Formal Aspects of Computing

Model-Driven Synthesis of Formally Precise, Stylized Software Architectures

Hamid Bagheri[1] and Kevin Sullivan[2]

[1]School of Information and Computer Sciences, University of California, Irvine
[2]Department of Computer Science, University of Virginia

Abstract. Reliably producing software architectures in selected architectural styles requires significant expertise yet remains difficult and error-prone. Our research goals are to better understand the nature of style-specific architectures, and relieve architects of the need to produce such architectures by hand. To achieve our goals, this paper introduces a formally precise approach to separate architectural style design decisions from application-specific decisions, and then uses these separate decisions as inputs to an automated synthesizer. This in effect supports a model-driven development (MDD) approach to architecture synthesis with style as a separate design variable. We claim that it is possible to formalize this separation of concerns, long implicit in software engineering research; to automatically synthesize style-specific architectures; and thereby to improve software design productivity and quality. To test these claims, we employed a combination of experimental systems and case study methods: we developed an MDD tool and used it to carry out case studies using Kitchenham's methods. Our contributions include: a theoretical framework formalizing our separation of concerns and synthesis approach; an MDD framework, Monarch; and results of case studies that we interpret as supporting our claims. This work advances our understanding of software architectural style as a formal refinement; makes application descriptions an explicit subject of study; and suggests that synthesis of architectures can improve software productivity and quality.

Keywords: Software Architecture; SAT-based Synthesis; Architectural Styles; MDD; Alloy Language.

1. Introduction

Software architecture is essential for managing complexity and meeting demanding requirements in developing complex software systems [SG96, TMD09]. Architectural styles systematize successful architectural designs in terms of constraints on architectural elements and their composition into systems [SG96].

Correspondence and offprint requests to: Hamid Bagheri, School of Information and Computer Sciences, University of California, Irvine, CA, 92612. E-mail: hamidb@uci.edu.

Developing a sound and appropriate architecture, however, remains a significant and intellectually challenging activity. To develop architectures effectively one must understand both the application domain in question and the discipline of software architecture. These bodies of knowledge are typically held by different people. Domain experts better understand requirements, while architects understand architectural styles, their implications, and techniques for mapping application requirements models to architectures in given styles.

The required communication and coordination between these roles, and the manual mapping of application requirements to architectures, are costly and error-prone activities. What is needed is a reliable, automated approach to mapping system models, expressed in terms closer to those of the domain expert, to software architectures in selected architectural styles.

This paper presents such a capability: a formal, model-driven approach to synthesis of architectures based on abstract application models and choices of architectural styles. We contribute a model-driven approach to modeling application properties abstracted from choices of specific architectural styles, and a formal and automated approach to mapping such models to architectures in given styles. Given an application model and a choice of style, our formal synthesis procedure generates architectures in that style in a standard architectural description language.

An important research challenge is that while we already have a theory of architectural styles [SG96, PW92], 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 in ways that admit precise definition of the *mappings* that combine application and architectural style choices to yield application realizations in given styles. Rather, the separation is informal and these mappings are typically learned by example and experience, and remain the domain of architectural *gurus*.

As an intellectual contribution, this paper introduces a formally precise approach to separate architectural style design decisions from application-specific decisions, and then uses these separate decisions as inputs to an automated synthesizer. The key to this separation is a means of reconciliation—an '*architectural map*'—connecting application descriptions to realizations in particular architectural styles. The study of architectural maps complements past work on architectural styles with new attention to how such styles combine with application models to yield architectures.

We develop a framework, called *Monarch*[1], that automates these architectural maps. Monarch supports meta-models for a range of application types, the graphical development of models of these types, formal synthesis of software architectures for chosen architectural styles, and representation of generated architectures in standard architecture description languages, including ACME [GMW00]. It uses *Alloy* as a back-end specification language [Jac02], and the Alloy Analyzer as the analysis engine. Alloy is a formal specification language based on set theory, optimized for automated analysis.

To evaluate our approach, we have conducted a set of case studies that formally model and replicate several published informal results in the architecture literature. Our results are either consistent with those previously derived informally, or reveal certain inconsistencies in earlier arguments. The data we have produced support the proposition that our theory and tool properly express and implement the key elements of previously informal and intuitively understood architectural maps.

To summarize, this paper makes three contributions:

- *Theoretical framework:* We develop a theoretical framework to make the notion of architectural maps precise (§ 2).
- *Model-driven tool implementation:* We show how to exploit the power of our formal abstractions by building an MDD framework, *Monarch* [Mon15], for formal model-driven development of software architecture (§ 3).
- *Experiments:* We present results from experiments run on over 10 case studies—formally replicating, among other things, prominent earlier architectural studies from the literature—corroborating Monarch's ability in correct-by-construction synthesis of style-specific architectures in the order of seconds (§ 5).

The remainder of the paper is organized as follows. Section 2 introduces a theoretical framework. Section 3 details the *Monarch* framework we have developed to support automated development of software architecture. In Section 4, we use a well-known and widely used example of the Lunar Lander application [TMD09] to demonstrate the key steps in the mapping process. Sections 5 and 6 then present the evaluation of the

[1] The name is intended to abbreviate **Mo**del a**n**d **Arch**itecture.

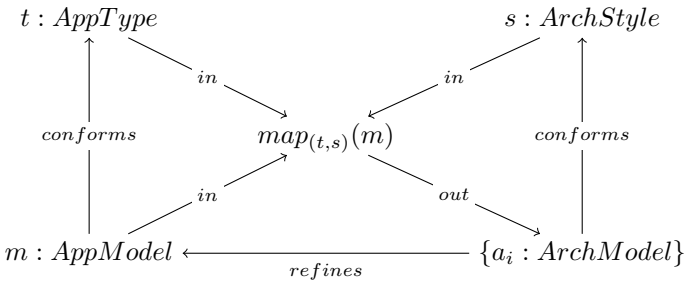

Fig. 1. Key entities and relations in *architectural maps*.

research, the experimental approach, and the results. Section 7 shows that comparing our work with related efforts has the potential to improve both. Finally, Section 8 presents our conclusion and a discussion of future work.

2. Approach Overview

This Section presents an overview of the notion of *architectural maps*, and then presents the correspondence of the proposed notion and the formal structure of model-driven development.

2.1. Architectural Style as a Separate Design Decision

The success of architecture development, today, depends heavily on the experience of human architects, and this manual process is unreliable, costly and labor-intensive. As our understanding of architectures grows, we can systematize and eventually formalize and automate synthesis of architectures.

In this paper, we take a more robust approach to automate the costly and unreliable process of transforming abstract application models into software architectures. Our approach introduces the *application type*–a formal specification of a family of applications–as a source modeling language, and uses style specifications as target languages[2]. Given an application model expressed in a particular application type, our approach refines the application model into software architectures in a selected architectural style.

Formula 1 makes explicit and elaborates the notion that an *architectural map* (*ArchMap*) combines an application model, m, of a given application type, with a specification, s, of a given architectural style, to produce a set of architectural models, $\{a_i\}$, for application m in style s. These architectural models refine the application model while complying with the rules implied by the architectural style.

$$\{a_i : ArchModel\} = ArchMap_{(t:AppType, s:ArchStyle)}(m : AppModel) \quad (1)$$

ArchMap captures architectural knowledge that we seek to formalize and automate. The study of *ArchMap* balances attention to architectural styles, with attention to how such styles combine with application descriptions to yield architectures. Knowledge of this mapping is crucial to expertise in software design. Given an application description, 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 $ArchMap$ is a complicated object. In some sense, it embodies knowledge of how to realize different types of applications in different styles. We need a way to study it in pieces. We decompose $ArchMap$ by treating it as a function polymorphic in both application type and architectural style. We then investigate it for specific pairs. This study requires to make explicit a notion of *application type*. Application descriptions come in a variety forms. Examples include *composition of functions* (which is in essence how Parnas characterized KWIC [Par72], for example), or *state machine*, or *sense-compute-control* [TMD09]. Each of these *application types* provides a vocabulary and structuring mechanisms for organizing application descriptions

[2] Architectural styles [SG96] are the results of earlier efforts to systematize successful architectures in terms of constraints on architectural elements and their patterns of composition.

Table 1. Correspondence between terms of Architectural maps and the formal structure of MDA.

Model-driven Development	Our Theory
Meta-model	Application type
Platform Independent Model (PIM)	Application Model
Model Transformation	Architectural Mapping
Platform Definition Model (PDM)	Architectural Style
Platform Specific Model (PSM)	application- and style-specific Architecture

prior to the choice of architectural style for the system implementation. An architectural map in essence converts the structure and content of such a description into a form consistent with a given architectural style choice. Specifically, we view *ArchMap* as parameterized by type (*AppType*) and style (*ArchStyle*), and develop separate mappings for each compatible *AppType*/*ArchStyle* pair. Compatibility captures the idea that not every architectural style is appropriate for every application type.

To make the idea precise, Figure 1 represents the fundamental elements of this model and their relationships: (1) $\{a_i\}$, a set of architectural models (architectures) derived by the processes we describe in this paper; (2) s, an architectural style specification; (3) *conforms*, a relation encoding the conformance of an architectural model, a_i, to an architectural style, s; (4) m, an application model; (5) t, an application type; (6) a (second) *conforms* relation, encoding the conformance of m to t; (7) $map_{(t,s)}$, a map parameterized by t and s that takes application model, m, to the set of architectural models, $\{a_i\}$; (8) a *refines* relation encoding the property that each such a_i refines the application model, m. Given input parameters, t, s, and m, our map yields a set of architectures, as, in general, multiple architectures in a given style satisfy the required conformance and refinement constraints.

With these terms in hand, we can now say more precisely what we mean by *architectural style as a separate design variable.* For a given application model and type, one can select among compatible styles and maps and automatically synthesize architectures in these styles. To the extent that the essence of modularity is a decoupling of design parameters, the approach realizes a new form of modularity: *it modularizes architectural style.*

2.2. Model-driven Automation

As software systems become larger and more complex, there is an ever greater need to employ higher levels of abstraction in application development. Model-driven development is centered around abstract, domain-specific models and transformations of abstract models into the constructs of specific underlying platforms. To be more precise, according to the specification for OMG's Model Driven Architecture (MDA)[3] [MM03], it is rooted in a mapping that takes a *platform-independent model* (PIM), p, and a *platform definition model* (PDM), s, to a *platform-specific model* (PSM), i:

$$i : PSM = map(p : PIM, s : PDM) \tag{2}$$

The analogy between two approaches is clear in the equations 1 and 2. Architectural styles plays the same role as platform descriptions in an MDA approach [Sch06]. We believe that this observation opens a path to model-driven tools that support architectural style as a separate variable in automated development of software architectures.

Table 1 shows the correspondence between terms of *Architectural maps* and the formal structure of MDA. We introduce *application types* as styles of application description that play the same role as application meta-models in MDA. Application models correspond to platform independent models in MDA; architectural maps, to MDA transformations; architectural styles, to MDA platforms; and synthesized software architectural models, to platform-specific models in MDA.

[3] Model Driven Architecture (MDA) is a registered trademark of the Object Management Group (OMG) for model driven development.

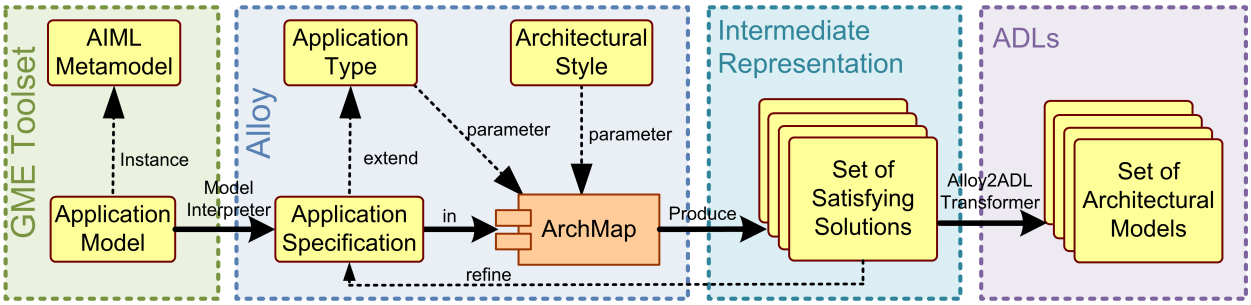

Fig. 2. High-level overview of the Monarch approach.

Given the concepts of application *type* and architectural *style*, we can now concisely describe the approach. A user selects an application type. This type selects a meta-model that parameterizes a model-based editing tool. Within such a tool one creates an application model as an instance of the selected application type. One then selects an architectural style. The combination of an application type and an architectural style selects the specification of a synthesis function: an *architectural map* for that particular pair of input specifications. Each architectural map specifies the mapping of any application model (instance) of the given application type to architectures in the given style. Application types, architectural styles, and architectural maps are all formally specified in a notation that supports automated analysis and synthesis. The approach involves the synthesis of an architectural description that satisfies the constraints of both type and style specifications, and that does so in a particular manner described by architectural mapping constraints.

3. The Monarch Framework

This section shows how the ideas for automated development of software architecture can be realized in practice. Figure 2 outlines the high-level overview of our approach. The Monarch framework comprises (1) an approach to architecture-independent application modeling using the Generic Modeling Environment (GME) [LBM+01], with application types realized concretely as GME architecture-independent modeling language (AIML) meta-models; (2) interpreters that transform application models, viewed as *concrete instances* of architecture-independent modeling languages, into application specification Alloy modules that *extend* the corresponding application types; (3) a mapping engine, based on the Alloy Analyzer, that takes such an application specification (represented in the Alloy language) and a formal specification of an architectural style and that finds architectural models that *refine* the application model in conformance with the given architectural style. The mapping employed (*ArchMap*) is based on the combination of the selected application type and the selected architectural style. The engine uses the Alloy constraint solver to compute intermediate architectural models, represented as satisfying Alloy solutions; (4) a final post-processing phase, *Alloy2ADL*, translates the resulting Alloy instances into human-readable architectural description languages (ADLs).

In the rest of this section, we first provide a brief overview of Alloy, and then describe each of the aforementioned modules of the Monarch framework.

3.1. Alloy Overview

Alloy is a lightweight formal specification language based on the first-order relational logic with transitive closure [Jac02, Jac12]. 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 better validate our claims because we use, as inputs, architectural style specifications, in Alloy, that others have published [KG10, WSWS08]. Reusing published models is important in that it shows our ideas and approach to be consistent with contemporary formal accounts of architectural style.

Essential data types, that collectively define the vocabulary of a model, are specified in Alloy by their type signatures (sig). Signatures represent basic types of elements, and the relationships between them are

captured by the declarations of *field*s within the definition of each signature. A signature declaration may also include a signature fact constraining elements of the signature. A signature declaration can also extend another signature. Signatures defined as abstract represent types of elements that cannot have an instance object without explicitly extending them.

The other essential constructs of the Alloy language include: *Facts*, *Predicates*, *Functions* and *Assertions*. Facts (fact) are formulas that take no arguments, and define constraints that must always hold. Predicates (pred) are named logical formulas used in defining parameterized and reusable constraints that are always evaluated to be either true or false. Functions (fun) are parameterized expressions. A function similar to a predicate can be invoked by instantiating its parameter, but what it returns is a relational value instead. An assertion (assert) is a formula required to be proved. It can be used to look for counterexamples of conjectures.

The Alloy language comes with a set of logical and relational operators. The dot (.) and tilde ($\sim$) relational operators denote a relational join of two relations and the transpose operation over a binary relation, respectively. The transitive closure ($\hat{}$) of a relation is the smallest enclosing relation that is transitive. The reflexive-transitive closure ($*$) of a relation is the smallest enclosing relation that is both transitive and reflexive.

The Alloy Analyzer is a constraint solver that supports automatic analysis of models written in Alloy. The analysis process is based on a translation of Alloy specifications into a Boolean formula in conjunctive normal form (CNF), which is then analyzed using off-the-shelf SAT solvers. With respect to the constraints in a given model, the Alloy Analyzer can be used either to find solutions satisfying them, or to generate counterexamples violating them. The Alloy Analyzer is a bounded checker, so a certain scope of instances needs to be specified. In the matter of architectural styles, the scope states the number of architectural elements of each type. The analysis is thus performed through exhaustive search for satisfying instances within the specified scopes. To take advantage of partial models, the latest version of the analyzer uses *KodKod* [Tor09] as its constraint solver so that it can support incremental analysis of models as they are constructed. The generated instances are then visualized in different formats such as graph, tree representation or XML. We use the Alloy Analyzer to compute architectural models given the conjunction of an architecture-independent model represented using a particular meta-model, and a choice of formal specifications of an architectural style, also represented in Alloy. We will introduce additional details of the Alloy language as necessary to present our model-driven architecture synthesis approach. For further information about Alloy, we refer the interested reader to [Jac12].

3.2. AIML Meta-model

To facilitate architecture-independent application modeling, we realize application types as meta-models, the development of which is supported thoroughly by many meta-modeling environments, e.g. Generic Eclipse Modeling System [WSNW07], MetaEdit+ [Met10] and Generic Modeling Environment [LBM+01]. We have developed Architecture-Independent Modeling Languages (AIMLs) on top of the GME to support the specification of application content at the abstract modeling level. The reasons for choosing GME for this study include its straightforward mechanisms for developing extensions, and its availability and proven success for MDD.

A meta-model specification describes a particular form of model. In our earlier work [BSS10, BS10b], we identified several possible forms of architecture-independent model, including *composition of functions*, *aspect-enabled composition-of-functions (ACF)*, *state-driven behavior (SD)* and *sense-compute-control (SCC)*. Our meta-model for the *sense-compute-control (SCC)* application type is shown in Figure 3a. We have developed GME meta-models for several previously identified but not well elaborated application types: *composition-of-functions (CF)*, *aspect-enabled composition-of-functions(ACF)* (Figure 3b), and *state-driven behavior (SD)* (Figure 3c). For brevity, and because it suffices to make our points, we describe only the *SCC* meta-model in this work. Monarch supports the others as well.

We view sense-compute-control (SCC) as an *application type* for embedded control systems. The SCC application type is used to model applications in which a set of sensors and actuators are connected to controllers that cycle through the steps of fetching sensors values, executing a set of functions, and emitting outputs to the actuators [TMD09]. Figure 3a shows a UML class diagram for the SCC meta-model as represented in GME. The *ApplicationDescription* class is common among application-type meta-models, and represents the top level container class, required by the MetaGME interpreter to facilitate modeling of all other application elements. The *Controller*, *Sensor* and *Actuator* classes represent three main elements

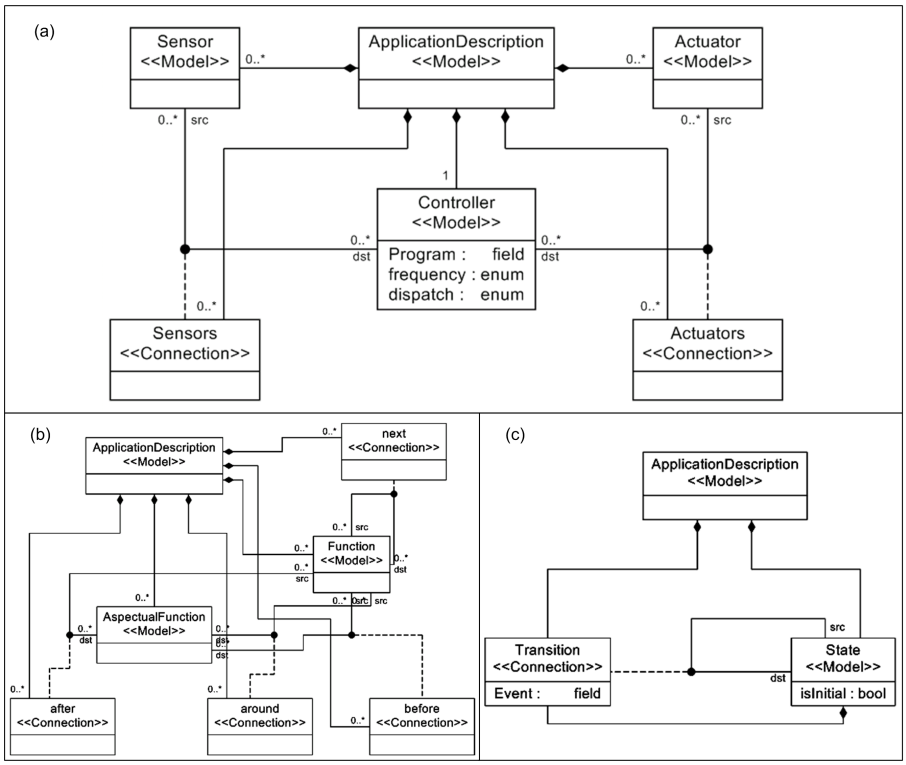

Fig. 3. Meta-models in GME for application types: a) Sense-Compute-Control b) Aspect-enabled Composition-of-Functions c) State-Driven behavior

of the SCC application type. The Controller's frequency is abstracted into discrete ranges of *slow* and *fast*. The Controller can also be specified as using a *periodic*, *aperiodic* or *sporadic* task.

Given such a meta-model specification of a modeling language for each architectural-style-independent application type, GME automatically creates an architectural-style-independent modeling environment. The designer of a system then specifies an application description as a model using the modeling environment. This approach promises to allow domain experts to model their applications abstracted from details of software architectural styles.

3.3. Transformation to Alloy Specification

Monarch translates an application model specified as a concrete instance of such an architecture-independent modeling language (AIML) for a given application type into an Alloy module. The GME provides several ways to process data from the model automatically. We used the Java version of the Builder Object Network (BON), providing us access to the internal representation of the model through Java objects. The BON Java interface mirrors the structure of the models, where each element, such as a *Model* or a *Connection*, has a corresponding builder class. In that way, the BON interface provides support for traversal of the models along the containment hierarchy.

We developed a *GME interpreter* for each meta-model as a syntax mapping, which elicits the model elements and transforms the constructs of the application model to formal specifications in the Alloy language. Specifically, for each meta-element, our interpreter contains a specific builder class that extends a corresponding general-purpose builder class provided by GME. For example, our builder class for *Controller*, tagged with the *Model* stereotype, extends the *JBuilderModel* class, and the builder class developed for *Sensors* of type *Connection*, extends the *JBuilderConnection* class.

The interpreter transforms elements of type *Model* into a singleton signature definition that represents the inheritance of the given element from its corresponding signature definition. The following code snippet

shows the Alloy representation of a Controller element, where the value for the *ControllerName* is captured from the application model (developed in the GME environment). The element's properties specified in the model are then transformed into signature facts (lines 2–4). If no value is specified for a field, it will be assigned as none. The *Connection* elements are also transformed into signature facts such that they specify the exact values of their corresponding signature field (lines 5–6).

```
1 one sig <ControllerName> extends Controller{}{
2   controller_dispatch_state = <dispatchValue>
3   frequency_state = <frequencyValue>
4   program = <programValue>
5   sensors = <Sensors>
6   actuators = <Actuators>
7 }
```

Given an application model, the interpreter thus creates an Alloy module that contains a signature definition for each *Model* element in the application model and additional sets of facts corresponding to the properties of those elements. In doing so, the interpreter starts from the root folder of the model, and then accesses all builder objects corresponding to the root model through recursive traversal of the children of model builders.

3.4. Architectural Mapping

After modeling application content, in this section we show how we use the Alloy Analyzer to enable automated formal derivation of software architectures. We first define the notions of conformance and refinement.

Definition 1 (conformance). Let S be a set of first-order sentences[4], we say that model m conforms to S, if m satisfies each sentence of S.

$$conform(m, S) \quad \equiv \quad \forall\, s_i \ \in S \,.\, m \ \models \ s_i \tag{3}$$

Definition 2 (architectural refinement). Let a be an architectural model, m be an application model, and T be a set of first-order sentences which model m conforms to, or $conform(m, T)$. We say that a refines m with respect to T, if and only if a would also conform to T, or $conform(a, T)$.

To ensure conformance of derived architectural models to the target architectural styles, an architectural style description specifies the *co-domain* of an architectural map that drives the architecture synthesis. The result of an architectural map, $ArchMap_{(T,S)}(m)$, thus is a set of architectural models that refine the input application model, m, while conforming to the rules implied by the architectural style, S. More precisely,

$$ArchMap_{(T,S)}(m) = \{a_i : ArchModel \mid conform(m, T) \ \wedge \ conform(a_i, S) \ \wedge \ refine(a_i, m)\} \tag{4}$$

To accomplish this, four pieces of Alloy specifications are conjoined in the process of mapping an application model to architectural models in a given style: (1) an *application type* represented in an Alloy module; (2) an *application model,* comprising an instance of an application type, represented in an Alloy module; (3) an *architectural style* specification module; and (4) *mapping predicates*. These predicates take types of applications as parameters, and define relationships required to hold between applications of given types and computed architectural models in the given style. As an example, Listing 7 presents part of the predicate for mapping application models in the sense-compute-control application type to architectures in the implicit invocation style. We describe it in more detail in Section 4.3. The Alloy Analyzer then computes satisfying solutions to the conjoined specification, yielding the synthesized architectures.

Formalizing the approach, elements, and mapping rules in an analyzable specification language, such as Alloy, not only enables automatic synthesis of architectural models, but also provides the basis to formally validate their correctness. We express essential properties of target architectural style, and use the automated relational logic analyzer, i.e., Alloy Analyzer, to verify them. We specify such implications required to be checked as Alloy assertions, a set of constraints intended to follow from specifications.

As a concrete example, Listing 1 illustrates the content of one such assertion specified for architectures synthesized in the component-and-connector (Cnc) architectural style (cf. Listing 5). The *correctCncConfiguration* assertion states that each component port in a configuration of a component-and-connector architecture should be attached to a connector role, where the component and connector connecting through their

[4] Note that Alloy's specification language can be viewed as first-order logic, as Jackson notes [Jac12].

```
assert correctCncConfiguration{
  all s: System| attachments[s].dom in s.connectors.roles &&
                 attachments[s].ran in s.components.ports
}
```

Listing 1. An example assertion specification checking the correct attachments of component ports and connector roles in a configuration of a system architecture modeled in component-and-connector architectural style.

```
<alloy ...>
<instance ...>
<field label="components">...</field>
<field label="connectors">...</field>
<field label="ports">...</field>
<field label="roles">...</field>
<field label="attachments">...</field>
<field label="handle">
  <tuple> <atom label="IIObject$0"/>
  <atom label="FlightControl"/> </tuple>
...</field>
</instance>
</alloy>
```

Listing 2. An example of the intermediate representation of a synthesized architectural model

respective port and role belong to the same system. The functions *dom* and *ran*, defined in the Alloy module util/relation, return the domain and range of a binary relation, respectively.

The analyzer performs scope-complete analysis [Jac12], where each assertion is exhaustively checked against a huge set of model instances up to a certain bound. In other words, the analyzer is a bounded checker, guaranteeing the validity of assertions only within a bounded instance space. We bound execution of assertion checking with the ultimate scope used for synthesis of stylized architectural models, and thus expect the validity of assertions for all generated model instances.

3.5. Transformation to Architecture Description Language

Having computed satisfying solutions, our *Alloy2ADL transformer* component parses and transforms these solutions from low-level, XML formatted Alloy objects to high-level architecture descriptions in human-readable ADLs.

Listing 2 presents an example of the intermediate representation of a synthesized architectural model represented as an XML formatted Alloy instance, which is automatically generated by Monarch. The result was edited manually for presentation (to remove inessential details). The XML model contains basic elements involved in the component-and-connector architectural models, such as, components, connectors, ports of components, and roles of connectors. The system's configuration is further specified in the *attachments* field. Finally, the field labeled *handle* describes the relationships between the architectural model elements and application model constituents. Transformation of these XML-formatted results into a desired format, such as ADLs that supports essential architectural constructs, is straightforward and can be realized using the FreeMarker template engines [Fre].

During the past several years, a considerable number of general and domain-specific ADLs have been proposed [MT00]. We use the ACME language in our prototype [GMW00]. ACME emerged as a generic language for describing software architectures, with particular support for architectural styles. It is also designed to work as an interchange format for mapping among other architecture description languages. In an earlier work [BS10a], we briefly reported on the feasibility of treating architectural style as a separate variable in an aspect-oriented setting, with AspectualACME [GCB+06]—an aspect-enabled extension of ACME—as a target ADL.

Listing 3 partially represents the template for transforming generated satisfying solutions into models in the ACME architecture description language. Template parameters, such as components, connectors, and their respective ports and roles, are delimited by ${ and } in the template definition. The values of these

```
System ${system.name} extended with {
<#list components as component>
  Component ${component.name} extended with {
  <#list component.ports>
    <#items as port>
    Port ${port.name} extended with {}
    </#items>
  </#list>
  }
</#list>
<#list connectors as connector>
  Connector ${connector.name} extended with {
  <#list connector.roles>
    <#items as role>
    Role ${role.name} extended with {}
    </#items>
  </#list>
  }
</#list>
<#list attachments as attachment>
  Attachment ${attachment.port} to ${attachment.role};
</#list>
}
```

Listing 3. Part of the template for transforming generated Alloy instances into models in the ACME architecture description language.

```
module SCC

sig Sensor extends needHandle{}
sig Actuator extends needHandle{}

sig Controller extends needHandle{
  sensors : set Sensor,
  actuators : set Actuator,
  controller_dispatch_state: dispatch_protocol,
  frequency_state: frequency,
  program: fileAddress
}

abstract sig dispatch_protocol{}
abstract sig frequency{}
one sig periodic, aperiodic, sporadic extends dispatch_protocol{}
...
```

Listing 4. Part of SCC application type as an Alloy module.

parameters are dynamically realized by FreeMarker while parsing each XML formatted Alloy instance that represents a synthesized architectural model (cf. Listing 2). The #list and #item tags instruct the template engine to enumerate values for each architectural element. For example, expressions in lines 4–8 enumerate ports of each component within the system architecture.

4. Illustrative Example

In this section, we use Monarch to formally illustrate the process of mapping a *SCC* description of the *Lunar Lander* application [TMD09] to architectural models in the implicit-invocation style [DGJN98].

4.1. Application Type: SCC

In Section 3, we presented the concrete and human-centric realization of *SCC* application type as an AIML meta-model developed atop GME. Here we focus on its representation as an Alloy module.

Listing 4 partially outlines the sense-compute-control application type represented in Alloy. We explain

```
1  // (a) Cnc style specification
2  module Cnc
3
4  abstract sig System {
5    components : set Component,
6    connectors : set Connector,
7    attachments : Role -> some Port
8  }
9  abstract sig Component {
10   ports : set Port,
11   handle : lone needHandle
12 }
13 abstract sig Port {
14   component : one Component
15 }{
16   this in component.ports
17   one c:Component | this in c.ports
18 }
19 abstract sig Connector {
20   roles : set Role,
21   handle : lone needHandle
22 }
23 abstract sig Role {
24   connector : one Connector
25 }{
26   this in connector.roles
27   one c:Connector|this in c.roles
28 }
```

```
1  // (b) OO style specification
2  module OO
3
4  open Cnc
5
6  abstract sig Call extends Port{}
7  abstract sig Procedure extends Port{}
8  abstract sig Provide extends Role{}
9  abstract sig Request extends Role{}
10 abstract sig Object extends Component{
11   calls: set Call,
12   procedures: set Procedure
13 }{
14   ports = calls + procedures
15 }
16 abstract sig procedureCall extends Connector{}{
17   one Provide & roles
18   one Request & roles
19   roles = Provide + Request
20 }
21 fact{
22   all port:Call| port[attachments].ran in Request
       && lone port[attachments].ran
23   all port:Procedure| port[attachments].ran in
       Provide
24   all provideRole:Provide| lone System.attachments[
       provideRole]
25 }
```

Listing 5. Part of (a) the component-and-connector (Cnc) and (b) object-oriented (OO) styles described in Alloy.

it by comparing it to the meta-model of Fig. 3a. Each meta-element tagged with the *Model* stereotype in the SCC metamodel has a corresponding Alloy signature definition, except for *ApplicationDescription* whose instances denoting specific application models are mapped to separate Alloy modules. In particular, three Alloy signatures (lines 3–12) represent basic elements of SCC application type, i.e. *Actuator*, *Sensor* and *Controller*. Recall from Section 3.3, the elements tagged with *Connection* in the SCC metamodel, i.e., Actuators and Sensors in our running example (cf. Fig 3a), represent as signature fields in the Alloy representation (lines 7–8). That is, each of such meta-elements has a corresponding field declaration within the definition of the signature relevant to the Connection source, here the Controller signature. The Alloy module further defines two abstract signatures of *dispatch_protocol* and *frequency*, which are used in defining the specific properties of *Controller* elements.

We describe later in section 4.4 that the automatically generated Alloy modules representing application models explicitly *import* such application type Alloy modules, e.g., SCC module of Listing 4.

4.2. Architectural style: II

Architectural styles systematize successful architectural design practices in terms of constraints on architectural elements and their composition into systems [SG96]. One can consider an architectural style as a domain specific language (DSL), or more specifically, a DSL embedded in the Alloy language, where an architectural model is defined in terms of elements within that language. Indeed, such tiny domain specific languages provide specialized constructs for a particular domain for which the DSL was explicitly designed—here, style-specific architecture synthesis—and eschews irrelevant features.

In some cases it is helpful to model one architectural style as inheriting rules from another. Here, we specify the II style as an extension of more generic styles. An implicit invocation object (*IIObject*) is an *Object* that provides both a collection of interfaces (as with Object) and a set of events. Procedures may also be called in the usual way. So, an IIObject extends the definition of an Object. It can, in addition, register some of its procedures with events of the system; so those procedures will be invoked when the events are announced.

Listing 5 partially shows Alloy representations of (a) the component-and-connector (Cnc) style, adopted from Wong et al. [WSWS08], and (b) the object oriented (OO) style that itself extends the Cnc style. The

```
1  module II
2
3  open OO
4
5  abstract sig Publish extends Role {}
6  abstract sig Subscribe extends Role {}
7  abstract sig PublishEvent extends Port {}{
8    one o:IIObject | this in o.ports
9    all port:PublishEvent| attachments.port.ran in Publish
10 }
11 abstract sig SubscribeEvent extends Port {}{
12   one o:IIObject | this in o.ports
13   all port:SubscribeEvent| attachments.port.ran in Subscribe
14 }
15 abstract sig IIObject extends Object{}{
16   some PublishEvent & ports
17 }
18 abstract sig EventBus extends Connector {}{
19   roles in Publish + Subscribe
20 }
21
22 pred PublishAnEvent(s:System, s':System,pub : Component, conn : EventBus, port : PublishEvent){
23   some r : Publish |
24   some port & pub.ports and r in conn.roles and attached[s,s',r,port]
25 }
26
27 pred SubscribeToEvent(s:System, s':System, comp:Component, conn:EventBus, port:SubscribeEvent){
28   some r : Subscribe |
29   some port & comp.ports and r in conn.roles and attached[s,s',r,port]
30 }
```

Listing 6. Part of II style described in Alloy.

Cnc specification includes 5 top-level signatures: System, Component, Connector, Port, and Role. A System contains two sets of components and connectors. The attachments field then specifies connections between different components and connectors under the system configuration, where a connector Role is attached to a component Port (lines 3–7). Each component represents a system's computation entity, and contains a set of ports as its interfaces to communicate with other components. It may also *handle* an element from the application model (style independent). The keyword lone indicates that the handle element is optional. A connector is similarly specified, yet contains a set of roles, used to describe a certain type of communication.

An Object (Listing 5b, lines 10–15), that extends the Component signature, includes two sets of procedures and calls that collectively define its ports. A ProcedureCall is defined as a specific type of Connector (lines 16–20), and contains two roles: Provide and Request. Each Call port of an Object connects to at most one Request role of a ProcedureCall connector (line 22). A Provide role of the connector is connected to at most one Procedure port of an Object whose procedure will be called (lines 23–24).

Listing 6 (eliding details) describes the II style. The II Alloy module specifies six signatures: Publish, Subscribe, PublishEvent, SubscribeEvent, IIObject and EventBus. IIObject, a component extension, has PublishEvent and SubscribeEvent as its ports. EventBus is further a special kind of connector—explicitly extending its definition—and has two roles, i.e. Publish and Subscribe.

The Alloy dot operator denotes a relational join. In expressions represented in lines 9 and 13, the attachments is a relation of type $System \times Role \times Port$ (cf. Listing 5, line 6). Therefore, the attachments.port relation is from System to Role. The function *ran*, defined in the Alloy module util/relation, returns the range of a binary relation. The "in" operator furthermore declares the subset relation. As such, the invariant under consideration specifies that each PublishEvent port of an IIObject should be attached to a role of type Publish, and each SubscribeEvent port of an IIObject, in a similar way, should be connected to a Subscribe role of an EventBus.

The next predicate, i.e. *PublishAnEvent* (lines 22–25) ensures a connection between a *PublishEvent* port of a component, given as an input parameter, and a *Publish* role of the given *EventBus* connector. Similarly, the *SubscribeToEvent* predicate (lines 27–30) ensures a connection between a *SubscribeEvent* port of a component, again given as an input parameter, and a *Subscribe* role of the given *EventBus* connector.

```
1  module SCC_II
2
3  open SCC
4  open II
5
6  pred mapping(){
7  all n:needHandle | one o: IIObject | o.handle = n
8
9  all a:Actuator | one port: Port|
10     (port in (a.~handle.ports & Procedure)) ||
11     (port in (a.~handle.ports & SubscribeEvent))
12
13 all s:Sensor | one port: Port|
14     (port in (s.~handle.ports & Procedure)) ||
15     (port in (s.~handle.ports & PublishEvent))
16
17 # (Controller.~handle.ports & SubscribeEvent) =
18 # (Sensor.~handle.ports & PublishEvent)
19
20 (#SubscribeEvent >0) =>
21  # (Controller.~handle.ports & PublishEvent) = 1
22 (#Procedure >0) =>
23  # (Controller.~handle.ports & Call) = 1
24
25 all port:Procedure| one conector: procedureCall|
26     port[attachments].ran = conector.roles & Provide
27
28 Controller.~handle.call[attachments].ran.connector =
29  Actuator.~handle.procedure[attachments].ran.connector
30  + Sensor.~handle.procedure[attachments].ran.connector
31
32 (Controller.~handle.ports &
33   SubscribeEvent)[attachments].ran.~roles=
34 (Sensor.~handle.ports &
35   PublishEvent)[attachments].ran.~roles
36 ...
37 }
```

Listing 7. Part of the mapping predicate for the pair of SCC application type and II architectural style.

4.3. Architectural Map: (SCC,II)

The architectural mapping process takes as inputs the abstract application model (transformed directly from the concrete model to the Alloy module), an Alloy module specifying the application type (meta-model), an architectural style Alloy module that specifies the constraints to which the computed results conform, and the architectural mapping Alloy predicates that define relationships required to hold between the application model and computed architectural models. Each mapping predicate for the given application type and architectural style is responsible for confirming that the satisfying solutions refine the given application model in conformance with the given style.

Listing 7 shows such a predicate for the SCC application type and the implicit invocation architectural style. At the top, the specification imports the Alloy modules for the SCC application type and implicit invocation architectural style (depicted in Listings 4 and 6, respectively). The mapping predicate then, in line 7, states that for each sensor, actuator and controller, declared as subtype of the *needHandle* abstract Signature, there is an IIObject that handles it. Expressions in lines 9–11, by using the Alloy inverse relation operator $\sim$, state that each Actuator's IIObject has a port of type SubscribeEvent or Procedure to be called implicitly or explicitly. Likewise, each Sensor's IIObject has a port of type PublishEvent or Procedure. The number of SubscribeEvent ports of the Controller's IIObject equals to the number of PublishEvent ports of the Sensors' IIObjects, as mentioned in lines 17–18. So, each SubscribeEvent port of the Controller could be connected to a Sensor's PublishEvent port to be called implicitly. In addition, the specification, in lines 20–23, states that the Controller's IIObject has at most one PublishEvent port and one Call port so that the procedures of Actuators' IIObjects could be called explicitly or could register to be invoked when the PublishEvent port of the Controller's IIObject announces an event.

The II architectural style provides two ways of invoking methods: *procedure call* and *implicit invocation*. For the purpose of the former method, lines 25–30 state that for each Procedure port, there is a ProcedureCall

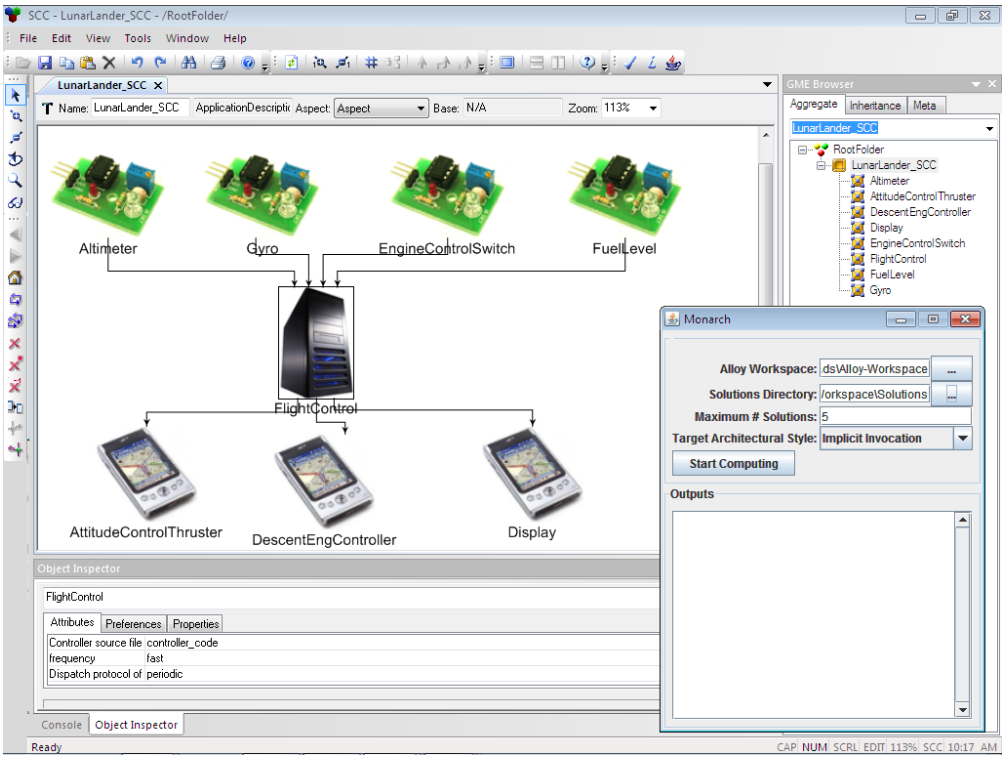

Fig. 4. Lunar-Lander application model in our GME modeling Environment.

connector connected to it, and the Call port of the Controller's IIObject is connected to the Procedure port of the IIObjects handling Sensors and Actuators via a connector of type ProcedureCall. For an implicit invocation, the SubscribeEvent ports of the controller's IIObject are connected to the PublishEvent ports of Sensors' IIObjects via an EventBus connector, as mentioned in lines 32–35. In a similar way, the SubscribeEvent ports of the Actuators' IIObjects are connected to the PublishEvent port of the Controller's IIObject through an EventBus connector.

4.4. Application Model: Lunar Lander in SCC application type

In their textbook [TMD09], Taylor et al. describe the informal mapping of a lunar lander application to architectures in a range of architectural styles. In this application, *FlightControl* maintains the state of a spacecraft based on the information provided by various sensors: Altimeter, Gyroscope, Fuel level indicator and the engine control switch. After processing control laws and computing values, FlightControl provides them for various actuators: Descent engine controller, Attitude control thruster and Display. Taylor et al. describe the lunar lander as an instance of a $sense-compute-control$ application. The notion of *application type* is implicit in their account. We make it explicit and formal in our theory. Figure 4 shows the lunar lander's application description modeled within GME using the generated modeling environment for our SCC meta-model. What is also shown in the figure is a screenshot of the Monarch architecture-synthesizer environment and how the synthesis process is started.

Listing 8 illustrates the Alloy representation of the lunar lander application model generated directly from its concrete model by *Monarch Interpreter* developed for the SCC meta-model (cf. Section 3.3). A synthesized Alloy module contains a signature definition for each element of type *Model* in the concrete model as well as a set of facts corresponding to the properties of those elements. More specifically, it starts by synthesizing the module name representing the name of the instance of the *ApplicationDescription* class within the concrete model. It then imports the Alloy specification module(s) for application type(s). For each instance of *Sensor*, *Actuator*, and *Controller* classes in a concrete model, it synthesizes a signature

```
module LunarLander_SCC

open SCC

one sig controller_code extends fileAddress{}
one sig Gyro extends Sensor{}
one sig Altimeter extends Sensor{}
one sig FuelLevel extends Sensor{}
one sig EngineControlSwitch extends Sensor{}
one sig Display extends Actuator{}
one sig DescentEngController extends Actuator{}
one sig AttitudeControlThruster extends Actuator{}
one sig FlightControl extends Controller{}{
  sensors = FuelLevel + EngineControlSwitch + Gyro + Altimeter
  actuators = DescentEngController + Display + AttitudeControlThruster
  controller_dispatch_state = periodic
  frequency_state = fast
  program = controller_code
}
```

Listing 8. Lunar Lander application model transformed automatically from its concrete model (cf. Fig. 4) into the Alloy language.

definition that represents the inheritance of a concrete element from its corresponding abstract class. The element's properties (if any) are also specified as Alloy facts for the corresponding signature of that element, e.g. *FightControl* has a periodic task with high frequency.

4.5. Satisfying Architectural Models

Using the Alloy Analyzer, Monarch computes architectural models, represented as satisfying solutions to the constraints of a map applied to an application model. Alloy Analyzer guarantees that computed descriptions *conform* to the given architectural style. The mapping predicates are responsible for ensuring that computed architectural models *refine* given application models.

Mapping the *SCC* description of the Lunar Lander to the II architectural style yields a set of satisfying solutions. Among them, for instance, Figure 5 depicts the internal structure of a result for the lunar lander example. In this diagram, the architectural description has eight *IIObject*s. The *FlightControl* element along with related sensors and actuators, inferred from the input specification, represents the Lunar Lander System. Each IIObject handles an element. As a case in point, *IIObject6* handles *FuelLevel* sensor and publishes a notification of new value through *PublishEvent1* that should be connected to an *EventBus* (connections are omitted for the sake of readability). On the other hand, *IIObject0*, that handles *FlightControl*, subscribes to input events through *SubscribeEvent*s ports, and will be implicitly invoked. This allows it to update the state of the spacecraft. This in turn causes *Display*, for example, to be invoked implicitly so that it refreshes its display based on new data.

To make the outputs humanly readable and useful, the *Alloy2ADL* transformer converts the Alloy-generated results, also available in an abstract XML-format (recall from Section 3.5), to a traditional architecture description language. Figure 6 represents one of the automatic computed instances of architecture description models in ACME, transformed from the model shown in Figure 5. These architectural models refine the Lunar Lander application description specified using the SCC application type, in conformance with the fully formal definition of the implicit invocation architectural style. The result is a set of formally derived architectural models for the given application in the selected architectural style.

According to the diagram, in this particular, arbitrarily selected case, the four top components, handling sensors, are connected to FightControl through EventBus connectors, i.e., using implicit invocation. The actuators components are also connected to the FlightControl through the EventBus4 connector.

The example illustrates the point that architectural styles, viewed as mappings to platforms, can be one-to-N. In general, there are many possible architectures, consistent with a given style, for a given application. This is mainly because style specifications to which application models were being mapped are under-specified, and in turn their corresponding architectural maps leave a family of architectural spaces all conforming to the target styles. Mappings to families of architectures could be useful in enabling optimizing search for properties influenced by architecture but not constrained in the modeling stage. Alternatively,

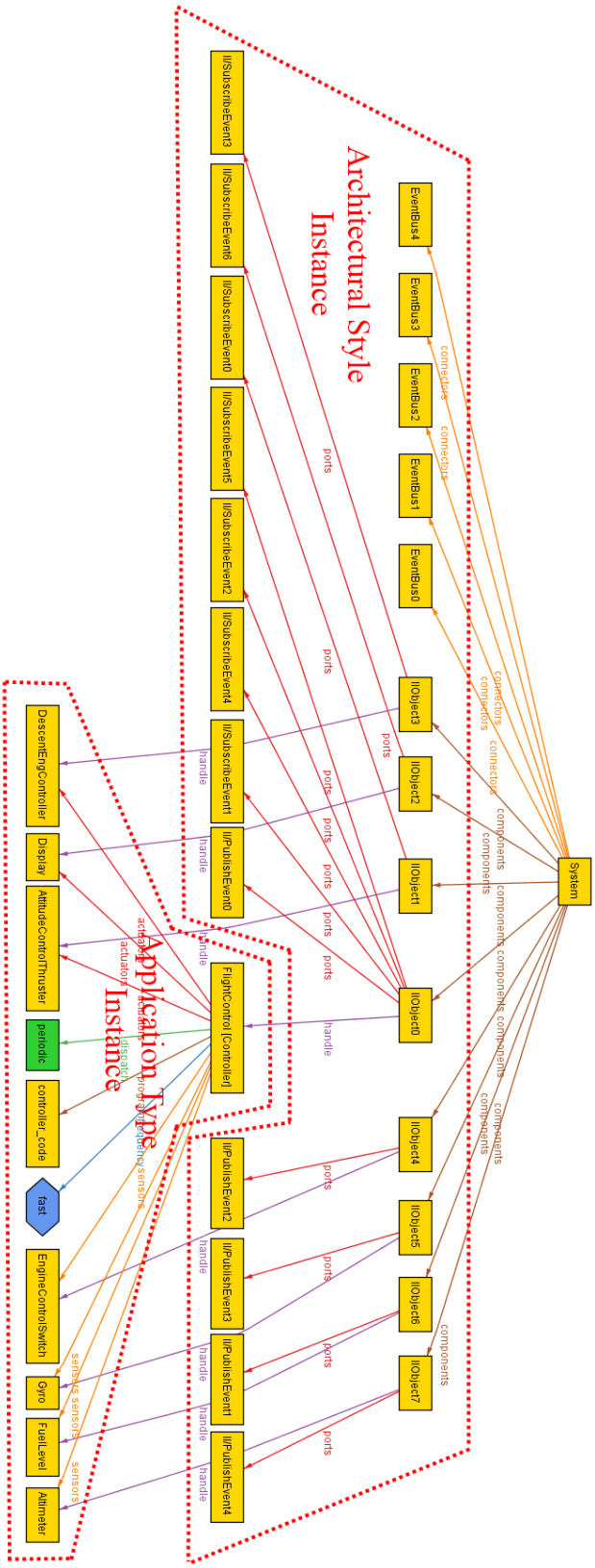

Fig. 5. The internal structure of a result of mapping Lunar Lander application into the II style

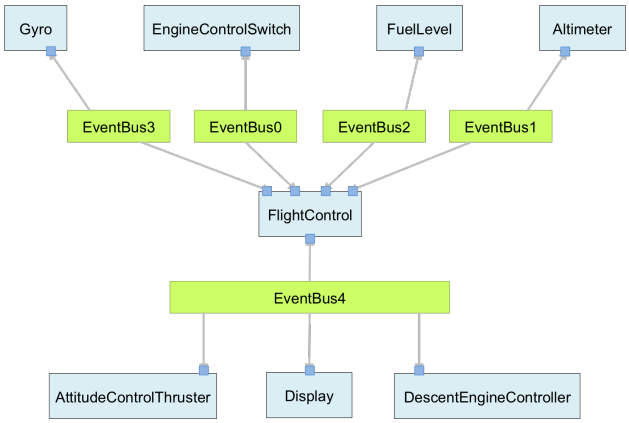

Fig. 6. A computed instance of mapping SCC description of the Lunar Lander into the implicit invocation architectural style in ACME.

one could strengthen the definition of the style or the application type and instance to limit the family of conforming instances.

5. Evaluation

To evaluate our approach, we have designed and conducted a number of case studies following guidelines from Kitchenham, Pickard and Pfleeger [KPP95]. Our evaluation addresses the following research questions:

RQ1. Does the approach enable formal, automatic synthesis of architectures from application descriptions and choices of architectural styles for representative applications from the software architecture literature?

RQ2. What is the performance of our prototype tool implemented atop the SAT solving technology for the considered applications?

To answer these questions, we use the Monarch apparatus we developed based on the synthesis approach for carrying out the case studies.

5.1. Subject Systems

We synthesized architecture in a variety of styles for four subject systems.

KWIC. Our first experimental case is KWIC, long used in studying architectural styles and their properties [Par72]: "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."

Lunar Lander. The Lunar Lander (LL) case study is adopted from Taylor et al's textbook on software architecture[TMD09]. The LL application description is discussed in the previous section.

MIDAS. Our third experimental subject is an alarm system of type sense-compute-control, called MIDAS [MSR+07]. This case study is inspired by Edwards et al. [ESM08]. They illustrated the structuring of the MIDAS application [MSR+07] from a family of embedded applications at Bosch, in different architectural styles, to assess the influence of architectural style on quality attributes.

EDS. Emergency Deployment System (EDS), adopted from Canavera et al. [CEM12], is designed for the deployment and management of personnel in emergency response scenarios. This subject system is representative of a class of component-based, distributed software systems, and has been deployed on more than 100 computing nodes [CEM12]. Figure 7 depicts the EDS's application model represented in the CF application type along with the dependency relationships among its components.

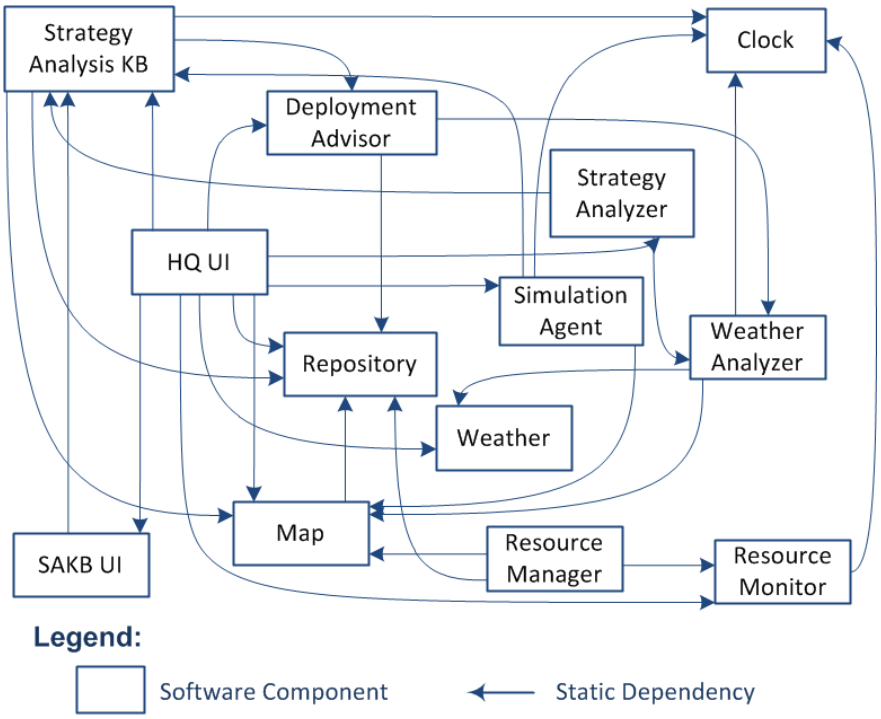

Fig. 7. Emergency Deployment System's application model.

5.2. Case Study Planning and Execution

We configured our tool by providing the input models. Specifically, we have developed formal specifications for involved application types and architectural styles. We then formalized architectural maps, such as the one shown in Listing 7, for these application types and architectural styles. For each experimental subject, we then configured Monarch by providing their application models using our GME-based modeling environment.

To address the first research question, we apply the formally specified architectural maps to case studies from the architecture literature [GKN92, TMD09, Par72, SG96], and test the consistency of formally generated results with the informal, manually derived architectures in the literature. To address the second research question, we measure the computational time required for deriving architectural models.

5.3. Results

In this section, we report and interpret data from our case studies. Table 2 summarizes a set of case studies formally replicating prominent earlier architectural studies from the literature. Each non-empty cell indicates an architectural map that we have implemented for the given type-style pair, and a corresponding experiment using the map. 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 developed maps for four application types and four architectural styles. The types are *composition-of-functions (CF)*, *state-driven behavior (SD)*, *sense-compute-control (SCC)* and *aspect-enabled composition-of-functions(ACF)*. The styles are *pipe-and-filter (PF), object-oriented (OO), implicit invocation (II)* and the *C2* architectural style.

These case studies attempt to replicate previously reported informal architectural mappings. In the following we first summarize the mapping results for the experiments, and then focus on the two experiments where the informally and manually produced results documented in the literature are not consistent with our formal and automated computations. That is, such architectural models do not properly refine the application model in conformance with the given architectural style (cf. Definition 2).

Table 2. Maps defined and experiments performed. (Rows represent architectural styles; Columns represent application types.)

	Comp. Fun.	State-Driven	SCC	ACF
PnF	KWIC, EDS			KWIC
OO	KWIC, EDS	KWIC	MIDAS, LL	
II	KWIC, EDS		MIDAS, LL	
C2	KWIC, EDS		MIDAS, LL	

(a) (b)

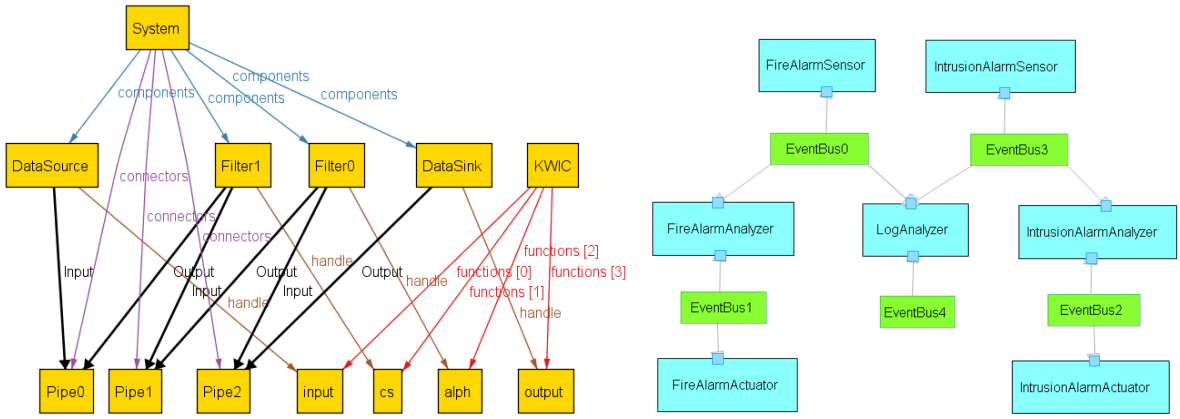

Fig. 8. Examples of computed results for two experiments: (a) $(CF, PnF, KWIC)$ and (b) $(SCC, II, MIDAS)$.

Figure 8a illustrates the computed result for the $(CF, PnF, KWIC)$ experiment. The diagram, that shows an intermediate representation of the generated architectural model, 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 KWIC *input* function. Its output 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.

In the $(CF, OO, KWIC)$ experiment, which is attempted to reproduce previous informal studies by Parnas [Par72] and later studies by Shaw and Garlan [SG96], we map a *CF* description of KWIC to an architectural description in the OO style. The $(SD, OO, KWIC)$ experiment addresses the work of Garlan, Kaiser & Notkin [GKN92], 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 type of application description, and that this change is what really drives the need for a new architectural style. We employ state-driven behavior as an application type for interactive application description.

Figure 8b represents one of the formally derived architectural models in ACME for the $(SCC, II, MIDAS)$ experiment. According to the model, among others, the *FireAlarmAnalyzer* component subscribes to the events of *FireAlarmSensor* and will be implicitly invoked. This allows it to perform the fire detection analysis, and in turn, causes the actuator component (*FireAlarmActuator*) to be invoked implicitly. The results of our formal and automated computations are consistent with the informally and manually produced results documented in the literature, except for two experiments of (SCC, OO, LL) and $(CF, C2, EDS)$.

The rest of this section reports on the execution and results of these two experiments. We elide discussion of the others, as they do not add anything new. For the (SCC, OO, LL) experiment, we note that Taylor et al., indicate that the application description of the lunar lander that they use in discussing the OO architectural style is not exactly the same as the one illustrated in the OO style section of the book.

We also applied similar experiments to the EDS system, formally mapping its application description (cf.

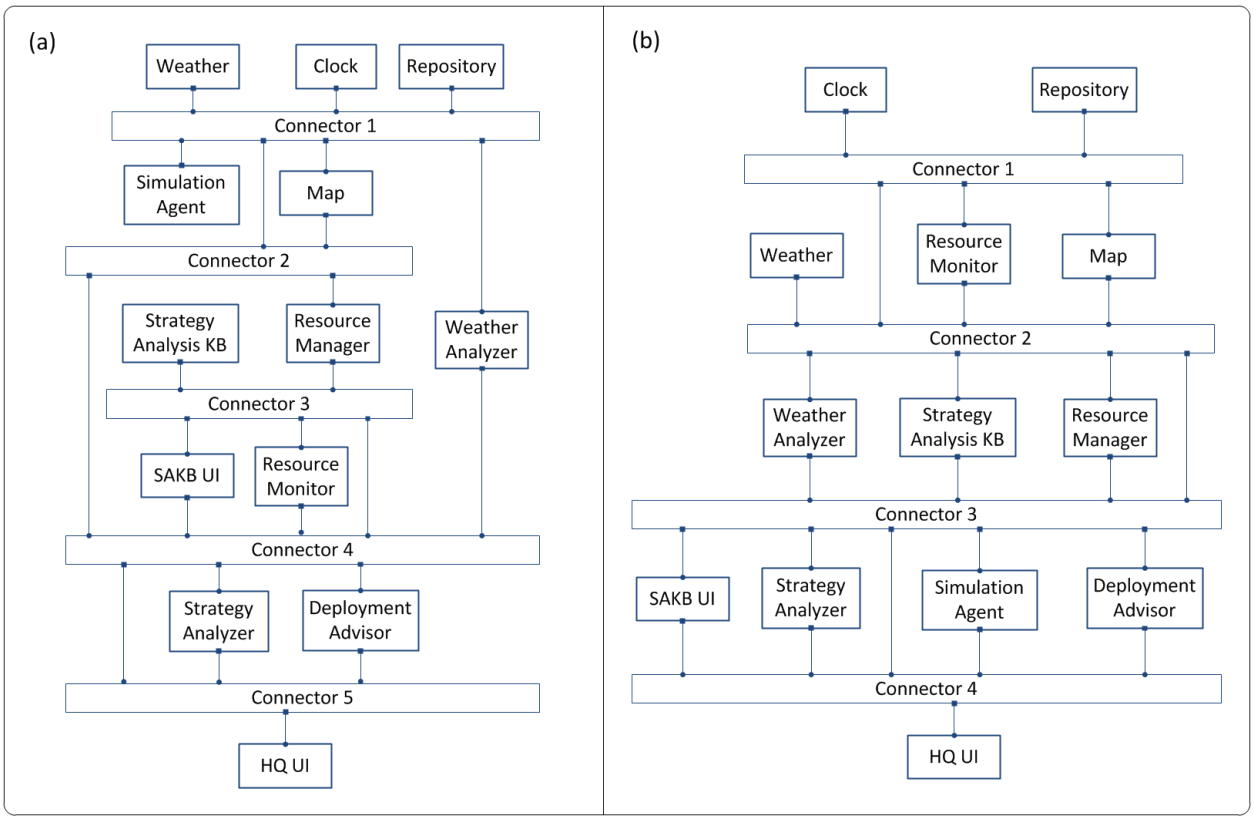

Fig. 9. Two versions of the EDS architectures: (a) the manually-developed version adopted from [EM12]; (b) formally-derived one computed through mapping of the component-dependency diagram (Figure 7) of the EDS into the C2 style.

Figure 7) into architectural models in different styles. In [CEM12], Canavera et al. provide the component-dependency diagram of the EDS system. The same authors, later, provide the EDS architecture in the C2 style [EM12], which in turn enables us to check consistency of the formally synthesized architecture with its manually-developed counterpart. Figures 9a and b depict the manually and formally-derived architectures for the EDS system, respectively.

Manual Inspection reveals inconsistencies between component-dependency diagram—from which we formally derived architecture—and the manually-developed architecture in C2 style. For example, in the component-dependency diagram, the `WeatherAnalyzer` component depends, among other things, on the `map` component, while in the C2 style architecture it is not connected to an appropriate connector that supports this component dependency. According to the C2 style specification [TMA+95], a component can send request messages only through its top interfaces, and notifications can only be sent through the bottom interfaces. Similarly, the `ResourceManager` component depends on the `ResourceMonitor` component, as shown in the component-dependency diagram of Figure 7, but the former component could not access the latter through its top interface, preventing it from sending request messages to the `ResourceMonitor` component.

Our discovery of such inconsistencies provides an example of how our formal synthesis technique can help designers in an error-prone task of developing architectures that refine the application model while complying with the rules implied by the target architectural style. A complete list of these case studies, including complete versions of Alloy models, are available for download [Mon15].

Table 3. Statistical results for architecture synthesis across subject systems.

Subj. Sys.	Arch. Style.	#Vars (thousands)	#Clauses (thousands)	Synthesis Time (Seconds) SAT Construction MiniSAT	SAT Construction SAT4J	SAT Solving MiniSAT	SAT Solving SAT4J
KWIC	PnF	73.086	136.306	1.218	0.191	0.81	0.150
	OO	39.034	93.573	0.174	0.111	0.42	0.090
	II	70.174	127.263	0.554	0.209	0.126	0.201
	C2	87.458	153.775	1.172	0.400	0.132	0.408
Lunar Lander	OO	196.087	600.119	0.875	0.476	1.219	1.867
	II	270.051	849.376	2.943	2.285	3.050	5.091
	C2	204.875	371.220	0.761	0.532	0.150	1.103
MIDAS	OO	222.648	696.725	0.952	0.526	0.486	2.529
	II	306.214	982.545	3.609	2.837	4.274	9.003
	C2	212.984	387.000	0.821	0.535	0.275	2.094
EDS	PnF	660.615	1372.933	3.412	2.636	7.257	21.361
	OO	526.187	1681.527	3.223	2.167	13.745	17.686
	II	1091.963	1987.840	3.623	3.343	3.462	147.372
	C2	823.166	1549.486	4.969	3.420	7.711	6.085

```
1  module performanceBenchmark
2
3  open CF as AppType
4  open C2 as ArchStyle
5  open CF_C2 as Map
6
7  pred show{
8    mapping[]
9  }
10
11 run show for 40 but 1 System
```

Listing 9. A performance benchmark predicate for the CF and C2 architectural map.

5.4. Performance

The next evaluation criterion is the performance benchmark of the architecture synthesis technology based on underlying SAT solver engines. We used a PC with an Intel Core i7 2.4 GHz CPU processor and 4 GB of main memory, and leveraged MiniSat and SAT4J as SAT solvers during the experiments. Table 3 shows statistical results for architecture synthesis across experiments targeting various architectural styles. Recall from Section 3, synthesizing architectural models using Monarch framework consists of four steps: (1) The application model is automatically transformed into an Alloy model from its concrete representation in the GME modeling environment (cf. section 3.3); (2) the Alloy model is automatically transformed into 3-SAT clauses using the Alloy Analyzer; (3) A SAT solver explores the space to find satisfying models; (4) The *Alloy to ADL transformer* automatically translates satisfying models to architectures in standard ADLs. Table 3 represents computational time involved in steps 2 and 3; the front- and back-end transformation time is trivial, thus omitted.

The `# Vars` and `# Clauses` columns delineate the number of variables and clauses generated in the boolean formula for the SAT-solver. These measured numbers for each experiment are representative of the size of the corresponding SAT-based synthesis problem. The next two columns, `SAT Construction` and `SAT Solving`, then present the computational time involved in steps 2 and 3 of our approach, respectively. The computation time is given for both MiniSAT [Min15] and SAT4J [SAT15] solvers.

The experimental data shows that both the choice of SAT solver and changes in a specification, e.g., simply swapping between architectural maps for the same application, have an impact on the synthesis time. It is also known that other parameters, such as the order of variables, have an influence on the performance of SAT solvers. However, regardless of the SAT solver employed, the results show that Monarch is able to formally generate architectures for all subject systems in less than 3 minutes.

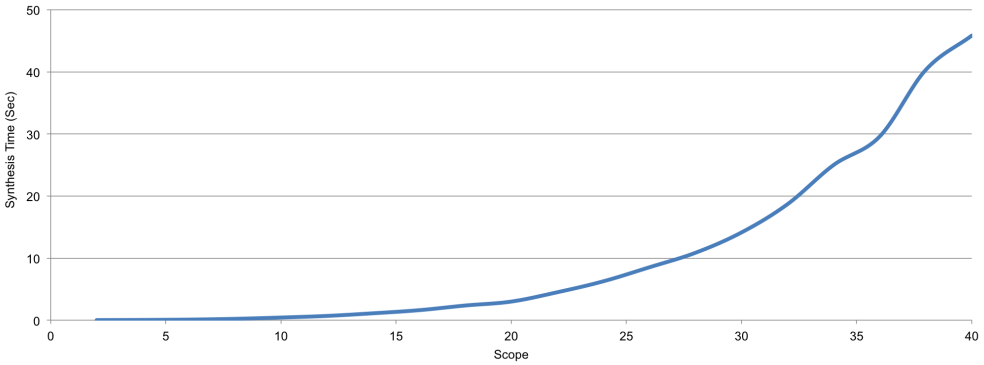

Fig. 10. Average architecture synthesis time over an increasing bound (Scope) for top-level element types.

We then evaluate scalability of Monarch's synthesis over the architectural mappings. Specifically, we measure how variation in size of an application affects the computational time required for deriving valid architectural models. As already mentioned, the synthesis relies on constraint solving over finite domains, and so it must be given an explicit scope to bound the number of elements of each top signature, such as components, connectors and ports. To instruct the analyzer to generate valid instances with specific number of elements, we formulate performance benchmark Alloy modules for various architectural maps, and run them several times within a scope increasing in each iteration. Listing 9 shows a performance benchmark predicate for the (CF,C2) architectural map. The *but* keyword specifies a separate bound for a signature following the keyword, here System. So it forces an upper bound of 40 elements for each top signature, but exactly within one System, for generated models.

Figure 10 shows the average synthesis time as the maximum number of elements varies from 2 to 40, with MiniSAT as the SAT solver. The results show that the synthesis time increases with increasing bound as expected. In the worst case, this increase could be exponential. While the experimental results confirm that the proposed technology based on a bounded model finder is feasible, given the apparent trend of advancements in SAT solvers we were witnessing in recent years, we expect that performance of synthesis techniques that leverages underlying off-the-shelf SAT solvers will likewise continue to improve.

6. Observations and lessons learned

This work provides evidence in support of the applicability of a specification-driven MDD in formally precise development of stylized software architectures from high-level application structures. It further enables the reuse of formal specifications for synthesis, including previously published specifications of architectural styles as well as mapping rules (cf. section 5). Reusing published style specifications is also important in validating consistency of synthesized architectures with contemporary formal accounts of architectural style. This work, thus, incorporates benefits of formal specifications of architectural styles to reliably synthesize style-specific architectural models.

6.1. Analyzable specification language

The main motivations for the use of Alloy as the back-end synthesis engine in this work are the gains in expressiveness and abstraction together with its automatic analysis capabilities favoring its adoption in implementation of our approach. More precisely, its simple set theoretic language was sufficiently expressive for formally defining syntax and semantics of languages for application and architecture modeling [KM08]. We find it very helpful to formalize both modeling languages, such as AIML (cf. section 3), and model instances. Furthermore, its ability to compute solutions that satisfy complex sets of constraints is useful as an automated synthesis mechanism.

Its effective module mechanism also facilitates reusing specifications and constraints in different contexts, and allows us to split the overall synthesis model among several tractable modules. Architectural maps and

styles are strictly separated and modularized in different specifications, which further facilitates reusability of such specifications. Specifically, this paper shows the promise of paying a one-time cost to formally specify them in an analyzable specification language to enable synthesis of stylized, application-specific software architecture.

Finally, we use the simulation capability of the Alloy Analyzer to compute application model instances, represented as satisfying solutions to application specifications automatically generated by our GME interpreters (cf. Section 3.3). This shows the validity of such transformed models, confirming that the application specifications are self-consistent, mutually compatible and consistent with the application type specification modeled in a separate module.

6.2. Architectural map: a means of reconciliation

Our experiments show that architectural maps can be formalized and implemented as executable specifications. 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 either consistent with manually produced results documented in the literature, or revealed certain inconsistencies.

A particularly important idea is that these reusable maps can be subject to one-time rigorous analysis to show that they properly preserve stated application properties. This use of validated maps is the key to enabling designers to work at the application level with confidence that architectures will preserve stated application properties.

This work also suggests that the concept of *application type* is important. The concept of application type leads naturally to an abstract, user-friendly approach to application modeling. We have demonstrated an approach taking fully formal specifications of application models and architectural styles as inputs and producing software architectures as outputs within the framework of MDD.

6.3. Supports for extension

The proposed framework tool suite supports reasonable extension for new types and mappings. To that end, one specifies an application type and its corresponding GME meta-model, as well as the architectural mapping predicates for relevant styles, so that by swapping between implementations of architectural maps, one can produce architectures in a range of styles for a given system from a high-level application description. This work appears to support the idea that being able to treat architectural style as a separate variable is a plausible aspiration. With automated architectural mapping, the software architect may also readily examine the feasibility of various architectural alternatives. The ease of examining more architectural alternatives will also increase the quality of software architecture. The more various alternatives are studied, the more likely it is that the most satisfying option will be found.

6.4. Multiple specification languages

To date the work has mainly considered structural refinements, and system behavior is addressed only to the extent that constraints on behavior are implicit or explicit in the specifications, e.g. of the employed architectural styles. As Alloy's emphasis is on specification and automatic analysis of structural properties of systems, it may not be the best option to address behavioral aspects of systems. We envisage that in an ultimate implementation of this technology, one uses several specification languages and corresponding synthesis technologies handling different aspects of the system. Utilizing heterogeneous modeling notations, however, calls for a modularization mechanism so that different models can be changed independently within certain constraints without breaking the whole system. Important steps in this direction are provided by the work on weaving aspect mechanisms in multi-language aspect-oriented frameworks [KL07].

Overall, the intellectual contribution of this work is the insight that software architectural styles can serve as analogs to choices of platforms in model-driven development. This idea then leads naturally to a new kind of tool: one that allows for modeling of applications independent of subsequent architectural style choices, and for the automated mapping of models to architectures once such style choices are made. Whereas traditional MDD work seeks to replace the programmer, this work points the way to a future in which MDD replaces at least some of the work of the software architect.

7. Related Work

We can identify in the literature four categories of research that are related to ours: research on formalization of architectural styles, architecture optimization research efforts, formal approaches to model transformation with special focus on software architecture, and approaches to separating application and architectural concerns.

Formalization of Architectural Styles. The notion of architectural styles has been present since the identification of software architecture as a discipline within software engineering [BCK03, PW92, SG96]. A variety of approaches have been proposed to model and analyze architectural styles. In this context, Alloy has been applied by numerous researchers to formal work in software architecture [GMK02, BS12, KG10, WSKW06, BS11, WSWS08]. Kelsen and Ma [KM08], comparing the traditional methods of formal specifications for modeling languages with an approach based on the Alloy language, argue that because of both lower notational complexity and automatic analyzability, Alloy provides more convenient facilities for defining the formal semantics of modeling languages. Wong et al. [WSWS08] proposed an approach based on the Alloy language for modeling and verification of complex systems that exploit multi-style structures. Our work differs in its focus on separating application description from style choices. Furthermore, we use Alloy not only to check the consistency of a given description against the rules of a style, but also to synthesize spaces of architectural models consistent with given styles. More recently, Maoz et al. [MRR13] developed a model merging approach for architecture synthesis. Given a set of component-and-connector views, each one essentially representing a partial model of the system, their approach, using Alloy, generates a satisfying component-and-connector model. We share with this approach the emphasis on leveraging constraint solving for architecture synthesis. However, our work differs fundamentally in its emphasis on the generation of stylized architectural models from architecture-independent application models.

Along the same line, Kim and Garlan [KG10] proposed an approach to translate architectural styles described formally in an architecture description language to the Alloy language, and in turn, to verify properties implied by architectural styles. Their focus is on transforming description of architectural styles to a relational model that can be automatically checked, whereas we concentrate on extending the concept of the software architectural style to include mappings from abstract problem descriptions to target software architectures.

Architecture Optimization. The other relevant line of research focuses on architecture optimization. Bondarev et al. [BCd07] proposed a framework, called *DeepCompass*, that analyzes architectural alternatives in dimensions of performance and cost to find Pareto-optimal candidates. Their approach, however, requires a manual specification of architectural alternatives, and provides no support for architecture synthesis.

Along the same line, Aleti et al. [ABGM09] developed *ArcheOpterix* for optimizing an embedded system's architecture. They applied an evolutionary algorithm to optimize architectures modeled in the AADL language with special focus on component deployment problems. Martens et al. [MKBR10] also developed *PerOpteryx* to automatically improve an initial software architectural model through searching for Pareto-optimal solution candidates. They applied a genetic algorithm to the Palladio Component Models of given software architectures.

Like many other research work we studied, these research efforts do not support architectural style as a co-domain for architecture synthesis. However, we believe this line of research is complementary to ours in that one can apply an optimizing search step to the synthesized architectural space to find Pareto-optimal solutions.

Architectural Transformation. A number of researchers have proposed formal approaches that can model architecture transformation [BG08, BS13, AK02, GBSC09, Gru05]. Among others, Ambriola and Kmiecik [AK02], described how preliminary architectural sketches can be incrementally refined into more mature architectural descriptions by the use of *architectural transformations* that add, remove, and move architectural elements and relationships. Their work focuses on *horizontal* architectural changes over time: from one architectural description to another. Our work by contrast focuses on vertical mappings: how architectural style choices map structured application descriptions to architectural descriptions.

Tamzalit and Mens [TM10] 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 differs in its focus on formal mappings of architecture-independent application models to a diversity of realized architectural models.

The other similar work is the recent work of Garlan et al. [GBSC09] on the evolution of one fully developed architectural description with respect to architectural style. The premise of this work is that it is sometimes necessary to change an architecture developed in one architectural style into a related architecture in another style. The approach they proposed involves incremental steps between models, 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.

Finally, these research efforts both focus on *horizontal architectural evolution*, whereas ours is on a formal study of the diversity of possible mappings from structured application specifications to architectural descriptions. That is, beyond just evolutionary transitions between pairs of architectural models, we focus on architectural style as a separate variable in design. Research efforts on horizontal and vertical mappings are complementary, in that success in understanding architectural style as a separate variable could, in practice, help to explain and perhaps automate horizontal evolution by treating the starting and ending points of horizontal evolution as images of a given application under different styles.

Separation of Application and Architecture Concerns. Researchers and practitioners have long separated application descriptions from choices of architectural form. Parnas's 1972 paper make this distinction [Par72], showing how one application, key word in context (KWIC), could be mapped to two distinct architectures: one based on a functional decomposition, and one on information hiding. In analogous work published in 2009, Taylor et al. [TMD09] described a landing control system and showed how it could be realized in a wide variety of architectural styles. Deline's work on Flexible Packaging (FP) [DeL99] is also related to our work. The basic idea was that a given function can be packaged up in different ways to work in systems with different architectural styles. He furthermore assumed that the stylistic differences mostly involved the ways in which a core function implementation would interact with its surrounding environment, or packaging. The FP method thus separates the interactive aspects of a component (*packaging*), from its core function (the *ware*).

In these research efforts, however, the mapping process has remained implicit, informal, and not itself subject to rigorous investigation or explicit representation and analysis. In most work to date, the focus is on *ex post* analysis of the relevant properties of the resulting architectures (e.g., performance). Key application properties are inadequately addressed until it is too late. Our work aims to move consideration of all essential application properties to the application description level, and to precisely determine a *priori* the checkable conditions under which particular architectural mappings are allowed. This change in perspective is significant.

8. Conclusion

This paper makes several contributions. First, we identified the treatment of architectural style as a separate variable as a key problem area and goal for software engineering. Second, we developed the concepts of application type and architectural map as key constructs needed, in addition to that of architectural style, to achieve such a separation of concerns. We showed that this separation of concerns gives rise to a natural form of model-driven development for synthesis of architectural models from abstract application descriptions and architectural styles expressed in Alloy. Third, we presented experimental data and a framework that support our claims of feasibility and the proposition that these ideas are worth pursuing.

References

[ABGM09] A. Aleti, S. Bjornander, L. Grunske, and I. Meedeniya. Archeopterix: An extendable tool for architecture optimization of aadl models. In *Proceedings of the International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*, pages 61–71, February 2009.

[AK02] Vincenzo Ambriola and Alina Kmiecik. Architectural transformations. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 275–278, 2002.

[BCd07] E. Bondarev, M. R. V. Chaudron, and E. A. de Kock. Exploring performance trade-offs of a jpeg decoder using the deepcompass framework. In *Proceedings of WOSP'07*, pages 153–163, February 2007.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2nd edition, 2003.

[BG08] Antonio Bucchiarone and Juan P. Galeotti. Dynamic software architectures verification using DynAlloy. In *Proceedings 7th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008)*, 2008.

[BS10a] Hamid Bagheri and Kevin Sullivan. Architecture as an independent variable for Aspect-Oriented application descriptions. In *Abstract State Machines, Alloy, B and Z (ABZ 2010)*, (LNCS 5977), Canada, 2010.

[BSS10] Hamid Bagheri, Yuanyuan Song, and Kevin Sullivan. Architectural style as an independent variable. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*, pages 159–162, 2010.

[BS10b] Hamid Bagheri and Kevin Sullivan. Monarch: Model-based Development of Software Architectures. In *Proceedings of the 13th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010)*, Lecture Notes in Computer Science 6395, pages 376–390, 2010.

[BS11] Hamid Bagheri and Kevin Sullivan. A Formal Approach for Incorporating Architectural Tactics into the Software Architecture. In *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*, pages 770–775, 2011.

[BS12] Hamid Bagheri and Kevin Sullivan. Pol: Specification-Driven Synthesis of Architectural Code Frameworks for Platform-Based Applications. In *Proceedings Proceedings of the 11th ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE'12)*, pages 93–102, 2012.

[BS13] Hamid Bagheri and Kevin Sullivan. Bottom-up Model-driven Development. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*, pages 1221–1224, 2013.

[CEM12] Kyle R. Canavera, Naeem Esfahani, and Sam Malek. Mining the execution history of a software system to infer the best time for its adaptation. In *Proceedings of the International Symp. on the Foundations of Software Engineering*, pages 1–11, November 2012.

[DeL99] Robert DeLine. Avoiding packaging mismatch with flexible packaging. In *Proceedings of the 21st international conference on Software engineering*, pages 97–106, 1999.

[DGJN98] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Towards a formal treatment of implicit invocation. *Formal Aspects of Computing*, 10:193–213, 1998.

[EM12] Naeem Esfahani and Sam Malek. Utilizing architectural styles to enhance the adaptation support of middleware platforms. *Inf. Softw. Technol.*, 54(7):786–801, July 2012.

[ESM08] George Edwards, Chiyoung Seo, and Nenad Medvidovic. Model interpreter frameworks: A foundation for the analysis of domain-specific software architectures. *Journal of Universal Computer Science*, 14(8):1182–1206, 2008.

[Fre] Freemarker java template engine. `http://freemarker.org/`.

[GBSC09] David Garlan, Jeffrey M. Barnes, Bradley Schmerl, and Orieta 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, September 2009.

[GCB+06] Alessandro Garcia, Christina Chavez, Thais Batista, Claudio Santanna, Uira Kulesza, Awais Rashid, and Carlos Lucena. On the modular representation of architectural aspects. In *Proceedings of the European Workshop on Software Architecture*, pages 82—97, Nantes, France, 2006. Lecture Notes in Computer Science.

[GKN92] D. Garlan, G. Kaiser, and D. Notkin. Using tool abstraction to compose systems. *Computer*, 25(6):30–38, June 1992.

[GMK02] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising software architectures for distributed systems. In *Proceedings of the first workshop on Self-healing systems*, pages 33–38, 2002.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. Acme: architectural description of component-based systems. In *Foundations of component-based systems*, pages 47–67. 2000.

[Gru05] L. Grunske. Formalizing architectural refactorings as graph transformation systems. In *Proceedings of the Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Networks SNPD/SAWN'05*, pages 324–329, 2005.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[Jac12] Daniel Jackson. Software Abstractions, 2nd ed. MIT Press, 2012.

[KG10] Jung Soo Kim and David Garlan. Analyzing architectural styles. *Journal of Systems and Software*, 83(7):1216–1235, 2010.

[KL07] Sergei Kojarski and David H. Lorenz. Identifying feature interactions in multi-language aspect-oriented frameworks. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 147–157, Washington, DC, USA, 2007. IEEE Computer Society.

[KM08] Pierre Kelsen and Qin 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, 2008.

[KPP95] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. Case studies for method and tool evaluation. *IEEE Softw.*, 12(4):52–62, 1995.

[LBM+01] Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing Domain-Specific design environments. *Computer*, 34(11):44–51, 2001.

[Met10] MetaEdit+. http://www.metacase.com/.

[Min15] MiniSat. Minisat website, 2015.

[MKBR10] A. Martens, H. Koziolek, S. Becker, and R. H. Reussner. Automatically improve software models for performance, reliability and cost using genetic algorithms. In *Proceedings of the 1st Int. Conf. on Performance Engineering*, pages 105–116, February 2010.

[MM03] J. Mukerji and J. Miller. MDA guide version 1.0.1. omg/2003-06-01. Technical report, 2003.

[Mon15] Monarch tool suite. `http://www.cs.virginia.edu/~hb2j/Downloads/Monarch-ToolSuite.zip`.

[MRR13] S. Maoz, J. O. Ringert, , and B. Rumpe. Synthesis of component and connector models from crosscutting structural views. In *Proceedings of the European software engineering conference held jointly with the ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE'13)*, pages 444–454, 2013.

[MSR+07] Sam Malek, Chiyoung Seo, Sharmila Ravula, Brad Petrus, and Nenad Medvidovic. Reconceptualizing a family of heterogeneous embedded systems via explicit architectural support. In *Proceedings of the International Conference on Software Engineering*, 2007.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[SAT15] SAT4J. Sat4j website, 2015.

[Sch06] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[TM10] D. Tamzalit and T. Mens. Guiding architectural restructuring through architectural styles. In *Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems*, pages 69–78, 2010.

[TMA+95] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, Jr E. James Whitehead, and Jason 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. ACM, 1995.

[TMD09] Richard N. Taylor, Nenad Medvidovic, and Eric Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[Tor09] Emina Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, MIT, February 2009.

[WSKW06] Ian Warren, Jing Sun, Sanjev Krishnamohan, and Thiranjith Weerasinghe. An automated formal approach to managing dynamic reconfiguration. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 37–46, 2006.

[WSNW07] Jules White, Douglas C. Schmidt, Andrey Nechypurenko, and Egon Wuchner. Introduction to the generic eclipse modelling system. *Eclipse Magazine*, 2007(6):11–18, 2007.

[WSWS08] Stephen Wong, Jing Sun, Ian Warren, and Jun 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.