Analyzing Architectural Styles

Jung Soo Kim * and David Garlan

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213, USA

Abstract

The backbone of most software architectures and component integration frameworks is one or more architectural styles that provide a domain-specific design vocabulary and a set of constraints on how that vocabulary is used. Today's architectural styles are increasingly complex, involving rich vocabularies and numerous constraints. Hence, designing a sound and appropriate style becomes an intellectually challenging activity. Unfortunately, although there are numerous tools to help in the analysis of architectures for individual systems, relatively less work has been done on tools to help in the design of architectural styles. In this paper we address this gap by showing how to map an architectural style, expressed formally in an architectural description language, into a relational model that can then be checked for properties, such as whether a style is consistent, whether a style satisfies some predicates over its architectural structure, and whether two styles are compatible for composition.

1 Introduction

The discipline of software architecture has matured substantially over the past decade: today we find growing use of standards [1,39,43], architecture-based development methods [9,13,32], and handbooks for architectural design and documentation [10,12]. 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 [7], performance [15], reliability [49], style conformance [44], and many others.

One of the important pillars of modern software architecture is the use of architectural styles [5,10,12,45]. An architectural style defines a family of related systems, typically by providing a domain-specific architectural design

* Corresponding author.
Email address: jungsoo@cmu.edu (Jung Soo Kim).

vocabulary together with constraints on how the parts may fit together. Examples of common styles range from the very generic, such as client-server and pipe-filter [45,10], to the very domain-specific, such as NASA's Mission Data Systems (MDS) style [18,17] and the IEEE Distributed Simulation Standard [31,2].

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 approaches, allowing the reuse of architectures across many products, and supporting component integration. By constraining the design space, they provide opportunities for specialized analysis [4]. And 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 (such as [22]).

Consequently an ever-increasing number of architectural styles are being defined. In many cases new styles are elaborations of existing styles. For example, a company might constrain Java EE-based architectures [37] to support particular types 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 needs for monitoring of automotive control software.

Defining a new style, however, is not an easy task. One must take care that the system building blocks fit together in the ways expected, 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 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 given style specifies an appropriate set of systems, whether it can be combined consistently with another, or whether it retains the essential properties of some parent style, is today largely a matter of trial and error. In fact, typically style designers 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.

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 an approach that does exactly that with respect to structural aspects of architectural styles. Specifically, we show how to map an architectural style, expressed in an architectural description language, into a relational model that can be checked tractably for various properties relevant to a style designer using existing model checkers and model generators. Additionally, significant parts of this approach can be automated, including the translation of a style into a relational model, invocation of a set of standard sanity checks on styles, and reverse translation of the output of the analysis tools (e.g., counterexamples) back into an architectural model where the results of the analysis can be viewed in terms of architectural structures.

The main contributions of this work include (a) a translation scheme from architectural style definitions into relational models, and specifically those supported by Alloy [26,27]; (b) an enumeration of a number of important classes of properties that a style designer should be concerned with, such as style consistency and compatibility of two styles for merging; (c) techniques to encode such properties in the relational model where they can be checked using tools such as the Alloy Analyzer; and (d) performance measurements that demonstrate the tractability of the techniques and explore size and complexity limitations.

In Section 2 we consider 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 a brief introduction to Alloy and the Alloy Analyzer, the target modeling language and analysis tool that we will be using to check properties of styles. Section 5 provides an overview of the approach and tools. Section 6 then presents the translation approach, showing how to map formal descriptions of architectural styles into Alloy, and highlighting places where that translation is non-trivial. Section 7 examines a variety of important style analyses and shows how to map these into Alloy where the Alloy Analyzer can be used to check them. Section 8 demonstrates how a realistic architectural style can be analyzed using the approach presented in this paper and explores some of the performance issues related to scaling to large-scale styles. Section 9 considers how the performance of the analysis scales with respect to several model complexity dimensions. Section 10 discusses the strengths and limitations of the approach, and considers future work.

2 Related Work

There are three broad areas of 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) [35]. 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 [7,34] to specify abstract behavior of an architecture, and to check for properties like deadlock freedom of connector specifications. Others have used rewriting rules [25], sequence diagrams [24], event patterns [33], and many other formal underpinnings.

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 architectural 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 [42], performance [15,47], and reliability [43]. While many of these analyses assume that a system is described in a particular style (such as one amenable to rate-monotonic analysis or queuing theoretic modeling), unlike our work, the issue of analyzing *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 carried out by Abowd et al. [5] modeled styles using Z [48]. In that approach one can specify general properties of architectural styles, but the work lacks explicit guidance on *what* properties should be evaluated, and it does not provide any tool-assisted support for analysis. Other work has investigated formal properties of particular styles, such as EJB [46] or publish-subscribe [16], but these have not provided any general style-independent 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, in particular [20]. 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 these have been targeted to standard general-purpose modeling notations, such as UML [39], Z [48], or B [6], as opposed to domain-specific modeling languages, such as architectural description languages. In this work we build on these general specification languages using Alloy [26], one such general-purpose modeling language, as the "assembly language" for our own analyses.

Also closely related to our research is work on model-based software engineering [40]. In particular, the work by Karsai et al. [28] adopts a similar approach to meta-modeling in which new kinds of modeling languages and analyses can

be defined for a particular domain. This 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 their approach, 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.

A third related area is that of formal languages [23]. Since an architectural style defines a kind of design language, some of the problems of understanding a style are related to those of understanding formal languages in general. For example, determining whether a given sentence belongs to the language generated by a grammar is similar to determining whether the architecture of an individual system conforms to an architectural style. Moreover formal language results have been extended to graphs grammars, which can also be used to represent architectural styles [36]. However, there is a significant difference between the kinds of properties that can be checked over such grammars and the properties we explore in this work. Specifically, the constraint languages for defining architectural styles in ADLs like Acme include full first-order predicate logic, rendering many properties of interest undecidable [38]. As a result, in this work we take a pragmatic approach to analysis of styles, by investigating the use of model checking (or more accurately, model generating) tools for carrying out analyses.

3 Architectural Style

Software architecture is concerned with the high-level structure and properties of a system [45,41]. Over time there has emerged a general consensus that modeling of complex architectures is best done through a set of complementary views [12,1]. 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 communications 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 [5,10,12,45]. We refer to these specialized modeling languages as *architectural styles*.[1]

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

Architectural styles have a number of significant benefits. First, styles promote design reuse, since the same architectural design is used across a set of related systems. Second, styles can lead to significant code reuse. For example many styles (like Java EE, .Net, and most commercial service-oriented architectures) provide prepackaged middleware to support connector implementations. Similarly, Unix-based systems adopting a pipe-filter style can reuse operating system primitives to implement task scheduling, synchronization, and communication through buffered pipes. Pipe-filter styles are also seen in modern web-based applications, such as Yahoo Pipes [3]. 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 a client-server style immediately conveys an intuition about the kinds of pieces in the system and how they fit together. Fourth, styles support component interoperability. Examples include CORBA component-based architectures [14] and event-based tool integration [8]. Fifth, by constraining the design space, an architectural style often permits specialized analyses. For example, it is possible to analyze certain 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 is a large number 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 creation of new styles. This is because product lines and frameworks implicitly define a family of systems, where architectural issues (such as what kinds of components can be added, and how they are composed with other components) are a significant part of the design. Many of these styles are specializations or combinations of existing styles. For example, a company specializing in inventory management might provide a specialization of Java EE 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 [35], as well as UML 2.0 [39].

- **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 might have reader and writer roles; 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 the 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, however, we need a formal language. In this work we use the Acme ADL [21], although other ADLs could have been used instead. Figure 1 illustrates the basic constructs in Acme for defining configurations. [3] The figure specifies a very simple repository architecture consisting of two components – a database (`db`) and a client (`client`) – connected by a database access connector (`db_access`). Each component has a single port (`request` and `provide`), and the connector through which they interact has two roles (`user` and `provider`). The client has a single property (`avg_trans_per_sec`), its average number of transactions per second.

Implicit in most ADLs are a set of basic common structural constraints on how ports, roles, components, and connectors may be combined. In Acme these include the following constraints:

(1) Each port is associated with a unique component. That is, there are no orphaned ports, and no port may be part of more than one component.
(2) Dually, each role is associated with a unique connector.
(3) A role may be attached to at most one port.
(4) Ports may *only* be attached to roles, and vice versa. Hence, components

[2] We use the term "elements" to refer generically to any kind of architectural structure: component, connector, port, or role.
[3] In this paper we discuss only the aspects of Acme that are necessary to explain our approach to style analysis. For a more details on the language see [21].

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

Fig. 1. Simple repository system

may only interact with other components through connectors, and connectors may not be directly connected to other connectors. [4]

Other potential topological restrictions are left unconstrained, but may be introduced later when defining specific styles, as noted in the next section. These include things such as the permissibility of unattached ports, unattached roles, the existence of at least one port on a component or one role on a connector, whether more than one role may be attached to a port, etc.

3.2 Formal Modeling of Architectural Styles

To define a style we add the following concepts:

- **Design vocabulary:** This can be specified as a set of component, connector, port and role types that are available for defining a specific architecture in that style. For example, a pipe-filter style would include a pipe connector type and a filter type, and a client-server style would include a client type and a server type, etc. Additionally, a style may define a set of property types. For example, a property type `BufferSize` might be defined to represent the buffer size of a pipe.
- **Constraints:** A style may also include constraints that describe the allowable configurations of elements from its design vocabulary. Some constraints may restrict overall topology. For example, a pipeline architecture might constrain the configurations to be linear sequences of pipes and filters. Or,

[4] This restriction is relaxed in some ADLs, such as C2, where components may be directly connected to other components[35].

a N-tiered style might restrict clients from interacting directly with a back-end database. Other constraints may be associated with specific elements. For example, a client-server connector may be constrained to connect only a client and a server. Constraints are therefore used to restrict the possible configurations that may be created using the design vocabulary. Thus constraints determine the set of possible systems that can be created using the building blocks defined by the design vocabulary.

A style can be formally specified as a set of architectural element types together with a set of constraints specified in first-order predicate logic. 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 supertypes. For instance, a UnixFilter component type may be declared to be a subtype of Filter, adding an additional constraint that there are exactly three ports, one each of type *stdIn*, *stdOut*, and *stdErr*.

A style naturally determines a collection of possible configurations: namely, those whose element types are defined by the style, and whose overall structure satisfies the predicates associated with the style. We say that a system *conforms to* a style if it is a member of the family of configurations determined by that style.

Styles can be formally 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 (i.e., subtypes) 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. As with element types, a substyle must respect the constraints of its superstyle, and hence defines a subset of the possible systems that the superstyle represents.

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.

Figure 2 illustrates the definition of a 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 type provide

```
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);
}
```

Fig. 2. Repository style specified in Acme

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 – that is, there are no dangling connectors. 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. Although not illustrated in this example, one can also specify properties in type definitions. For example, the `Provide` port type might include a property such as `max-trans-per-sec`, with the meaning that

```
System simple_repository_system: RepositoryStyle = {
  Component client = {
    Port request: Use = new Use;
    Property avg_trans_per_sec: int;
  }
  Component db:Database = new Database;
  Connector db_access:Access = new Access;
  Attachments = {
    client.request as db_access.user;
    db.provide as db_access.provider;
  }
}
```

Fig. 3. Repository system specified using the style

every instance of this port must provide a value for this property.

As noted earlier, constraints are first-order predicates. Acme uses a notation similar to UML's OCL to specify these, for example, representing universal and existential quantification with `forall` and `exists`.

To simplify specifications of constraints Acme also provides a number of built-in functions. For example, in Figure 2 `attachedPorts` returns the ports attached to the role represented by `self`, while `declaresType(e,T)` returns true if an element `e` is declared to have type `T`. The term `self` refers to the entity to which the constraint is associated.

To see how this style would be used, Figure 3 shows how the system of Figure 1 would be described using the style. Note how the declaration of that system is simplified, at the same time making explicit its commonality with other systems that use the same style.

Although the example used above is relatively simple, in practice styles may be quite complex. They may define a rich vocabulary of elements, and the rules for configuration may be numerous. For example, the Mission Data Systems (MDS) defined by NASA JPL as a style for space systems [18,17,42] includes nine component types (actuators, sensors, etc.), twelve connector types, and seventy nine constraints over configurations. Figure 4 shows one such constraints: when associated with an MDS system, it specifies that it is possible to connect at most one controller to any of an actuator's ports. (We will revisit this style in Section 8.)

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

Fig. 4. Example constraint of 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 [5] is a modeling language based on first-order relational logic [26,27]. An Alloy model represents a set of instances, and is defined using *signatures* and *facts*. Signatures provide the vocabulary of a model by defining the types of objects in the model and relations between those objects. Facts are Boolean expressions that every instance of a model must satisfy, thus restricting the instance space of the model. Consider the following Alloy model:

```
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` signature is defined using signature extension as a subtype of the `Book` type. A global fact in the above example prescribes that a person cannot write two autobiographies. It uses the Alloy keyword `disj` which enforces the two quantified objects are distinct.

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

Models written in Alloy can be analyzed by the Alloy Analyzer, which searches for an instance of an Alloy model. Depending on the type of command to execute, the Alloy Analyzer can be used either as a prover which finds a *solution* that satisfies a given *predicate*, or as a refuter which finds a *counterexample* that violates a given *assertion*.

[5] For this research we used Alloy Version 4.

The Alloy Analyzer is a bounded checker, guaranteeing the correctness of the result only within a bounded instance space. The maximum number of objects for each top-level type can be specified when giving the Analyzer a command to execute.

Consider the following Alloy model, which extends the previously defined `publication` model by additionally defining a command to execute:

```
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 searches for a solution that satisfies the predicate `good_world`, which claims that it is possible for every person to write at least one book. The solution should have no more than five objects for each top-level type (`Person` and `Book` in this example).

We will introduce additional details of the Alloy language as necessary to explain our use of it for modeling architectural styles. For further information about Alloy refer to [27].

5 Overview of Approach

As illustrated in Figure 5, our overall approach is to provide a translation scheme to map Acme style specifications into Alloy models. In addition, we provide a way to specify (in architectural terms) analyses that we wish to carry out on that style. These are translated into Alloy analysis specifications that, in combination with the translated style and a set of baseline shared definitions (`Common Model` in the figure) can be performed by the Alloy Analyzer. Results from the Alloy analysis are then automatically mapped back to architectural specifications in Acme, where style designers can use them to improve their style specifications.

In the next section we examine in detail how we map styles to Alloy. Examples of common analyses for architectural styles and their translation to Alloy are discussed in Section 7.

6 Representing Styles in Alloy

We now present our style translation scheme, illustrating the ideas with the repository style presented in Section 3. We begin with a high-level overview and then consider in more detail how styles are translated.

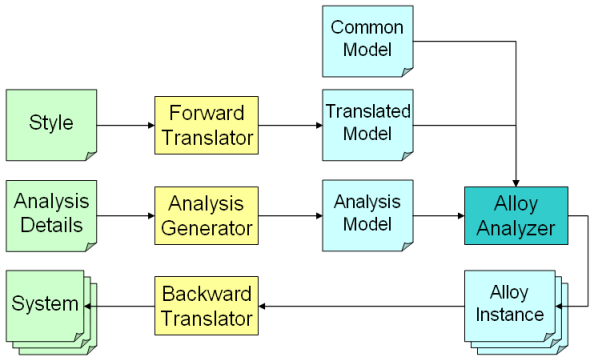


Fig. 5. Overall flow of style analysis

To translate a style into a relational model such as Alloy one must handle four things.

1. **Basic architectural types and relations:** Our approach is to encode the four basic element types (components, connectors, ports and roles) as top-level Alloy *signatures*. Containment and attachment relations are encoded as *binary relations* between the respective types. A system can then be encoded as a top-level signature containing sets of components and connectors.

2. **Style-specific architectural types:** The element types defined by a specific style are modeled as Alloy signatures *extending* the signature of their respective supertypes. For example, the `Database` type in the repository style is modeled as a signature extending the `Component` signature. This approach naturally accommodates new architectural types defined as subtypes of other style-specific types. Here we exploit the crucial point of agreement between an ADL (like Acme) and Alloy that subtyping is viewed as a subset relationship.

3. **Constraints:** Architectural constraints are modeled in a straightforward way as Alloy formulae, since both are expressed in predicate logic. Common constraints (in the `Common Model` of Figure 5) are modeled as Alloy *facts* that must be satisfied. Style-specific constraints are modeled as Alloy *predicates* that can be selectively enabled depending on the type of analysis.

4. **Functions and Expressions:** As noted earlier, Acme provides a set of built-in functions (such as `connected(c1,c2)` that test for the existence of a direct connection between two components) that are crucial for making it easy to express constraints in natural architectural terms. To handle these, we use Alloy *functions*. Additionally one must handle the data types in constraint expressions. Integers and sets are straightforward, because Alloy supports these directly. However, care must be taken for other expressions like floats, strings.

To carry out the translation we begin by defining a common Alloy model, `cnc_view`, that models the generic style-independent architectural types (component, connector, port, role, and system) and the implicit constraints that every architectural model must obey (as enumerated in Section 3.1). Next we translate an individual style into an Alloy model that imports the common model so that generic style constraints are enforced in the translated model. If a style is derived from other styles, those superstyles must also be translated into their own models, and imported into the translated substyle.

Mechanically, the translation of a style is a multi-step process. First the target style is parsed. The parsed style is then preprocessed to mitigate some of the modeling incompatibilities between Acme and Alloy and to reduce the size of the translated model. For example, as explained later, certain Acme data types that are missing in Alloy are converted into integers. The translation scheme is then applied to the preprocessed Acme style. Once translated, the resulting Alloy model is optimized to make analysis run faster.

In the remainder of this section we first show how generic element types and implicit constraints are translated into Alloy. Then we show how a style is translated. [6] Along the way, we consider some of the areas where style translation is problematic, or where there are alternative approaches for translation that might have been chosen.

6.1 An Alloy Model for Shared Architectural Concepts

We model the basic element types as top-level Alloy signatures. The architectural relations, specifically containment and attachment relations, are modeled as Alloy relations. Additionally, Alloy facts are added to enforce implicit architectural constraints. Consider the following Alloy model:

```
module cnc_view
  sig Component {ports: set Port, system: System}
  sig Connector {roles: set Role, system: System}
  sig Port {comp: Component}
  sig Role {conn: Connector, attachment: lone Port}

  one sig self extends System {}
  abstract sig System {components: set Component,
                       connectors: set Connector}

  fact {~ports = comp && ~roles = conn}
  fact {~components = Component<:system}
```

[6] In this paper we define the translation rules informally. Formal translation rules can be found in [30].

```
fact {~connectors = Connector<:system}
```

There are four top-level signatures to model the basic element types and a `System` signature to model a system. Note that `System` signature is defined as abstract, meaning that the signature cannot have an object without explicitly extending it. In the model above the `self` signature extends the `System` signature, and it is defined with the Alloy keyword `one`, thereby restricting the signature to be a unique. This naming scheme allows us to use the term `self` when we want to refer to the system being analyzed.

Containment relations (e.g., between ports and components) are defined as Alloy relations. First, `ports` and `roles` relations identify the set of ports or roles a component or connector has, respectively. `comp` and `conn` relations identify the parent component or connector that a port or role belongs to. Likewise, `components`, `connectors`, and `system` relations model the containment relation between components or connectors and systems.

The attachment relation between ports and roles is defined as the `attachment` relation, which identifies a port to which a role may be attached. Since, in general, a role need not be attached to a port, it is modeled as an optional relation using the Alloy keyword `lone`, which indicates zero or one.

By themselves, the signatures of `cnc_view` model above provide the basic vocabulary of generic architectural structure. They naturally encode the fact that ports are part of components. (For example, we may access the ports of a component `c` with the expression `c.ports`.) The same applies to connectors and roles.

However, we also need to make sure that any declared port or role is associated with the right component or connector, respectively. To do this we add an additional `fact` in the model, which states that `comp` is the inverse relation of `port` and that `conn` is the inverse relation of `roles` by using the Alloy inverse relation operator `~`. This fact guarantees that if a port appears in the set of ports owned by a component, that component will also be the parent of the port. The same applies to a role and a connector. Similarly, it is guaranteed that if a component or connector belongs to a system, the system will also be the parent of the component or connector. The Alloy operator `<:` denotes domain restriction, which is used here to disambiguate two `system` relations.

This encoding is certainly not the only way of modeling the basic architectural concepts in Alloy. In an earlier version of our translation scheme, for example, we included a shared supertype of all architectural element types, called `Element` [29]. However, we found that removing the shared supertype gave more control over the search space when executing the Alloy Analyzer, resulting in larger scope of analysis in the same execution time. In our current approach `Element` is translated into the union `Component+Connector+Port+Role`

for the same purpose.

Another alternative would be to remove redundant relations. In the model above, for example, `ports` and `comp` depend on each other. Therefore, formally speaking, one is redundant and could be removed by replacing its occurrence with the inverse relation of the other. If `comp` is removed, however, one would have to add additional facts to ensure that ports and roles have unique owners. Without those facts the inverse of `ports` may not be a function, i.e., a port may belong to more than one component or to no component at all. The same reasoning for redundant relations applies to other containment relations.

6.1.1 Built-in functions

As illustrated earlier, Acme provides a collection of built-in functions that are used for various purposes in constraint expressions: to check type conformance or structural connectivity, to access the parent of an element, or to manipulate sets. For example `attached(r,p)` returns true if role `r` is attached to port `p`.

Alloy allows one to define functions, thereby providing a natural way to encode such built-in functions. An exception is that for the built-in functions returning a Boolean value we use Alloy predicates rather than functions that return a Boolean. Since Alloy has no built-in support for Booleans, one would otherwise have to create these as user-defined signatures. Hence the use of predicates makes analysis more efficient, not requiring extra signatures and the need to encode (and decode) Boolean values.

The following are the Alloy translations of the built-in functions used in the examples of this paper.[7] These predicates augment the `cnc_view` model presented above.

```
pred declaresType [element: univ, type: set univ] {
  element in type
}
pred attached [r: Role, p: Port] {
  r -> p in attachment
}
pred attached [n: Connector, c: Component] {
  n -> c in roles.attachment.component
}
pred connected [c1: Component, c2: Component] {
  some c1.ports.~attachment.connector & c2.ports.~attachment.connector
}
```

The Alloy operator `->` represents a binary relation between scalars. The operator `in` tests the subset relation. The dot operator is relational join. In the pred-

[7] For the full translations of built-in functions see Appendix A.

icates above, the `roles` relation is from `Connector` to `Role`, the `attachment` relation is from `Role` to `Port`, and the `component` relation is from `Port` to `Component`. Therefore, the `roles.attachment.component` relation is from `Connector` to `Component`.

As illustrated above, some built-in functions are overloaded. Here `attached` can be applied to a role and a port, returning true if they are attached. It can also be applied to a component and a connector, returning true if a role of the connector is attached to a port of the component. Because these functions have different parameter types, Alloy can resolve these overloaded function names automatically.

6.2 Translating a Style

We now look at how we translate an Acme style into an Alloy model using the repository style introduced in Figure 2 as a running example.

The translation of a style consists of the translation of its types and its constraints. Note that the common model `cnc_view`, which contains basic definitions for built-in features of Acme language, is imported using the `open` command. The overall structure of a translated Alloy model will look like this:

```
module RepositoryStyle
open cnc_view
// translations of type definitions
...
// translations of constraints
...
```

6.2.1 Translating type definitions

After translation each new port and role type becomes an Alloy signature extending the top-level signature, `Port` or `Role` respectively, either as an immediate subtype or a subtype of another port or role signature. The same is true for each component and connector type with respect to the top-level signature `Component` or `Connector`, respectively.

To illustrate, the translation of the port, role, component and connector types of the repository style is as follows:

```
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}
```

When a subtype is defined in Alloy, all the fields defined in its supertypes are automatically made available in the subtype definition. For example, suppose there is a role `user` of `User` type. Then an expression like `user.attachment` is valid since `attachment` is defined in the supertype `Role`.

The same principle holds for style-specific types that are defined as subtypes of other style-specific types: fields of the supertype are available to the subtype. Thus we exploit the natural congruence between "subtyping as subsetting" that exists in both Acme and Alloy.

One important issue for this translation step is that the uniqueness of ports and roles is not guaranteed with the translation above. Suppose there are two instances `a1` and `a2` of the `Access` connector type. With the model above it is possible that `a1.user` and `a2.user` could refer to the same role. Furthermore, it is also not enforced that the type-specific ports and roles are included in the `ports` and `roles` relations defined in the `Component` and `Connector` signatures, respectively. Hence, to enforce the rules of valid structure extra constraints are needed. Consider the additional Alloy fact added to the translation above:

```
fact {
  (Database<:provide) in ports
  (Access<:provider + Access<:user) in roles
}
```

The Alloy binary operator `<:`, denoting domain restriction, is used to disambiguate overloaded relations by giving a specific domain, which in this case is either a component type or a connector type. The keyword `in` denotes set inclusion. Thus by using two such facts we can guarantee that all the type-specific ports and roles are included in the `ports` and `roles` relations. As a result, `a1.roles` always gives all the roles that belong to just the connector `a1` regardless of its type. The same is true for components.

From this it can be inferred that all such type-specific ports and roles are unique. Recall that the inverse of the `ports` and `roles` relations, which are `comp` and `conn` relations respectively, are functions. Thus, there cannot be a port or a role that belongs to more than one component or one connector.

Every port or role of components or connectors in a style is similarly included in Alloy facts as above. The translation function automatically generates the facts by scanning the type definitions of a style.

6.2.2 Translating property types and expressions

Like other ADLs, Acme supports various property types. Primitive property types in Acme are integer, string, float, enumeration, and boolean. Compound type constructors supported in Acme are set, sequence, and record. Alloy, on the other hand, currently supports only integer, boolean, and object expressions. Thus, the translator must map the property types and expressions of Acme into the supported types in Alloy.

During preprocessing, string, enumeration, and float types are converted into integer types, while their values are encoded as integer values. Thus after preprocessing there remain only two primitive property types and expressions to translate: integer and boolean. During the translation both types are translated into integer object types in Alloy.

To translate Acme's three compound property type constructors (set, sequence, record) is straightforward: set and sequence types are directly translated into the corresponding types in Alloy; record types are translated into a new signature with each field recursively translated as necessary.

6.2.3 Translating constraints

Translation of Acme constraints is straightforward because constraints are modeled as boolean expressions in both Acme and Alloy. When carrying out the translation we translate "local" constraints separately – those that are attached to individual types – from "global" constraints – those that are included at the system level. Each of these is included in its own named predicate. This is done to support analyses that need to refer to these separately, such as showing that some global constraint is implied by the local constraints.

To illustrate, the following Alloy constraints represent the translation of the local and global constraints in the repository style:

```
pred RepositoryStyle_constraints_local[] {
  all self: Provide |
    (all r: self.~attachment | declaresType[r, Provider])
  all self: Use |
    (all r: self.~attachment | declaresType[r, User])
  all self: Provider |
    (#(self.attachment) = 1) &&
    (all p: self.attachment | declaresType[p, Provide])
  all self: User |
    (#(self.attachment) = 1) &&
    (all p: self.attachment | declaresType[p, Use])
}
pred RepositoryStyle_constraints_global[] {
  some c: self.components | declaresType[c, Database]
```

```
  some n: self.connectors | declaresType[n, Access]
}
pred RepositoryStyle_constraints[] {
  RepositoryStyle_constraints_local()
  RepositoryStyle_constraints_global()
}
```

The translation of each local constraint always begins with a universal quantification using `self` as the name of bound variable. Use of the variable `self` simplifies the translation functions because it is a built-in object name in Acme, representing the element that the local constraints are applied to. Using this approach, constraints can be translated in a uniform way regardless of whether they are local or global.

6.3 Optimization

A key issue when using any tool that relies on exhaustive state exploration is tractability. If representations are not chosen carefully, the size of model that can be explored in a reasonable amount of time may be seriously limited.

While we defer the discussion of overall performance analysis until later, we note that an important step in our translation scheme is optimization. One notable optimization that we apply is to exploit Alloy's built-in capability to perform relational joins. We can use this capability to eliminate certain forms of quantifier nesting. [8]

Specifically, a common form of constraint in a style definition is to specify restrictions on ports or roles. For instance one might restrict the types of ports that can be placed on a given type of component. Because a port or role can only be accessed through its parent, such constraints result in two levels of quantification, as illustrated below.

```
invariant forall c:T in self.components |
            forall p in c.ports | predicate(p)
```

This constraint asserts that `predicate(p)` must hold for every port of every component of `T` type. (Here `self` refers to the system as a whole, so that `self.components` is the set of components in that system.) Note that two levels of quantification are needed, and the quantified variable `c` is not used in the body of the predicate. Without optimization the corresponding Alloy predicate will look like this:

```
all c in (self.components & T) | all p in c.ports | predicate[p]
```

[8] As we will see later, quantifier nesting causes an exponential increase in analysis time.

During optimization, however, the outer quantification is removed, and the occurrences of its bound variables are replaced with the expression for its range. Thus, after optimization the predicate above is simplified to this:

```
all p in (self.components & T).ports | predicate[p]
```

Specific examples of this kind of optimization are illustrated in Section 7.5.

7 Analyzing Styles

We now show how translated architectural styles can be analyzed using the Alloy Analyzer. Our approach is to provide style designers with an *analysis generator* that can generate various forms of Alloy specifications for analysis. Supported analyses include checking of:

- consistency of a style
- constructability of specific architectural configuration
- properties of a style
- equivalence of global and local constraints
- style compatibility
- consistency of hybrid types

To carry out an analysis over a style the analysis generator takes one or more styles as input, together with a specification of predicate to check against instances of that style. The analysis generator can also take as input a predicate, and then checks that the predicate implies the satisfaction of the constraints of the styles. The output of the analysis generator in this case is an Alloy model that imports the translated styles and includes a command to test the predicate.

The Alloy Analyzer will take the generated Alloy model and execute the included command over the instances of the translated style, each of which corresponds to a constructible system using the style. There are conceptually two types of command: one is to find an instance that passes the test, and the other is to find a counterexample that fails the test. These correspond to different kinds of Alloy analysis models, as detailed later. In both cases, instances are back-translated to systems of the original style for further investigation.

As noted in Section 4 the Alloy Analyzer is a bounded checker and therefore requires the user to set scope of analysis in the form of the maximum number of objects for each top-level signature. Under our translation scheme there are five top-level signatures: component, connector, port, role, and system. The system signature is constrained to be a singleton, reflecting the fact that we are interested in analyzing individual systems. A style designer can specify the bounds of the other top-level elements when generating an analysis model. If

left unspecified, the analysis generator uses a default bound of 10 components, 10 connectors, 20 ports, and 20 roles. [9]

Let us now see how this works for specific kinds of analyses that are relevant to style designers.

7.1 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., that satisfies the style's architectural constraints). Consistency checking is the most basic analysis to make sure that there are no internal contradictions within a style.

Style consistency can be checked simply by using the Alloy Analyzer to find an instance of the translated model of the style: if an instance is found, the style is consistent. If no instance can be found, the style cannot be satisfied by any model up to the specified bound on model size.

Consider the previous example of the repository style shown in Figure 2. Suppose the first two universal quantifications of the constraints had been mistakenly written, erroneously interchanging `User` and `Provider`:

```
Style RepositoryStyle = {
  Port Type Provide = {
    invariant
      forall r in self.attachedRoles | declaresType(r, User);
  }
  Port Type Use = {
    invariant
      forall r in self.attachedRoles | declaresType(r, Provider);
  }
  ...
}
```

The input to the analysis generator is simply:

```
Check RepositoryStyle
```

The output of the analysis generator is the Alloy description:

```
module analysis
  open cnc_view
  open RepositoryStyle
  run  RepositoryStyle_constraints for 10 but 20 Port, 20 Role
```

[9] It is usually the case that the number of ports and roles required to construct a system is more than the number of components and connectors.

Since no specific bounds were indicated the scope is taken to be the default: up to 10 components and 10 connectors, and up to 20 ports and 20 roles. When executed, the Alloy Analyzer reports that no instance can be found within the specified scope. At this point a style designer must decide whether to investigate the style for conflicting constraints or to expand the scope of search.

7.2 Checking the Constructability of Specific Architectural Configurations

While consistency checking formally establishes the existence of *some* systems in the style, in most cases this is not as strong a check as we might like. In particular, we may want to explore whether there exist systems that include *certain kinds of structure*. Such structures may represent configurations that we would expect to be consistent with the style specification. Or, they might be configurations that we would expect *not* to be representable in the style.

To check if a specific architectural configuration is constructible using a style, it is necessary to translate the architectural configuration of interest into Alloy facts that require this configuration be present in the instances of the translated style. If the Alloy Analyzer can find an instance of the translated style, supplemented with the additional facts, the specific architectural configuration is constructible.

As an example, suppose that the repository style designer wants to make sure that it is possible for two components to access a database at the same time. To check this, we specify as input to the analysis generator the following command. (Note the use of Acme syntax to specify the structure.)

```
Check RepositoryStyle Using

  component c1 = {port use:Use = new Use;}
  component c2 = {port use:Use = new Use;}
  connector a1:Access = new Access;
  connector a2:Access = new Access;
  component d:Database = new Database;

  attachment a1.user to c1.use;
  attachment a2.user to c2.use;
  attachment a1.provider to d.provide;
  attachment a2.provider to d.provide;
```

The analysis generator will create an Alloy model that augments the translated style with Alloy facts that require any instance of the translated model to have two components connected to the same database. Specifically, the following Alloy model will be produced:

```
module analysis
  open cnc_view
  open RepositoryStyle

  one sig c1 extends Component { use: Use }
  one sig c2 extends Component { use: Use }
  one sig a1 extends Access {}
  one sig a2 extends Access {}
  one sig d extends Database {}

  fact { c1<:use + c2<:use in ports }
  fact { a1.user.attachment = c1.use
         a2.user.attachment = c2.use
         a1.provider.attachment = d.provide
         a2.provider.attachment = d.provide
  }
  run RepositoryStyle_constraints for 10 but 20 Port, 20 Role
```

When executed, the Alloy Analyzer returns an instance that satisfies the additional facts, confirming that it is possible to have two components connected to the same database.

In a similar way one can check that some undesirable configuration cannot exist. For instance, to check that no architecture in this style can have two databases the style designer would request the system to produce an instance of a system in this style that *does* have that property. Any system produced by Alloy will represent a counterexample.

7.3 Checking Properties of a Style

While constructability checking demonstrates the existence of a system (in a given style) that includes specific structures, it requires style designers to explicitly describe those structures. As a more general alternative, style designers can check if certain *properties* of a style hold for all of its constructible systems.

A property of a style can be checked by invoking the Alloy Analyzer with an assertion that all instances of the translated style must satisfy the property. The Alloy Analyzer then attempts to find a counterexample – i.e., an instance that does not satisfy the property. If one is found it indicates that the property is not guaranteed by the style.

Consider the property of the repository style that every system of the style must have at least one component with a `Use` port. The input to the analysis generator is as follows. (Note the use of an Acme predicate to specify the

property.)

```
Check RepositoryStyle Satisfies
  exists c in self.components |
    exists p in c.ports | declaresType(p, Use)
```

The output of the analysis generator is the following analysis model, which asserts that all instances of the repository style within the scope of analysis have at least one component that has a `Use` port. [10]

```
module analysis
  open cnc_view
  open RepositoryStyle
  check { RepositoryStyle_constraints[] =>
    some p: self.components.ports | declaresType[p, Use]
  } for 10 but 20 Port, 20 Role
```

When executed the Alloy Analyzer reports that there is no counterexample that violates the assertion within the specified scope. As an aside, note that we would expect this result to be the case, since the Acme constraints require that there be a connector of `Access` type that has a role of `User` type, and that it must be attached to a port of `Use` type. Thus, there must be a component that has a `Use` port.

7.4 Checking Global and Local Constraint Equivalence

There are often several ways in which one can incorporate constraints into a style. One is to do it at a global level. For example, Figure 4 illustrated a global constraint (or universally quantified predicate) for the MDS style that expresses a property about all attachments to ports of actuators. An alternative is to include local constraints associated directly with the ports and roles to achieve an equivalent effect.

In general, global constraints are 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 sometimes useful to specify constraints locally, and then check that they collectively imply some global constraint. For this reason we may be interested in checking the equivalence of global and local constraints.

Consider the example of the repository style, where each port and role type has local constraints restricting the way they can be attached – specifically that `Use` roles can only be attached to `User` ports, and that `Provide` roles can

[10] The nested existential quantifications have been simplified using the optimization technique explained in Section 6.3.

only be attached to `Provider` ports. Suppose the style designer is considering an alternative using global constraints to specify the same thing. The style designer can check the equivalence between these two forms as follows:

```
Check RepositoryStyle Locally Iff
  forall c in self.components | forall p in c.ports |
  forall n in self.connectors | forall r in n.roles |
    attached(r,p) =>
    (declaresType(p, Use)     <=> declaresType(r, User)) &&
    (declaresType(p, Provide) <=> declaresType(r, Provider))
```

As the result the following Alloy specification will be generated. The local constraints of the repository style should imply the alternative global version, and vice versa. Note that the original global constraints of the style are also included in the assertion.

```
module analysis
  open cnc_view
  open RepositoryStyle
  check {
    RepositoryStyle_constraints_global[]
    RepositoryStyle_constraints_local[] <=>
    { all p: self.components.ports |
      all r: self.connectors.roles |
        attached[r,p] =>
        (declaresType[p, Use]     <=> declaresType[r, User]) &&
        (declaresType[p, Provide] <=> declaresType[r, Provider])
    }
  } for 10 but 20 Port, 20 Role
```

When executed, the Alloy Analyzer reports that there is a counterexample that violates the assertion. This means that the local constraints of the repository style and the alternative in global format are, in fact, not equivalent. Inspection of the counterexample reveals that the alternative global constraints permit a system to have unattached roles of `User` or `Provider` type, while this is prohibited by the original local constraints.

7.5 Checking Style Compatibility

It is often the case that real world systems combine multiple styles. Unfortunately, not all styles can be used together because they might not be compatible. Two styles are *compatible* if any system satisfying the constraints of one style also satisfies the constraints of the other. Note that this definition still allows each style to have its own types and constraints over them. But where constraints refer to types in both styles, they must be consistent. In other words, when styles are compatible, any system of an individual style should

still be constructible using the merged set of styles.

Consider the case of merging the previously defined repository style and a new pipe-filter style shown below (after translation and simplification). We will test if these two styles are compatible.

```
module PipeAndFilter
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 PipeAndFilter_constraints[] {
    all r:Input .~attachment | declaresType[r, Sink]
    all r:Output.~attachment | 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 p:Filter.ports | declaresType[p, Input] || declaresType[p, Output]
    all r:Pipe.roles   | declaresType[r, Source]|| declaresType[r, Sink]
  }
```

The last two predicates constrain the allowed port or role types in a filter or a pipe. In the original Acme specification they were expressed using two levels of quantification (i.e., `forall f:Filter in self.components | forall p in self.ports |...`). After translation and optimization this is reduced into one level of Alloy quantification (cf., Section 6.3).

The following command is provided as input to the analysis generator.

```
Check RepositoryStyle,PipeAndFilter Compatibility
```

The following Alloy model will be generated, asserting bi-implication between the constraints of the repository style and the constraints of the pipe-filter style.

```
module analysis
  open cnc_view
  open RepositoryStyle
  open PipeAndFilter
  check {
    RepositoryStyle_constraints[] <=> PipeAndFilter_constraints[]
  } for 10 but 20 Port, 20 Role
```

When executed the Alloy Analyzer reports that there is a counterexample, indicating that certain systems are not constructible using the merged style. Further inspection reveals that constraints in the pipe-filter style imply that every system must contain at least one `Filter` component and `Pipe` connector – something that systems of the pure repository style cannot satisfy.

7.6 Checking Hybrid Types

A more interesting (and practical) use of multiple styles is the situation in which a system contains architectural elements of *hybrid types* – created by combining types from different styles. When merging multiple styles, it is helpful to check if instances of such types can exist.

Building on the previous example of checking the compatibility of the repository style and the pipe-filter style, let us assume that we want to check if there can be a filter that at the same time can access to the database. The following is the input to the analysis generator to check this. A filter component with a `Use` port (to access the database) is declared as being part of the required structure (similar to the constructability analysis above).

```
Check RepositoryStyle,PipeAndFilter Using
  component userFilter: Filter
    = new Filter extended with {port use: Use = new Use;}
```

The result is the following analysis model, which checks the existence of such a hybrid filter.

```
module analysis
  open cnc_view
  open RepositoryStyle
  open PipeAndFilter

  one sig userFilter extends Filter {use: Use}
  fact{ userFilter<:use in ports }
  run {
    RepositoryStyle_constraints[] && PipeAndFilter_constraints[]
  } for 10 but 20 Port, 20 Role
```

When executed, the Alloy Analyzer reports that there is no solution within the specified scope, indicating that it is not possible for a filter to access a database. Inspection of the constraints of the pipe-filter style reveals that the original filter definition allows only ports of `Input` or `Output` type, which prevents a filter from having an extra port of `Use` type.

8 Case Study: MDS Style

To investigate the applicability of our approach to realistic examples, we formalized and analyzed NASA's Mission Data System (MDS) [18,17,19,42]. MDS is an experimental architectural style for NASA space systems developed at the Jet Propulsion Lab. It consists of a set of component types (e.g., sensors, actuators, state variables), and connector types (e.g., sensor query). It also includes a number of rules that define legal combinations of those types. When modeled in Acme the MDS style consists of 9 component types, 12 connector types, 22 port types, 22 role types. The component types included in MDS are summarized in Figure 6.

Figure 7 illustrates a simple system constructed using the MDS style. It uses 7 component types and 7 connector types of the MDS style to define a simple temperature control system. A temperature sensor (`TSEN`) feeds sensed data to a temperature estimator (`TEST`), which in turn updates a temperature state variable (`CTSV`). The health status of the temperature sensor is stored in `SHSV`. An executive component (`EXEC`) sets the target temperature, while a controller (`TCON`) sends commands to a heater actuator (`SACT`).

MDS also included a set of 19 rules that specify what is a valid MDS configuration. These were initially defined in English and had to be hand translated into Acme constraints, which resulted in 79 Acme constraints for the style. (Appendix B contains a list of the rules as specified by NASA engineers, and [30] contains the full Acme specification of this style.)

A simple example of such a rule is *"For any given Sensor, the number of Measurement Notification ports must be equal to the number of Measurement Query ports (rule R5A)"*. In general, however, MDS rules are much more complex. For example, one of their rules reads:

> Every estimator requires 0 or more Measurement Query ports. It can be 0 if estimator does not need/use measurements to make estimates, as in the case of estimation based solely on commands submitted and/or other states. Every sensor provides one or more Measurement Query ports. It can be more than one if the sensor has separate sub-sensors and there is a desire to manage the measurement histories separately. For each sensor provided port there can be zero or more estimators connected to it. It can be zero if the measurement is simply raw data to be transported such as a science

SchedulerT	A scheduler component is used to coordinate real-time scheduling of taskswith and MDS architecture.
ExecutableT	A component type that is combined with other component types to designate a component as one that is associated with a separate thread
ExecutiveT	A component that sets high-level objectives of and MDS system.
ControllerT	A controller is responsible for delegating the goals to other states, or for issuing commands to adaptors to achieve the state, if there are goals associated with a state variable.
ActuatorT	An actuator represents the interface between a controller and the hardware. Commands are issued to actuators to get the spacecraft to do something.
EstimatorT	An estimator is responsible for examining all of the available cues (other states, sensors, or goals) and updating state variables periodically to provide a current best estimate of the states value based on available evidence (command history, other states, sensor values, etc.).
SensorT	A sensor represents an interface to hardware sensor, for use by estimators.
StateVarT	A state variable contains the record of the state over time, and goals associated with the state.
HealthStateVarT	A health state variable stores a discrete set of data. For example, they may be used to store the history of commands sent to an actuator by a controller.

Fig. 6. Summary of component types in MDS Style

image. It can be more than one if the measurements are informative in the estimation of more than one state variable.

Formulating such constraints by hand may introduce mistakes and/or unwanted side-effects. Therefore, automated analysis can potentially provide significant benefits in dealing with the complexity of the style and gaining insight into it.

Our first check was to establish that MDS was consistent for a reasonable bound on the model size. (Section 9 contains details.) Although formal consistency of the MDS style was easily demonstrated, a more interesting kind of check was to investigate the existence of candidate architectures that contain certain minimal structures: for example, at least one controller, actuator, estimator, sensor, and state variable. As illustrated earlier, we can do this using the analysis generator by providing the following input.

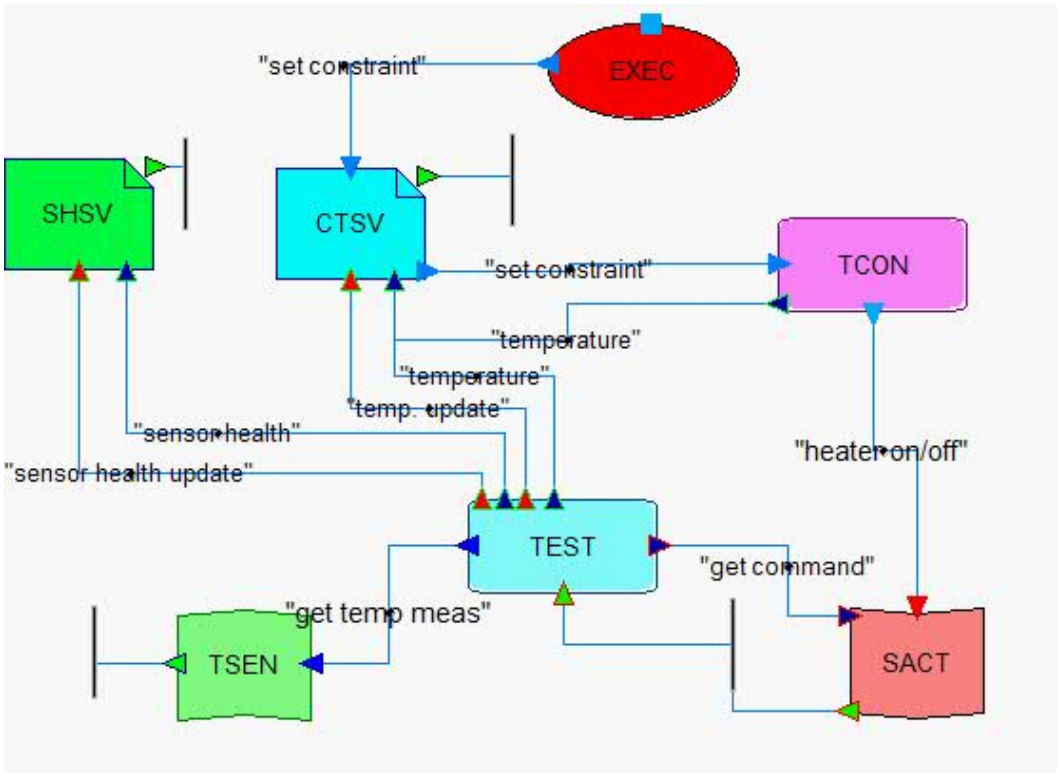


Fig. 7. An example MDS system

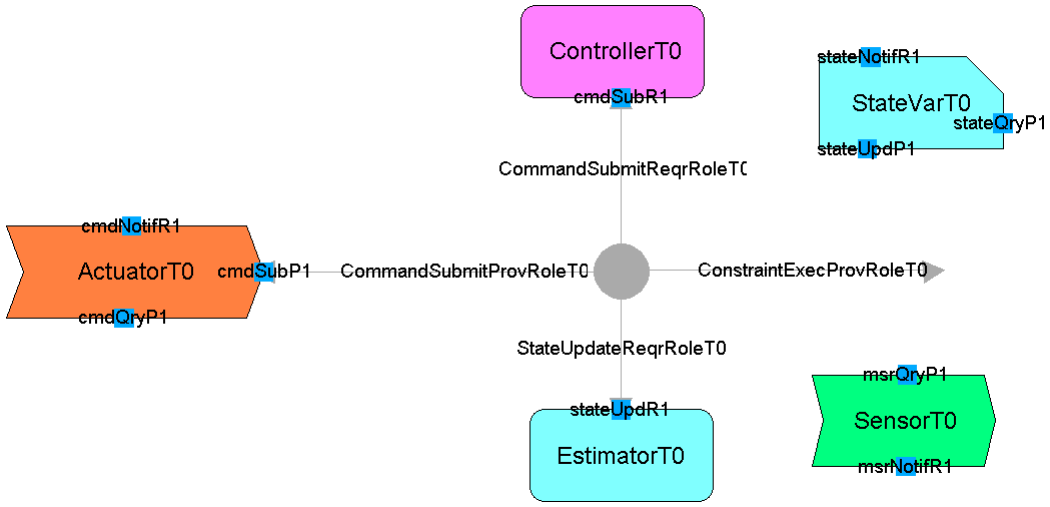


Fig. 8. An MDS system constructed by the Alloy Analyzer and translated into Acme

```
Check MDSFam Using
  component controller: ControllerT = new ControllerT;
  component actuator  : ActuatorT   = new ActuatorT;
  component estimator : EstimatorT  = new EstimatorT;
  component sensor    : SensorT     = new SensorT;
  component statevar  : StateVarT   = new StateVarT;
```

The Alloy Analyzer found the instance shown in Figure 8, confirming that such a structure is constructible. One interesting consequence of this process was that while the analysis confirmed the existence of the requested structure within the style, it yielded a counterintuitive example. Specifically, the generated model uses a single connector to connect a controller, actuator and estimator. This was hardly the intention of the original MDS style, and indicates that despite the large number of rules in MDS, the given formalization is under-constrained. This caused us to go back and refine the style specification, adding new rules to eliminate structures that should not be included.

9 Performance

Whenever one uses automated tools for analysis, a critical question is how well those tools scale to the task at hand. This is particularly critical for exhaustive state exploration-based tools such as model checkers, whose execution time may vary dramatically depending on the encodings of models and the nature of the properties to be checked.

In this section we examine the way in which architecture style analysis scales relative to three specific factors. First is the number of architectural element types in the style. Second is the number and complexity of architectural constraints in the style, where complexity is primarily determined by quantifier nesting depth. Third is the maximum model size (or bound) in terms of numbers of components and connectors.

Before we begin, we first note that there are conceptually two types of analysis in Alloy. One is *existential* analysis, which attempts to find an instance of a model that satisfies a given predicate. For example, this kind of analysis is used to perform a style-consistency check, since consistency is guaranteed by the existence of a single system satisfying the style. The other is *universal* analysis, which checks *all* instances (within the bound set for the analysis) against a predicate that every instance must satisfy.

In either case, the analyzer may have to search the entire state space: for an existential analysis, if no model exists; and for a universal analysis, if every model satisfies the predicate. In conducting our experiments, we carried out both existential and universal analyses on valid styles. In the former case typically a single system is required to be constructed, constituting a best-case analysis. In the later case, the analysis is required to examine every system instance in the search space, and hence provides a worst-case analysis.

Analysis consists of four phases. First, the architectural style is translated into an Alloy model. Second, the Alloy Analyzer translates an Alloy model into Conjunctive Normal Form (CNF). Next, it passes the translated CNF to one of several off-the-shelf SAT solvers to do the actual analysis. Finally, the result

of the SAT solver is translated back to Alloy and presented to the user. Since the translations involved in the first and last steps take negligible time, our experiments focus on how the phases two and three affect performance.

9.1 Experimental Setup

To evaluate performance we use the following experimental architectural style template, which can be instantiated with a variable number of component types and constraints. Each component type has two ports. Associated with each component type is an optional local constraint (i.e., a component type-specific constraint) stating that each port should be attached to at least one role. Additionally a global constraint (i.e., a system-wide constraint) is specified; it states that every connector should have exactly two roles.

```
Style Experiment = {
  Component type CT;
  Component type CT1 extends CT with { port p1; port p2;
    invariant forall p:self.ports | size(p.attachedroles)>0; }
  Component type CT2 extends CT with { port p1; port p2;
    invariant forall p:self.ports | size(p.attachedroles)>0; }
  ...
  Component type CTn extends CT with { port p1; port p2;
    invariant forall p:self.ports | size(p.attachedroles)>0; }
  invariant forall n in self.connectors | size(n.roles) == 2;
}
```

In all of the experiments we use an equal number of components and connectors for the model bound. The number of ports and roles is set to be double the number of components and connectors, for reasons indicated earlier. We use MiniSat as the SAT solver.

9.2 Existential Analysis

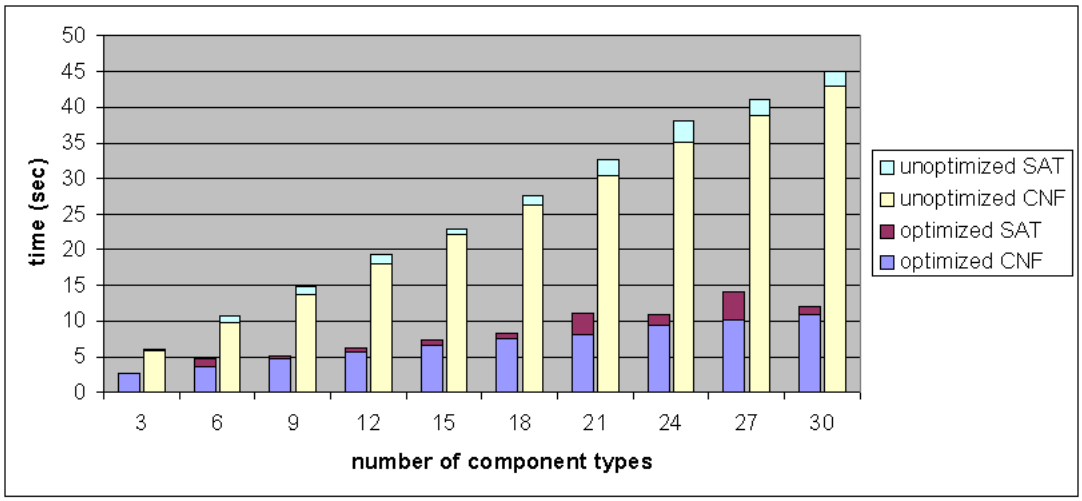
To evaluate the best-case performance of existential analysis we check the consistency of the (valid) styles instantiated from the template by varying the number of component types, the number of local constraints, and the model bound. In each case the consistency check succeeds, because the empty system – without any components or connectors – is valid for any style that has been instantiated from the style template.

9.2.1 The number of element types

Since each element type in a style is translated into an Alloy signature (as described in Section 6), the more element types a style has, the more Alloy

signatures the translated model has. Hence, one would expect performance to be affected by the number of element types in a style.

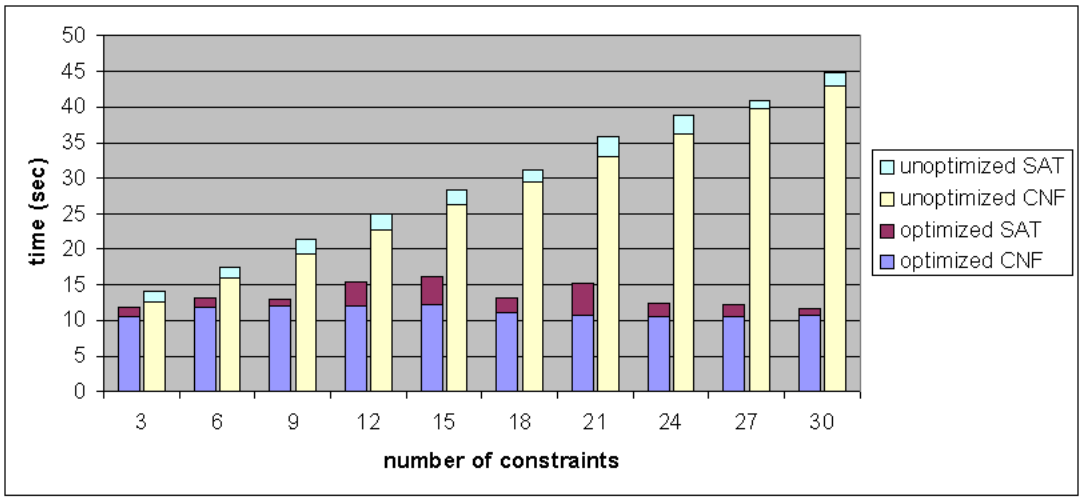
The following graph shows how performance varies according to the number of component types in the style, ranging from 3 to 30, and using the model bound of 30. For the analysis we consider two cases: with and without optimization (cf., Section 6.3). As shown, an optimized analysis for a style with 30 component types completes in under 10 seconds, and the bulk of that time is spent in CNF generation. Further, performance scales roughly linearly in the number of component types, and the slope is affected by the complexity of the constraints associated with those types. [11]



9.2.2 The number of constraints

Each constraint in a style is translated into an Alloy predicate. One would expect that the more complex the constraints are, the longer it should take to generate and evaluate predicates (with respect to both CNF generation and SAT solving). The following graph shows how the performance grows with the number of local constraints in the style. For the analysis the number of component types is fixed at 30, but we vary how many of these have an associated (local) constraint. Furthermore, to see how the performance is affected by the complexity of constraints, we ran the experiments using both optimized and unoptimized constraints. As can be seen, performance increases in a linear fashion for the unoptimized version, and is roughly constant for optimized analysis of the test styles.

[11] Deviations from a strictly linear slope are caused by the variable nature of SAT solving.



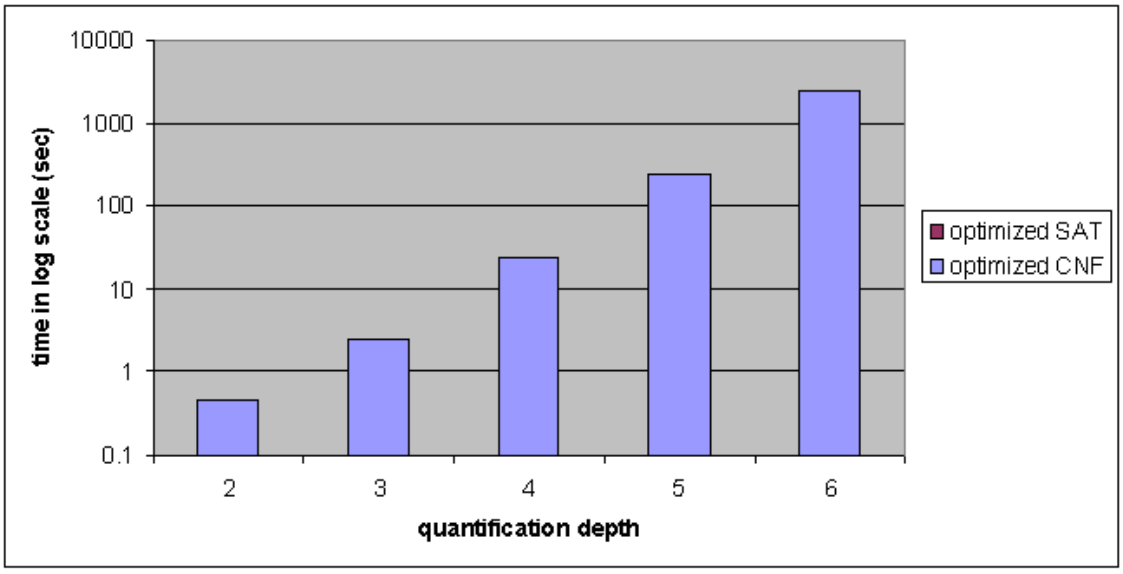
At this point it is worth asking how the complexity of an individual constraint affects performance, especially in the case of a quantified predicate. When a quantifier is encountered during CNF generation, the Alloy Analyzer expands the body of the quantification using all the possible objects in the quantified range. Therefore, when quantifiers are nested, the body at inner quantification levels will be expanded multiple times depending on the quantifier depth. [12] This suggests that the quantification level will affect performance exponentially.

To confirm this, we checked the consistency of the following style template by varying the number of quantified variables in a single global constraint with model bound set to 10.

```
Style Experiment2 = {
  invariant forall c1,c2,...,cn in self.components |
    connected(c1, c2) &&
    connected(c2,...) &&
    connected(...,cn) => connected(cn,c1)
}
```

The following graph shows that the execution time does indeed grow exponentially. (Note that the vertical axis has a logarithmic scale.)

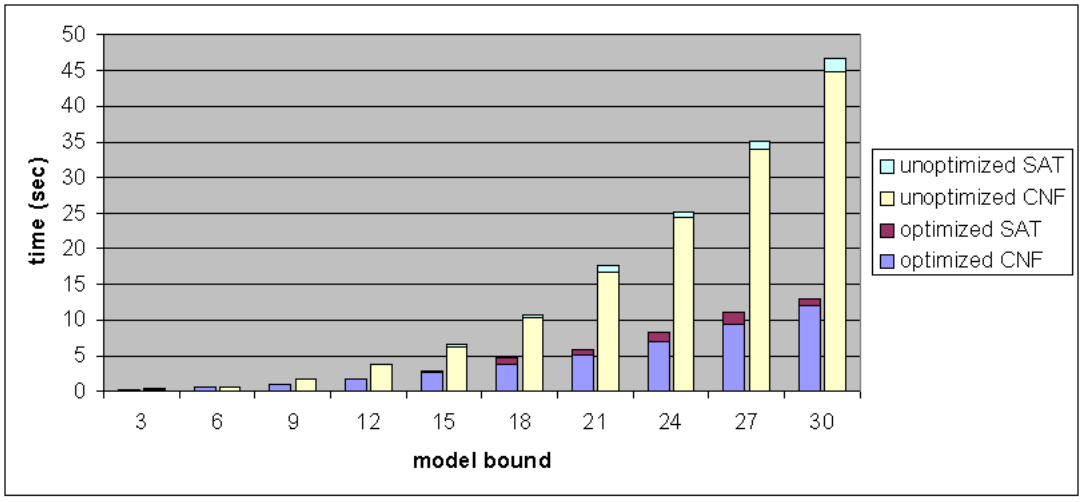
[12] The same is true for quantified expressions using multiple variables, since these are equivalent to nested quantifications.



9.2.3 The model bound

Larger model bounds (which determine the maximum number of architectural elements in a system) can increase analysis time in two ways. First, as noted in the previous experiment, it increases the CNF generation time when a quantified predicate is encountered. Second, as we will see in more detail later, it increases SAT solving time when a universal analysis is performed.

The following graph shows how the experimental style scales in terms of the model bound ranging from 3 to 30, using 30 component types with local constraints. We again consider two cases: with and without optimization. When the optimization is on growth is linear. When the optimization is not used, growth is exponential, which is consistent with the earlier experiment regarding constraint complexity.

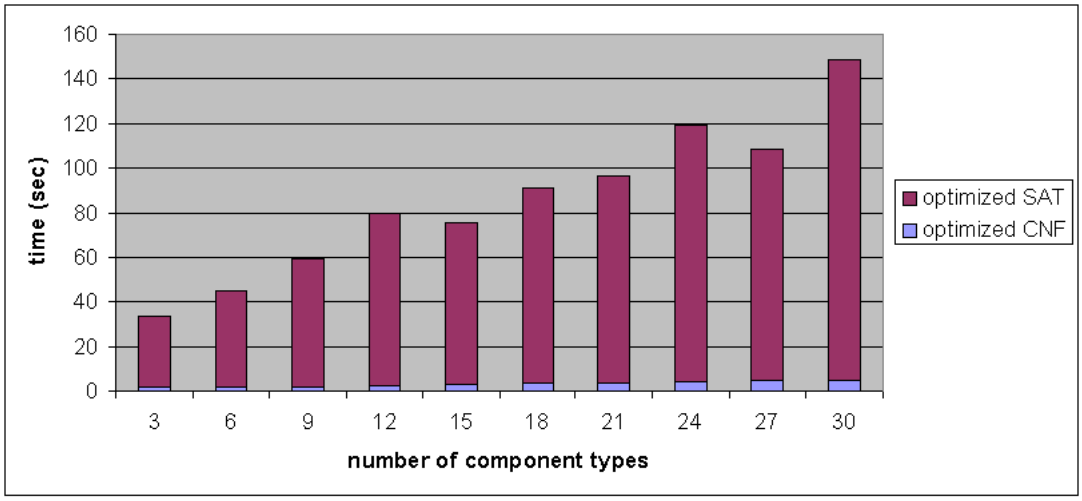


9.3 Universal analysis

To evaluate worst-case performance we performed a universal analysis by checking a property that states that a component of type `CT` must be attached to a connector, for each of the styles instantiated from the template by varing the number of component types and the model bound. The property holds if all the local constraints are present, because the local constraints require all the ports of a component of type `CT` to be attached to a role. For the styles in the experiment this triggers an exhaustive analysis of the state space of possible architectures for that style.

9.3.1 The number of element types

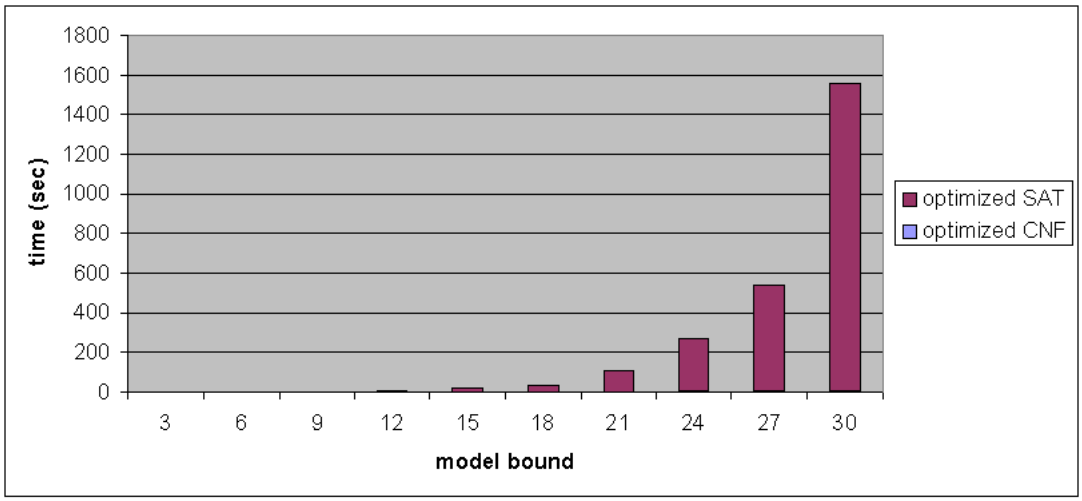
Although the number of element types does not affect the number of instances to check, it does affect the complexity of constraints to check for each instance. The following graph shows how the analysis scales in terms of the number of element types when checking a property, where the model bound is set to 20. A linear increase in time is observed, consistent with the previous consistency check experiment. Note, however, in this case SAT solving time dominates because in an exhaustive analysis much more work is being done by that phase.



9.3.2 The model bound

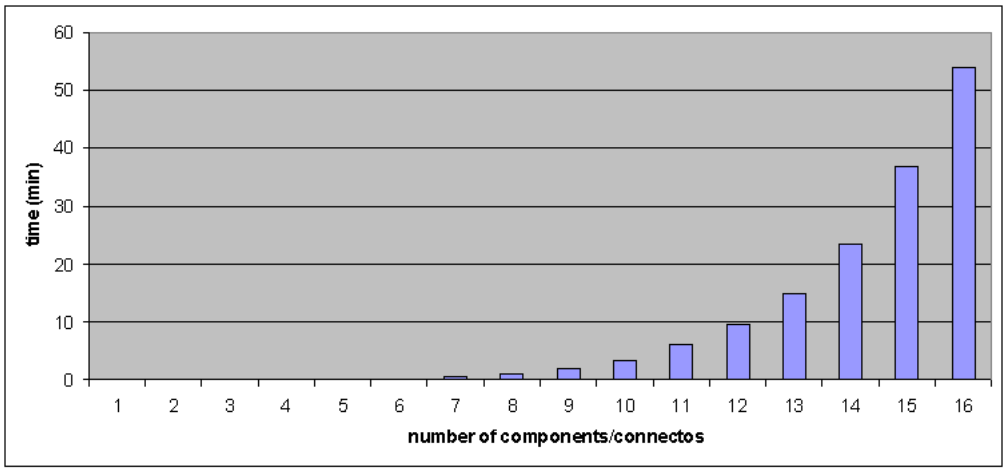
Larger model bounds affect performance significantly in the case of universal analysis for a valid style, because as the model bound grows, there are more instances to check. As the number of components grows, the number of other top-level types (connectors, ports, and roles) also grows. Therefore it is expected that the analysis time increases exponentially. The following graph shows how the experimental style scales in terms of the model bound using ten component types. As expected exponential increase is observed, although for

up to 27 model elements (of each top-level type) the analysis remains under 10 minutes.



9.4 Evaluating MDS

To investigate the scalability of our analysis approach to a larger example, we applied similar experiments to the MDS style (Section 8). Specifically, we performed a consistency check on the translated Alloy model allowing the maximum number of components and connectors to range from 1 to 16. The results are shown below.



Within an hour we were able to check its consistency with up to 16 components, 16 connectors, 32 ports and 32 roles. Considering that the machine that we used for the experiment was a modest one,[13] and that the number and complexity of constraints in this style is quite high, this is a reasonable stress

[13] We used a 1.6GHz Core2 CPU processor.

test for consistency checking of a valid style.

9.5 Performance summary and discussion

Based on our benchmarking examples, we would argue that Alloy-based analysis of architectural styles is tractable for realistic-sized styles. Specifically, best-case analysis scales tractably to styles with (a) up to 30 element types; (b) constraints with complexity of up to 6 levels of quantifier nesting, and (c) model generation bounds of up to 30 top-level instances of both components and connectors. These figures were confirmed on a style developed for practical space flight systems (MDS), with many types and constraints.

There are a number of additional factors worth noting. First, optimization played a significant role in improving analysis performance by removing one level of depth in quantification (when applicable) and thereby reducing both CNF generation time (with one less nested loop) and SAT solving time (with simpler predicates). Since in practice many style constraints are represented as quantifications with at least two levels of depth, as explained in 6.3 and illustrated in 7.5, this kind of optimization has a large overall impact.

Second, existential analysis for a valid style depends heavily on CNF generation, because SAT solving stops immediately once a satisfying instance is found. On the other hand, universal analysis depends heavily on SAT solving, because all instances within bound must be exhaustively tested. Since the Alloy Analyzer uses external off-the-shelf SAT solvers, we can expect that as the performance of SAT solvers improves – a trend that has been evident over the past few years – we can expect that performance of style analysis will likewise continue to improve.

10 Conclusions and Future Work

In this paper we have described a technique for formal analysis of the structural aspects of architectural styles. As we have illustrated, a relational model generator, such as the Alloy Analyzer, can be used to automatically check critical properties of styles, including consistency, existence of specific structures, implied properties, equivalence of global and local constraints, and compatibility between multiple 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 be inclined to check these properties by hand. Hence, having a tool to assist in this effort represents a major advance.

However the approach has some limitations. First, since the Alloy-based 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 bound, 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 a relatively small counterexample.

A second issue is the need to relate counterexamples back to the source specification. While our tools automatically map Alloy counterexamples back into an architectural model, a 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 in that community [11].

A final limitation of our current tool is the fact that it deals only with structural properties of architectural styles. It does not handle, for example, architectural behavior, dynamic changes to architectural models, and quality attributes. However, these 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 important area for future work.

Acknowledgements

This work is supported in part by the Office of Naval Research (ONR), United States Navy, N000140811223 as part of the HSCB project under OSD, by the US Army Research Office (ARO) to CyLab under grant numbers DAAD19-02-1-0389 and DAAD19-01-1-0485, and by the National Science Foundation under Grants No. CNS-084701 and IIS-0534656. We thank Dan Dvorak, Kirk Reinholtz, Kenny Meyer, and Robert Rasmussen for their help in understanding MDS. Thanks also to Daniel Jackson and his research group for answering questions regarding Alloy, and understanding its performance. Earlier versions of this work benefited greatly from comments by Orieta Celiku, as well as Marcelo Frias, Juan Galeotti, and Nazareno Aguirre from the University of Buenos Aires. We also thank members of the ABLE research group and anonymous reviewers for their constructive comments.

References

[1] IEEE recommended practice for architectural description of software intensive systems (IEEE std 1471-2000), 2000.

[2] IEEE standard for modeling and simulation high level architecture (HLA) - framework and rules. *IEEE Std 1516-2000*, pages i–22, Sep 2000.

[3] Yahoo Pipes website. http://pipes.yahoo.com, 2009.

[4] Gregory Abowd, Robert Allen, and David Garlan. Using style to understand descriptions of software architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering*, Software Engineering Notes 18(5), pages 9–20. ACM Press, December 1993.

[5] Gregory Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, October 1995.

[6] J. R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, 1996.

[7] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

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

[9] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, Second Edition.* Addison Wesley, 2003. ISBN 0-201-19930-0.

[10] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, 1996.

[11] Sagar Chaki, Alex Groce, and Ofer Strichman. Explaining abstract counterexamples. In *Foundations of Software Engineering (SIGSOFT FSE)*, pages 73–82. ACM SIGSOFT, October 2004.

[12] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond.* Addison Wesley, 2002.

[13] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies.* Addison Wesley Longman, 2001.

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

[15] 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.

[16] Jürgen Dingel, David Garlan, Somesh Jha, and David Notkin. Towards a formal treatment of implicit invocation. *Formal Aspects of Computing*, 10:193–213, 1998.

[17] Daniel Dvorak. Challenging encapsulation in the design of high-risk control systems. In *Proceedings of the 2002 Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA92)*, Seattle, WA, November 2002.

[18] Daniel Dvorak, Robert Rasmussen, G. Reeves, and Alan Sacks. Software architecture themes in JPL's Mission Data System. In *Proceedings of the AIAA Space Technology Conference and Expo*, Albuquerque, NM, 1999.

[19] Daniel Dvorak and Kirk 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.

[20] David Garlan, Robert Allen, and John 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.

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

[22] David Garlan, William K. Reinholtz, Bradley Schmerl, Nicholas Sherman, and Tony Tseng. Bridging the gap between systems design and space systems software. In *Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop (SEW-29)*, Greenbelt, MD, 6-7 April 2005.

[23] John E. Hopcroft and Jeffrey D. Ullman. *Introuduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

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

[25] Paola Inverardi and Alex 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.

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

[27] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

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

[29] Jung Soo Kim and David Garlan. Analyzing architectural styles with Alloy. In *Workshop on the Role of Software Architecture for Testing and Analysis 2006 (ROSATEA 2006)*, Portland, ME, July 2006.

[30] Jung Soo Kim and David Garlan. Analyzing architectural styles with alloy. Technical Report CMU-ISR-09-111, School of Computer Science, Carnegie Mellon University, April 2009.

[31] Frederick Kuhl, Richard Weatherly, and Judith Dahmann. *Creating computer simulation systems: an introduction to the High Level Architecture*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[32] Anthony J. Lattanze. *Architecting Software Intensive Systems: A Practitioner's Guide*. Auerbach Publications, 2008.

[33] David C Luckham, Lary M. Augustin, John J. Kenney, James Veera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):336–355, April 1995.

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

[35] 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, January 2000.

[36] Daniel Le Metayer. Software architecture styles as graph grammars. In *Proceedings of the Fourth ACM Symposium on the Foundations of Software Engineering*. ACM SIGSOFT, October 1996.

[37] Sun Microsystems. J2ee information site. URL: http://java.sun.com/javaee/.

[38] Robert T. Monroe. *Rapid Develpomentof Custom Software Design Environments*. PhD thesis, Carnegie Mellon University, July 1999.

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

[40] OMG. Unified modeling language. URL: http://www.omg.org/mda.

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

[42] Roshanak Roshandel, Bradley Schmerl, Nenad Medvidovic, David Garlan, and Dehua Zhang. Understanding tradeoffs among different architectural modelling approaches. In *Proc. of the 4th Working IEEE/IFIP Conf. on Software Architectures*, Oslo, Norway, June 2004.

[43] SAE. SAE AADL information site. URL: http://www.aadl.info/.

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

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

[46] Joao Pedro Sousa and David 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.

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

[48] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1989.

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

Appendices

A cnc_view Model Specification

```
////////////////////////////////////////////////////////////////////////
// This model includes the built-in features (types and functions) of
// the Acme language, and additionally contains several predefined
// predicates for convenience. This model should be imported by any
// translated model.
////////////////////////////////////////////////////////////////////////

module cnc_view'

//..............................................................
// Built-In Types
//..............................................................

sig Component {ports: set Port, system: System}
sig Connector {roles: set Role, system: System}
sig Port {component: Component}
sig Role {connector: Connector, attachment: lone Port}

one sig self extends System {}
abstract sig System {components: set Component,
                     connectors: set Connector}

fact {~ports = component && ~roles = connector}
fact {~components = Component<:system}
fact {~connectors = Connector<:system}

//..............................................................
// Built-In Type Test Functions
//..............................................................

pred declaresType   [e:     univ, t: set univ] { e in t }
pred satisfiesType  [e:     univ, t: set univ] { e in t }
pred declaredSubtype[t: set univ, u: set univ] { t in u }

// "satisfiesType" is modeled the same as declaresType.
// "typesDeclared" is not modeled because it returns high-order value.
// "superTypes" is not modeled because it returns high-order value.

//..............................................................
// Built-In Connectivity Funcitons
//..............................................................

pred attached[r: Role, p: Port] { r->p in attachment }
pred attached[p: Port, r: Role] { r->p in attachment }
```

```
pred attached[n: Connector, c: Component] {
  n->c in roles.attachment.component
}
pred attached[c: Component, n: Connector] {
  n->c in roles.attachment.component
}
pred connected[p1: Port, p2: Port] {
  some p1.~attachment.connector & p2.~attachment.connector
}
pred connected[c1: Component, c2: Component] {
  some c1.ports.~attachment.connector & c2.ports.~attachment.connector
}
pred reachable[c1: Component, c2: Component] {
  c1->c2 in ^(ports.~attachment.connector.roles.attachment.component)
}

//.................................................................
// Built-In Ownership Funcitons
//.................................................................

fun parent[p: Port]: Component { p.component }
fun parent[r: Role]: Connector { r.connector }
fun parent[c: Component]: System { c.system }
fun parent[n: Connector]: System { n.system }

// "parant" is not modeled for systems, properties, and representations

//.................................................................
// Built-In Set Functions
//.................................................................

pred contains     [e:     univ, s: set univ]       { e in s }
pred isSubset     [s: set univ, t: set univ]       { s in t }
fun  union        [s: set univ, t: set univ]: univ { s +  t }
fun  intersection [s: set univ, t: set univ]: univ { s &  t }
fun  setDifference[s: set univ, t: set univ]: univ { s -  t }
fun  size         [s: set univ             ]: Int  { #s     }
fun  sum          [s: set Int              ]: Int  { sum i:s|i }

// "flattern" is not modeled because the input is high-order.
// "product" is not modeled because they require recursion.

//.................................................................
// Predefined properties
//.................................................................
```

```
pred predefined_no_unattached_port[cs: set Component] {
  all p: component.cs | some r: Role | attached[r,p]
}
pred predefined_no_unattached_role[ns: set Connector] {
  all r: connector.ns | some p: Port | attached[r,p]
}
pred predefined_no_unattached_component[cs: set Component] {
  all c: cs | some n: Connector | attached [n,c]
}
pred predefined_no_unattached_connector[ns: set Connector] {
  all n: ns | some c: Component | attached [n,c]
}
pred predefined_no_unconnected_component[cs: set Component] {
  all c: cs | some d: Component | connected[c,d]
}

pred predefined_topology_star[cs: set Component] {
  one core: cs |
    ( all c: cs-core | connected[c, core] ) &&
    ( no disj c1,c2: cs-core | connected[c1, c2] )
}

pred predefined_topology_linear[cs: set Component] {
  some disj c1, c2: cs |
    ( one c: cs-c1 | connected[c1, c] ) &&
    ( one c: cs-c2 | connected[c2, c] ) &&
    all c: cs-c1-c2 |
      some disj c3, c4: cs-c | connected[c, c3] && connected[c, c4] &&
                               no c5: cs-c-c3-c4 | connected[c, c5]
}

pred predefined_topology_ring[cs: set Component] {
  all c: cs | predefined_topology_linear[cs-c]
}

pred predefined_no_cycle[cs: set Component] {
  no cs': set cs | predefined_topology_ring[cs']
}
```

B Mission Data Systems Configuration Rules

The following configuration rules and guidelines were provided by NASA for the Mission Data Systems architectural style.

Rule 1: Every controller requires 1 or more Command Submit ports. Every actuator provides 1 or more Command Submit ports. An actuator can have more than 1 if it has separately commandable sub-units and there is a desire to manage the command histories separately. There must be exactly one controller port connected to each actuator port.

Rule 1A: If actuator A has more than one Command Submit ports then all such ports must be connected to the same controller C. This ensures that control of A is the responsibility of a single controller C.

Rule 2: Every actuator requires 1 or more Command Notification ports, one per Command Submit port. Every estimator provides 0 or more Command Notification ports; it can be 0 if the estimator has no need to be event driven. For every actuator Command Notification Port there may be 0 or more estimators connected to it.

Rule 2A: For any given actuator the number of Command Submit ports and Command Query ports and Command Notification ports must be equal, i.e., there must be a 1:1:1 association among them.

Rule 2B: If estimator E has a Command Notification connection with actuator A, then it must also have a Command Query connection with A for the same command history. Otherwise it will be getting command notifications with no way to obtain the commands that were submitted.

Rule 3: Every estimator requires 0 or more Command Query ports; it can be 0 if there is no command evidence for a particular state. Every actuator provides 1 or more Command Query ports. For each actuator-provided port there can be 0 or more estimators connected to it. It can be more than 1 if command submittal is informative to more than one estimator. Though unusual, it can be 0 if no estimator chooses to use command submittal evidence, instead relying on other sources of evidence.

Rule 4: Every estimator requires 0 or more Measurement Query ports. It can be 0 if the estimator does not need/use measurements to make estimates, as in the case of estimation based solely on commands submitted and/or other states. Every sensor provides 1 or more Measurement Query ports. It can be more than one if the sensor has separate sub-sensors and there is a desire to manage the measurement histories separately. For each sensor-provided port there can be 0 or more estimators connected to it. It can be zero if the measurement is simply raw data to be transported, such as a science image. It can be more than one if the measurements are informative in the estimation of more than one state variable.

Rule 5: Every sensor requires 1 or more Measurement Notification ports, one per Measurement Query port. Every estimator provides 0 or more Measurement Notification ports. It can be 0 if the estimator has no need to be event

driven. For every sensor port there may be 0 or more estimators connected to it.

Rule 5A: For any given sensor the number of Measurement Notification ports must equal the number of Measurement Query ports.

Rule 5B: If estimator E has a Command Notification connection with actuator A, then E must also also have a Command Query connection with A for the same command history. Otherwise it will be getting command notifications with no way to obtain the commands that were submitted.

Rule 6: Every state variable provides exactly 1 State Query port. There can be any number of components connected to that port. It would be suspicious if there were no connections at all, but it is valid for states that are estimated purely for transport to another deployment.

Rule 7: Every estimator requires 1 or more State Update ports, one per state variable that it estimates. Every state variable provides exactly 1 State Update port, and has exactly 1 estimator connected to it.

Rule 8: Every state variable requires exactly 1 State Notification port and can have any number of other components connected to it.

Rule 8A: If component C has a connection to the State Notification port of state variable S, then C must also have a connection to the State Query port of S. Otherwise, C will be getting notifications with no way to obtain the updated state.

Rule 9: Every executable component provides exactly 1 Execute port. There must be a 1-to-1 association of thread scheduler ports to executable component ports.

Rule 10: No two ports of a component should be connected to the same target port. This could happen due to a cut-and-paste error where, say, a controller has two State Query ports connected to the same state variable.

Rule 11: Every connection must be "property compatible." System engineers will specify properties at four levels: component type, component instance, port type, and port instance. For example, suppose that there is a single position and heading controller for the six wheels of a rover, and that it requires 12 Command Submit ports (6 for steering and 6 for driving). Here are examples of properties at each of the four levels:

- component type: property = executable, controller
- component instance: property = position and heading
- port type: property = steering, multiplicity=1
- port instance: property = left-front

Thus, in attempting a connection, property checking at the port instance would prevent a left/right error, and property checking at the port type would prevent a steering/driving error as well as a multiplicity error (0 connections where exactly 1 is required).

Note for Consideration: There ought to be enough information specified in properties such that there is only one valid arrangement of connections (or that all possible arrangements are equivalent). Otherwise, system engineers are leaving something unsaid, open to creative interpretation by

others downstream in the development process. Thus, we should view the property information as part of a prescription. In addition to the prescription, we need to specify requirements on evolution so that thread management (for example) while creating, configuring, and connecting components is specified.

Rule 12: There are several "don't do this" kind of rules. For example, never connect a controller to a sensor (because then it is doing its own private estimation), and never connect an estimator to a Command Submit port of an actuator (because only a controller is allowed to submit commands).

Rule 13: For every estimator/controller pair where the controller controls a state variable that is estimated by that estimator, the order of dispatch for the pair of executables must be deterministic. If either the estimator or controller or both are connected to executable hardware adapter(s), then the hardware adapter(s) must be included in the deterministic ordering check.

C Abstract Syntax of Acme Language

The following abstract syntax is a subset of Acme language that is translatable to Alloy using the technique presented in this paper.

```
Style  ::= style id [ extends id+ ] Decl*

          | fun id ( ( id : Type )* ) : Type Expr
          | type id extends ElemType+ Decl*
          | type id = [ ( id : PropType )* ]
Decl   ::= | element id : ElemType
          | property id : PropType
          | attachment id.id to id.id
          | invariant Expr

          | unary_op Expr
          | Expr binary_op Expr
          | Xid ( Expr* )
          | Xid
          | id
          | Expr.id
Expr   ::= | Expr.reference
          | select id [ : Type ] in Expr | Expr
          | collect id.id in Expr | Expr
          | { Expr* }
          | < Expr* >
          | quantifier id [ : Type ] in Expr | Expr
          | integer

ElemType  ::= Xid | element | component | connector | port | role
PropType  ::= Xid | int | set int | seq int
Type      ::= ElemType | PropType | set ElemType | system
Xid       ::= [ id / ] id
id        ::= identifier

unary_op   := ! | -
binary_op  := or | -> | <-> | and | == | != | < | > | <= | >= | + | - | * | / | %
reference  := components | connectors | ports | roles | attachedports | attachedroles
quantifier := forall | exists | unique
```