In Proc. Workshop on the Role of Software Architecture for Testing and Analysis 2006 (ROSATEA06), Portland, Maine, July 17, 2006.

Analyzing Architectural Styles with Alloy

Jung Soo Kim
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
jungsoo@cmu.edu

David Garlan
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
garlan@cs.cmu.edu

ABSTRACT

The backbone of many architectures is an architectural style that provides a domain-specific design vocabulary and set of constraints on how that vocabulary can be used. Hence, designing a sound and appropriate architectural style becomes an important and intellectually challenging activity. Unfortunately, although there are numerous tools to help in the analysis of individual architectures, relatively less work has been done on tools to help the style designer. In this paper we show how to map an architectural style, expressed formally in an architectural description language, into a relational model that can be automatically checked for properties such as whether a style is consistent, whether a style satisfies some predicate over the architectural structure, whether two styles are compatible for composition, and whether one style refines another.

Categories and Subject Descriptors

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

Keywords

software architecture, architectural style, style analysis

1. INTRODUCTION

The discipline of software architecture has matured substantially over the past decade: today we find growing use of standard notations [22, 25], architecture-based development methods [7, 11], and handbooks for architectural modeling and design [4, 6]. And, as a significant indicator of engineering maturity, we are also seeing a growing body of research on ways to formally analyze properties of architectures, such as component compatibility [2], performance [9], reliability [30], style conformance [26], and many others.

One of the important pillars of modern software architecture is the use of architectural styles [1, 4, 6, 27]. An architectural style defines a family of related systems, typically by providing a domain-specific architectural design vocabulary together with constraints on how the parts must fit together. Examples of common styles range from the very generic (such as client-server or pipe-filter [27]) to the very specific (such as MDS [24], J2EE [21]).

The use of styles as a vehicle for characterizing a family of software architectures is motivated by a number of benefits. Styles provide a common vocabulary for architects, allowing developers to more easily understand a routine architectural design. They form the backbone of product-line frameworks, allowing the reuse of architectures across many products. By constraining the design space, they provide opportunities for specialized analysis. In many cases they can be linked to an implementation framework that provides a reusable code base, and, in some situations, code generators for significant parts of the system.

Consequently more and more architectural styles are being defined every day. In many cases styles will be elaborations of existing styles. For example, a company might constrain J2EE-based architectures to support a particular set of business services. In other situations new styles may be combinations of other styles. For example, one might combine a closed-loop control architecture with a publish-subscribe style to satisfy emerging needs for automotive software.

Defining a new style, however, is not an easy task. One must take care that the component building blocks fit together in appropriate ways, that each instance of the style satisfies certain key properties, and that constraints on the use of the style are neither too strong, nor too weak. Thus defining styles becomes an intellectual challenge in its own right. Indeed, in many ways the need for careful design of architectural styles far exceeds the needs for individual systems, since flaws in an architectural style can potentially impact *every* system that is built using it.

Unfortunately, despite significant progress in formal analysis of the architectures for *individual systems*, there is relatively little to guide the style designer. Answering questions like whether a style specifies a non-empty set of systems, whether it can be combined consistently with another, or whether it will retain the essential properties of some parent style, is today largely a matter of trial and error. In fact, today style designers typically cannot detect fundamental errors in a style until someone actually tries to implement a particular system in that style, when the cost of change is very high.

ROSATEA '06, July 17, 2006, Portland, Maine, USA.

To address this gap, ideally what we would like to have is a way to formally express and verify properties of architectural styles. Even better would be a set of standard sanity checks that every style designer should consider. Better still, we would hope that many of these checks could be carried out automatically.

In this paper we describe a technique that does exactly that with respect to structural aspects of architectural styles. Specifically, we show how to map an architectural style, expressed formally in an architectural description language, into a relational model that can be checked for various properties relevant to a style designer. We illustrate the approach by showing how to analyze four crucial kinds of properties: whether a style is consistent, whether a style satisfies some predicate over the architectural structure, whether two styles are compatible for composition, and whether one style refines another.

In Section 2 we describe related work. Section 3 discusses architectural styles: what they are, how they can be characterized formally, and what kinds of properties we would like them to have. Section 4 provides an overview of Alloy and the Alloy Analyzer, the target modeling language and tool that we will be using to check properties of styles. Section 5 then presents the translation schema, showing how to map formal descriptions of architectural styles into Alloy, and highlighting places where that translation is non-trivial. Section 6 shows by example how to to use the Alloy Analyzer to check properties of an architectural style. Section 7 discusses the strengths and limitations of the approach, and considers future work.

2. RELATED WORK

There are two broad areas of closely related work. The first is formal representation and analysis of software architecture. Since the inception of software architecture over a decade ago, there have been a large number of researchers interested in formal description of architectures. These efforts have largely focused on the definition and use of architecture description languages (ADLs) [20]. Many of these languages were explicitly defined to support formal analysis, often using existing (non-architectural) formalisms for modeling behavior. For example, a number of ADLs have used process algebras [2, 19] to specify abstract behavior of an architecture, and to check for properties like deadlock freedom of connector specifications. Others have used rewriting rules [15], sequence diagrams [14], and many others.

This existing body of work on analysis of software architectures has primarily focused on the problem of analyzing the properties of *individual* systems. That is, given an architectual description of a particular system, the goal is to formally evaluate some set of properties of that system. Properties include things like consistency of interfaces [24], performance [9, 29], and reliability [25]. While many of these analyses assume that a system is described in a particular style (such as one amenable to rate monotonic analysis), unlike our work, the issue of evaluating *the style itself* is not directly addressed.

There has been some research on ways to formally model architectural styles and their properties. Early work on this was carried out by Abowd et al. [1], who modeled styles using Z. In that approach one can specify general properties of architectural styles, but the work lacked explicit guidance on what properties should be evaluated, and it did not provide any tool-assisted support for analysis. Other work has investigated formal properties of particular styles, such as EJB [28] or Pub-Sub [10], but these have not provided any general style-oriented analyses. Finally, the Acme ADL was developed with the specific intent of providing a way to formally define architectural styles, in general, and to check conformance between the architecture of a system and its purported styles [12]. As we describe later, our work builds directly on that formalism, extending the possibilities of analysis to the styles themselves.

The second area of related research is model-based design. Independently of software architecture, there has been a lot of research on using models to develop and gain confidence in systems. Most of this work has been targeted at standard general-purpose modeling notations, such as UML, Z, or B, as opposed to domain-specific modeling languages such as architectural styles. In this work we build on these general specification languages using Alloy [16], one such general-purpose modeling language, as the assembly language for our own analyses.

Most closely related to our research is work on model-based software engineering, such as work by Karsai et al. [17] on GMS, that adopts an approach to meta-modeling in which new modeling languages and analyses can be defined for a particular domain. That work shares the general goals of our approach: that it should be possible to provide customized modeling notations and analyses to take advantage of them. However, unlike the work on GMS, ours is focused specifically on software architectural styles and their properties. This makes our work less general, but at the same time allows us to tailor our approach specifically to the needs of the architectural style designer.

3. ARCHITECTURAL STYLE

Software architecture is concerned with the high-level structure and properties of a system [23, 27]. Over time there has emerged a general consensus that modeling of complex architectures is best done through a set of complementary views [6]. Among the most important types of views are those that represent the run-time structures of a system. This type of view consists of a description of the system's components – its principle computational elements and data stores – and its connectors – the pathways of interaction and communication between the components. In addition, an architecture of a system typically includes a set of properties of interest, representing things like expected latencies on connectors, transaction rates of databases, etc.

While it is possible to model the architecture of a system using generic concepts of components and connectors, it is often beneficial to use a more specialized architectural modeling vocabulary that targets a family of architectures for a particular domain. We refer to these specialized modeling languages as *architectural styles*.[1]

[1]Styles are sometimes also referred to as "families," "patterns," or "frameworks."

Architectural styles have a number of significant benefits. First, styles promote design reuse, since the same architecture design is used across a set of related systems. Second, styles can lead to significant code reuse. For example many styles (like those associated with J2EE or .Net) provide prepackaged middleware to support connector implementations. Similarly, Unix-based systems adopting a pipe-filter style can take advantage of operating system services to implement task scheduling, synchronization, and communication through pipes. Third, it is easier for others to understand a system's organization if standard architectural structures are used. For example, even without specific details, knowing that a system's architecture is based on "client-server" immediately conveys an understanding of the kinds of pieces and how they fit together. Fourth, styles support interoperability. Examples include CORBA object-oriented architecture [8] and event-based tool integration [3]. Fifth, by constraining the design space, an architectural style often permits specialized analyses. For example, it is possible to analyze systems built in a pipe-filter style for schedulability, throughput, latency, and deadlock-freedom. Such analyses might not be meaningful for an arbitrary, ad hoc architecture – or even one constructed in a different style.

Consequently, there are hundreds, if not thousands, of architectural styles that are in use today (even if they are not formally named or defined as such). Indeed, the recent industrial interest in product lines and frameworks invariably results in the definition of new styles. Many of these styles are specializations or combinations of existing styles. For example, a company specializing in inventory management might provide a specialization of J2EE that captures the common structures in that product domain. Other styles may be defined from scratch.

3.1 Formal Modeling of Architectures

Building on the large body of existing formal modeling techniques for component-and-connector architectures, we model an architecture using the following core concepts that appear in most modern ADLs [20], as well as UML 2.0 [22].

- **Components:** Components represent the principal computational elements and data stores of a system. A component has a set of run-time interfaces, called *ports*, which define the points of interaction between that component and its environment.

- **Connectors:** Connectors identify the pathways of interaction between components. Connectors may represent simple interactions, such as a single service invocation between a client and server. Or they may represent complex protocols, such as the control of a robot on Mars by a ground control station. A connector defines a set of *roles* that identify the participants in the interaction. For example, a pipe connector might have a reader and a writer role; a publish-subscribe connector might have multiple announcer and listener roles.

- **Configurations:** An architectural configuration (or simply *architecture* or *system*) is a graph that defines how a set of components are connected to each other via connectors. The graph is defined by associating component ports with the connector roles in which they participate. For example, ports of filter components are associated with roles of the pipe connectors through which they read and write streams of data.

- **Properties:** In addition to defining high-level structure, most architectures also associate properties with their constituent elements[2] of an architectural model. For example, for an architecture whose components are associated with periodic tasks, properties might define the period, priority, and CPU usage of each component. Properties of connectors might include latency, throughput, reliability, protocol of interaction, etc.

To make such definitions precise, we need a formal language. In this work we use the Acme ADL [13], although many other ADLs could have been used as well. (See also Section 7.) Figure 1 illustrates the basic constructs of Acme for defining configurations.[3] The figure specifies a very simple repository architecture consisting of a single database and single client connected by a database access connector. Both components have a single port, and the connector through which they interact has two roles. The client has a single property, its average number of transactions per second.

```
System simple_repository_system: Repository = {
  Component client = {
    Port request;
    Property avg_trans_per_sec: int;
  }
  Component db = {
    Port provide;
  }
  Connector db_access = {
    Role user;
    Role requester;
  }
  Attachments = {
    client.request as db_access.user;
    db.provide as db_access.provider;
  }
}
```

Figure 1: Simple repository system

3.2 Formal Modeling of Architectural Styles

To define a style we need to add the following concepts:

- **Design vocabulary:** This can be specified as a set of component and connector types that are allowed to be used in defining a specific architecture in that style. For example, a pipe-filter style would include a pipe connector type and a filter type, a client-server style would include clients and servers, etc.

- **Constraints:** A style may also include constraints that describe the allowable configurations of elements from its design vocabulary. For example, a pipeline architecture might constrain the configurations to be linear sequences of pipes and filters. A J2EE style might restrict clients from interacting directly with the underlying database. Formally constraints are predicates over the set of all possible topologies allowed by

[2]The term "elements" refers generically to any kind of architectural structure: component, connector, port, or role.
[3]In this paper we use only those aspects of Acme that are necessary to illustrate the main points of style analysis. For a more thorough description of the language see [13].

the design vocabulary, defining which of those configurations are permitted by the style. In this respect, a style can be viewed as the "type" of a configuration.

Styles can be related to each other in various ways. One relationship is specialization. A style can be a substyle of another by strengthening the constraints, or by providing more-specialized versions of some of the element types. For instance, a pipeline style might specialize the pipe-filter style by prohibiting non-linear structures and by specializing a filter element type to a pipeline "stage" that has a single input and output port. An N-tiered client-server style might specialize the more general client-server by restricting interactions between non-adjacent tiers.

A second important relationship is conjunction. One can combine two styles by taking the union of their design vocabularies, and conjoining their constraints. For example, one might add a database component to a pipe-filter system by conjoining a pipe-filter style with a database style. In such cases it may be necessary to also define new types of components or connectors that pertain to more than one style, such as a component type that has filter-like behavior, but that can also access a database.

To specify styles formally, we again use Acme. In Acme a style is defined as a set of architectural element types together with a set of constraints specified in first-order predicate logic, augmented with some helper functions to ease the definition of architecturally-relevant predicates. Types may be subtypes of other types, with the interpretation that a subtype satisfies all of the structural properties of its supertype(s) and that it respects all of the constraints of those types. For instance, a UnixFilter component type may be declared to be a subtype of Filter. In Acme a new element type can be a subtype of several types, meaning that instances of the new type must satisfy the constraints of all supertypes.

In addition, Acme allows one to define a substyle by specializing one or more existing styles. As with element types the substyle must respect the invariants of the superstyle. When more than one style is used as a supertype, the new style must be a substyle of the *conjunction* of the parent styles.

Figure 2 illustrates the definition of a very simple repository style in Acme. The `RepositoryStyle` style includes definitions of various types of interfaces: `provide` and `use` port types for components, and `provider` and `user` roles for connectors. The `database` component type and the `access` connector provide the component and connector design vocabulary.

In the example the port and role types specify constraints (termed "invariants") that constrain attachments between ports and roles. Specifically, constraints on ports specify that a `provide` port must be attached to a `provider` role, and that a `use` port must be attached to a `user` role. The constraints on the roles specify that each role must be attached to some port. The style also includes constraints on configurations dictating that at least one database and one access connector must exist in any system in this style.

```
Style RepositoryStyle = {
  Port Type Provide  = {
    invariant Forall r:role in self.attachedRoles |
      declaresType(r, Provider);
  }
  Port Type Use = {
    invariant Forall r:role in self.attachedRoles |
      declaresType(r, User);
  }
  Role Type Provider = {
    invariant size(self.attachedPorts) == 1;
    invariant Forall p: port in self.attachedPorts |
      declaresType(p, Provide);
  }
  Role Type User   = {
    invariant size(self.attachedPorts) == 1;
    invariant Forall p: port in self.attachedPorts |
      declaresType(p, Use);
  }
  Component Type Database = {
    Port provide: Provide = new Provide;
  }
  Connector Type Access = {
    Role provider: Provider = new Provider;
    Role user: User = new User;
  }
  invariant
    Exists c: component in self.components |
      declaresType(c, Database);
  invariant
    Exists n: connector in self.connectors |
      declaresType(n, Access);
}
```

Figure 2: Repository style described in Acme

To simplify invariant specifications Acme also provides a number of built-in functions. Here `attachedPorts` and `attachedRoles` return the ports or roles (resp.) attached to a role or port (resp.), while `declaresType(e,T)` returns true if an element `e` is declared to have type `T`.

Although the example used in this paper are relatively simple, in practice styles may be quite complex. They may define a rich vocabulary of elements, and the rules for configuration may be complicated. For example, the Mission Data System (MDS) defined by NASA JPL as a style for space systems [24] includes eight component types (actuators, sensors, etc.), eight connector types, and over sixty rules constraining configurations. Figure 3 shows one such rule: it specifies that it is possible to connect only one controller to any of an actuator's ports.

```
(forall compA: ActuatorT in sys.Components |
  numberOfPorts(compA,CommandSubmitProvPortT) > 1
    -> (exists unique compC: ControllerT
     in sys.components |connected(compA, compC)))
```

Figure 3: Example constraint for the MDS style

Definition of architectural styles such as MDS is a challenging intellectual effort. The style designer must worry about providing an expressive and appropriate vocabulary, as well as making sure that the style contains appropriate constraints. (If the constraints are too strong, it will rule out systems that should be included; if too weak, it will allow configurations that should not be permitted.)

4. ALLOY

Alloy is a modeling language based on first-order relational logic [16]. An Alloy model consists of signature definitions and constraint definitions. Signatures define the basic types of elements and relations between them to be used in a model, and constraints restrict the instance space of the model. Consider the following example:

```
module publication
 sig Person {}
 sig Book {author: Person}
 sig Autobiography extends Book {}
 fact {
  all disj b1,b2:Autobiography | b1.author!=b2.author
 }
```

Three Alloy signatures are defined: `Person`, `Book`, and `Autobiography`. The `author` relation is defined over `Book` and `Person`. The `Autobiography` type is defined using signature extension as a subtype of the `Book` type.

An Alloy *fact* is a boolean expression that any instance of a model must satisfy. A fact might be *local* to a signature or *global* to the whole model. The fact in the above example prescribes that a person can't be the author of two distinct autobiographies.

The semantics of Alloy's subtyping is that of subsets. Additionally, subtypes partition the elements of the supertype: no two immediate subtypes of a type can share any element. There are two built-in types in Alloy: `univ` and `none`. The `univ` type is the supertype of all types, while `none` is the subtype of all types (and includes no elements).

Models written in Alloy can be analyzed by the Alloy Analyzer. Depending on the type of model, the Alloy Analyzer can be used as either a *prover* that finds a solution which satisfies the constraints in a given model, or a *refuter* that finds a counterexample that violates the assertions in a given model.

Since the Alloy Analyzer is a bounded checker, it guarantees the correctness of the result only within a specified finite bound of numbers of elements. To illustrate, the following module contains a predicate and a command to analyze whether the publication module (defined earlier) satisfies that predicate:

```
module analysis
open publication
pred good_world() {
  all p: Person | some b: Book | b.author = p
}
run good_world for 5
```

When executed, the Alloy Analyzer checks the predicate that it is possible for everyone to write at least one book. The command, `run good_world for 5`, directs the analysis to be performed within the bound of at most five instances for each of the top-level types (`Person` and `Book` in this example).

5. REPRESENTING STYLES IN ALLOY

We now describe how we can use Alloy to analyze properties of architectural styles. Our approach will be to provide a set of translation rules from Acme style specifications to Alloy models. After applying the rules we can then analyze various style-related properties, finally mapping counterexamples back to the Acme source.

There are three important representational requirements for any translation scheme from an architecture style specification language (like Acme) into a more general modeling language (like Alloy). First, one must be able to represent the four basic architectural element types that a style defines: component, connector, port, and role types.

Second, one must be able to represent relations between types. These fall into two sub-categories. One is containment relationships: components contain their ports, connectors contain their roles, and systems contain instances of components, connectors and attachments. The other type relationship is subtyping, which in Acme can involve multiple supertypes.

Third, one must be able to represent constraints over elements and configurations. These constraints include the invariants declared by the style. In addition they include implicit constraints, such as the facts that ports can cannot be directly connected to other ports, and that ports and roles cannot exist in isolation (i.e., independent of a parent component or connector, resp.).

We now present our translation scheme for each of these, illustrating the ideas with the repository style.[4]

We start by defining a mapping of the generic element types as a common definition module `cnc_view`. These definitions are independent of any particular style, which will allow the use of these definitions by inclusion as building blocks in any specific style definition. This shared module will also contain all of the implicit constraints on architectures that every architectural model must satisfy.

```
module cnc_view
 sig Element {Parent: lone Element,
              Attached: set Element}
 sig Component extends Element {ports: set Port}
 sig Connector extends Element {roles: set Role}
 sig Port extends Element {component: Component}
 sig Role extends Element {connector: Connector,
              attachment: lone Port}
 fact {~ports = component && ~roles = connector}
 fact {Parent = component + connector}
 fact {Attached = attachment + roles.attachment.component}
```

In the Alloy model `Element` is the supertype of all basic architectural types. It has a placeholder for a set of attachments. It also keeps track of a unique parent element, through the relation `Parent`. The other basic types provide the obvious extensions and relations. The Alloy "facts" guarantee that ports/roles have unique components/connectors as parents.

With these definitions in hand we then define the built-in functions that are used to define Acme invariant expressions.

[4]In fact, there are many possible ways of encoding styles in Alloy. Because of space limitations, we present only the final version of our translation. For a discussion of alternatives, the reasons for choosing this particular scheme, and a formal definition of the translation as a set of rewrite rules, see [18].

These are also contained in the common `cnc_view` definition module.

```
fun parent(e: Element): Element { e.Parent }
pred attached(e1: Element, e2: Element)
  { e1 -> e2 in Attached }
pred declaresType(element: Element,type: set Element)
 { element in type }
...other built-in functions...
```

Subtype relationships for element types can be translated directly into subtypes in Alloy. However, since Alloy supports only single inheritance, we are not able to use this scheme to translate Acme element type definitions where multiple inheritance is used.[5]

Using subtyping the translation of any style-specific port or role type becomes be a subtype of the built-in type (`Port` or `Role` respectively), either as an immediate subtype or a subtype of another port or role type. Similarly component and connector types become subclasses of the built-in type `Component` or `Connector`, respectively. Finally, constraints are translated as corresponding Alloy facts using the built-in functions defined in `cnc_view`, as illustrated earlier.

To see how this works consider `RepositoryStyle`, shown above. After translation of the declared types, we get:

```
module repositorystyle
open cnc_view
// translation of type definitions
 sig Use      extends Port {}
 sig Provide  extends Port {}
 sig User     extends Role {}
 sig Provider extends Role {}
 sig Database extends Component{provide:Provide}
 sig Access extends Connector
              {provider:Provider, user:User}
// translation of constraints
...
```

The constraint definitions are relatively straightforward to translate. For convenience of reference we separate these into *local* constraints on individual element types and *global* constraints that apply to the configuration as a whole.

```
pred database_constraints_local() {
  all self:Use | all r: self.~attachment |
    declaresType(r, User)
  all self:Provide | all r: self.~attachment |
    declaresType(r, Provider)
  all self:User | all p: self.attachment |
    declaresType(p, Use)
  all self:Provider | all p: self.attachment |
    declaresType(p, Provide)
}

pred database_constraints_global() {
  all r:User | #attachedPorts(r) = 1
  all r:Provider | #attachedPorts(r) = 1

  some c:Component | declaresType(c, DataBase)
  some n:Connector | declaresType(n, Access)
}

pred database_constraints_local() {
  database_constraints_local()
  database_constraints_global()
}
```

[5]As we will see, however, we *are* able to support style definitions where multiple inheritance of *styles* occurs.

Using this general strategy the translation scheme is relatively straightforward. Our current implementation carries out the translation automatically, using standard parsing and unparsing tools.

There are however, certain features in Acme which cannot be handled in Alloy using the translation scheme just outlined. These include: use of multiple inheritance for individual type definitions (as noted earlier), certain arithmetic expressions (including those that involve multiplication or negative numbers), and higher-order expressions and signatures (e.g., sequences of sets).

6. ANALYZING STYLES

We now show how translated architectural styles can be effectively analyzed using the Alloy Analyzer. Supported analyses include checking consistency of a style, checking for satisfaction of properties of a style, checking equivalence between global and local constraints, checking style compatibility, and checking whether one style refines another.

The Alloy Analyzer works by looking for an instance of a specified model within a *scope* that indicates the maximum number of elements for each top-level signature. In the case of architectural styles, the scope indicates the number of architectural elements of each type.

The strategy for analyzing translated architectural styles is to produce an analysis module that imports all the translated styles and that checks either a predicate or an assertion, depending on the type of analysis to perform (see below). The interpretation of the result will vary depending on the kind of analysis performed.

```
module analysis
open styles_to_analyze
...
run predicate_to_check for N
// or
check assertion_to_check for N
```

6.1 Checking the Consistency of a Style

A style is consistent if there exists at least one architectural configuration that conforms to the style (i.e., satisfies the style's structure and invariants). Consistency checking is important to make sure that a style's definition is internally consistent.

Although consistency errors can arise in a single style definition, more typically they occur when combining other styles, since the other styles may have been written by different people with conflicting assumptions.

Style consistency can be checked by using the Alloy Analyzer to generate a solution to the Alloy model of the style: if a model is found, the style is consistent.

Consider the previous example of the repository style. Suppose that the first two universal quantifications in the constraints had been mistakenly written as shown below (`User` and `Provider` are erroneously interchanged).

```
pred database_constraints() {
  all p:Use | all r:attachedRoles(p) |
    declaresType(r, Provider)
```

```
  all p:Provide | all r:attachedRoles(p) |
    declaresType(r, User)
  all r:User | all p:attachedPorts(r) |
    declaresType(p, Use)
  all r:Provider | all p:attachedPorts(r) |
    declaresType(p, Provide)

  all r:User | #attachedPorts(r) = 1
  all r:Provider | #attachedPorts(r) = 1

  some c:Component | declaresType(c, DataBase)
  some n:Connector | declaresType(n, Access)
}
```

The following is the Alloy analysis module that we use to check the consistency of the repository style.

```
module analysis
open repositorystyle
run database_constraints for 10
```

When the command is executed, the Alloy Analyzer reports that no instance can be found within the specified scope number. Therefore the repository style with the mistakenly written constraints above is inconsistent (for the size of model specified).

At this point we may not be sure whether the failure to find a model is the fact that bound for the analysis (here 10) is too small, or whether there is indeed a consistency. In practice, however, we find that if models can be found at all, they can be found with relatively small models.

6.2 Checking Properties of a Style

Frequently it is useful to check whether the systems that conform to a style also satisfy certain additional derived properties. This allows us to investigate whether the constraints of a style imply other desired properties.

Such a property of a style can be checked using assertion and implication. Specifically, if a property P of a style is valid for the constraints Q, the logical expression $Q => P$ should be true of every instance of the style.

Consider the example of the repository style. Let us define a property of the repository style that every instance of the style must have a component that has a `Use` port. The following is the analysis module to check this property of the repository style.

```
module analysis open repositorystyle
assert database_property_check {
  database_constraints() =>
    some c: Component | some p: c.ports |
      declaresType(p, Use)
}
check database_property_check for 10
```

When the command is executed, the Alloy Analyzer reports there is no counterexample that violates the assertion within the specified scope number. This follows from the fact there must be a connector of `Access` type that has a role of type `User`, and it must be attached to a port of type `Use` that belongs to a certain component. Therefore, the property of the repository style that there must be a component that has a `Use` port is valid.

6.3 Global and Local Constraint Equivalence

There are often several ways in which one can add constraints to a style. One is to do it at a global level. For example, Figure 3 illustrated a global constraint (or universally quantified predicate) for the MDS style that expresses a property about attachments to ports of actuators. Another alternative would have beeen to include local constraints associated directly with the ports and roles to achieve an equivalent effect.

In general, a global constraint is easier to understand and specify. However, local constraints are more efficient to evaluate incrementally by tools, and may provide better error reporting when they are violated. As a consequence, it is often useful to specify constraints locally, and then check that they collectively imply some global constraint.

The equivalence of local and global constraints can be checked using bi-implication and assertion. If a conjunction of local constraints L of a style is equivalent to a global constraint G, the boolean expression $L <=> G$ must be true for every instance of the style.

Again, consider the example of the repository style. The first four universal quantifications in the constraints are local constraints associated with types of ports and roles. Let us assume there are alternative global constraint as shown below. The following is the analysis module to check the equivalence between them.

```
module analysis
open repositorystyle
pred database_constraints_global() {
  all p: Port | all r: Role | attached(r,p) =>
    (declaresType(p, Use)     <=>
               declaresType(r, User)) &&
    (declaresType(p, Provide) <=>
               declaresType(r, Provider))

  some DataBase
  some Access
}
assert equivalence_check {
  database_constraints() <=>
        database_constraints_global()
}
check equivalence_check for 10
```

When the command is executed, the Alloy Analyzer reports that there is a counterexample that violates the assertion. This means that the local and the global constraints of the repository style are not equivalent. Closer inspection of the counterexample (not shown here) reveals that the global constraints permit a system to have unattached roles of `User` type or `Provider`, while this is prohibited in case of the local constraints. After adding the boolean expressions below to the body of the global constraints and executing the command again, the Alloy Analyzer reports that there is no counterexample within the specified scope, indicating that the local and the new global constraints of the repository style are equivalent.

```
  all p: Port | all r: Role |
    (declaresType(r, User) =>
         #attachedPorts(r) > 0) &&
    (declaresType(r, Provider) =>
         #attachedPorts(r) > 0)
```

6.4 Compatibility of Styles

Often real world systems employ multiple styles to describe a single system. Different styles can be used in various ways. One is to use different styles at different levels of architectural hierarchy, so that the "internal architecture" of a component is defined in a different style than its surroundings. In that case the encapsulation boundary of a component can insulate one style from another. However, in other cases, it is desirable to mix several styles at the same level of abstraction. In that case we might like to check that the styles can be consistently mixed together – i.e., that their constraints do not conflict.

Compatibility of a set of styles can be checked by evaluating the consistency of a new style that merges those styles. That is, a new merged style is written by importing all the styles and then checking consistency as before. The constraints of the merged style are the conjunction of the constraints of included styles. If the merged style is consistent, the imported styles are compatible.

Consider the previous example of the repository style and a new pipe-and-filter style shown below. We will check whether these two styles are compatible.

```
module pipe_and_filter
open cnc_view
 sig Input  extends Port {}
 sig Output extends Port {}
 sig Source extends Role {}
 sig Sink   extends Role {}

 sig DataSource extends Component{output: Output}
 sig DataSink   extends Component{input: Input}
 sig Filter     extends Component{input: Input,
                                  output: Output}
 sig Pipe extends Connector
                 {source: Source, sink: Sink}

fact {
  (Filter<:input + Filter<:output +
   DataSource<:output + DataSink<:input) in ports
  (Pipe<:source + Pipe<:sink) in roles
}

pred pipe_and_filter_constraints() {
  all p:Input  | all r:attachedRoles(p) |
    declaresType(r, Sink)
  all p:Output | all r:attachedRoles(p) |
    declaresType(r, Source)

  all r:Source | one p:Output | attached(r, p)
  all r:Sink   | one p:Input  | attached(r, p)

  some Filter
  some Pipe

  all c:Filter | all p:c.ports |
    declaresType(p, Input) || declaresType(p, Output)
  all n:Pipe   | all r:n.roles |
    declaresType(r, Source) || declaresType(r, Sink)
}
```

The following is the analysis module to check if the repository style and the pipe-and-filter style are compatible.

```
module analysis
open respositorystyle
open pipe_and_filter
pred compatibility_check() {
  database_constraints() &&
     pipe_and_filter_constraints()
}
run compatibility_check for 10
```

When executed the Alloy Analyzer reports that there is a solution, indicating that the repository style and the pipe-and-filter style are compatible, and that it is safe to use both of them when defining a system.

6.5 Checking Overlapping Styles

A more interesting and practical use of multiple styles in a system is to create architectural elements that have multiple types, each type taken from a different style. Such architectural elements form an overlapping zone of styles, and they must satisfy the constraints from multiple styles. It is desirable to know in advance if such an overlapping zone of multiple styles can exist.

To check if styles can overlap in this way, in addition to what was done to check if styles are compatible, it is necessary to define new types for the architectural elements in the overlapping zone and include a boolean expression that states the existence of instances of the new types and all the constraints from the imported styles. Then checking the consistency of the modified merged style is sufficient to check if the styles can overlap.

Building on the previous example of checking the compatibility of the repository style and the pipe-and-filter style, let us assume we want to have a new element that can act both as a filter and also can access the database. We need to check if such a filter can exist. The following is the analysis module to check this.

```
module analysis
open cnc_view open repositorystyle open
pipe_and_filter

sig UserFilter extends Filter {use: Use}
fact{ UserFilter<:use in ports }

pred overlapping_check() {
  some UserFilter
  database_constraints() &&
      pipe_and_filter_constraints()
}
run overlapping_check for 10
```

Note that the new filter type `UserFilter` has an extra port of `Use` type. Since Alloy does not allow multiple inheritance, we use the technique of adding new interfaces to exiting types, similar to the way the Java programming language deals with multiple inheritance. That is why we do not have a user component type in the repository style. Here new interfaces are represented by new ports.

When the command is executed, the Alloy Analyzer reports that there is no solution within the specified scope, indicating that it is not possible to instantiate the new `UserFilter` type. Closer inspection of the constraints of the pipe-and-filter style reveals that a filter can only have ports of `Input` type or `Output` type, which prevents a filter from having an extra port of `Use` type. Since the filter port constraint is stronger than is needed, we decide to remove that constraint from the pipe-and-filter style. Executing the com-

mand again, Alloy Analyzer reports there is a solution, indicating that we can use the repository style and the pipe-and-filter style with the revised `UserFilter` type.

6.6 Checking Style Refinement

A style S_r is a refinement of a style S_a if all instances of S_r also satisfy the constraints of the S_a. When a style is directly declared to be a substyle of another style, it is sufficient to check the consistency of the substyle to guarantee the refinement relation because all the constraints of superstyle also apply to the substyle automatically.

However in some situations there may be no explicitly declared substyle relation. Consider the previous pipe-and-filter style and a new Unix-pipe style. We will check whether these two styles are compatible.

```
module unix_pipe
open cnc_view

sig StdIn  extends Port {}
sig StdOut extends Port {}
sig StdErr extends Port {}
sig ErrIn  extends Port {}

sig Source extends Role {}
sig Sink   extends Role {}

sig Filter     extends Component {si: StdIn,
                        so: StdOut, se: StdErr}
sig CharInput  extends Component {so: StdOut}
sig CharOutput extends Component {si: StdIn}
sig ErrOutput  extends Component {ei: ErrIn}

sig Pipe extends Connector {source: Source,
                            sink: Sink}

fact {
  (Filter<:si + Filter<:so + Filter<:se +
   CharInput<:so + CharOutput<:si + ErrOutput<:ei)
          in ports
  (Pipe<:source + Pipe<:sink) in roles
}

pred unix_pipe_constraints() {
  all p:StdIn  | some n:Pipe | attached(n.sink,  p)
  all p:StdOut | some n:Pipe | attached(n.source,p)
  all p:StdErr | all r:attachedRoles(p) |
              declaresType(r, Source)
  all p:ErrIn  | all r:attachedRoles(p) |
              declaresType(r, Sink)

  all r:Source | some p:attachedPorts(r) |
    declaresType(p, StdOut) || declaresType(p,StdErr)
  all r:Sink | some p:attachedPorts(r) |
    declaresType(p, StdIn) || declaresType(p,ErrIn)

  some Filter && some Pipe && lone ErrOutput

  all n:Pipe | declaresType(
                attachedPorts(n.source), StdErr)
    =>
    (some c:CharOutput | attached_nc(n,c)) ||
    (some c:ErrOutput  | attached_nc(n,c))
}
```

The following is the analysis module to check if the Unix-pipe style is a refinement of the pipe-and-filter style.

```
module analysis
open pipe_and_filter
open unix_pipe
pred modified_constraints() {
  all p:(StdIn+ErrIn)  | all r:attachedRoles(p) |
    declaresType(r, Sink)
  all p:(StdOut+StdErr) | all r:attachedRoles(p) |
    declaresType(r, Source)

  all r:Source | one p:(StdOut+StdErr) |attached(r,p)
  all r:Sink   | one p:(StdIn+ErrIn)   |attached(r,p)

  some Filter
  some Pipe
}
assert refinement_check {
  unix_pipe_constraints() => modified_constraints()
}
check refinement_check for 10
```

The result is that it is indeed valid refinement for models within the specified scope.

7. CONCLUSION AND FUTURE WORK

As we have tried to illustrate, the use of a model generator, such as the Alloy Analyzer, can provide substantial benefits to the architectural style designer by providing a way to check critical properties of styles. These properties include style consistency, implied properties, refinement, equivalence of global and local constraints, and checking for compatibility between styles.

Since specification languages such as Acme have the expressive power of first-order predicate logic, such properties are in general undecidable and typically require mathematical proof. This makes it unlikely that in practice style designers will actually be able to check these properties by hand. Hence, having a semi-automated tool to assist in this effort represents a major advance.

However the approach has some limitations. First, since the model generator can only work over finite models, for many systems one can only approximate a solution. That is, if the tool says there is no problem within a given model size, it may be that this holds for all models, or only for those of that finite size. Experience has shown, however, that if a specification has a flaw, it can usually be demonstrated by relatively small counterexample.

A second potential limitation is the degree of automation. In general, a tool like the Alloy Analyzer requires the specification of both a model and a property to check against it. In our approach the model comes for free: once you have specified an architectural style, our tool automatically generates the Alloy model. However, our current implementation requires one to specify the properties that Alloy must check. In some cases this is trivial, such as checking for style consistency, but in others (such as checking whether a style implies some property or whether global and local constraints are equivalent) the style designer must specify the property to check in Alloy. In future work we hope to provide a set of automated properties specified in the Acme source language, and have the tool also automate their translation.

A third issue is the need to relate counterexamples back to the source specification. At present this requires some detailed knowledge of the details of the Alloy Analyzer and

some knowledge of the conventions used by our translator. We believe that automating the reverse translation should not be difficult, and plan to do that in future work. A more tricky issue is understanding what flaw in the design caused the counterexample to be generated in the first place. This problem is common to any model checking approach, and is an area of active research [5].

A fourth area is that of performance. While today's SAT solver-based model checkers (like Alloy) can handle a large number of variables, they are still limited in the size of the model that can be checked. Our own experience with Alloy is that when the model bound approaches 20 top-level architectural elements, or when the model contains a large number of component and connector types, it may take some time to check a property, if it is indeed tractable at all. Thankfully, most of the flaws we find in practice require relatively simple models to generate.

A final limitation of our current tool is the fact that it only deals with structural properties of architectural styles. It does not handle, for example, architectural behavior, dynamic changes to architectural models, and expressions over Acme properties and other quality attributes. However, those extensions are not intrinsic limitations, and we believe the current structural analysis techniques can be naturally extended to include other kinds of analyses. This remains an active area for future work by us and other research groups.

Acknowledgements

This research was sponsored by the US Army Research Office (ARO) under grants DAAD19-01-1-0485 and DAAD19-02-1-0389, and by the National Science Foundation under Grant No. CCR-0113810. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the ARO, the U.S. government or any other entity. We would like to thank the members of the ABLE research group at CMU for their comments on this work.

8. REFERENCES

[1] G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, October 1995.

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[3] D. J. Barrett, L. A. Clarke, P. L. Tarr, and A. E. Wise. A framework for event-based software integration. *ACM Trans. on Software Engineering and Methodology*, 5(4):378–421, October 1996.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[5] S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. In *Foundations of Software Engineering (SIGSOFT FSE)*, pages 73–82. ACM SIGSOFT, October 2004.

[6] P. Clements, F. Bachmann, L. Bass, D. GArlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison Wesley, 2002.

[7] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison Wesley Longman, 2001.

[8] The Common Object Request Broker: Architecture and specification. OMG Document Number 91.12.1, December 1991. Revision 1.1 (Draft 10).

[9] A. DiMarco and P. Inverardi. Compositional generation of software architecture performance qn models. In *4th Working IEEE/IFIP Conf. on Software Architecture (WICSA04)*, Oslo, Norway, June 2004.

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

[11] D. Dvorak and K. Reinholtz. Separating essential from incidentals, an execution architecture for real-time control systems. In *Proc. 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Austria, 2004.

[12] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proc. of SIGSOFT'94: The 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 179–185. ACM Press, December 1994.

[13] D. Garlan, R. T. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Ontario, Canada, November 1997.

[14] P. Inverardi and M. Tivoli. Deadlock-free software architectures for com/dcom applications. *Elsevier Journal of Systems and Software*, 65(3):173–183, 2003.

[15] P. Inverardi and A. Wolf. Formal specification and analysis of software architectures using the chemical, abstract machine model. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):373–386, April 1995.

[16] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineerng and Methodology*, 2002.

[17] G. Karsai and J. Sztipanovits. A model-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):46–53, May 1999.

[18] J. S. Kim and D. Garlan. Automating the analysis of architectural styles. Technical Report CMU-ISRI-06-106, Carnegie Mellon University, 2006.

[19] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. John Wiley and Sons, 1999.

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

[21] S. Microsystems. J2ee information site. URL: http://java.sun.com/javaee/.

[22] OMG. Unified modeling language. URL: http://www.uml.info/.

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

[24] R. Roshandel, B. Schmerl, N. Medvidovic, D. Garlan, and D. Zhang. Understanding tradeoffs among different architectural modelling approaches. In *Proc. of the 4th Working IEEE/IFIP Conf. on Software Architectures*, Oslo, Norway, June 2004.

[25] SAE. Sae aadl information site. URL: http://www.aadl.info/.

[26] B. Schmerl and D. Garlan. Acmestudio: Supporting style-centered architecture development. In *Proc. of the 26th International Conference on Software Engineering (ICSE)*, 2004.

[27] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[28] J. P. Sousa and D. Garlan. Formal modeling of the Enterprise JavaBeans component integration framework. In *Proc. of FM'99 – Formal Methods: World Congress on Formal Methods in the Development of Computing Systems*, number 1709, pages 1281–1300, Toulouse, France, November 1999. Springer Verlag, LNCS.

[29] B. Spitznagel and D. Garlan. Architecture-based performance analysis. In *10th International Conf. on Software Engineering and Knowledge Engineering (SEKE'98)*, San Francisco, CA, June 1998.

[30] B. Tekinerdogan and H. Szer. Software architecture reliability analysis using failure scenarios. In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 203–204, 2005.