Challenges in Developing a Software Architecture Evolution Tool as a Plug-In

Jeffrey M. Barnes and David Garlan
Carnegie Mellon University, Pittsburgh, PA
jmbarnes@cs.cmu.edu · garlan@cs.cmu.edu

***Abstract*—Recent research has developed a theoretical basis for providing software architects with better support for planning and carrying out major evolutions of software systems. However, these theoretical models have yet to be implemented in a complete, integrated tool. In this paper, we consider some of the challenges involved in developing such a tool as a plug-in to an existing architecture modeling framework. We present an experience report based on the development of a prototype architecture evolution tool as a plug-in to MagicDraw, a commercial UML tool. We find that there are many practical obstacles to developing tools for architecture evolution modeling as plug-ins, and we discuss some of the key factors that plug-in developers should weigh when considering frameworks.**

I. Introduction

Software architecture is today widely accepted as an essential means of designing software systems effectively. However, one topic that current software architecture practices do not address well is *software architecture evolution*. Software architecture evolution is a phenomenon that occurs in virtually all software systems of significant size and longevity. As a software system ages, it often needs to be structurally redesigned to accommodate new requirements, new technologies, or changing market conditions. In addition, many systems over the years tend to accrue a patchwork of architectural workarounds, makeshift adapters, and other forms of architectural detritus that compromise the architectural integrity and maintainability of the system, requiring some sort of architectural overhaul to address. At present, however, software architects have few tools to help them plan and carry out these kinds of architecture evolution.

In recent years, we and other researchers have been developing approaches and models to support software architects in reasoning about such evolutions [1], [2], [3], [4]. Tools based on these approaches could help architects to make better decisions when planning software evolutions. To date, however, research in this area has focused on the theoretical aspects of architecture evolution modeling, rather than the practical challenges involved in developing usable and useful evolution modeling tools. While some rudimentary prototypes and demonstrations have been presented [5], [6], [7], [8], they serve chiefly as illustrations of the theoretical concepts they embody; there has been no significant effort to make these tools practical for industry use, nor to examine the challenges in doing so.

Last year, we undertook a project to develop a prototype architecture evolution tool as a *plug-in* to an existing, commercial UML modeling tool. A plug-in-based approach is a natural and sensible one for developing an architecture evolution tool because it allows us to leverage the facilities that the tool provides for architecture modeling (see Section II-D). In the course of developing this prototype, however, we encountered a number of unforeseen challenges. In this paper, we describe the difficulties we encountered and consider the general lessons that may be drawn from this experience.

Section II provides necessary background on our research and on the context for the project. Section III describes how we developed the prototype plug-in. Section IV discusses what we learned from this experience. Section V describes related work, and Section VI concludes.

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 from which they are made. At a basic level, 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 previous work, we have developed an approach for understanding and modeling software architecture evolution, supported by automatable formal methods [1]. Here we give a brief overview.

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 *intermediate architectures* leading from the initial state to the target state. We can represent and relate these evolution paths within an *evolution graph* whose nodes represent intermediate architectures and whose edges represent the possible transitions among them (Fig. 1).

The architect's goal, in this model, is to find the optimal path—the one that best meets the goals of the evolution, while adhering to any relevant technical and business constraints, and subject to concerns such as cost and duration. To facilitate the architect in making informed trade-offs among the candidate paths, we provide two kinds of analysis: *evolution path constraints*, which define which paths are legal with respect to the rules of the evolution domain, and *path evaluation functions*, which provide quantitative assessments of path qualities such as duration and cost. Each transition in an evolution graph may be understood

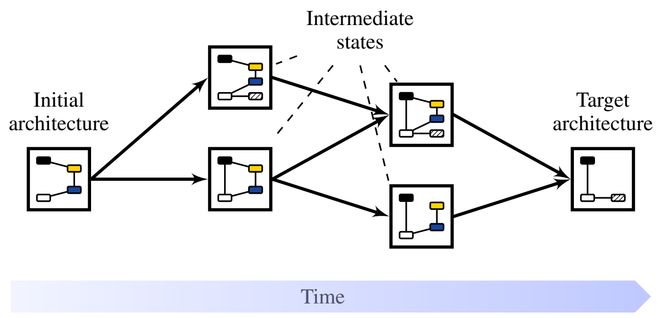


Fig. 1. A depiction of an evolution graph. Each node is a complete architectural representation of the system. Edges represent possible evolutionary transitions. The architect's task is to select the optimal path through the graph.

as a sequence of *evolution operators*—reusable architectural transformations such as "add adapter" or "migrate database."

These concepts are fairly simple, but there is a substantial formal framework supporting them. To formalize path constraints, for example, we have developed a language based on linear temporal logic. The use of formal methods has several advantages, the most important of which are precision (by using a formal approach, we minimize ambiguity, which helps to pin down what project stakeholders mean when they talk about the project) and automation (a formal approach allows us to develop tools to make it easier to plan and analyze the evolution).

C. *Our Previous Prototypes*

In previous work, we developed two partial prototypes illustrating features of our approach. The first, Ævol [6], was based on the architecture description language Acme [10] and its associated tool, AcmeStudio [11]. Using Ævol, evolution plans could be represented as sequences of intermediate architectures. These plans could be related by means of an evolution graph, and simple analyses could be applied to evaluate the validity and quality of an evolution path. However, the prototype had significant limitations. It lacked a number of key features of our approach, such as the ability to define an evolution path in terms of operators. In addition, because it was intended only as a demonstration of our approach, it lacked qualities that would be necessary in a practical, adoptable tool. In particular, usability was not a focus of this prototype; substantial effort was required to define and analyze even a simple evolution path.

The second prototype was developed in support of a case study at NASA's Jet Propulsion Laboratory [7]. The aim of the case study was to model a planned software evolution at JPL, using modeling languages and tools already in use there. We hoped to demonstrate that our approach could be adapted to industrial languages and tools—that it was not tied to AcmeStudio. To that end, we developed facilities to support modeling software architecture evolutions in MagicDraw, a commercial UML tool that is widely used at JPL. These facilities included representational conventions for modeling architecture evolutions in UML (actually SysML, a widely used UML profile), model transformation macros to implement evolution operators, and SysML models of particular intermediate states of evolution. Together, these mechanisms served as a prototypical illustration of how our approach could be adapted to commercially prevalent languages and tools such as UML and MagicDraw.

Both these early prototypes had serious limitations. They both exhibit only a subset of the features of our approach. They were both developed to demonstrate or evaluate particular points and were intended as illustrative prototypes rather than tools that could be readily adopted for use in real-world software organizations. We therefore sought to develop a third prototype that would allow us to better understand the challenges involved in developing a practical architecture evolution tool based on our approach; that is the work we describe here.

D. *Vision: A Plug-In for Architecture Evolution Planning*

Our case study at JPL demonstrated that it was possible to adapt our approach to an off-the-shelf tool based on a commercially popular modeling language. However, the JPL prototype had been developed ad hoc for the purpose of supporting a case study; it was an assemblage of scripts, macros, and modeling conventions specialized to a particular domain—indeed, to a particular system—rather than a general, reusable tool. The evolution operators, in particular, were defined by means of hard-coded, system-specific transformation macros. Thus, although the prototype had demonstrated how specific concepts from our research could be adapted to a practical context, it had sidestepped the significant tooling issues involved in doing so.

To extend MagicDraw so that it could be used as a general tool for evolution planning, we recognized that it would be necessary to take a *plug-in-based* approach. A plug-in-based approach has several advantages. First, developing an architecture evolution tool as a plug-in to a tool such as MagicDraw allows us to leverage the facilities that tool provides for architectural modeling. This frees us to focus our development efforts on implementing the novel aspects of our approach, rather than reinvent the wheel by building our own framework for architecture modeling. In addition, a plug-in-based approach can facilitate user adoption, because users are already likely familiar with the interface and idioms of the framework on which the plug-in is based (particularly a widely used framework such as MagicDraw). Finally, plug-ins are significantly more powerful than other extension mechanisms, such as the macro-based approach we used in the JPL case study, allowing us much more freedom in implementing our approach.

Perhaps the best way to describe the main features we envisioned in a plug-in for evolution planning is to consider a sample work flow, illustrating how a software architect might use such a plug-in to define and analyze an evolution space:

1) The architect begins by defining the nodes and transitions of the evolution graph. This can be done using the architectural modeling tools already present in MagicDraw. As in our previous MagicDraw work [7], we can represent an evolution graph as a package diagram, where the evolution nodes are represented by packages and transitions are modeled as dependencies. To define the evolution graph, the architect simply creates a package for each intermediate state and draws the transitions as dependencies among the packages. Once the graph is represented in this way, the plug-in can use the MagicDraw API to inspect the evolution graph, then determine the set of candidate evolution paths.
2) At this point, the evolution graph has been laid out, but all the evolution nodes are undefined, in the sense that all the packages representing intermediate states are empty—they do not yet contain representations of intermediate architectures. The architect must now "fill in" the evolution graph by specifying these intermediate nodes. The first node that the architect fills in is the initial node, which represents

the system architecture at the outset of the evolution. (A plug-in can identify the initial node algorithmically. The initial node is simply the graph's source node—the node with in-degree 0, which should be unique in a well-formed evolution graph.) To do so, the architect uses MagicDraw's existing facilities to create an architectural model contained within the package representing the initial node.

3) Now that the first node has been "filled in"—the initial architecture has been specified—the architect can begin using the plug-in to define the other intermediate states. To do so, the architect selects one of the transitions leaving the initial state, then clicks a "Specify Transition" button provided by the plug-in. This launches transition specification mode, in which the architect applies a sequence of operators capturing the architectural transformations that make up the transition. The architect selects these operators from a predefined palette and specifies the parameters necessary to apply them to the evolving model. As the architect does so, the effects of the operators are displayed. Once the definition of the operators making up the transition is finished, the architect exits transition specification mode. The newly defined architecture is now associated with the relevant state in the evolution graph; thus, there are now two nodes "filled in."
4) The architect continues specifying transitions as in step 3 until all transitions in the graph have been specified. At this point, all nodes have been "filled in."
5) With the evolution graph fully defined, the architect now clicks a "Validate Evolution Paths" button, which causes the plug-in to validate all the evolution paths in the graph with respect to a predefined set of constraints. If a path fails to satisfy any of the constraints, the user is notified of the violation and given the opportunity to correct it.
6) Finally, the architect can run a set of predefined evaluation functions, which provide quantitative assessments of each valid path. The architect is now equipped to make an informed decision about the optimal evolution path.

With this operational vision in mind, we set out to create a new prototype, the development of which would give us insight into the previously unexplored engineering issues involved in developing an evolution planning tool as a plug-in to an existing architecture modeling tool (viz. MagicDraw).

III. Developing a Plug-In for Architecture Evolution Planning

The development of this prototype plug-in is described in the following sections. Section III-A describes the context and setting for this project. Section III-B describes the MagicDraw OpenAPI, on which the plug-in was based. Section III-C describes our development process, and Section III-D describes the outcome of the project.

A. Project Setup

This work was conceived as a follow-on to the JPL case study and structured as a practicum project within Carnegie Mellon University's software engineering master's program. The development of the prototype was carried out by a team of two master's students, directed by the authors. At the beginning of the project, we spent several weeks carrying out initial tasks necessary to prepare for the main plug-in development phase, including familiarizing the master's students with our research and with MagicDraw, communicating project requirements, and setting up the development environment.

B. The MagicDraw API

There are several ways in which the functionality of MagicDraw can be extended. The most basic and limited is by defining *UML profiles*. This allows users to enrich the available modeling vocabulary by extending UML itself; however, UML profiles cannot be used to modify the user interface or behavior of MagicDraw. A much richer extension mechanism is the *MagicDraw Macro Engine*, which we used in our JPL case study to generate an evolution graph [7]. Finally, the most powerful means of extension that MagicDraw provides are *plug-ins*, the approach we used in this project.

Both macros and plug-ins interact with MagicDraw via the *MagicDraw OpenAPI* [12], a Java API that provides numerous means of extending or enhancing MagicDraw's functionality. With the OpenAPI, it is straightforward to add user interface features (such as buttons and menu items), manipulate model and presentation elements, listen for events, and so on.

C. The Development Process

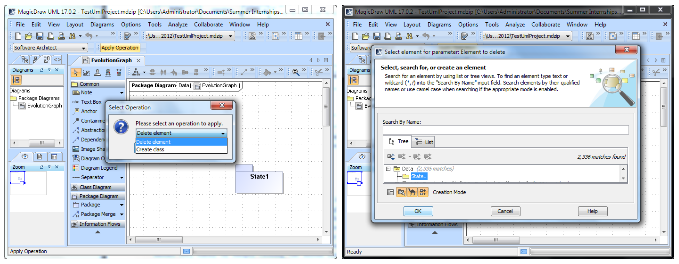
Development of the prototype proceeded in stages, each designed to result in a particular unit of functionality. These units—"feature prototypes," we called them—would be pieced together at the end of the project, in a final integration phase. This staged development approach allowed us to familiarize ourselves with the relevant portions of the OpenAPI rapidly. In addition, developing the plug-in by way of a series of partial prototypes made sense because the plug-in's responsibilities were clearly delineated; thus, the components of the plug-in's planned architecture were well defined and interacted with each other in well-understood ways. The development stages were:

Feature 1: Manipulation of the evolution model. Our first feature prototype was intended to familiarize us with the fundamentals of MagicDraw plug-in development and to acquaint us with MagicDraw's model manipulation API. We therefore created a simple plug-in that made a predefined sequence of API calls to manipulate a predefined evolution model in MagicDraw. This prototype was similar in function to the transformation macro that was used to define evolution operators in our previous MagicDraw work [7]. Here, however, this was accomplished by means of a plug-in rather than a macro.

Feature 2: Identification of evolution paths. The second feature prototype was intended to demonstrate basic reasoning about an evolution model. Given a MagicDraw model of an evolution graph, this feature prototype would analyze the model and identify all the evolution paths in the graph.

Feature 3: Interface for applying evolution operators. Next, we wanted to familiarize ourselves with MagicDraw's user interface extension capabilities. The third feature prototype was thus an interface for applying an evolution operator (Fig. 2). To apply an operator, the user firsts select an operator from a palette, then specifies any required parameters. Each operator can define an arbitrary set of typed parameters. For example, a "delete element" operator might take one parameter: the element to be deleted. A "create component" operator might have multiple parameters: the name of the component, its type, and so on.

Feature 4: Operator parser. We wanted operators to be specifiable via a configuration file that would be parsed and



Select Operation

Please select an operation to apply.

Delete element

Create class

Select element for parameter: Element to delete

Select, search for, or create an element

Search for an element by using list or tree views. To find an element type text or wildcard (*,?) into the "Search By Name" input field. Search elements by their qualified names or use camel case when searching if the appropriate mode is enabled.

Search By Name:

2,336 matches found

Data (2,335 matches)

State1

Fig. 2. Screenshots of feature prototype 3, the operator palette interface. At left, the user selects an operator from the palette. In this demo, two operators are defined. The user selects "Delete element." At right, the user is prompted to specify the parameter required by the operator: the element to be deleted.

interpreted at the time the plug-in is loaded, so that plug-in users could add, remove, or modify operators easily. In this stage, we developed a parser for a highly simplified version of the operator specification language we developed in previous work [1].

Feature 5: Metadata handling. This feature prototype focused on storage of *metadata* pertaining to the model—that is, information that is auxiliary to the evolution model but necessary to support analysis, such as the *sequence of evolution operators* that makes up each transition in the model. We considered several mechanisms for storing metadata. The most principled approach would have been to develop a UML profile for the purpose of representing architecture evolution graphs. However, while elegant, this solution would have taken too long to implement given the limited time available. For this feature, we adopted the cruder but more expedient approach of storing metadata in existing fields intended for other purposes.

Feature 6: Positioning of presentation elements. A significant challenge of developing an evolution planning tool as a plug-in to a tool like MagicDraw is dealing with the positioning of presentational elements in intermediate states. For example, when a user applies an operator that introduces a new component, how should we position that component in any diagrams that are associated with the new state? We considered a number of possible approaches, including making use of the layout algorithms built into MagicDraw, requiring the user to specify the placement of new presentation elements, or even developing layout algorithms of our own. Ultimately, we again adopted the most expeditious approach; to determine the layout of an evolution state, we cloned the layout of the state from which it was derived, then allowed MagicDraw to position any new elements in the default location.

Feature 7: Evolution path constraints. Evolution path constraints, like operators, are defined in a configuration file that is read by our plug-in when it is loaded. The constraints are written in a simplified subset of the constraint specification language we developed in previous work [1]. The final feature prototype comprised a parser and interpreter for this language.

Integration. The feature prototypes were designed to allow us to develop, in isolation, the main features required for an evolution planning plug-in based on our approach. The final phase of development was an integration phase aimed at tying these features together into a unified tool.

D. Project Outcomes

For the most part, the project proceeded as expected. All the planned feature prototypes were completed. However, there were a number of unexpected challenges.

First, the MagicDraw API turned out to be less flexible than we had hoped, making it difficult or impossible to implement several of the features that we had envisioned. For example, the API does not provide much flexibility for plug-in developers to overhaul or replace the default user interface, which made implementing the kind of work flow described in Section II-D impractical. We will discuss this further in Section IV.

In addition to limitations in API *functionality*, we also experienced problems pertaining to API *discoverability*. Much of this was due simply to the significant size of the MagicDraw OpenAPI. With such a large API, it is difficult for a small team of part-time developers to learn it quickly. Thus, it was often difficult to find the features we needed, or even to know *which* features the API supported. Even when we found the relevant part of the API, the documentation was sometimes inadequate, so trial and error were required to understand a class's function.

These delays (along with technical difficulties in the project set-up phase) slowed our progress significantly. Although we completed all planned feature prototypes, many of them were not as fully fleshed out as we had anticipated, as mentioned above. This also left insufficient time for the integration phase to be satisfactorily completed. Thus, at the time the practicum project ended, the plug-in was more like an agglomeration of features than a unified tool with a consistent design.

IV. Lessons Learned

Our experience provided a number of insights about developing a tool for software architecture evolution modeling as a plug-in to an existing architecture modeling framework. When developing a plug-in for architecture evolution modeling, it is crucial to ask the following questions of the architecture modeling tool on which the plug-in is to be based:

What architecture modeling languages does the tool support? Our framework for architecture evolution is not tied to a particular modeling language; in our previous work, we have used Acme [6], UML [1], and SysML [7]. Nonetheless, there are important differences between a formal architecture description language like Acme and a commercially popular modeling language like UML (which is often used quite informally). Among the most significant is that Acme is tailored for capturing component-and-connector views of software architectures, while UML is used for a variety of purposes and has many different diagram types, from state charts to timing diagrams. Another important difference is size: the UML specification is hundreds of pages long, while Acme has only a handful of concepts. Because UML is such a large, complex language, any adaptation of UML to a narrow purpose like architecture evolution modeling must involve the establishment of specific conventions for using a subset of the language to capture relevant concerns. (An alternative approach would be to develop a UML profile for architecture evolution.) Because the choice of modeling language has a significant effect on the process of modeling an evolution, plug-in developers should weigh these points carefully when evaluating an architecture modeling framework.

How extensive is the framework API? MagicDraw's OpenAPI is quite large (especially relative to a research tool such as AcmeStudio). Such an extensive API is a double-edged sword; while it affords significant flexibility to plug-in developers, it also has a significant learning curve. Thus, while the MagicDraw OpenAPI boasts a large array of features, figuring out how to achieve a particular goal—or whether it is achievable at all—can be difficult. Based on our experience, both large, complex frameworks (like MagicDraw) and small, simple frameworks (like AcmeStudio) can serve as suitable platforms for the development of an architecture evolution plug-in. However, the size and complexity of a framework have a tremendous impact on plug-in development. Large, complex frameworks provide more features to address plug-in developers' needs, but small, simple frameworks are much easier to learn and search.

How comprehensive and useful is the framework documentation? Documentation is an important attribute of any API. However, this is especially true for the API of a framework for software architecture modeling. Due to a number of factors—the highly abstract nature of the domain, terminological differences among different schools of software-architectural thought, and the complex and many-layered interactions between framework components that tend to be necessary to support the features required by modern software architecture tools—an API for a software architecture modeling framework cannot realistically be "self-documenting" in any meaningful sense, the way an API for date manipulation or ray tracing might be.

Documentation of the MagicDraw API is uneven. While a great volume of MagicDraw API documentation exists, there remain substantial swaths of the API that have little or no documentation, and even the best-documented parts of the API are inferior in documentation quality to, say, the standard Java libraries. This means that developing a MagicDraw plug-in necessarily involves a great deal of trial and error.

What technical support is available? Fortunately, although MagicDraw's API documentation is somewhat fragmentary, its technical support is reliable and responsive. MagicDraw developers respond quickly to questions on MagicDraw's public forums as well as more formal channels such as bug reports and support tickets. Good support can compensate, to a limited extent, for problems with a framework's usability or documentation.

What are the intended uses of the framework? What kinds of variation does it support? Plug-ins are, by nature, subject to the limitations of the frameworks they are based on. A framework, as the name implies, establishes the boundaries within which plug-ins may operate and rules to which they must adhere, and a plug-in can subvert these boundaries and rules only with great difficulty. Therefore, when considering developing a plug-in to some framework, it is important to consider whether the intended features of the plug-in align with the intended use of the framework—whether the framework's variation points are appropriate to the needs of the plug-in. In this project, we found that some of the features we wished to implement (e.g., traversing and analyzing the evolution paths in a model) aligned well with the capabilities of the MagicDraw OpenAPI, while some others (e.g., ensuring the consistency of the evolution model) did not.

For a software architecture evolution plug-in, one especially important kind of variation is variation in the user interface, which we discuss next.

To what extent can a plug-in exert control over the tool's basic user interface? We had hoped it would be possible to present the user with custom interfaces for defining evolution paths, such as a "transition specification mode" with which users could select operators from a palette and visualize their effects on the architecture. This turned out to be infeasible. The MagicDraw OpenAPI is ideally suited to adding simple interface elements such as new toolbars and menus; it does not provide an easy way to define new user work flows with custom views of the system, or to overhaul the basic user interface.

For an architecture evolution plug-in to be usable, it is essential that significant modifications to the basic user interface of the modeling tool be possible. For example, to prevent the evolution model from becoming inconsistent, the plug-in must be able to prevent users from making arbitrary changes to the model. This appears to be impractical with the MagicDraw OpenAPI.

Does the framework provide good support for manipulating the presentational elements of an architectural model? A significant practical challenge of developing an architecture evolution tool based on our approach—or any approach involving representation of intermediate states—is positioning presentational elements. Because intermediate states are generated from earlier states by means of operators, it is the plug-in's responsibility to ensure those newly generated states are rendered appropriately (e.g., by deciding where to position newly added components). A mature software architecture evolution tool might have to make use of fairly intelligent positioning algorithms to produce a satisfactory user experience. Ideally, we might hope that the architecture modeling platform on which our plug-in is based would provide advanced diagram layout features. MagicDraw provides only slight help in this area; although it has some automatic layout tools, they are quite limited.

Is the framework open-source? MagicDraw's "OpenAPI" is so called to distinguish it from the rest of the MagicDraw code, which is very much *closed*. MagicDraw's core libraries are not only unsupported and undocumented, but obfuscated so that they are unusable (and resistant to reverse engineering). When developing a plug-in for closed, proprietary software such as MagicDraw, one is constrained to operate within a "walled garden"—it is possible to modify the system only in ways that were anticipated and supported by its developers. This is a disadvantage relative to open-source systems.

Of course, many of these questions are not specific to software architecture evolution. Questions about a framework's size, documentation, licensing, and so on are relevant when developing any plug-in. Our experience suggests the questions above may be especially important when developing an architecture evolution plug-in, as we have argued above. However, some of these lessons are more general, as we discuss in Section VI.

V. Related Work

There is a growing body of architecture evolution research. (For a survey, see recent literature reviews [13], [14] or our journal paper [1].) However, most of this work has been theoretical; practical issues of tool development have been largely ignored.

For example, Tamzalit et al. [2], [8] present a theoretical framework, based on *evolution styles*, to support software architecture evolution modeling. However, they do not appear to have implemented their approach in a tool. The examples and case studies they present are all either paper-based or one-off adaptations of off-the-shelf tools.

Similarly, Wermelinger and Fiadeiro's category-theoretic approach for representing architectural transformations [3], Grunske's hypergraph-based approach for defining automatable architectural refactorings [4], and Brown et al.'s approach for iterative release planning [5] are all presented in theoretical terms; in each case, implementation was left for future work.

Outside the domain of software architecture evolution, there is a great deal of work on framework design issues, from framework and API design advice from practitioners [15], [16] to academic investigations of the causes of API usability problems [17], [18] to research on novel approaches and tools to make framework use easier [19], [20] to experience reports [21], [22]. This literature is too vast to summarize here. However, many of the general points we touched on in Section IV appear as recurring themes in this body of work, such as the role of documentation in framework usage, the importance of understanding a framework's variation points, and the effects of API size and complexity on usability.

VI. Conclusion

In this experience report, we have sought to shed light on the practical issues involved in implementing software architecture evolution tools as plug-ins to existing architecture modeling frameworks. Over the course of this project, we learned a number of lessons about factors that can simplify or complicate the development of an architecture evolution plug-in: the size and complexity of the framework, the documentation and support available from the framework developer, the kinds of variation that the framework allows, the framework's licensing, and so on.

Of course, many of these concerns are more broadly applicable. Here, we have focused on how these factors are especially relevant to architecture evolution, but the basic lessons are more general. For example, we noted earlier that the ability to control the user interface (e.g., by locking out the user from making model changes that cause inconsistencies) is a crucial platform feature for an architecture evolution plug-in. This specific lesson is particular to architecture evolution. However, the broader lesson—before adopting a framework, be sure to understand its intended uses and variation points—is quite general.

Similarly, the lesson that good framework API documentation is very useful to plug-in developers is, in its basic form, extremely broad; all frameworks can benefit from clear, comprehensive documentation. But the architecture modeling domain has some special attributes that make clear documentation particularly crucial for a framework like the MagicDraw OpenAPI: a very large feature set; inconsistent use of terminology within the field; complex, many-layered relationships among concepts; and the need to support proprietary features.

Our results are especially relevant to other tools that involve definition, comparison, and analysis of multiple software architectures. For example, a plug-in to support software architecture differencing [23] would have many of the same needs as a plug-in for architecture evolution. Like an architecture evolution plug-in, an architecture differencing plug-in would need to represent relations among multiple architectural models, provide a substantially customized user interface (e.g., a comparison view allowing users to visualize differences between models), and apply architectural transformations to a model (e.g., to support merging). As a result, all the considerations in section IV would likely apply to that domain as well.

Much work remains to be done on the topic of architecture evolution tool development. For example, there are a number of unanswered questions about architecture evolution tool usability that could be addressed from a human-computer interaction perspective. (E.g., what kind of user interface would best facilitate architects in understanding an evolution graph? What kind of specification languages would allow architects to define evolution analyses most easily?) Within the last several years, several theoretical frameworks for software architecture modeling have appeared; but we are only beginning to understand how to make these approaches practical.

Acknowledgments

We would like to express our gratitude to Nivedhitha Manjappur Narayanaswamy and Vignesh Venkatachala Perumal, the master's students who worked on the development of the plug-in. We would also like to thank Brian Giovannoni at NASA JPL for his support and guidance. Finally, we are grateful to the Software Engineering Institute for providing support for this work.

References

[1] 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

[2] O. Le Goaer *et al.*, "Evolution styles to the rescue of architectural evolution knowledge," in *Proc. SHARK'08*, 2008, pp. 31–36.

[3] 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.

[4] L. Grunske, "Formalizing architectural refactorings as graph transformation systems," in *Proc. SNPD'05*, 2005, pp. 324–329.

[5] N. Brown *et al.*, "Analysis and management of architectural dependencies in iterative release planning," in *Proc. WICSA'11*, 2011, pp. 103–112.

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

[7] J. M. Barnes, "NASA's Advanced Multimission Operations System: A case study in software architecture evolution," in *Proc. QoSA'12*, 2012, pp. 3–12.

[8] D. Tamzalit and T. Mens, "Guiding architectural restructuring through architectural styles," in *Proc. ECBS'10*, 2010, pp. 69–78.

[9] P. Clements *et al.*, *Documenting Software Architecture: Views and Beyond*, 2nd ed. Upper Saddle River, NJ: Addison-Wesley, 2010.

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

[11] B. Schmerl and D. Garlan, "AcmeStudio: Supporting style-centered architecture development," in *Proc. ICSE'04*, 2004, pp. 704–705.

[12] *MagicDraw Open API Version 17.0.1 User Guide*, No Magic, Inc., 2011. [Online]. Available: http://nomagic.com/files/manuals/MagicDraw%20OpenAPI%20UserGuide.pdf

[13] 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.

[14] P. Jamshidi *et al.*, "A framework for classifying and comparing architecture-centric software evolution research," in *Proc. CSMR'13*, 2013, to appear.

[15] J. Bloch, "How to design a good API and why it matters," in *Proc. OOPSLA'06*, 2006, pp. 506–507.

[16] K. Cwalina and B. Abrams, *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*, 2nd ed. Upper Saddle River, NJ: Addison-Wesley, 2009.

[17] M. P. Robillard, "What makes APIs hard to learn? Answers from developers," *IEEE Software*, vol. 26, no. 6, pp. 27–34, 2009.

[18] M. F. Zibran, F. Z. Eishita, and C. K. Roy, "Useful, but usable? Factors affecting the usability of APIs," in *Proc. WCRE'11*, 2011, pp. 151–155.

[19] C. Jaspan and J. Aldrich, "Checking framework interactions with relationships," in *Proc. ECOOP'09*, 2009, pp. 27–51.

[20] D. S. Eisenberg, J. Stylos, and B. A. Myers, "Apatite: A new interface for exploring APIs," in *Proc. CHI'10*, 2010, pp. 1331–1334.

[21] S. Roy Choudhary *et al.*, "Platform support for developing testing and analysis plug-ins," in *Proc. TOPI'11*, 2011, pp. 16–19.

[22] L. Prechelt and K. Beecher, "Four generic issues for tools-as-plugins illustrated by the distributed editor Saros," in *Proc. TOPI'11*, 2011, pp. 9–11.

[23] M. Abi-Antoun *et al.*, "Differencing and merging of architectural views," in *Proc. ASE'06*, 2006, pp. 47–58.