Architecturally Correct-by-Construction Synthesis for Contemporary Frameworks and Middleware Platforms

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

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

ABSTRACT

We recently contributed correct-by-construction synthesis of architectural descriptions as a form of model-based development (MBD), in which architecture-independent application models are combined with specifications of architectural styles to yield families of architectural descriptions for given applications in given styles. We have demonstrated MBD tool support for such styles as implicit invocation and pipe-and-filter. We now show this approach can work for contemporary architectural styles induced by modern standards, frameworks and middleware, and that it can be combined with code synthesis to enable the synthesis of architecturally correct-by-construction code targeted to modern architectural frameworks and middleware. We validate our claims by demonstration. Starting with a formalization of a model of a sense-compute-control system from the literature, we synthesize code for two frameworks: in the REST style for the Restlet framework on a network of Android phones, and in the implicit invocation style for the Prism middleware.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architecture

General Terms

Design, Architectural Styles

1. INTRODUCTION

Architectural decisions, including decisions about the architectural styles in which systems will be developed, are generally understood as the earliest and most difficult to change [37]. Yet style decisions are increasingly volatile [17], for at least two reasons. First, they are increasingly being driven by rapidly changing industrial standards, platforms and middleware technologies that induce or define the styles in which applications are written [29]. Second, we face increasing demands for the mapping of application models to diverse platforms: e.g., interoperability protocols for electronic health records systems. We need new concepts, methods and tools for abstracting from, and for later and more diverse bindings of, otherwise hard to change architectural style decisions.

Our earlier work addressed this problem with a new approach to correct-by-construction synthesis of architectural descriptions from formal, abstract application models and separate choices of architectural style specifications [5]. We also showed that our approach is consistent with model-based development (MBD) perspectives and technologies [6]. However, this work left open questions about support for modern, industrially relevant architectural styles such as representation state transfer (REST), and about the synthesis of code for platforms that support such styles. This paper resolves these issues.

The contribution of this paper is the demonstration that our notion of *architectural maps* promises to support architectural styles induced by modern architectural standards, platforms, and frameworks, and the synthesis of architecturally correct-by-construction code for such platforms and frameworks. Among other things, this work provides experimental support for the proposition that it can be practical to pay a one-time cost to formalize the architectural styles defined by modern standards to enable use of our specification-driven approach to architecture and code synthesis.

Validation of these claims of feasibility is by demonstration of a working MBD tool and its use. We present an example of a sense-compute-control application model system from the published literature and its mapping to executable code frameworks for two distinct targets: the Prism-MW middleware and the Restlet framework for RESTful web-based applications. Our RESTful application instance is deployed on a network of Android phones, and its synthesis was based on a formalization of the REST architectural style developed for this work.

The rest of the paper is organized as follows. Section 2 presents our approach for dealing with architectural style as a variation point with a model-based synthesis approach. Section 3 describes two middleware platforms and a running case study that has been developed for them using the proposed approach and an extension to our Monarch tool. Section 4 surveys related work. Section 5 concludes and discusses future work.

2. APPROACH

This section presents our approach, which builds on work in three areas: our previous work on architectural maps [5,

University of Virginia, Technical Report # CS-2010-15.

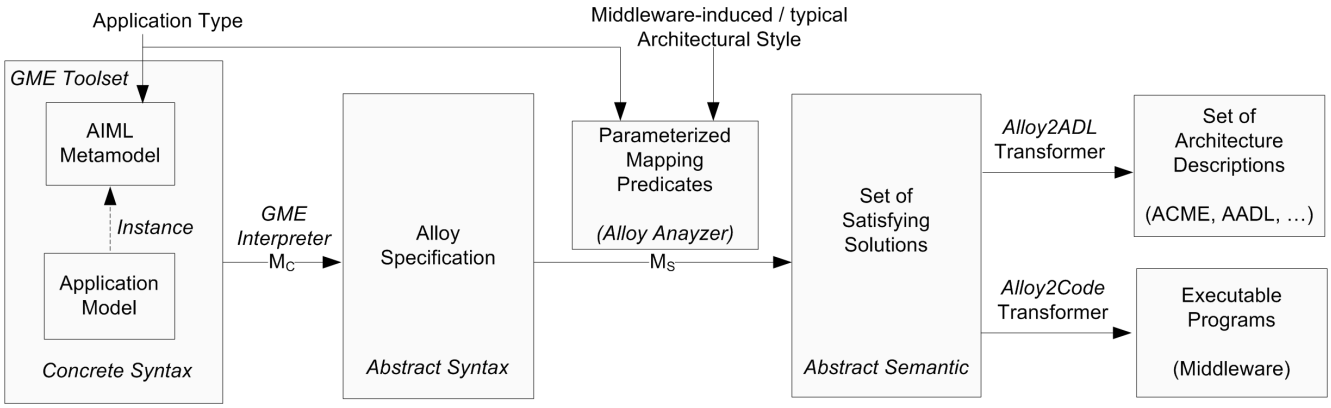

Figure 1: High-level overview of the approach

6]; work starting with Di Nitto and Rosenblum [29] that recognized that application frameworks and platforms induce or support architectural styles; and traditional model-based synthesis of code from architectural models.

2.1 Architectural Maps

Our earlier work introduced and formalized the *architectural map*, based on the idea that choices of architectural style can and should make separately from choices of other application properties. That is, architectural style is a separate variable in design. In our approach, one develops an application model of a given *application type* then selects an *architectural map* for that type and a given architectural style. Applying this map to the application model yields a family of correct-by-construction architectural descriptions for the given application in the given style. The concept of *application type* comes to parallel that of *architectural style*. Our approach builds on these ideas, extending them in two dimensions. First, we formalize the architectural styles required by modern frameworks. Second, we use standard model-based techniques to synthesize code from our synthesized architectural descriptions.

The overall mapping occurs in five steps, as illustrated in Figure 1. First, we define architecture-independent application properties in a model of a given application type. We customize the Generic Modeling Environment (GME) [25] such that application types are realized concretely as GME meta-models, providing *architecture-independent modeling languages (AIML)*. Second, we transform application models, viewed as *concrete instances* of application types, to an *abstract syntax*. We use Alloy [20] to specify application types and as an abstract syntax for application models. Alloy is appropriate for defining the formal semantics of modeling languages [22]. Third, we combine application models with choices of architectural style specifications and use the Alloy Analyzer to compute architectural descriptions satisfying the conjunction of the constraints. This paper first shows that this can be done for industrially important styles. Fourth, we transform the resulting Alloy solutions (architectural descriptions) into concrete architecture description languages (ADL). The new transformation in this paper (bottom right) uses traditional model-based code synthesis to map synthesized architectural models to executable code that runs on industrial frameworks and platforms.

2.2 Middleware-Induced Architectural Styles

Software-intensive systems are continuously growing in size and complexity. In recent years, they have migrated further from traditional enterprise settings to distributed and embedded environments. Middleware infrastructures are further emerging to be used as a major building of large-scale distributed systems. There are many middleware platforms, such as CORBA and DCOM, and application frameworks, such as Ruby on Rails, Microsoft.net, Enterprise JavaBeans (EJB), and Restlets.

A design method commonly used in practice is to select a platform for the services it offers and then to design software in the de facto architectural style it requires. This approach can have a counterproductive impact on system architecture. Deferring middleware decisions promotes separation of concerns and a high level of abstraction in early software design [29, 37]. On the other hand, middleware decisions are not independent of architecture. Decisions regarding architecture can limit decisions about the middleware that can be used to implement a system.

Problems can occur when the architectural styles chosen for an application conflict with the assumptions of the chosen middleware. Blair et al. [8] argue that the architectural models can be used in systematic synthesis of middleware configurations. It would be helpful to consider structural and behavioral constraints implied by middleware infrastructures as architectural styles [29]. Formal definition of these styles will allow architects to exploit them in a way that avoids mismatches between required application properties and the constraints imposed by middleware-induced architectural styles.

Numerous approaches have been explored to separate and relate middleware infrastructures and the architectural styles they induce: for embedded systems [26], web-based systems [18], etc. However, insufficient progress has been made on mapping architecture-independent application models into the modern and practical, middleware-induced architectural styles and in turn, into architecturally compliant implementations. Our vision is to map architecture-independent application models, considered as platform independent models, to implementations in conformance with the architectural styles induced by platforms and other complex and practical application frameworks. This paper shows that this vision can be realized in practice.

2.3 Architectural Frameworks

In practice, platforms, frameworks and design approaches often emerge before their underlying architectural styles are fully developed, understood or characterized precisely. The architectural styles they induce are clarified later. Platforms are then refined to support the refined styles in every respect [9]. More precisely, *accidental architectures* emerge; and once they succeed, their architectural styles and patterns are studied more systematically [15].

The architectural styles so derived promise benefits for both development and maintenance. However, even formal specifications of architectural styles generally lack bindings to implementation-level constructs. It is thus difficult to verify the fidelity of the developed software system with respect to the architectural models. To use architectural models and stylistic guidelines extracted from the middleware platforms in an effective manner, such models should also provide support for implementation [31]. Implementing architectural models further is a problem of mapping abstract design decisions to implementation elements that realize those decisions [37].

Frameworks can provide the required bridge. In concrete terms, an architectural framework is a software technology built upon the functionalities provided by the programming language and the operating system that provides services with respect to a supported architectural styles. Architectural frameworks are practical technologies that facilitate system development in conformance to a specific architectural style. They are considered as a significant strategy for bridging the gap between architectural models and associated implemented technologies. In this paper we fill the gap between architectural models and executable code by synthesizing code for specific architectural frameworks from synthesized architectural models in the supported architectural styles.

There are many architectural frameworks for different languages, platforms and styles [37]. Examples include the standard I/O library, java.io, and java.nio for pipe-and-filter architectures; Lightweight and Flexible C2.frameworks for the C2 style [36]; the Aura framework [33] for the Aura style for ubiquitous computing; and numerous architectural frameworks in various platforms for REST architectural style. We envision—and this paper demonstrates the potential for—synthesis of architecturally correct-by-construction code from synthesized architectural models for a wide variety of such frameworks based on the formalizations of the architectural styles they support.

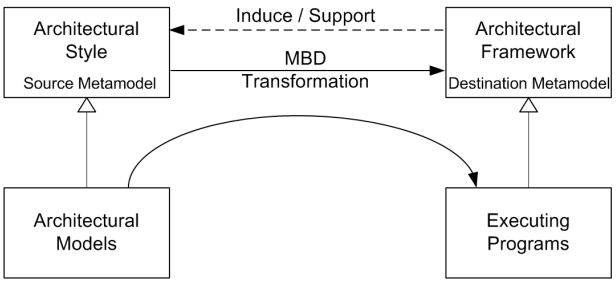

Figure 2: Code Synthesis as MBD model transformations

Tools do exist to support synthesis of code for various languages and frameworks, but to our best knowledge previous work has not exploited the idea that architectural styles can play a key role in bridging the implementation gap. This paper proposes a model-driven code synthesis approach with a focus on architectural style. As represented in Figure 2, our approach is to synthesize code from synthesized architectural models, viewing architectural styles as source metamodels in this second phase of synthesis, and architectural frameworks and middleware constructs as destination metamodels. As such, every (architectural) model which is an instance of the source (style) metamodel can be transformed to an instance of the destination model. More precisely, architectural models conforming to the constraints implied by an architectural style can be transformed to executable code developed atop a middleware supporting the given architectural style.

3. EVALUATION

In this section we report and interpret data from experimental testing of our approach and hypothesis. In a nutshell, we support our claims of the feasibility and potential profitability of our approach with a practical demonstration.

Using our Monarch tool [1], available for download, we have mapped an application model taken from a case study in the published research literature to practical platforms supporting two architectural styles: the Restlet platform supporting the REST style for World Wide Web applications, and the Prism-MW middleware system supporting (among other things) an implicit invocation architectural style for embedded software. Prism-MW [26] is a middleware platform supporting architecture-based software development especially in distributed embedded environments. The Restlet middleware [2] an open source Java framework that supports the REST style. Reasons for choosing Prism-MW and Restlet for this study include their straightforward mechanisms for mapping of architectural constructs into implementations, and their availability and success in their respective industrial and research domains.

The following subsection introduces the running case study which is used extensively throughout the experiments. The next two subsections report on the execution and results of two experiments. We conclude this section by discussing and interpreting the results of our experimental systems effort.

3.1 MIDAS: A Case Study

Our application case study is inspired by Edwards et al. [11]. They illustrate the structuring of an application called MIDAS [27] from a family of embedded applications at Bosch, in two typical architectural styles, client-server and implicit-invocation, to assess the influence of architectural style on quality attributes. Three analyzer components, *FireAlarmAnalyzer*, *IntrusionAlarmAnalyzer* and *LogAnalyzer*, analyze the information provided by *FireAlarmSensor* and *IntrusionAlarmSensor* components for fire and intrusion detection. In case of fire or intrusion, the analyzer transmits messages to the corresponding components.

We view this application as an instance of a sense-compute-control (SCC) as an *application type* for embedded control. We use a simple formal model of the SCC type to model applications in which sensors and actuators are connected to controllers that cycle through the steps of fetching sensor values, computing function values, and sending outputs to

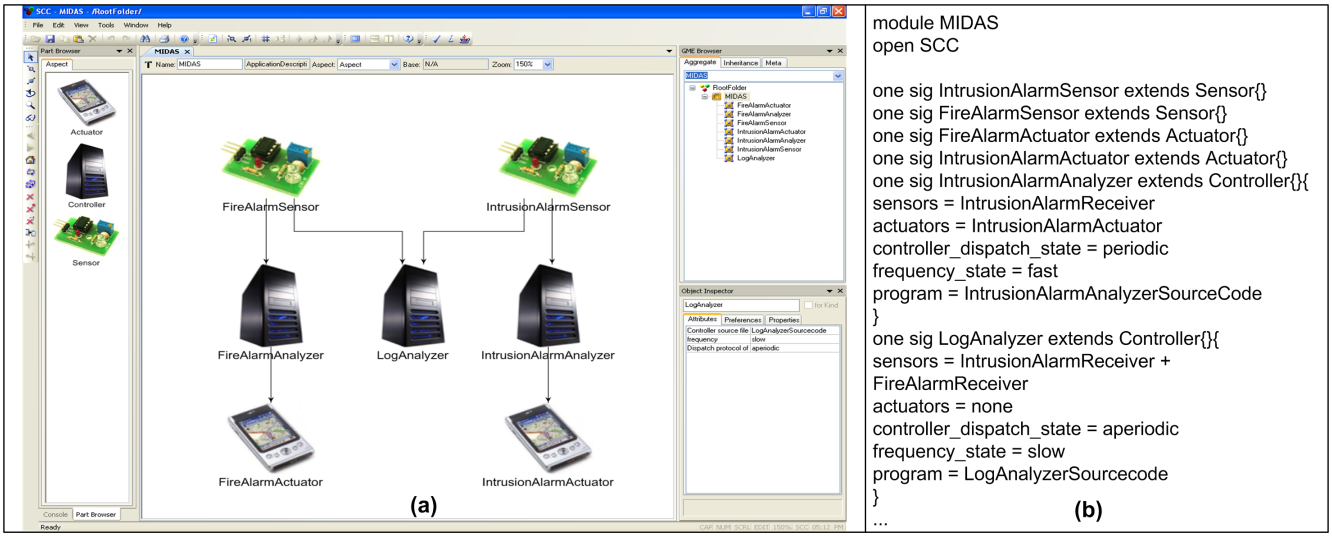

Figure 3: application model of MIDAS example in: a) GME modeling Environment b) Alloy

actuators [37]. Figure 3a illustrates our application as modeled in GME using our generated modeling environment for our SCC meta-model.

Figure 3b presents the Alloy representation of the application model generated from its concrete model by the GME-Interpreter. For presentation purposes, some details are omitted. The Alloy module starts by synthesizing the module name. It then imports the Alloy specification module(s) for application type(s). For each instance of *Sensor*, *Actuator*, and *Controller* in a concrete model, we synthesize a signature definition that represents the inheritance of a concrete element from its associated abstract class. The element's properties (if any) are also specified as Alloy facts for the corresponding signature of that element, e.g. *IntrusionAlarmAnalyzer* has a periodic high frequency task. The following subsections represent the results of mapping this application model to two different platforms.

3.2 Experiment: MIDAS, II, Prism-MW

After modeling the application, Monarch uses the Alloy analyzer, Alloy formalizations of the REST and implicit-invocation architectural styles, Alloy formalization of SCC application type, and corresponding architectural maps to synthesize satisfying architectural models. In this subsection, we have leveraged the architectural mapping for the SCC application type and implicit-invocation style specified in earlier work [6]. The output models refine the application description in conformance with the fully formal definition of the implicit invocation architectural style. Monarch, using our *Alloy2ADL* transformer, converts the abstract XML-format outputs to a traditional architecture description language (ADL). Figure 4 represents one of the formally derived architectural models in Acme.

3.2.1 Implicit-Invocation and Prism-MW

Prism-MW [26] is a middleware platform that provides support for development of software systems by means of the architectural constructs. Having its origin in *programming-in-the-small-and-many* (Prism), the middleware focuses on the embedded systems environment. Besides the typical

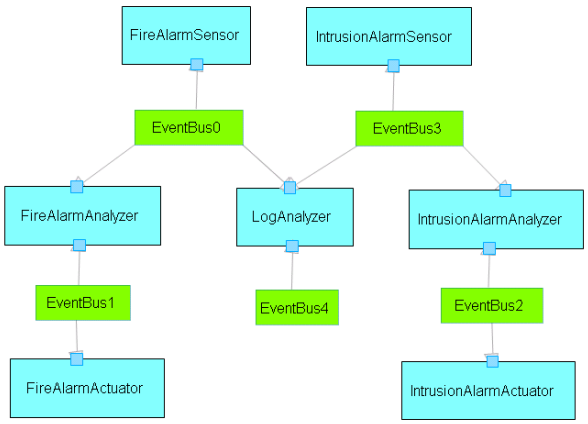

Figure 4: One of the formally derived architectural models in implicit-invocation style represented in Acme

support for architectural concepts, such as component, connector and configuration, Prism-MW provides explicit support for architectural styles. Application architectures designed in selected architectural styles can be implemented by extending Prism-MW classes with application-specific details. We illustrate the process of transforming an architectural model into its corresponding implementation developed atop Prism-MW framework using the running example.

The first step is to declare an implementation class for each component. The synthesized class for each component extends the *AbstractImplementation* base class. The base class provides the code to both send and handle events. It also enables dynamic association between the component and its real implementation [26]. As a case in point, the *FireAnalyzer* component subscribes to the events of *FireAlarmSensor* and will be implicitly invoked. Then, its *handle* method is called automatically by the framework as a request of a recognized type arrives for this component. This allows it to perform the fire detection analysis. This, in turn,

causes the actuator components to be invoked implicitly, as the FireAnalyzer component sends an appropriate message in the necessary case.

Listing 1 presents an automatically synthesized implementation of a formally computed architectural model for this application. For the sake of readability, we omit details. First, an *ExtensibleArchitecture* is created, an object that represents an architectural structure conforming to the rules implied by an architectural style: here, the *publish-subscribe* style. Next, components of the application are created. Each component is an instance of *ExtensibleComponent* with its associated implementation of *AbstractImplementation* installed on it. Then, the instances of each of the connectors are created and added to the architecture. Afterwards, components and connectors are linked together according to the generated architectural model, and the architecture's *start* method is called. The result is an automatically synthesized, architecturally correct-by-construction implementation of the MIDAS application that runs on Prism.

```
1  public class MIDAS {
2    public static void main(String[] args) {
3      ...
4      // Create the MIDAS architecture
5      ExtensibleArchitecture arch = StyleFactory.
           generateArchitecture("MIDAS",
           PrismConstants.PUB_SUB_ARCH);
6
7      // Create components
8      FireAnalyzer fireAnalyzerImpl = new
           FireAnalyzer();
9      ExtensibleComponent fireAnalyzerComp =
           StyleFactory.generateComponent("
           fireAnalyzerComp", PrismConstants.
           PUB_SUB_COMP, fireAnalyzerImpl);
10     fireAnalyzerComp.scaffold = s;
11     ...
12     // Create connectors
13     ExtensibleConnector EventBus0 = StyleFactory.
           generateConnector("EventBus0",
           PrismConstants.PUB_SUB_CONN);
14     EventBus0.scaffold = s;
15     arch.add(EventBus0);
16     ...
17     // Add components to the architecture
18     arch.add(fireAnalyzerComp);
19     arch.add(fireAlarmSensorComp);
20     arch.add(alarmLoggerComp);
21     ...
22     // Create links btw. components and connectors
23     arch.weld(fireAlarmSensorComp, EventBus0);
24     arch.weld(fireAnalyzerComp, EventBus0);
25     arch.weld(alarmLoggerComp, EventBus0);
26     ...
27     arch.start();
28   }
29 }
```

Listing 1: Code snippet of the automatically synthesized MIDAS application based on Prism-MW

3.3 Experiment: MIDAS, REST, Restlet

This section introduces REST, explains the mechanisms that RESTful applications use, and touches on the Restlet [2] architectural framework.

3.3.1 REST and Restlets

Representational State Transfer (REST) is an example of an architectural style derived from a successful technology: the World Wide Web [12]. REST is style for distributed systems: from applications on embedded systems to large enterprise systems. Rather than focusing on details of components, REST emphasises the roles of components and the communication between them. The main element of REST is called *resource*. The information in a resource is called state, and method invocations are performed by transferring state representations.

Similar to the other architectural styles, REST consists of a set of rules. Applications complying with the rules of the REST styles are called *RESTful*. The following assert briefly the principle rules of REST [12, 37]. A Resource is the fundamental abstraction of information, and is identified by a resource identifier. A resource representation is a sequence of bytes along with metadata explaining those bytes. Interactions are stateless: each method invocation contains all the required information without any dependency on previous messages. Resources are manipulated using a few primitive operations defined over representations. In other words, information are accessed via a fixed set of interfaces, and components communicate by transferring representations through these interfaces rather than operating directly upon resources. More precisely, instead of adding particular interfaces to the architecture, new resources add new pieces of information to be manipulated using generic operations. For the purpose of caching and representation reuse, idempotent operations and representation metadata are of utmost importance. Finally, Intermediary components, such as *proxy* and *gateway*, handle requests and responses transparently, based on their representations and corresponding metadata, for filtering or redirection purposes among others.

```
1  module REST
2
3  sig Entity{}
4
5  sig MediaType{}
6
7  sig Representation extends Entity{
8    metadata: MediaType
9  }
10
11 sig Identifier extends Entity{
12   identifies: one Resource
13 }
14
15 sig Resource{
16   reps: set Representation,
17   state: needHandle
18 }
19
20 abstract sig Component{
21   connectors: set Connector
22 }
23 sig OriginServer extends Component{
24   contains: set Resource
25 }
26 sig UserAgent extends Component{}
27 sig Proxy extends Component{}{...}
28 sig Gateway extends Component{}{...}
29
30 abstract sig Connector{
31   connects: set Connector
32 }
33 sig Client extends Connector{}
34 sig Server extends Connector{}
35 sig Server_Cache extends Server{}
36 sig Client_Cache extends Client{}
37 sig Resolver extends Connector{}
38 sig Tunnel extends Connector{}
39
40 ...
```

Listing 2: part of REST style described in Alloy

A key contribution of this paper is evidence in the form of this case study suggesting that it is practical to formalize such modern industrial architectural standards so that they can be employed within our synthesis framework. Listing 2 (eliding details) demonstrates this claim for the REST style using Alloy. *Entity* is a basic type consisting of a sequence of bytes. *Representation* and *Identifier* inherit from the Entity signature. The representation of a resource is an Entity plus *metadata* describing it. An Identifier is also an Entity uniquely identifying a Resource. Each *Resource* has a set of representations. The state is defined by an encapsulated *needHandle* object which is an abstract signature that the other objects, such as sensors, extend. Each *Component* handling a set of resources has a set of *Connectors*. REST components are specified by roles. *OriginServer* is the container of resources. *UserAgant* using *Client* connector initiates a request. *Proxy* is an intermediary component providing interface encapsulation of other services such as security for UserAgents. On the contrary, *Gateway* intermediate components provide interface encapsulation for OriginServers.

The remaining parts of the module specify connector types. *Connectors* in the REST style are like ports in Component-Connector based styles. They provide an abstract interface for component communication. *Client* and *Server* are the key types of Connectors. Another connector type is *Cache*, which itself could be a server connector or a client connector. A *Resolver* is in charge of translating resource identifiers into the network address, e.g. DNS. *Tunnel* is the final connector type that relays communication over a connection boundary [12].

3.3.2 Architectural Map

We now exhibit (in elided form) the main result of this work: an Alloy implementation of an architectural map, $map_{(SCC,REST)}$, that takes SCC application models to architectural models in the REST style. An architectural style specification defines the *co-domain* of an architectural map. We represent a map, such as $map_{(SCC,REST)}$, as a set of predicates that define the relationships required to hold between an application description and computed architectural descriptions.

```
1  module SCC_REST
2
3  open SCC
4  open REST
5
6  pred mapping(){
7    all n:needHandle | one r: Resource| r.state = n
8    all r:Resource | one c: OriginServer| r in c.
         contains
9
10   all s:Sensor| one cConnector:Client| cConnector
         in s.~state.~contains.connectors
11   all a:Actuator| one sConnector:Server|
         sConnector in a.~state.~contains.connectors
12   all c:Controller| one sConnector:Server|
         sConnector in c.~state.~contains.connectors
13   all c:Controller| (#c.actuators>0) => {
14     one cConnector:Client| cConnector in c.~state
           .~contains.connectors }
15
16   all c:Sensor| (c.~sensors.~state.~contains.
         connectors & Server) in
17   (c.~state.~contains.connectors & Client).^
         connects
18
19   all c:Controller| (c.actuators.~state.~contains.
         connectors & Server) in
20     (c.~state.~contains.connectors & Client).^
           connects
21 ...
22 }
```

Listing 3: The (elided) mapping predicate for the SCC application type and the REST architectural style

Listing 3 presents the elided $map_{(SCC,REST)}$ Alloy predicate. It states that there is a *Resource* for each sensor, actuator and controller. Line 8 assigns a component of type *OriginServer* to each resource. Each component handling a sensor resource has a *Client* connector to post its latest state to the controllers. According to lines 11–15, each component handling either an actuator or a controller resource has a connector of type *Server* to receive the latest state from the corresponding sensors. Provided that having connected to an actuator, each component handling a Controller resource has also Client connector. Lines 16–20 state that *Client* connector of each Sensor's component is connected to *Server* connectors of the corresponding Controllers' components, and *Client* connector of each Controller's component is eventually connected to *Server* connectors of the corresponding Actuators' components.

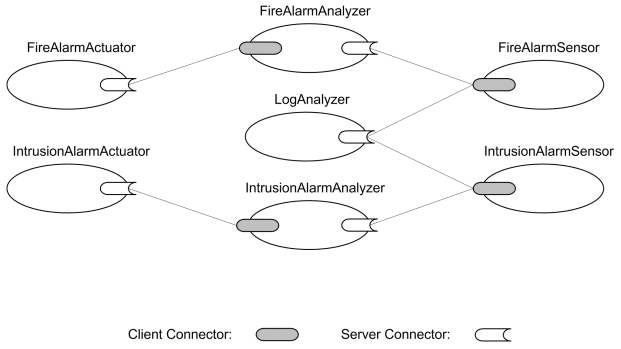

Figure 5: Process view of a REST-based architecture for an example of MIDAS application

Using the Alloy Analyzer, Monarch computes architectural models, represented as satisfying solutions to the constraints of the architectural map applied to the application model. Alloy guarantees that computed outputs *conform* to the REST architectural style. The mapping predicates also ensure that the resulting architectural models *refine* the given application model. Figure 5 represents the process view [30] of a REST-based architecture for our example of MIDAS application. The result in Alloy XML-format was computed automatically, but redrawn manually for presentation. To simplify the diagram, we omit some details.

3.3.3 Implementing Architectures in Restful Framework

Numerous architectural frameworks have been proposed for the REST style, some of which have found their way into industrial practice. *Restlet* is one. In contrast to Prism-MW, which was rooted in an academic project, Restlet is an industrial product. It supports the REST architectural style with an open source Java framework. As such, an architectural model in REST style can be mapped comfortably to an executable program based on the Restlet framework.

Like to other architectural frameworks, it supports architectural abstractions that provide a means of direct mapping between architectures and implementations.

The synthesized class for each resource extends the *Resource* base class. Each resource class implements the required REST methods, i.e. GET, POST, PUT and DELETE. Support of POST and PUT requests is indicated by the implementation of the methods *acceptRepresentation(Representation)* and *storeRepresentation(Representation)* respectively, which provide the facilities to process the posted entity. Similarly, the implementation of the *storeRepresentation(Representation)* method indicates the support for GET method.

According to the generated architectural description, the Client connector associated to the component of each sensor resource is connected to the Server connector of the components of its corresponding controllers. As such, each controller's resource supports receiving of the latest alarm message through an explicit call of its PUT method invoked by its corresponding sensors. That is, each sensor calls explicitly the method of its associated Controllers as a new alarm fires. This allows the recipient to perform the required detection analysis and to call the PUT method of the corresponding actuators if required. All the resources furthermore responds to GET requests with a representation that lists the alarm messages.

Figure 6: Deployment of automatically generated programs in Android platform

Our hand-written model-based code synthesizer generates resource identifiers (URIs) automatically based on the name of a resource's host machine and name. Host name management is handled manually. By assigning URIs to resource classes, each incoming call is handled by a dedicated instance of each resource class. Figure 6 shows the deployment of automatically synthesized programs to the Restlet framework running on a distributed system of Android cell phones. To test the synthesized application framework code, we synthesized sufficient test driver code to see the application actually run. It works.

3.4 Discussion

Malek has suggested that the lack of support by traditional middleware platforms for architectural abstractions leads to uncertainty about the consistency of implemented systems with their software architectures, making such traditional platforms a poor fit to architecture-driven development [26, 27]. This paper shows by contrast that *architectural middleware* can support effective, correct-by-construction synthesis of both architectures and corresponding implementations. Moreover, our work supports reasonable extension for new platforms and mappings. To that end, one can pick a platform, identify and formalizes the style it requires, and specify the map required to support high-assurance architecture and code synthesis for a range of architectural styles.

It is also important to acknowledge that, in practice, a system rarely completely conforms to a single style: different parts conform to different styles. Although in this work we focus on illustrating how to map a common application model to different styles and thereby to different platforms, there are other research studies which focus on that issue. As a case in point, Wong et al. [38] proposed an approach based on the Alloy language for modeling and verification of the complex systems that exploit multi-style structures. In an implementation, each subsystem can be considered as a separate *architecture* in the context of middleware platforms: a system is implemented with a number of *architecture* objects that may be comply with different architectural styles. We plan to extend our work to multi-style systems and do not foresee great difficulty in doing so.

Overall, we view this paper as providing significant evidence in support of the proposition that a formal, architecturally correct-by-construction approach to synthesis of both architectural models and executable code for architectural frameworks and middleware is a promising approach. This is the primary intellectual contribution of this work. All of the software assets needed for an independent verification of our claims are available on the World Wide Web.

4. RELATED WORK

We identify four areas of research that are most related to our work. The first area concerns research dealing with middleware-induced architectural styles. The second encompasses all the researches on filling the gap between architectures and implementations. The next area is on formal approaches to model transformation. The last is our own earlier work on *architectural mappings*.

Considering the first category, the notion that an application framework induces one or more architectural styles and that systems using an application framework are required to conform to those styles in order to both take the advantages of the framework and avoid unintentional mismatch side-effects was first considered by Di Nitto and Rosenblum [29]. The basic ideas are replicated in numerous publications.

Gall et al. [14], considering the correspondence between component technologies and software architectures, identified the architectural style (ComPAS) induced by component programming, and claimed that it can be used in developing software systems. Baresi et al. [7] argued that constraints imposed by a certain choice of platform could be considered as an architectural style. They focused on service-oriented architectures, and defined refinement rules based on graph transformations for checking the consistency of an application with constraints implied by the styles to which it conforms. We share with these researches the problem domain, namely architectural styles imposed by middleware platforms, but our focus is on developing end-to-end transformations that map architecture-independent application models to executable program implementations in selected platform-induced architectural styles.

Sousa and Garlan [32] using the Wright ADL, modeled

structural and behavioral constraints imposed by the Enterprise JavaBeans framework. Sullivan et al. [34] similarly used Z to model and analyze composition mechanisms provided by Microsoft's Component Object Model (COM). These works use formal modeling and verification for detecting architectural mismatch [16], whereas we focus on preventing such mismatches in the first place with a correct-by-construction synthesis approach.

Giesecke et al. [19] discussed the styles imposed by web-based middleware frameworks, such as Spring and Apache Cocoon, but they did not present the model or results in detail. It is unclear how far this work has progressed.

A notable attempt to bridging the gap between software architecture and its correspondence implementation is ArchJava [3], an extension to Java to provide capabilities of architecture description languages within a programming language. It can verify conformance of an implementation to the architectural constraints. However, it does not explicitly support architectural styles or application synthesis.

Kruger and Mathew [24] proposed an approach to develop software implementations in service-oriented architecture. Using aspect-oriented programming [23], they provided a mapping from their service-enabled architecture description language (ADL) to the implementation of the architecture. In fact, they realize the system by assigning a Java class to each role and component type of the architecture. Services are then specified as aspects that relate components and the roles they implement. Although their work has partially common intent with ours on mapping architectural models to implementations conforming to the constraints of given styles, our work introduces a new concept of application type along with the architectural mappings that specify the refinement rules based on not only architectural style, but also on application type of system.

Medvidovic et al. [28] focused on relating the modeling facilities provided by software architectures and the implementation abilities of the application frameworks and proposed an approach to using middleware to implement architectural connectors. Malek et al. [26] extended the same line of work by proposing that using application frameworks supporting architectural styles is a crucial approach to bridging the gap between concepts in architecture descriptions and the system implementation. They devised a mature architectural middleware infrastructure (Prism-MW) for embedded systems, supporting architectural styles. Our approach builds on such architectural frameworks but we have focused on end-to-end mapping of architecture-independent application models into realized applications conforming to architectural styles.

Regarding the third area of related work, Tamzalit and Mens [35] recently proposed an approach for the evolution of an architecture description under the guidance of architectural style, rooted in the idea that the evolution follows certain common patterns, which they identified as *architectural evolution patterns*. To this extent, they rely on a formalism based on graph transformation. This work shares with ours an emphasis on applying architectural styles, but our work focuses on formal mappings of architecture-independent application models to a diversity of realized architectural models in practical architectural styles.

The other similar work is the recent work of Garlan et al. [17] 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. The approach they propose involves incremental steps between programs, each step being affected by the application of a well defined incremental *architectural operator*. The key ideas that remain implicit in Garlan et al., and which are the central focuses of our work, are (1) we are dealing with one or more architecture-independent application models, (2) architectural models are obtained from application models by way of architectural maps, and (3) making architectural maps explicit. Beyond just evolutionary transitions between pairs of architectural models, we focus on architectural style as a separate variable in design. That said, until one can rely entirely on models, viewing synthesized code as a detail that can be regenerated when needed, it will be important to consider requirements for architecture evolution. We plan to investigate the potential for our approach to assist in such tasks.

Bucchiarone and Galeotti [10] also proposed an approach based on graph grammars and *DynAlloy* [13] to verify programmed dynamic software architectures in which all possible architectural changes are defined before run-time. Like other work we have studied, this work lacks an explicit notion of mappings from application models to architectures via choices of styles. That said, this work appears to have the potential to help us extend application models to include richer semantics.

Another interesting approach to model transformation, though not directly relying on formal methods, is by Almeida et al [4]. They defined the notion of an abstract platform as an abstraction of platform attributes to be used in the role of meta-model for platform-independent models (PIMs) of an application. Their work stems from the observation that PIMs must be balanced against realization of PSMs. They argued that an abstract platform should specify characteristics that are required to be mapped into the concrete target platforms. In other words, they developed *platform-independent* application descriptions in the language of the *abstract platform*. By contrast, we view the platform (even the abstract platform) as the target to which we map platform and architecture-independent descriptions. Our work thus develops a notion of application type explicitly, while they do not, and in the first stage of synthesis takes architectural styles specifically as abstract platforms.

Finally, our earlier work [5] formalizes the concept of *architectural mappings*, and it shows that it is possible to separate, and combine formal representations of application contents and architectural styles respectively, and provide MBD tool support for architectural model synthesis for simple styles. Recent work [6] presenting a proof of concept supporting tool showed how the proposed separation of concerns can be placed within the formal framework of model-based development. The current work extends our notion of architectural mappings to platform-induced styles, and composes are initial synthesis of architectural models from application models with synthesis of platform code from architectural models to provide an end-to-end approach to synthesis of architecturally correct code from abstract application models. This work this provides a critical advance toward an industrially viable theory and technology.

5. CONCLUSION

In this paper, we proposed a formally specified approach

to end-to-end transformation of formal application models into architecturally correct code, averting the difficult problem of mapping application models to diverse architectural styles and platforms with assurance of consistency between the code and the architecture. Our contribution builds on prior work in architectural maps, research studies dealing with middleware-induced architectural styles, and work on filling the gap between architectural model and executable implementations. As an experimental evaluation, we have mapped a common application model to two platforms supporting two styles: Prism-MW in an implicit invocation style, and Restlet in the REST style. We demonstrated the viability of this approach by among other things synthesizing an architecturally correct RESTful application running on a network of Android phones.

We identify three key goals for future work. First, we intend to integrate architectural analysis into our theory and tools. Second, once this is done, it will enable automated optimizing design space search over architectural variants not only within a style but across styles, yielding a new approach to finding appropriate styles and architectures for given applications. Work by Jackson et al. [21] suggests a promising approach that we intend to explore. Third, as part of our future work, we plan to extend our work to support *forgetful* back-mappings, i.e., abstraction from code back to architectural models and thence to application models. With such a technology in hand, combined with the forward mappings we have already demonstrated, we will have a technology for automated architectural evolution of modern applications to further address the needs we articulated at the beginning of this paper.

6. ACKNOWLEDGMENTS

This work was supported by a grant from the National Science Foundation.

7. REFERENCES

[1] Monarch tool suite. `http://monarch.cs.virginia.edu/`.

[2] Restlet. http://www.restlet.org/.

[3] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 187—197, Orlando, Florida, 2002. ACM.

[4] J. P. Almeida, R. Dijkman, M. van Sinderen, and L. F. Pires. On the notion of abstract platform in MDA development. In *Proceedings of Eighth IEEE International Enterprise Distributed Object Computing Conference*, pages 253—263. IEEE Computer Society, 2004.

[5] H. Bagheri, Y. Song, and K. Sullivan. Architectural style as an independent variable. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*, 2010.

[6] H. Bagheri and K. Sullivan. Monarch: Model-based development of software architectures. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (Models'10)*, 2010.

[7] L. Baresi, R. Heckel, S. Thöne, and D. Varŕo. Style-based modeling and refinement of service-oriented architectures. *Software and Systems Modeling*, 5(2):187–207, June 2006.

[8] G. S. Blair, L. Blair, V. Issarny, P. Tuma, and A. Zarras. The role of software architecture in constraining adaptation incomponent-based middleware platforms. In *IFIP/ACM International Conference on Distributed systems platforms*, pages 164–184, New York, New York, United States, 2000. Springer-Verlag New York, Inc.

[9] G. Booch. The accidental architecture. *IEEE Software*, 23(3):9—11, 2006.

[10] A. Bucchiarone and J. P. Galeotti. Dynamic software architectures verification using DynAlloy. In *Proceedings 7th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008)*, 2008.

[11] G. Edwards, C. Seo, and N. Medvidovic. Model interpreter frameworks: A foundation for the analysis of domain-specific software architectures. *Journal of Universal Computer Science*, 14(8):1182—1206, 2008.

[12] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. In *Proceedings of the 22nd international conference on Software engineering*, pages 407–416, Limerick, Ireland, 2000.

[13] M. F. Frias, J. P. Galeotti, C. G. L. Pombo, and N. M. Aguirre. DynAlloy: upgrading alloy with actions. In *Proceedings of the 27th international conference on Software engineering*, pages 442–451, St. Louis, MO, USA, 2005.

[14] H. Gall, M. Jazayeri, R. Kloesch, and G. Trausmuth. The architectural style of component programming. In *21st International Computer Software and Applications Conference (COMPSAC '97)*, pages 18—25, Los Alamitos, CA, USA, 1997. IEEE Computer Society.

[15] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[16] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is still so hard. *IEEE Software*, 26(4):66—69, 2009.

[17] D. Garlan, J. M. Barnes, B. Schmerl, and O. Celiku. Evolution styles: Foundations and tool support for software architecture evolution. In *Joint 8th Working International Conference on Software Architecture and 3rd European Conference on Software Architecture*, Cambridge, UK, Sept. 2009.

[18] S. Giesecke and J. Bornhold. Style-based architectural analysis for migrating a web-based regional trade information system. In *First International Workshop on Web Maintenance and Reengineering (WMR 2006) in conj. with CSMR 2006*, volume 193, pages 15—23, Bari, Italy, 2006. CEUR Workshop Proceedings.

[19] S. Giesecke, J. Bornhold, and W. Hasselbring. Middleware-Induced architectural style modelling for architecture exploration. In *Working IEEE/IFIP Conference on Software Architecture*, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[20] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[21] E. K. Jackson, W. Schulte, and J. Sztipanovits. The

power of rich syntax for model-based development - microsoft research. http://research.microsoft.com/apps/pubs/default.aspx?id=77374, 2008.

[22] P. Kelsen and Q. Ma. A lightweight approach for defining the formal semantics of a modeling language. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 690–704, Toulouse, France, 2008. Springer-Verlag.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented programming. In *ECOOP'97 âĂŤ Object-Oriented Programming*, pages 220—242. 1997.

[24] I. H. Krüger and R. Mathew. Systematic development and exploration of Service-Oriented software architectures. In *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture*, pages 177—187. IEEE Computer Society, 2004.

[25] A. Lédeczi, A. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific design environments. *Computer*, 34(11):44–51, 2001.

[26] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A Style-Aware architectural middleware for Resource-Constrained, distributed systems. *IEEE Trans. Softw. Eng.*, 31(3):256–272, 2005.

[27] S. Malek, C. Seo, S. Ravula, B. Petrus, and N. Medvidovic. Reconceptualizing a family of heterogeneous embedded systems via explicit architectural support. In *Proceedings of the 29th international conference on Software Engineering*, pages 591—601. IEEE Computer Society, 2007.

[28] N. Medvidovic, E. M. Dashofy, and R. N. Taylor. The role of middleware in Architecture-Based software development. *International Journal of Software Engineering and Knowledge Engineering*, 13(4):367—393, 2003.

[29] E. D. Nitto and D. Rosenblum. Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In *Proceedings of the 21st international conference on Software engineering*, pages 13—22, Los Angeles, California, United States, 1999. ACM.

[30] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[31] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng.*, 21(4):314–335, 1995.

[32] J. P. Sousa and D. Garlan. Formal modeling of the enterprise JavaBeans(TM) component integration framework. *Information and Software Technology*, 43(3):171—188, Mar. 2001.

[33] J. P. Sousa and D. Garlan. Aura: An architectural framework for user mobility in ubiquitous computing environments. *In Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture*, pages 29—43, 2002.

[34] K. J. Sullivan, M. Marchukov, and J. Socha. Analysis of a conflict between aggregation and interface negotiation in microsoft's component object model. *IEEE Transactions on Software Engineering*, 25(4):584—599, 1999.

[35] D. Tamzalit and T. Mens. Guiding architectural restructuring through architectural styles. In *17th IEEE International Conference and Workshops on Engineering of Computer Based Systems*, pages 69—78, 2010.

[36] R. N. Taylor, N. Medvidovic, K. M. Anderson, J. E. J. Whitehead, and J. E. Robbins. A component- and message-based architectural style for GUI software. In *Proceedings of the 17th international conference on Software engineering*, pages 295—304, Seattle, Washington, United States, 1995. ACM.

[37] R. N. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[38] S. Wong, J. Sun, I. Warren, and J. Sun. A scalable approach to multi-style architectural modeling and verification. In *Proceedings of the 13th IEEE International Conference on on Engineering of Complex Computer Systems*, pages 25–34. IEEE Computer Society, 2008.