraints. Another common class of constraints comprises constraints that govern the order of the operations that are to be carried out in the course of an evolution. For example, a firewall must be installed before connections to a protected resource are permitted; a high-priority client should receive a service upgrade before a low-priority one. Such constraints are also generally easy to model in PDDL. If an operator B must be preceded by an operator A, then we can have action A set a predicate, aExecuted, that is a precondition for operator B.

Timing constraints. Constraints on the time at which evolution operations are carried out, or the time by which certain goals must be achieved, are extremely common in real-world evolution. In the simplest case, there may be a requirement that the evolution be completed by a specific date. In more complex cases, there may be a set of such requirements: feature A must be available for client 1 by April, feature B for client 2 by July,

[2]For durative actions, this is generalized to include other kinds of conditions—not only preconditions (conditions that must hold at the start of an action), but also conditions that must hold at the end of an action, or over its entire duration. These are specified with the temporal annotations **at start**, **at end**, and **over all**. (These can also be applied to effects.)

and so on. These can be modeled in PDDL by setting appropriate conditions on durative actions.

A more complex kind of timing constraint is a constraint that certain actions can be performed only at certain times. A real-world example is that many retailers, such as Amazon.com and Costco, refrain from making major software changes during the Christmas shopping season, so as not to introduce bugs during a period of heavy use. In Section IV, we will see another example, in which certain operations can be carried out only on certain days of the week. These are also expressible in PDDL. There are some challenges, however, which we explore in Section IV.

There are many constraints that do not fit into these categories, but in our experience, many of the constraints that arise in real-world evolutions do fall into these groups. In Section IV, we will see how various constraints can be represented in PDDL.

A final point to note is that PDDL3 has its own notion of a constraint. Like our constraints, PDDL3 constraints express conditions that must be met by an entire plan (in contrast with conditions in PDDL2.1, which are evaluated locally, with respect to a particular point in time). Moreover, these constraints are expressed in a syntax reminiscent of temporal logic, with operators such as always, sometime, at-most-once, and so on. However, there are substantial restrictions; most significantly, these modalities may not be nested. As a result, this constraint language is less expressive than LTL. Nonetheless, PDDL3 constraints would be a useful way of expressing a broad class of evolution path constraints. However, because we are targeting PDDL2.1 here, we do not discuss them further.

D. Representing Path Evaluation Functions

As described earlier, an evaluation function provides a quantitative evaluation of a path. There may be evaluation functions for various dimensions of concern, such as cost and availability. Ultimately, the architect's aim is to select the path that best meets the goals of the evolution. In an evolution with multiple competing concerns, we can define an evaluation function that captures a notion of *overall path utility*, which may be a weighted composite of primitive functions such as cost and availability.

All of this can be translated into PDDL. Evaluation functions such as cost and availability can be modeled as nullary functions in PDDL, and their values can be modified by actions as appropriate. Finally, we can use these values to set a *plan metric* in the problem description, which planners will try to optimize in generating a plan. This metric can simply be a reference to a function, or it can be an arbitrary arithmetic expression. The metric can also incorporate the total duration of the plan by using the built-in variable total-time.

IV. Application

To show how this approach can be used in practice, and to provide a demonstration of its applicability, we applied it to an evolution scenario. The scenario is based loosely on a real-world data migration experience that we had previously elicited (for other purposes) from a practicing software engineer. We elaborated this experience into a complete description of an architecture evolution problem, so that it would be specific enough to operationalize as a planning problem (Section IV-A). Then, using the approach described above, we translated this scenario into PDDL (Sections IV-B through IV-F).[3] Finally, we used two different off-the-shelf planners to generate plans and evaluated the results (Section IV-G).

A. Evolution Scenario

Our example is based loosely on a real-world data migration scenario, in which a company had to migrate a number of services from an old data center to a new data center. The planning for this migration was nontrivial, because there were a number of interacting constraints governing how the various services had to be moved. For example:

- Different services had different kinds of availability requirements. For example, some services had to be continuously available for regulatory reasons (zero planned downtime). In other cases, there were periods when certain services were required to be online (e.g., the payroll system had to be online at the end of each payroll period).
- Different services had to be moved in different ways. Some services (particularly those hosted on Unix systems) could be easily cloned into the new data center using the corporate storage area network. Other services were more finicky and could not be cloned automatically; manual intervention was required to migrate these services. And there were a few unique legacy services that were running on custom-built, special-purpose hardware. These services were so closely tied to the machines on which they were running that the only practical way to migrate them was to load the machines onto a truck and drive them to the new data center.
- No services could be established in the new data center until a firewall was installed there.

In the real-world experience on which our scenario was based, the architects experienced significant difficulty in managing these interacting constraints to develop a satisfactory plan. The planning process ultimately took roughly six months, and the migration itself was carried out over several weekends.

We elaborated this scenario by adding additional architectural details as necessary to create a complete specification of an architecture evolution problem. For example, although we had general information about the kinds of architectural elements and evolution constraints, we did not have a list of specific service names and locations, so we invented fictitious service names and assigned them to hosts at will.

The initial architecture is shown in Fig. 3. There are five hosts in data center DC1, each with one or more services, all of which must ultimately be migrated to DC2. We defined a number of specific evolution constraints based on the real-world constraints above. For example, we specified that the payroll service in Fig. 3 must be available on Mondays to permit payroll processing, and we defined rules governing how the services could be moved (e.g., Unix services can be cloned to

[3]Space constraints permit us to show only snippets of our PDDL specification here. However, for the sake of replicability, we have made our entire specification available at http://www.cs.cmu.edu/~jmbarnes/pddl/.

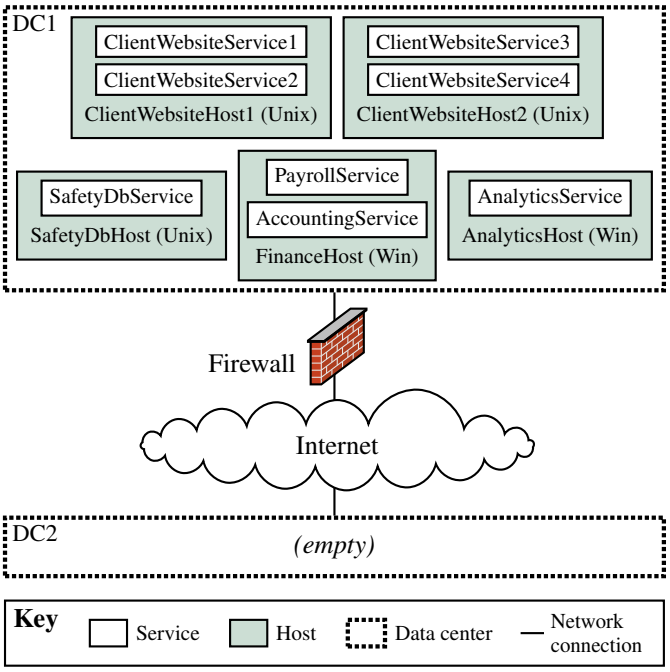

Fig. 3. Deployment view of the initial architecture of the data migration scenario.

a new data center over the network, but the analytics engine is tied to special-purpose hardware that must be physically relocated). We defined six evolution operators: *install network switch*, *install firewall*, *decommission host*, *clone host*, *manually transfer service*, and *physically relocate host*. Finally, we defined two ways of evaluating path quality: cost and duration. *Cost* is affected by when and how systems are migrated (migrating a system on weekends is more expensive than during normal working hours, and physically moving a host is much more expensive than cloning a host over the network). *Duration* refers to the overall time to complete the evolution.

B. Representing the Initial and Target Architectures

We represented the initial architecture following the approach described in Section III. In the domain description, we defined PDDL types for the architectural element types: DataCenter, Service, and Host (with subtypes UnixHost and WindowsHost). We defined predicates to indicate relationships among elements, such as an is-in predicate that holds when a given host is in a given data center and an is-on predicate that holds when a given service is on a given host. With these types and predicates defined, we were able to translate the initial architecture in Fig. 3 into a PDDL description of the initial state.

Representing the target architecture as a set of goal conditions, on the other hand, entails some subtleties. In principle, we could define the target architecture by the same method that we defined the initial architecture—specify exactly which services are on which hosts and which hosts are in which data center. In practice, however, this would be a bit too restrictive. Because services can be migrated in multiple ways—cloning, manual service-by-service migration, or physical relocation—there are actually multiple legal end states. For example, we could clone ClientWebsiteHost1 onto a new host in DC2 and decommission ClientWebsiteHost1, or we could instead move ClientWebsiteHost1 itself to DC2. These would result in slightly different end states, but either is permissible from the standpoint of path correctness; the choice should be left to the planner. Thus, we defined the goals of the evolution in more general terms; we defined a permissible end state to be one in which (1) all services end up in DC2 and (2) no hosts remain in DC1.

In principle, these goals are easy to represent in PDDL:

```
(:goal (and
  ; All services end up in DC2.
  (forall (?s - Service)
    (exists (?h - Host) (and (is-on ?s ?h) (is-in ?h DC2))))
  ; No hosts remain in DC1.
  (not (exists (?h - Host) (is-in ?h DC1)))))
```

Unfortunately, practical considerations prevent such a straightforward approach. Many planners—including the OPTIC planner that we use in Section IV-G—do not support goals with negative or existential operators. To get around this, we defined helper predicates such as was-migrated (to indicate that a service has been migrated) and was-removed-from (to indicate that a host has been removed from a data center) and modified the actions to update them throughout the evolution. We then declared our goals as follows:

```
(:goal (and
  ; All services end up in DC2.
  (forall (?s - Service) (was-migrated ?s))
  ; No hosts remain in DC1.
  (was-removed-from ClientWebsiteHost1 DC1)
  (was-removed-from ClientWebsiteHost2 DC1)
  (was-removed-from SafetyDbHost DC1)
  (was-removed-from FinanceHost DC1)
  (was-removed-from AnalyticsHost DC1)))
```

This complicated the domain description, but it allowed us to express our goals crisply despite the limitations of planners.

C. Representing the Evolution Operators

We represented the operators as actions in accordance with the approach described in Section III. Fig. 4 shows an example: the action for manual migration of a service from one host to another. Much of this is straightforward. The action first defines its parameters: the service being migrated, the hosts it is moving from and to, and the current day (we will explain this parameter in Section IV-D). Then it defines its duration: 3.9 hours.

Many of its conditions correspond directly to preconditions of the evolution operator. For example, to migrate a service s from host h_1 to host h_2, clearly s must be on h_1 at the outset. We also require that h_1 is in DC1 and h_2 is in DC2 (we only want to move services from DC1 to DC2), and we require that the firewall and network switch are already installed.

The conditions that reference no-work-in-progress, today, and time-since-last-day are used in modeling the passage of time, which we describe in Section IV-D.

Many of the effects are straightforward specifications of the evolution operator's architectural transformations. When the manuallyMigrateService operator is applied, the effect on the architecture is that service s is now on host h_2. We

```
(:durative-action manuallyMigrateService
  :parameters (?s - Service ?h1 ?h2 - Host ?d - Day)
  :duration (= ?duration 3.9)
  :condition (and
    (at start (is-on ?s ?h1))
    (over all (is-in ?h1 DC1)) (over all (is-in ?h2 DC2))
    (over all (has-firewall DC2))
    (over all (network-switch-installed))
    (at start (not-yet-migrated ?s))
    (over all (can-be-migrated-individually ?s))
    (over all (ok-to-move-on ?s ?d))
    (at start (no-work-in-progress))
    (over all (today ?d))
    (over all (>= (allowed-downtime ?s) 3.9))
    (at start (<= time-since-last-day 4.1)))
  :effect (and
    (at end (is-on ?s ?h2))
    (at end (was-migrated ?s))
    (at end (not (not-yet-migrated ?s)))
    (at end (not (is-unused ?h2)))
    (at start (not (no-work-in-progress)))
    (at end (no-work-in-progress))
    (at start (increase (total-cost) (* 20 (cost-multiplier ?d))))
    (at end (increase current-hour 3.9))
    (at end (increase time-since-last-day 3.9))))
```

Fig. 4. Expression of an evolution operator in PDDL.

also must set here a number of helper predicates, as mentioned in Section IV-B, such as was-migrated, not-yet-migrated, and is-unused. The effects that mention no-work-in-progress, current-hour, and time-since-last-day are used to support the modeling of the passage of time and will be discussed in Section IV-D. Finally, the effect that increases total-cost is used for cost optimization, described in Section IV-F.

D. Representing Time

The most difficult part of representing this scenario in PDDL was capturing its temporal aspects. The temporal features that PDDL provides fall well short of this scenario's needs. In particular, this scenario (like many evolution problems) is steeped in references to real-world time—that is, clock time, or calendar time. The payroll service must be available on Mondays; the accounting service can be moved only on weekends; operations are most expensive when carried out on weekends. PDDL is ill suited to representing such considerations. PDDL's conception of time is a continuous timeline, extending from zero to infinity. To reason about concepts such as Mondays and working hours, we must model them ourselves—and do so in a way consistent with PDDL's own model of time. This is rather difficult.

In this scenario, we are interested only in working hours; the company, in this scenario, has only day employees, and all work takes place between 9 a.m. and 5 p.m. We therefore interpret PDDL's timeline within an eight-hour day cycle; time indices between 0 and 8 represent Monday, times between 8 and 16 represent Tuesday, and so on.

This simplifies the specification because we do not need to model the empty nighttimes. However, it also creates some difficulties. We now must prohibit actions from spanning day boundaries; we do not want the planner to schedule a four-hour action as beginning at time 6 (Monday at 3 p.m.) and ending at time 10 (Tuesday at 11 a.m.). An action must be completed within a single day. Enforcing this rule in PDDL is difficult.

First we need a way to keep track of time. In PDDL, an action does not know when it is occurring; that is, it has no way to refer to the current time. If we want to keep track of time, then, we must do it ourselves. To do so, we define a nullary function, current-hour, and we add to each action an effect, (**at end** (**increase** current-hour ?duration)), to set its value.[4]

If PDDL provided a sufficiently rich set of arithmetic operators, this alone would be sufficient to prevent actions from crossing day boundaries; each action could have a precondition

$$\text{?duration} + (\text{current-hour} \bmod 8) \leq 8.$$

Unfortunately, PDDL does not have a modulo operator. Instead we must further complicate the specification with a time-since-last-day function. As with current-hour, every action has an effect that increments this value by the action's duration. We also create a special action, waitTillNextDay, that waits until the next multiple of 8 and resets the value of time-since-last-day to 0. Finally, we give each action a precondition, <= time-since-last-day (- 8 ?duration), that prevents each action from being scheduled when there is less time remaining in the day than is required to carry out the action.[4]

Days of the week pose yet another challenge. Recall that some services can be moved only on certain days; for example, the accounting service can be moved only on weekends. Again, with a modulo operator, this would be easy; weekends are those times such that $40 \leq \text{current-hour} \bmod 56 < 56$. Since PDDL lacks a modulo operator, we must again complicate the domain description, this time by defining a Day type (with values Monday, Tuesday, etc.) and a predicate over days, today, that indicates the current day. We then modify the waitTillNextDay action to set this predicate. This will permit us to express constraints pertaining to days of the week in Section IV-E.

A final temporal rule that we enforced was to forbid concurrency. This scenario describes a single, small team of engineers evolving a simple information system; they can work on only one thing at a time. PDDL2.1, on the other hand, is based on an action execution model that is concurrent by default; a planner will gladly schedule all actions to occur simultaneously at time 0 if allowed to do so. To prevent this, every action has a condition that prevents it from executing when the no-work-in-progress predicate is true; every action sets this predicate to false at the beginning of its execution and resets it to true at the end.

E. Representing the Path Constraints

We have already seen the representation of some of the constraints in this scenario. The prohibition on concurrency, for example, is achieved by means of action conditions and effects. The requirement that a firewall must be installed before any services are migrated is similarly simple to model via a has-firewall predicate that is set by the installFirewall action and appears as a precondition for all the migration actions.

[4]As can be seen in Fig. 4, we actually hard-code the duration rather than using the ?duration parameter, because some planners have trouble when the ?duration parameter appears in effects and conditions.

The availability constraints were more challenging to model. We saw in Section IV-D that a substantial infrastructure is required to model days of the week in a way that can support the expression of these constraints. With this infrastructure in place, we can specify which services may be moved on which days by defining a predicate, ok-to-move-on, over services and days, and setting its values in the problem description (e.g., ok-to-move-on AccountingService Saturday). Then, we set on each migration action a condition that the given service may be moved on the current day. To do so, we add a parameter ?d - Day to the action to represent the current day (which we enforce with a today ?d condition) and then add the condition ok-to-move-on ?s ?d (see again Fig. 4 for a full example).

We use a similar strategy to define the constraints governing how different services may be migrated. For example, to define which services can be manually migrated over the network, we define a can-be-migrated-individually predicate over services, which is a condition of the manuallyMigrateService action.

Most of these constraints are specified using the same general idiom: the constraints themselves often appear as action conditions, but they often are supported by predicates that keep track of state (which is maintained through the use of action effects). This is an ad hoc version of the kind of state-based reasoning that would occur if we were to adopt a more formal means of translating constraints expressed in temporal logic into PDDL à la Cresswell & Coddington, as suggested in Section III-C.

F. Representing the Path Evaluation Function

In this evolution, the goal is to minimize cost; thus we have a single evaluation function, which we model in PDDL by the nullary function total-cost. The value of this function is incremented by the actions, and the function is defined as the goal metric in the problem description.

The main complication is that the costs of actions are not fixed. Actions are more expensive on weekends than during normal working hours. The straightforward way to model this would be with conditional effects. Unfortunately they are not well supported (by now a familiar refrain).

Instead we introduce a cost-multiplier function over days, which we value at 1 for weekdays and 3 for weekends. Since each action has a parameter representing the current day of the week (see Section IV-E), each action can incorporate this cost multiplier in its effect on total-cost, as shown in Fig. 4.

G. Generating an Evolution Path

The final PDDL specification was of moderate size: 24 objects, 14 predicates, 8 functions, 130 initial conditions, and 9 durative actions (each with, on average, 8 conditions and 9 effects). With the specification complete, the next step was to generate a plan.

We used two different planners to demonstrate a key advantage of PDDL: its status as a lingua franca supported by many planners. We chose LPG-td [16] and OPTIC [17] as the two planners due to their feature sets, ease of installation and use, maturity, planning quality, and general good reputation.

Both planners work by first attempting to generate a correct (but possibly low-quality) plan, then progressively refining the plan to improve its quality. Both planners correctly interpret our PDDL specification, find a correct solution within a few seconds, and refine it into an optimal solution soon thereafter.

```
0.000: (installswitch monday) [1.900]
1.901: (installfirewall dc2 monday) [0.900]
2.802: (clonehost2 clientwebsitehost2 unusedunixhost1 clientv
4.703: (clonehost2 clientwebsitehost1 unusedunixhost2 clientv
6.604: (waittillnextday monday tuesday) [1.399]
8.004: (physicallymovehost1 analyticshost analyticsservice tu
13.905: (clonehost1 safetydbhost unusedunixhost3 safetydbse
15.806: (waittillnextday tuesday wednesday) [0.201]
16.008: (decommissionhost safetydbhost dc1 wednesday) [3.
19.908: (decommissionhost clientwebsitehost2 dc1 wednesda
23.809: (waittillnextday wednesday thursday) [0.200]
24.011: (decommissionhost clientwebsitehost1 dc1 thursday)
27.911: (manuallymigrateservice payrollservice financehost a
31.812: (waittillnextday thursday friday) [0.200]
32.013: (waittillnextday friday saturday) [8.000]
40.015: (manuallymigrateservice accountingservice financeh
43.915: (waittillnextday saturday sunday) [4.100]
48.016: (waittillnextday sunday monday) [8.000]
56.017: (decommissionhost financehost dc1 monday) [3.899]
```

Fig. 5. Output from OPTIC showing an optimal evolution plan. In bold are action names, which are followed by the action parameter assignments. At the beginning of each line is the time at which the action is executed.

Fig. 5 shows an optimal plan generated by OPTIC. Observe that services are always moved by the cheapest means permissible—cloning is preferred, with manual migration and physical host transfer used only when required. In addition, the planner avoids scheduling any unnecessary activity on weekends; the only service migrated on the weekend is the accounting service, which is forbidden to be moved during weekdays.

To reliably quantify the planners' performance on our specification, we ran our specification on each planner ten times on an Amazon EC2 medium instance (which has 3.75 GiB of memory and processing power roughly equivalent to a single 2.2-GHz core). Conducting multiple runs was particularly important because LPG-td's plan generation is highly nondeterministic; the initial plan is created based on a random seed.

In all ten runs, OPTIC was able to find a correct plan within 8 seconds and an optimal one (i.e., one that achieves the minimum possible cost) within 10 seconds. LPG-td was much slower at finding an optimal plan (unsurprisingly, since it is a much older planner than OPTIC), but it did succeed consistently within a few minutes, and it always found a correct, nonoptimal solution very quickly. Table II presents summary statistics.

We also ran a modified version of the problem in which we asked the planners to minimize plan duration and ignore cost. Times for these runs appear in the lower part of Table II.

V. Findings

Our experience demonstrated the viability of using automated planning tools to solve architecture evolution problems, but it also revealed challenges. We now discuss our main findings.

PDDL is expressive enough to capture the significant concerns of an evolution problem. Despite some challenges, we were able to capture the evolution scenario in its entirety. The PDDL model of a planning problem is broadly consistent with a

TABLE II
TIME TO GENERATE EVOLUTION PATH

	Time to Generate Initial Solution			Time to Discover Optimal Solution		
	Min	Median	Max	Min	Median	Max
Minimizing cost						
LPG-td	3.6	3.8	3.9	40.6	126.2	278.4
OPTIC	6.6	6.6	7.9	7.6	7.6	9.3
Minimizing time						
LPG-td	3.7	3.7	4.1	49.8	90.5	287.4
OPTIC	6.6	6.6	7.8	15.3	15.4	17.4

All figures are in seconds and are calculated over ten runs.

path-oriented view of architecture evolution; the correspondence between evolution operators and PDDL actions, for example, is satisfyingly direct.

We did have to contend with some limitations of PDDL. Most significantly, PDDL's simplistic model of time makes it difficult to specify constraints based on calendar time or clock time. Modeling constraints about which actions could occur on which days of the week, for example, posed significant challenges.

There has, incidentally, been considerable research on improved methods for expressing temporality in planning problems [18], [19], and languages have been developed that increase the temporal expressivity of PDDL [20], [21]. However, PDDL is far more widely supported than any other planning language.

Automated planners can effectively and efficiently generate evolution paths. Both automated planners we tried were able to quickly generate high-quality solutions to a moderately complex architecture evolution problem. This kind of automated path generation has the potential to ameliorate one of the most significant burdens of a path-based approach to software architecture evolution: the need for the architect to manually specify the evolution graph in full detail before beginning analysis. By taking advantage of automated planners, we are able to capitalize on decades of research in artificial intelligence, which allows paths to be generated quickly and intelligently.

Of course, we should be cautious about overgeneralizing based on a single experience. Some architecture evolution problems may be more amenable to solution by automated planners than others. More work is needed to evaluate the scalability and applicability of this approach.

Current off-the-shelf planners have limited feature sets. Although PDDL provides a powerful array of features for specifying complex and intricate planning problems, few (if any) existing planners support the language fully. In Section IV, we mentioned numerous instances where we had to adapt our specification to accommodate the limitations of planners. In most cases, we were able to circumvent these limitations. That is, in many cases, planner limitations do not reduce the practical expressivity of the specification language, but they do make specifications more verbose and awkward. For example, poor support for negative and existential conditions forced us to clutter our specification with many otherwise unnecessary declarations.

This somewhat compromises PDDL's effectiveness as a lingua franca. Ideally, we should be able to write a PDDL specification once and use any PDDL-based planner to analyze it. In practice, planners' limitations are so idiosyncratic and poorly documented that adapting a specification to work with a particular planner is a frustrating and time-consuming process.

Debugging planning specifications is difficult. PDDL specifications are inherently difficult to debug. If a specification author forgets, for example, to specify a necessary initial condition in the problem description, causing the problem to be unsolvable, the planner will simply say that it is unable to generate a plan. There is no good way to track down the cause of the problem.

The experimental nature of available off-the-shelf planners exacerbates this problem. Even the most stable planners are fairly buggy and have limited documentation. We chose LPG-td and OPTIC for their relative maturity and stability, but while using them, we encountered many bugs and undocumented limitations. Error messages were often unhelpful, and it can be difficult to tell whether a problem is caused by a specification error, a limitation of the planner, or an outright bug. When a planner encounters something its designers did not anticipate, it is as likely to crash with a segmentation fault as it is to display any useful explanation of the problem.

VI. RELATED WORK

A. Software Architecture Evolution

In this paper, we have described how automated planning techniques can be used in support of the approach we developed in our previous research. However, ours is not the only approach to software architecture evolution. In recent years, a number of software architecture researchers have turned their attention to evolution. (For a general survey of this body of work, see recent literature reviews [22], [23] or our journal paper [2].)

Many of these approaches bear fundamental similarities to our own work. For example, Le Goaer [5] frames the architecture evolution problem in terms of a directed graph over architectural configurations that is very similar in substance to our evolution graph. Grunske [7], in his work on architectural refactorings, and Wermelinger & Fiadeiro [24], in their work on architecture reconfiguration, do not have anything like an evolution graph, but they do have means of describing composable architectural transformations that capture steps in an architectural evolution—very much like our evolution operators. And Brown et al. [6], although they do not focus on capturing architectural transformations, speak of architecture evolution in terms of "development paths" that are essentially the same concept as our evolution paths. Thus, the general ideas we have presented in this paper are more broadly applicable beyond our own research; the approach we have described here could be easily adapted to these other methods for reasoning about architecture evolution.

B. Automatic Generation of Evolution Paths

The only previous effort to tackle the problem of automatic generation of candidate architecture evolution paths is a recent paper by Ciraci et al. [25]. They confront the same problem as we do, and their conception of architecture evolution is explicitly based on our previous work [1]. However, their approach is based on graph transformation rather than automated planning.

The most significant difference between our approach and theirs is that in their approach, evolution paths are generated by beginning with the initial architecture, then applying (blindly, without a goal) any evolutionary transformations that are applicable to the current architecture. The tool continues to apply transformations until there are no more that can be applied. This stable state is then deemed to be a final architecture. This process is repeated to generate other paths (whose final architectures may be different). Finally, the various alternatives are scored to see which of the final architectures match the desired structural characteristics and which paths have the desired properties.

Our approach is much more goal-directed. Instead of beginning with the initial architecture and blindly applying operators in the hope of attaining a suitable path with a suitable end state, we begin by defining both the initial architecture and the target architecture. Then, we use a planner to generate a path by which the initial architecture can be evolved into the target architecture. This requires more sophisticated reasoning, but automated planners are excellent at exactly this type of reasoning. Because our approach is based on an intelligent, goal-directed search, we expect it to be more scalable than brute-force generation of all possible evolution paths. However, further work is needed to evaluate the performance of different approaches.

Nonetheless, this highlights what we view as a key advantage of our approach to evolution path generation, namely that it draws on decades of research on creating efficient, intelligent planners. This allows us to solve the challenging problem of composing elementary operations together to reach a predefined goal state that may be quite different from the initial state.

C. Other Applications of Automated Planning

Although this is the first attempt to apply automated planning techniques to software architecture evolution, there have been applications of automated planning to other topics related to software architecture and software reconfiguration, such as planning dynamic reconfigurations of software architectures [26], computing workflows that effect autonomic changes to the configuration of a computing system [27], and composing services in a service-oriented architecture [28]. Although the techniques that support these approaches are analogous to ours, this previous work does not address the problem of assisting an architect in planning long-term software evolution.

VII. Conclusion

We have presented an approach for automated generation of evolution paths that draws on research on automated planning to conduct an intelligent, goal-directed, optimizing search. We demonstrated this approach on a scenario based on a real-world example and showed that existing off-the-shelf planners can be employed to generate evolution paths effectively and efficiently. While this is an encouraging first step, many questions remain unanswered, and we believe this area is ripe for future work. Topics of particular interest include:

Modeling uncertainty. Architecture evolution often involves uncertainty—about the architecture of the system, about the effects of the evolution effort, about exogenous events such as technological changes. There are various ways that uncertainty could be incorporated into the evolution model, from risk analyses to contingency planning. There has been a tremendous amount of work on uncertainty in automated planning, but it is an open question how to apply this to architecture evolution.

Modeling transitions comprising multiple evolution operators. Evolution operators capture fairly fine-grained architectural changes—installing an adapter, migrating a database, upgrading a message bus. In conceptualizing an evolution path, it is often helpful to think of evolution transitions as being somewhat coarser, with each transition potentially comprising multiple operators. This streamlines the evolution graph, making it more comprehensible to architects and simplifying analysis. Here, we have treated evolution operators as synonymous with the evolution graph transitions, but in principle a planner could aggregate operators into larger transitions by identifying particularly significant points within the evolution to serve as the nodes of the evolution path. However, it is not immediately clear on what basis it should select these significant points.

Generating multiple candidate paths. In this work, we have delegated to the planner not only the task of generating candidate paths, but also that of selecting an optimal path. However, it might be desirable to keep the architect in the loop. Rather than present a single, supposedly optimal path to the architect, it might be better to present multiple candidate paths, allowing the architect's experience and judgment to play a role in path selection. However, it is not clear how we might best generate multiple paths. One option might be to run the planner multiple times, each time optimizing a different metric.

In addition to these theoretical challenges, there are practical issues that must be addressed to make this kind of approach useful to practitioners, including not only the issues we have identified with existing planning technologies, but also questions on how planners can be integrated into the architecture planning process and adapted to be usable by practicing architects. One significant challenge in that vein is automating the translation of architecture evolution problems into PDDL specifications. Another is facilitating reuse of portions of planning specifications within domains of evolution, perhaps based on the concept of evolution styles that we have described previously [1], [2].

Finally, automated planning is only one possible approach to evolution path generation. Other approaches, such as constraint satisfaction or techniques from operations research, might well be fruitful avenues for exploration.

Acknowledgments

This work was supported by the Software Engineering Institute and by the United States Navy under contracts N000141310401 and N000141310171. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Software Engineering Institute, the Office of Naval Research, or the U.S. government.

We would like to thank Bradley Schmerl and Ipek Ozkaya for their helpful comments as we were carrying out this research.

REFERENCES

[1] D. Garlan, J. M. Barnes, B. Schmerl, and O. Celiku, "Evolution styles: Foundations and tool support for software architecture evolution," in *Proc. Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA'09)*, Cambridge, UK, Sep. 14–17, 2009, pp. 131–140.

[2] J. M. Barnes, D. Garlan, and B. Schmerl, "Evolution styles: Foundations and models for software architecture evolution," *Softw. & Syst. Model.*, in press. [Online]. Available: http://dx.doi.org/10.1007/s10270-012-0301-9

[3] D. Garlan and B. Schmerl, "Ævol: A tool for defining and planning architecture evolution," in *Proc. International Conference on Software Engineering (ICSE'09)*, Vancouver, BC, May 16–24, 2009, pp. 591–594.

[4] J. M. Barnes and D. Garlan, "Challenges in developing a software architecture evolution tool as a plug-in," in *Proc. Workshop on Developing Tools as Plug-Ins (TOPI'13)*, San Francisco, CA, May 21, 2013, pp. 13–18, to appear.

[5] O. Le Goaer, "Styles d'évolution dans les architectures logicielles," Ph.D. dissertation, Univ. of Nantes, Oct. 9, 2009. [Online]. Available: http://hal.archives-ouvertes.fr/docs/00/45/99/25/PDF/these.pdf

[6] N. Brown, R. L. Nord, I. Ozkaya, and M. Pais, "Analysis and management of architectural dependencies in iterative release planning," in *Proc. Working IEEE/IFIP Conference on Software Architecture (WICSA'11)*, Boulder, CO, Jun. 20–24, 2011, pp. 103–112.

[7] L. Grunske, "Formalizing architectural refactorings as graph transformation systems," in *Proc. International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'05)*, Towson, MD, May 23–25, 2005, pp. 324–329.

[8] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: Theory and Practice*. Amsterdam: Morgan Kaufmann, 2004.

[9] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, *Documenting Software Architecture: Views and Beyond*, 2nd ed. Upper Saddle River, NJ: Addison-Wesley, 2010.

[10] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL—the Planning Domain Definition Language, version 1.2," Ctr. for Computational Vision and Control, Yale Univ., Tech. Rep. TR-98-003, Oct. 1998.

[11] M. Fox and D. Long, "PDDL2.1: An extension to PDDL for expressing temporal planning domains," *J. Artif. Intell. Res.*, vol. 20, pp. 61–124, 2003.

[12] S. Edelkamp and J. Hoffmann, "PDDL2.2: The language for the classical part of the 4th International Planning Competition," Dept. of Computer Science, Univ. of Freiburg, Tech. Rep. 195, Jan. 24, 2004. [Online]. Available: http://tr.informatik.uni-freiburg.de/2004/Report195/

[13] A. Gerevini and D. Long, "Preferences and soft constraints in PDDL3," in *Proc. ICAPS'06 Workshop on Planning with Preferences and Soft Constraints*, 2006, pp. 46–53. [Online]. Available: http://strathprints.strath.ac.uk/3149/

[14] J. Frank, K. Golden, and A. Jonsson, "The loyal opposition comments on Plan Domain Description Languages," in *Proc. ICAPS'03 Workshop on PDDL*, 2003. [Online]. Available: http://users.cecs.anu.edu.au/~thiebaux/workshops/ICAPS03/proceedings/frank.pdf

[15] S. Cresswell and A. Coddington, "Compilation of LTL goal formulas into PDDL," in *Proc. European Conference on Artificial Intelligence (ECAI'04)*, Valencia, Spain, Aug. 22–27, 2004, pp. 985–986.

[16] A. Gerevini, A. Saetti, and I. Serina, "An approach to temporal planning and scheduling in domains with predictable exogenous events," *J. Artif. Intell. Res.*, vol. 25, pp. 187–231, 2006.

[17] J. Benton, A. Coles, and A. Coles, "Temporal planning with preferences and time-dependent continuous costs," in *Proc. International Conference on Automated Planning and Scheduling (ICAPS'12)*, Atibaia, Brazil, Jun. 25–29, 2012, pp. 2–10. [Online]. Available: http://www.aaai.org/ocs/index.php/ICAPS/ICAPS12/paper/view/4699

[18] W. A. Cushing, "When is temporal planning *really* temporal?" Ph.D. dissertation, Arizona State Univ., Dec. 2012. [Online]. Available: http://sagarmatha.eas.asu.edu/cushing-dissertation.pdf

[19] G. Sciavicco, "Reasoning with time intervals: A logical and computational perspective," *ISRN Artif. Intell.*, 2012. [Online]. Available: http://dx.doi.org/10.5402/2012/616087

[20] M. Fox and D. Long, "PDDL+: Modelling continuous time-dependent effects," in *Proc. International NASA Workshop on Planning and Scheduling for Space*, 2002.

[21] F. Maris and P. Régnier, "TLP-GP: Solving temporally-expressive planning problems," in *Proc. International Symposium on Temporal Representation on Reasoning (TIME'08)*, Montreal, QC, Jun. 16–18, 2008, pp. 137–144.

[22] H. P. Breivold, I. Crnkovic, and M. Larsson, "A systematic review of software architecture evolution research," *Inform. & Software Tech.*, vol. 54, no. 1, pp. 16–40, Jan. 2012.

[23] P. Jamshidi, M. Ghafari, A. Ahmad, and C. Pahl, "A framework for classifying and comparing architecture-centric software evolution research," in *Proc. European Conference on Software Maintenance and Reengineering (CSMR'13)*, Genoa, Italy, Mar. 5–8, 2013, pp. 305–314.

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

[25] S. Ciraci, H. Sözer, and M. Aksit, "Guiding architects in selecting architectural evolution alternatives," in *Proc. European Conference on Software Architecture (ECSA'11)*, Essen, Germany, Sep. 13–16, 2011, pp. 252–260.

[26] J.-E. Méhus, T. Batista, and J. Buisson, "ACME vs PDDL: Support for dynamic reconfiguration of software architectures," in *Proc. Conférence Francophone sur les Architectures Logicielles (CAL'12)*, Montpellier, France, May 30–31, 2012. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00703176/

[27] H. Herry, P. Anderson, and G. Wickler, "Automated planning for configuration changes," in *Proc. Large Installation Systems Administration Conference (LISA'11)*, Boston, MA, Dec. 4–9, 2011. [Online]. Available: http://www.usenix.org/events/lisa11/tech/full_papers/Herry.pdf

[28] H. Schuschel and M. Weske, "Automated planning in a service-oriented architecture," in *Proc. IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'04)*, Modena, Italy, Jun. 14–16, 2004, pp. 75–80.