Architecture as an Independent Variable

Hamid Bagheri  
University of Virginia  
151 Engineer’s Way  
Charlottesville, VA 22903 USA  
hb2j@virginia.edu

Yuanyuan Song  
University of Virginia/Amazon  
151 Engineer’s Way  
Charlottesville, VA 22903 USA  
ys8a@virginia.edu

Kevin Sullivan  
University of Virginia  
151 Engineer’s Way  
Charlottesville, VA 22903 USA  
sullivan@virginia.edu

ABSTRACT

The idea that we can separate application content from architectural form, or style, is a mainstay of modern software engineering. Architectural styles have themselves been a subject of intensive study. The problem is that we do not yet have an adequate account of the mappings that combine choices of application description independent of style, and of architectural style independent of application, to produce style-specific, application-specific architectural descriptions. An account of such mappings would deepen our understanding of architecture and provide a foundation for technologies for manipulating architecture as an independent variable. We contribute a validated early model of this kind.

Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architecture

General Terms

Design, Architectural Styles

1. INTRODUCTION

Researchers and practitioners have long known that application content can be considered largely separately from architectural form. As far back as 1972, Parnas [8] described a key word in context (KWIC) system and consequences of choosing to design it in one of two forms: functional decomposition (FD), or information hiding (IH). As recently as 2009, in their textbook on architecture, Taylor, et al. [11] described a lunar landing control system and showed how programs implementing it could be written in many styles.

In many related works (e.g. [9, 2, 10, 11, 3]) we find the same basic assumption: a choice of architectural form, or style, is a mainstay of modern software engineering. As we will argue, there is a viable theory of architectural styles, and of specifications as application descriptions abstracted from architectural form, and while we understand that we should seek to separate applications and architectural style, we do not have an adequate theory or technologies for truly making this separation, or for combining formal application and style descriptions to yield system-specific architectures. We have not yet reached a point where mechanically we can treat architecture as an independent variable.

This paper makes four basic contributions in this area. First, we formulate this problem and make the case it is worth addressing. Second, we present a vocabulary and graphical notation that support reasonably precise discussion of issues in this space. Third, we show that our ideas can be realized in a concrete, computationally effective form. We have developed an approach for combining formal application descriptions with architectural style descriptions to synthesize architectural descriptions. Fourth, we present an assessment of the viability of our approach by using it to replicate earlier architectural studies from the literature. We show our formally and automatically derived architectural descriptions to be consistent with results derived previously, informally and manually.

The rest of this paper is organized as follows. Section 2 presents our approach in theory. Section 3 shows that the theory can be automated. Section 4 hypothesizes that our automated approach produces results that are consistent with previously informal results, and presents experimental data in support of this claim. Section 5 shows that comparing our work with related efforts has the potential to improve both. Section 6 concludes.

2. THEORY

This paper makes explicit and elaborates the notion that an architectural map is what combines an application description, p, with a style description, s, to produce an architectural description \( i(p, s) \), for application \( p \) in style \( s \).

\[ i : \text{ArchDesc} = \text{archMap}(p : \text{AppDesc}, s : \text{ArchStyle}) \]
This map is the principal object of our study. Putting it at the center begins to balance attention to architectural styles, with attention to how style choices combine with application descriptions to yield architectures. Knowledge of this mapping is crucial to expertise in software design. Given an application description in some style, the experienced designer knows both what architectural style to pick, and how to map an application description of the given kind to an architectural description in the chosen style.

Clearly map is a complicated object. In some sense it embodies all our knowledge about how to realize different kinds of systems in different architectural styles. We need a way to study it in pieces.

One contribution of this work is an approach to doing this: We decompose map by treating it as a function polymorphic in both application style and architectural style. We then investigate map for specific pairs of styles. We thus make explicit a notion of application style. Application descriptions come in many forms, e.g., composition of functions, or state machine. Different architectural maps are, in practice, used in different styles, e.g., composition of functions, or state machine.

We make our idea more precise as a dispatching relation, \(disp\). Given an application style, \(t\), and an architectural style, \(s\), we view \(disp\) as indexing, or dispatching to, a \(t\)-\(s\)-specific map, \(\text{disp}(t,s) = \text{map}_{(t,s)}\), that in turn takes an application description, \(p_i\) in style \(t\), and an architectural style description, \(s\), and that yields an architectural description, \(i(p_i,s)\), in style \(s\), that refines \(p_i\).

Decomposing map into style-specific pieces enables us to build up a theory and automated style-specific tools in an incremental fashion.

Our theory and language for discourse in this area includes the following basic constructs: a set, \(\text{ArchStyle}\) of architectural styles; a set of \(\text{ArchDesc}\) of architectural descriptions; a binary relation, \(\text{confoms}\), on the cross product, that encodes the conformance (or not) of an architectural description, \(i\), to an architectural style, \(s\); a set \(\text{AppStyle}\) of application styles, a set, \(\text{AppDesc}\), of application descriptions; an analogous binary relation, \(\text{refines}\), encoding the refinement proper.

We chose Alloy for this study for two reasons. First, its ability to compute solutions that satisfy complex sets of constraints is useful as an automation mechanism. Second, and more importantly, it allows us to use published architectural style specifications written in Alloy as inputs \([6,12]\). Reusing published models is not only convenient, but is an important part of our approach to validating our ideas: It shows them to be consistent with contemporary formal accounts of architectural style.

We view the technology that we describe here as an early demonstration prototype only. We do not believe Alloy will ultimately prove to be the best, or at least the only, technology for reducing our ideas to useful tools. The novel concept in this work encompasses any technology that takes separate application and architectural style descriptions, broadly construed, and that automatically computes, or otherwise manipulates, corresponding architectural descriptions of systems with the required refinement and conformance properties.

### Figure 1: Key relations in a commutative diagram.

- \(t : \text{AppStyle}\)
- \(p_i : \text{AppDesc} \xrightarrow{\text{in}} \text{disp}(t,s) \xrightarrow{\text{in}} s : \text{ArchStyle}\)
- \(i(p_i,s) : \text{ArchDesc}\)
- \(\text{apply}\)
- \(\text{refines}\)
- \(\text{confoms}\)

### Figure 2: Architecture as an independent variable.

- \(\text{AppDesc}_{\text{cf}}\)
- \(\text{ArchDesc}_{\text{cf}}\)
- \(p_{(s,t)}\)
- \(i_{(p_{(s,t)},o)}\)
- \(i_{(p_{(s,t)},f)}\)

### 3. PROTOTYPE

In this section we show that our ideas can be reduced to practice. We can implement tools that support architecture as an independent variable. There are many possible approaches to implementing tools that compute architectural maps. Here we describe one approach, in which we use Jackson's Alloy specification language \([5]\) to represent application descriptions, architectural style descriptions, architecture maps, and computed architectural descriptions.

We chose Alloy for this study for two reasons. First, its ability to compute solutions that satisfy complex sets of constraints is useful as an automation mechanism. Second, and more importantly, it allows us to use published architectural style specifications written in Alloy as inputs \([6,12]\). Reusing published models is not only convenient, but is an important part of our approach to validating our ideas: It shows them to be consistent with contemporary formal accounts of architectural style.
3.1 Application Description

We illustrate our approach with a simple example, drawing on a widely known but previously informal case study from the literature: the mapping of a description of Parnas’s KWIC application to an architectural description in the pipe-and-filter style [8, 10]. In terms of application style, we describe KWIC as a composition of functions, a structure implicit in Parnas’s original paper [8]: “The KWIC [Key Word in Context] index system accepts an ordered set of lines. . . . Any line may be circularly shifted by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.”

\[ KWIC(\text{text}) = (output \circ sort \circ shift \circ input)(\text{text}). \]

For purposes of demonstrating the feasibility of reducing our approach to practice, we represent this composition of functions as a sequence of function objects in Alloy. To be specific, we specify this sequence in Alloy, and run the Alloy Analyzer to produce it as a solution. Figure 3, illustrates the result produced by Alloy: the application description consumed by our implementation of the architecture map. We elide the simple Alloy specification of this sequence of functions from this paper.

Figure 3: Alloy representation of application description for KWIC in composition-of-functions style.

We have not yet attached semantic specifications to the constituent functions, or to the KWIC function as a whole; but we hypothesize that it is feasible to do so and that such specifications could be carried through our maps into the resulting architectural descriptions, to provide semantic guidance to programmers. We identify this idea as an opportunity for future research and development.

3.2 Architectural Map

In this subsection we continue our explanation of how our ideas can be reduced to practice by exhibiting an Alloy implementation of architectural map, \(map_{(cf,pf)}\), taking Alloy-encoded application descriptions in the composition-of-functions \((cf)\) style, and producing architecture descriptions in the pipe-and-filter \((pf)\) style.

A pipe-and-filter architecture describes a system as a collection of components, filters, and connectors, pipes. A filter consumes data from an upstream pipe connected to an input port, transforms it in some way, and passes the results to a downstream pipe connected to an output port. Filters operate concurrently. Pipes connect filters to other filters.

We exploit existing Alloy formalization of such styles. Figure 4 outlines the pipe-and-filter style description, which in turn extends Wong’s component-and-connector meta-model of architectural style in Alloy [12]. The style description defines eight signatures: Input, Output, Source, Sink, Data-Source, DataSink, Filter and Pipe. In details elided from this paper, Filter is specified as having Input and Output ports. Pipe, a kind of Connector, has two roles: source is the input role of a pipe; sink is the output role. Data-Source, DataSink model data sources and sinks, respectively, in pipe-and-filter descriptions.

abstract sig Input extends Port {}
abstract sig Output extends Port {}
abstract sig Source extends Role {}{...}
abstract sig Sink extends Role {}{...}
abstract sig DataSource extends Component{}{...}
abstract sig DataSink extends Component{}{...}
abstract sig Filter extends Component{}{...}
abstract sig Pipe extends Connector{}{...}

Figure 4: Pipe-and-filter style (elided) in Alloy.

An architectural style description of this kind specifies the co-domain of an architecture map. To represent a map, itself, such as \(map_{(cf,pf)}\), we extend the style description with mapping predicates. These predicates take application descriptions as parameters (such as the KWIC structure illustrated above), and define relationships required to hold between them and computed architectural descriptions. Given an application description, and a map, Alloy computes corresponding architectural descriptions. Alloy guarantees that all computed descriptions conform to the given architectural style. Our predicates are responsible for ensuring that computed architectural descriptions refine given application descriptions. Our ideas are thereby reduced to practice. While noting that architectural styles include semantic constraints, here we only consider structural refinements. In practice it will be important to represent and refine semantics as well. We do plan to develop this observation in future work.

Figure 5 presents handleFunctions, the parameterized predicate that represents \(map_{(cf,pf)}\). It accepts application descriptions in the composition-of-functions style and produces pipe-and-filter architecture description. It specifies that for each function in the composition of functions there is a component in the architectural description that handles it. The Output port of filter is connected to the Source role of a downstream pipe, the Sink role of which is attached to the Input port of subsequent filter. DataSource and DataSink are specified as we described previously. The handleFunctions predicate additionally uses scope-PnP parameterized predicate to ensure the correct Alloy scope for the System.

3.3 Architecture Description

We use the Alloy Analyzer to compute architecture descriptions, represented as satisfying solutions to the constraints of a map given an application description. Figure 6 illustrates the computed result for our KWIC example. The diagram is accurate for the result that Alloy computed, but we have edited it to omit some details for readability (ports of filters, and roles of pipes, for example). In this diagram, DataSource, a specific type of Filter, handles the input function. Its input port is connected to Pipe0. Filter1 handles the CS function. Its input and output ports are connected to Pipe0 and Pipe1, respectively. Similarly, the other filters handle alph and output functions.
We presented a software tool for taking undue complexity, how we have reduced our ideas to practice for two reasons. First, it helps us explain, without undue complexity, how we have reduced our ideas to practice. Our theory supports a formal, automated approach to deriving architectural descriptions from application descriptions and architectural style definitions. The entries in the table indicate the case studies from the literature to which we have applied our maps, to test the consistency of our results with the informal, manually derived results in the literature. We have applied our work to two case studies: KWIC, long used in studying architectural styles and their properties; and the Lunar Lander (LL) case study of Taylor et al. Overall we have thus performed seven experiments to date. We described the (CF, PF, KWIC) in the preceding section. We elide discussion of (CF, PF, LL) because it does not add anything new. For reasons of space, we also elide (CF, II, KWIC). The following four subsections thus report on the execution and results of four experiments: (CF, OO, KWIC), (SD, OO, KWIC), (CF, II, LL), and (CF, OO, LL). In each case we cite the study that we are recapitulating in a formal, automated fashion.

4.1 Experiment: CF, OO, KWIC

This experiment attempted to reproduce previous informal studies by Parnas [8] and later studies by Shaw and Garlan [10]. We map a CF description of KWIC to an architectural description in the OO style. We have already presented our simple CF description of KWIC. The rest of this subsection defines our OO architectural style, discusses our architectural map, map_{cf,oo}, and presents and analyzes results of applying it to our KWIC architectural description in the CF style.

One of the most common architecture styles is object-oriented. Figure 8 presents part of our Alloy model of a constrained OO style. The specification defines three Alloy signatures: Interface, Implementation, and Object. An Object has an interface, an implementation, and a set of other interfaces on which the object depends. Elided predicates state that no object may depend upon the implementation of any other Object (in our constrained formalization of this style). The implementation of one Object may depend on the Interfaces of other Objects, except that (as we define the style here) no object may depend on its own interface.

We next define a parameterized predicate for mapping any composition of functions to an OO architecture. It takes a sequence of functions (representing a composition of functions), and constrains the architectural descriptions to have a single object to be represented.

```
pred handleFunctions[cons:seq Function]{
  all f:Function | one x: Component | x.handle = f
  one DSource:DataSource, DSink:DataSink |
  DSource.handle = cons.first
  && DSink.handle = cons.last
  one s:System | all f:Filter |
  one r1:Source | one r2:Sink |
  one p1:Output | one p2:Input |
  (f.handle.next!=none) &&
  (f.handle.'next'!=none) =>{
    nFilter.handle = f.handle.next
    && r1 in apipe.roles && r2 in apipe.roles
    && p1 in f.ports && p2 in nFilter.ports
    && r1->p1 in s.attachments
    && r2->p2 in s.attachments
  }
}
```

Figure 5: map_{cf,pf} represented in Alloy.

Figure 6: The map of composition-of-functions of KWIC into the PF style
We note that change can be seen as involving, at a more abstract level, a change in the style of application. Notkin [4], who explored, among other things, how changing an application description might demand corresponding changes in architectural style. This helps us organize the states in a linear fashion, which is all we need in this case. Figure 11 presents our description of an interactive version of KWIC in the state-driven style. (We wrote an Alloy specification, which the Alloy Analyzer solved to produce this structure.)

The initial state of the interactive-KWIC is Wait, in which the system waits to receive a command. There are transitions from the Wait state to the Add, Delete, Print and Exit states (and back) driven by interactive commands. The line buffer is updated in the Add and Delete states. As the line buffer is altered, the circular shifter is invoked to create the shifted lines. The Alphabetizer state has also access to the shift lines buffer. It is triggered by the completion of the shifter activities to sort lines in the buffer. The Alphabetized lines buffer is used to hold the alphabetized shifts. Finally, the Print state displays the latest buffer.

Figure 12 depicts the computed interactive-KWIC architectural description, in object-oriented style, that we computed. For clarity of presentation, we again omit details. The System here consists of a collection components. There is an object to handle each of the shared data objects, namely LineBuffer, ShiftedLines and AlphLines as well as the InteractiveKWIC.

Our mapping function uses the State pattern [2]. This pattern is useful when an object can be in one of several states, with different behaviors in each state. The pattern implements states as classes that are instantiated as an object enters each state. Operations invoked on the object are delegated to those instances, which have their own methods for the operations. In the same way, as can be seen from the diagram, each state is assigned to an object to be handled. Every Object has an Interface and Implementation (elided from the Figure). Each of the state Object depends on the Interface of the Object handling the associated shared data object. At a cross-path point, Object4 handles the Alphabetizer state, having an access to ShiftedLines and AlphLines. Object4 thus depends on the Interfaces of Object10 and Object9, which handle the two shared data objects, respectively.

### 4.2 Experiment: SD, OO, KWIC

This experiment addresses the work of Garlan, Kaiser & Notkin [4], who explored, among other things, how changing the KWIC application from batch-sequential to interactive might demand corresponding changes in architectural style. We note that change can be seen as involving, at a more abstract level, a change in the style of application description, and that this change is what really drives the need for a new architectural style.

We employ state-driven behavior as a style for interactive application description. The rest of this subsection introduces this style of description, an architectural map from this application style to the OO architectural style, and the results of applying this map to a description of an interactive version of KWIC.

Our interactive-KWIC system accepts a line of input at a time and outputs an alphabetized list of the current collection of lines. We modeled interactive-KWIC as a state-driven system, where each state represents one of the behavioral modes of the system (inputting, sorting, etc), and where transitions represent sequencing among modes. To do this in Alloy we use its polymorphic linear ordering module.
Figure 10: The map of the composition of functions of KWIC into the OO style

Figure 11: Interactive-KWIC described in state-driven style.

Figure 13: part of II style described in Alloy

\begin{verbatim}
abstract sig Publish extends Role {}  
abstract sig Subscribe extends Role {}  
abstract sig PublishEvent extends Port {}{...}  
abstract sig SubscribeEvent extends Port {}{...}  
abstract sig IIObject extends Object{}{...}  
abstract sig EventBus extends Connector {}{...}  
\end{verbatim}

Figure 14 presents predicates that define our architectural map. The handleIIFunctions parameterized predicate ensures a correct architectural structure. It takes a sequence (representing a composition) of functions. It uses three other predicates: scope-OO, handleIIObjects and handleEvents. The former predicate sets the Alloy scope of the System. handleIIObjects specifies the ports of each IIObject based on its role as a Publisher and/or Subscriber. The handleEvents predicate states that for each publisher IIObject, its PublishEvent port is attached to the publish role of an EventBus. The subscribeEvent port of a relevant subscriber, on the other hand, is attached to the subscribe role of that EventBus.

According to the informal description of the lunar lander, the Spacecraft component maintains the state of the spacecraft (its altitude, fuel level, velocity, and throttle setting). After calculating the altitude, fuel level and velocity, it emits those values to the event bus. Receipt of events providing the spacecraft’s altitude, fuel and velocity cause the Display component to update based on those values. The GetData component obtains new burn data settings from the user;
when this happens the component emits a notification of this new value onto the event bus. Upon receipt of this notification the Spacecraft component updates its internal model of the spacecraft and emits the updated state back to the bus.

Figure 15 presents the architectural description of the System, generated by Alloy. To simplify the diagram, we omit some details, such as interface and implementation of objects as well as the roles of EventBus elements. The architectural description has three IIObjects. The LunarLander element, inferred from the input specification, represents the function set of the Lunar Lander System. GetData, Spacecraft and Display constitute LunarLander. Each IIObject handles a function. As a case in point, IIObject2 handles GetData function. Object1, on the other hand, depends on the Interface of IIObject2 and handles the Spacecraft function. The last function of Lunar Lander is Display which is handled by Object0 depending on the Interface of Object1.

4.4 Experiment: CF, OO, Lunar Lander

Mapping our composition-of-function description of the Lunar Lander to the OO architectural style yields the architectural description depicted in Figure 16. As an aside, we note that Taylor et al., indicate that the application description of the lunar lander that the use in discussing the OO architectural style is not exactly the same as the one illustrated in the OO style section of the book.

According to Figure 16, generated again by Alloy, the System comprises three components. LunarLander along with the system’s functions, namely GetData, Spacecraft and Display, can be seen at the right side of the diagram. Each Object has its Interface and Implementation. Their connections are mentioned by labeled arcs. Object2 handles GetData function. Object1, on the other hand, depends on the Interface of Object2 and handles the Spacecraft function. The last function of Lunar Lander is Display which is handled by Object0 depending on the Interface of Object1.

4.5 Discussion

Our experiments show that architectural maps can be implemented in a computationally effective manner. We have used this technology to recapitulate studies of architectural style and choice from the research literature. The results of our formal and automated computations are consistent with the informally and manually produced results documented in the literature. By simply swapping between implementations of architectural maps, we are able to produce difference architectural descriptions for a given system from a high-level application description. Although we have not yet attempted experiments beyond those replicating studies from the literature, we are encouraged. Our early work appears to support the idea that being able to treat architecture as an independent variable is a plausible aspiration.
pred handleIIObjects()
all o:IIObject| 
{(o.handle.next!=none) &&
  (o.handle.~next!=none)} =>
  { one SubscribeEvent & o.ports
    one PublishEvent & o.ports}
else {(o.handle.next=none) } =>
  { one SubscribeEvent & o.ports
    no PublishEvent & o.ports}
else {(o.handle.~next=none) } =>
  { no SubscribeEvent & o.ports
    one PublishEvent & o.ports}
}

pred handleEvents()
all o:IIObject| 
{(o.handle.next!=none) &&
  (o.handle.~next!=none)} =>
  { some s:System| one e:EventBus |
    one sRole:Subscribe| one pRole:Publish|
    one sPort:SubscribeEvent| one pPort:PublishEvent|
    sRole in e.roles && pRole in e.roles
    && pRole-> pPort in s.attachments
    && sRole-> sPort in s.attachments
    && sPort in o.handle.next.~handle.ports
    && pPort in o.ports }
}

pred handleIIFunctions(cons: seq Function){
  scope_OO[1,#cons]
  handleIIObjects[]
  handleEvents[]
}

Figure 14: Selected Alloy predicates for the II style.

5. RELATED WORK

The work we present here is related to many other research efforts. In this paper, we discuss two especially close connections: to model-driven architectures [7] for program synthesis from abstract models, and to recent work of Garlan et al. [3] on the evolution of programs in terms of architectural style.

5.1 Model-Driven Architecture

The term model-driven architecture (MDA), refers to the architecture of a kind of programming system that support mapping of high-level, platform-independent application descriptions to executable code specialized to run on specific, often distributed and highly constrained, hardware-software platforms. The MDA architecture is rooted in a mapping that takes a platform-independent model, \( p \), and a platform definition model, \( s \), to a platform-specific model, \( i \). That is, 
\[
i : \text{PSM} = \text{map}(p : \text{PIM}, s : \text{PDM})
\]

The analogy with our approach is clear in this equation. Compare it with our mapping equation: 
\[
i : \text{ArchDesc} = \text{map}(p : \text{AppDesc}, s : \text{ArchStyle})
\]

The key similarity is of course that both approaches map high-level descriptions of applications, by way of choices of some target domain (platforms in MDA work, and architectural descriptions in our work) to detailed descriptions

in the given target space. In particular, both methods take high-level, target-independent application descriptions as inputs: PDM-independent models in the MDA context, and architectural style-independent application descriptions in our work.

The key differences between MDA work and our own are also clear. First, our target domain is one of architectural descriptions, whereas the target domain for MDA work is that of executable program representations (PSMs). PSM’s are often Java or C++ programs, for example. Second, MDA mappings are parameterized by descriptions of computing platforms as targets for code generation, placing MDA work broadly in the tradition of retargetable compilers for high-level languages. Our maps, by contrast, are parameterized by formal architectural style descriptions. In a sense, abusing terminology, what we propose could be seen an an MDA for generating, from high-level application descriptions, not programs for given platforms but architectural descriptions of programs in given architectural styles.

In this context we identify two areas for future work. First, we plan to explore the hypothesis that research in \( \text{PIM} \) languages can inform our own work on formal application descriptions. Perhaps we can literally share some kinds of application models.

Second, it seems clear that the MDA’s can perhaps be described overall in terms of our diagram in Figure 1, including, in particular, notions of \( \text{PIM styles} \), analogous to application styles. Furthermore, we might explore the analogy between PSM’s and architectural style descriptions.

The incongruity in the analogy is that PSM’s describe platform \( \text{instances} \) while architectural styles, in a manner of speaking, seem to define \( \text{types} \) of architecture descriptions. Perhaps both MDA and our own work can be helped by completing the picture on both sides. First, we can see an
architectural style description, such as OO, an instance of a broader class, architectural style, that explicitly captures, at a meta-level, what we mean by the term style. Such a meta-model is in fact present even in this work, as Wong’s Alloy architectural style meta-model, which introduces terms such as component and connector. In either case, what we’re doing is introducing a higher-level abstraction and a new conformance relation.

Similarly, on the MDA side, we have a clear notion of platform instance, but lack a notion of platform style, which we now make explicit. We thus treat each PSM as having some style. The symmetrization of our diagram in Figure 17 is graphically nice; it suggests that we can increase reuse by characterizing the polymorphic nature of our maps in terms of high level styles on both sides; and it now clearly exhibits the style-pair-based polymorphism that provides the basis for the modular implementation and incremental development of specialized map functions.

5.2 Incremental Evolution of Architecture Style

The second piece of related work that we discuss is the recent work of Garlan et al. on the evolution of programs with respect to architectural style. The premise of this work is that it is sometimes necessary to change a program written in one architectural style into a related program in another style. Garlan et al. note that such changes are often hard to make because they involve substantial disruption to the existing code base. The approach that they propose is one involving incremental steps between programs, each step being effected by the application of a well defined incremental architectural operator.

We can use the style notation that we have developed in this paper both to characterize the work of Garlan et al., and also to compare and contrast it with our work. Thus, Garlan et al. start with an architectural description $i_{(p_s,s)}$ and decompose its transformation into a new system, $i_{(p_s',s')}$.

The key ideas that remain implicit in Garlan et al., and which are the central focus of our work, are (1) we are dealing with are one or more architecture-independent application descriptions ($p_t$ and $p_t'$), (2) architectural descriptions ($i_{(p_s,s)}$ and $i_{(p_s',s')}$) are obtained from such application descriptions by way of architectural maps, and (3) we can and should make the architectural maps explicit: $map_{(t,s)}(p_t)$ and $map_{(t',s')}(p_t')$. Beyond just evolutionary transitions between pairs of architecture descriptions, involving changes in architectural style, we focus on architectural style as an independent variable in design. The following tree depicts a choice between architectural styles implicit in kinds of evolutionary transitions that Garlan et al. discuss.

Of course a fundamental motivation for our work is an identical concern with the difficulty of selecting and changing decisions about architectural style. We are thus fully "on board" with the notion that one will sometimes wish to change the architectural style of a system. Success in making architectural style an independent variable would, in practice, ease such transitions by allowing for the automated regeneration of a system from a given application description: as a kind of architectural retargeting. Adding a double arrow to our diagram illustrates this idea:
achieve the same effect. Of course the latter is indeed what we do today, and so having a theory to describe it is very helpful, while the previous diagram represents an aspiration for a fundamentally better approach.

As a last note in this section, we observe that the preceding diagram does not capture the exact evolutionary scenario of Garlan et al. They assumed that some change in the application, \( p_t \), and perhaps in its description style, \( t \), drove the need for a change in architectural style. What the diagram above depicts is a fundamental change in architectural style independent of any change in either the meaning of the application, \( p_o \) or in the application style, \( t \), in which that meaning is described.

The notation that we have described makes these separate dimensions of evolution explicit, and thus provides what appears to be a useful and interesting new category-theoretic-like, graphical language for describing a wide variety of software change scenarios, involving changes in important independent dimensions: application style, architectural style, application content, and, indeed, mapping strategies.

Here then is a diagram capturing the scenario of Garlan et al. expressed in our terms and related to our concept. A change in the application, \( p_t \), and perhaps (we assume) even in the application style, \( t \), is driving the need to re-architect the system, taking it from \( i(p_t,s) \) to \( i(p'_t,s') \). The change also includes a transition from architectural style \( s \) to \( s' \). Garlan decomposes the bottom path into intermediate system states. Without depicting this decomposition, we use a dashed double arrow to represent the idea that, with the kind of automation we envision and have demonstrated in a prototype form, the amount of manual work at this level might be substantially reduced or even eliminated.

\[
\begin{align*}
\text{map}(t,s)(p_t) & \xrightarrow{\Delta_{\text{app}}} \text{map}(t',s)(p'_t) \\
\text{map}(t,s)(p_t) & \xrightarrow{\Delta_{\text{arch}}} i(p_t,s) \xrightarrow{\Delta_{\text{arch}}} i(p'_t,s')
\end{align*}
\]

6. CONCLUSION

In summary, the principal contributions of this work are in four areas. First, we identified the treatment of architecture as an independent variable as a key problem area and goal for software engineering. Second, we presented a conceptual architecture to make this idea precise, including a category-theory-like graphical notation showing how the key concepts relate to each other. Third, we demonstrated the feasibility of automated computation of architectural descriptions with an executable prototype developed in Alloy, exploiting previously peer-reviewed, formal definitions of architectural styles as inputs. Fourth, we presented data that appear to support our hypotheses, and the proposition that these ideas are perhaps worth pursuing further.

We identify three key goals for future work. The first is to further develop what at present are clearly rudimentary application description styles. Extending such descriptions to include richer semantics, and then refining these semantics through mappings down to architectural descriptions appears to be quite important. Second, we are considering exploring the extension of our work to include subsequent mappings from architectural descriptions to code. Ultimately we want not only to map high-level descriptions to code, but for these maps to be invertible, so that we can abstract high-level descriptions from code, prior to remapping back to code with a new architecture. Finally, at an appropriate point, we intend to undertake experimental tests of the viability of these ideas for realistic system design, implementation, and architectural evolution.

7. ACKNOWLEDGMENTS

We thank Professor David Garlan of Carnegie Mellon University for helpful discussion of this work, especially on the relationship between the ideas in this paper and his recent work on how systems evolve from one style to another. This work was supported by a grant from the National Science Foundation.

8. REFERENCES