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 many software architectures and component integration frameworks is an architectural style that provides a domain-specific design vocabulary and a set of constraints on how that vocabulary is used. Given the increasing number and complexity of architectural styles, designing a sound and appropriate style becomes an important and 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 the style designer. In this paper we show how to map an architectural style, expressed formally in an architectural description language, into a relational model that can be automatically checked for properties such as whether a style is consistent, whether a style satisfies some predicate over the architectural structure, whether two styles are compatible for composition, and whether one style refines another.

1 Introduction

The discipline of software architecture has matured substantially over the past decade: today we find growing use of standard notations [28,32,17], architecture-based development methods [8], and handbooks for architectural design and notation [5,7]. 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 [3], performance [10], reliability [38], style conformance [33], and many others.

One of the important pillars of modern software architecture is the use of architectural styles [1,5,7,34]. An architectural style defines a family of related

* Corresponding author.
Email addresses: jungsoo@cmu.edu (Jung Soo Kim), garlan@cs.cmu.edu (David Garlan).

systems, typically by providing a domain-specific architectural design vocabulary together with constraints on how the parts may fit together. Examples of common styles range from the very generic, such as client-server or pipe-filter, to the very domain-specific, such as MDS [13,12,31] and J2EE [27].

The use of styles as a vehicle for characterizing a family of software architectures is motivated by a number of benefits. Styles provide a common vocabulary for architects, allowing developers to more easily understand a routine architectural design. They form the backbone of product-line frameworks, allowing the reuse of architectures across many products, and supporting component integration. By constraining the design space, they provide opportunities for specialized analysis. 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.

Consequently more and more architectural styles are being defined every day. In many cases new styles are elaborations of existing styles. For example, a company might constrain J2EE-based architectures 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 emerging needs for automotive 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 will 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 a non-empty set of systems, whether it can be combined consistently with another, or whether it will retain the essential properties of some parent style, is today largely a matter of trial and error. In fact, today style designers typically cannot detect fundamental errors in a style until someone actually tries to implement a particular system in that style, when the cost of change is very high.

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

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

In Section 2 we describe related work. Section 3 discusses architectural styles: what they are, how they can be characterized formally, and what kinds of properties we would like them to have. Section 4 provides an brief introduction to Alloy and the Alloy Analyzer, the target modeling language and tool that we will be using to check properties of styles. Section 5 then presents the translation approach, showing how to map formal descriptions of architectural styles into Alloy, and highlighting places where that translation is non-trivial. Section 6 shows by example how to use the Alloy Analyzer to check properties of an architectural style. Section 7 demonstrates how an architectural style in real world 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 8 discusses the strengths and limitations of the approach, and considers future work.

2 Related Work

There are two broad areas of closely related work. The first is formal representation and analysis of software architecture. Since the inception of software architecture over a decade ago, there have been a large number of researchers interested in formal description of architectures. These efforts have largely focused on the definition and use of architecture description languages (ADLs) [26]. 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 [3,25] to specify abstract behavior of an architecture, and to check for properties like deadlock freedom of connector specifications. Others have used rewriting rules [19], sequence diagrams [18], event patterns [24], and many others.

This existing body of work on analysis of software architectures has primarily focused on the problem of analyzing the properties of *individual* systems. That is, given an 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 [31], performance [10,36], and reliability [32]. 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 Abowd et al. [1] modeled styles using Z [37]. In that approach one can specify general properties of architectural styles, but the work lacked explicit guidance on *what* properties should be evaluated, and it did not provide any tool-assisted support for analysis. Other work has investigated formal properties of particular styles, such as EJB [35] or publish-subscribe [11], but these have not provided any general style-oriented analyses. Finally, the Acme ADL was developed with the specific intent of providing a way to formally define architectural styles, in general, and to check conformance between the architecture of a system and its purported styles, in particular [15]. 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 [28], Z [37], or B [2], as opposed to domain-specific modeling languages, such as architectural description languages. In this work we build on these general specification languages using Alloy [20], 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 [29]. In particular, the work by Karsai et al. [22] 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.

3 Architectural Style

Software architecture is concerned with the high-level structure and properties of a system [34,30]. Over time there has emerged a general consensus that modeling of complex architectures is best done through a set of complementary views [7,17]. 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. We refer to these specialized modeling languages as *architectural styles.* [1]

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 J2EE or .Net) 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. 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 interoperability. Examples include CORBA object-oriented architecture [9] and event-based tool integration [4]. Fifth, by constraining the design space, an architectural style often permits specialized analyses. For example, it is possible to analyze systems built in a pipe-filter style for schedulability, throughput, latency, and deadlock-freedom. Such analyses might not be meaningful for an arbitrary, ad hoc architecture - or even one constructed in a different style.

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

3.1 Formal Modeling of Architectures

Building on the large body of existing formal modeling techniques for component-and-connector architectures, we model an architecture using the following core concepts that appear in most modern ADLs [26], as well as UML 2.0 [28].

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

- **Components:** Components represent the principal computational elements and data stores of a system. A component has a set of run-time interfaces, called *ports*, which define the points of interaction between that component and its environment.
- **Connectors:** Connectors identify the pathways of interaction between components. Connectors may represent simple interactions, such as a single service invocation between a client and server. Or they may represent complex protocols, such as the control of a robot on Mars by a ground control station. A connector defines a set of *roles* that identify the participants in the interaction. For example, a pipe connector might have 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 [16], although many other ADLs could have been used as well. 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`). Both components have 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.

3.2 Formal Modeling of Architectural Styles

To define a style we add the following concepts:

[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 use only those aspects of Acme that are necessary to explain our approach to style analysis. For a more thorough description of the language see [16].

```
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 as db_access.user;
    db.provide as db_access.provider;
  }
}
```

Fig. 1. Simple repository system

- **Design vocabulary:** This can be specified as a set of component, connector, port and role types that are allowed to be used in defining a specific architecture in that style. For example, a pipe-filter style would include a pipe connector type and a filter type, and a client-server style would include a client type and a server type, etc.
- **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, a J2EE style might restrict clients from interacting directly with a backend database. Other constraints may be associated with specific elements. For example, a client-server connector may be constrained to only connect clients and servers. Constraints are therefore used to restrict the possible configurations that may be created using the design vocabulary. In this respect a style can be viewed as the "type" of a configuration.

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

A second important relationship is *conjunction*. One can combine two styles by taking the union of their design vocabularies, and conjoining their constraints.

For example, one might add a database component to a pipe-filter system by conjoining a pipe-filter style with a database style. In such cases it may be necessary to also define new types of components or connectors that pertain to more than one style, such as a component type that has filter-like behavior, but that can also access a database.

```
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 described in Acme

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 types. For instance, a UnixFilter component type may be declared to be a subtype of Filter, adding an additional constraint

that the ports must be named *stdIn*, *stdOut*, and *stdErr*.

In addition, we can define a substyle as a specialization of one or more existing styles. As with element types the substyle must respect the constraints of the superstyle(s). (When more than one style is used as a supertype, the new style must be a substyle of the *conjunction* of the parent styles.)

Figure 2 illustrates the definition of a 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 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 specify a property such as `max-trans-per-sec`.

```
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 declared using a style

To simplify specifications of constraints Acme provides a number of built-in functions. Here, for example, `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. For a complete list of such functions see Appendix D of this paper and [16].

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 complicated. For example, the Mission Data System (MDS) defined by NASA JPL as a style for space systems [31] includes nine component types (actuators, sensors, etc.), twelve connector types, and over seventy rules constraining configurations. Figure 4 shows one such rule: it specifies that it is possible to connect only one controller to any of an actuator's ports. (We will revisit this style in Section 7.)

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

Fig. 4. Example constraint for the MDS style

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

4 Alloy

Alloy is a modeling language based on first-order relational logic [20,21]. [4] An Alloy model consists of signature definitions and constraint definitions. Signatures define the basic types of elements and relations between them to be used in a model, and constraints restrict the instance space of the model. Consider the following example:

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

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

[4] We use Alloy Version 4 in this paper.

An Alloy *fact* is a Boolean expression that any instance of a model must satisfy. A fact might be *local* to a signature or *global* to the whole model. The (global) fact in the above example prescribes that a person can't write two autobiographies. It uses the Alloy operator `disj` which returns true if the two objects are distinct.

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

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

The Alloy Analyzer is a bounded checker, guaranteeing the correctness of the result only within a specified bound of numbers of elements. To illustrate, the following module contains a predicate and a command to analyze the publication module:

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

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

We will introduce additional details about the Alloy language as necessary to explain our use of it for modeling software architectural styles. For further details about Alloy refer to [21].

5 Representing Styles in Alloy

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

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

Second, one must be able to represent *relations between types*. These fall into two sub-categories. One is containment relationships: components contain their ports, connectors contain their roles, and configurations contain instances of components, connectors, and attachments. The other kind of relationship is subtyping.

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

We now present our translation scheme for each of these categories, illustrating the ideas with the repository style presented in Section 3. First we show how to translate generic element types (component, connector, etc.) and implicit constraints into Alloy. Then we show how specific styles can be translated. Finally, we consider some of the areas where style translation is problematic.

5.1 Translation Overview

We begin by defining an Alloy module, `cnc_view`, that models the basic (style-independent) architectural types (component, connector, etc.) and a set of implicit constraints that any architectural model must obey. For example, a port must belong to exactly one component, ports may not be directly attached to other ports, etc. This common module is automatically included by the translator for every translated style. (Appendix D contains the a complete listing of this module.)

Next we translate a given style into an Alloy module. If that style is derived from other styles, those superstyles must also be translated into their own modules, and imported into the target module. Finally we create a separate module that captures the desired analyses to be performed on the target style.

The translation of a specific 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. For example, as explained later, certain Acme types must be converted into integers in Alloy. Type resolution to resolve naming conflicts from multiple inheritance also happens at

this stage. The generator then applies translation rules to the preprocessed abstract syntax tree. During translation each Acme element type and property type is translated into an Alloy signature. Each Acme constraint is translated into a named Alloy predicate. Additional Alloy facts are then automatically added to enforce certain rules for constructing valid structures, as we will illustrate later. [5]

5.2 An Alloy Model for Shared Architectural Concepts

To create the shared architectural concepts, we model the four basic element types as follows:

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

  fact {~ports = comp && ~roles = conn}
```

Each component includes a set of ports, and each connector includes a set of roles. Ports have a relation `comp` that identifies their parent component. Roles have two relationships: one identifies the parent connector (`conn`), and the other (`attachment`) can be zero or one port (indicated by the Alloy keyword `lone`) to which the role is attached..

The `fact` at the end of the module guarantees that if a port appears in the set of ports owned by a component, that component will also be the parent associated by `comp` for the port. (In Alloy $\sim$ represents the inverse relation.) Similarly for roles of connectors.

A logical consequence of this model is the fact that attachments may only be between ports and roles (i.e., a port may not be directly attached to another port), and that a role may not be attached to more than one port. However, the model does permit various other kinds of flexibilities: roles need not be attached to any port; a connector may have no roles at all; more than one role may be attached to a port, etc. If a given style chooses to further restrict topologies to eliminate these possibilities it can do so using its own constraints.

[5] We use Alloy predicates instead of facts to translate constraints because predicates are named and can be used to define other predicates for analysis. For example to check if local constraints `LC` and global constraints `GC` of a style are equivalent, it is necessary to check if the predicate `LC <=> GC` is true.

This model is not the only possible way of modeling shared architectural concepts in Alloy. In an earlier version of our translator, for example, we included a shared supertype of all of the above types, called `Element` [23]. However, we found that the added model complexity decreased performance of the analyzer. In our current model we can use Alloy's more-efficient built-in type `univ` for the same purpose.

Another alternative is to model ports and roles without referring to their parent component or connector, which in the model above is redundant information. But then one has to add additional facts to ensure that ports and roles have unique owners.

In addition to the four element types we also model the `System` type. A system is the collection of other architectural elements that construct a functional unit, and pulls together the various parts for analysis. In style definitions, we restrict the model to include only one system instance, denoted as `self`, which represents the system to be instantiated using the defined style. It is modeled as shown below, and augments the previous `cnc_view` module. [6]

```
abstract sig System {components: set Component,
                     connectors: set Connector}
one sig self extends System {}
fact{ self.components = Component &&
      self.connectors = Connector }
```

5.2.1 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 model such built-in functions, with the exception that for Boolean functions we use Alloy predicates. The use of predicates reduces the size of an instance of the model, thus making analysis more efficient. Below are the built-in functions used in the examples of this paper, as modeled in Alloy. The full set of all such functions augments the previous `cnc_view` module. (Definitions of all supported built-in functions are included in Appendix D.)

```
pred declaresType [element: univ, type: set univ] {
  element in type
}
```

[6] The modifier `abstract` ensures that the Analyzer will not directly create an instance of that signature, but only a specialization of it – which in this case is `self`.

```
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 r1,r2: Role |
  some p1,p2: Port |
    disj[r1, r2] &&
    attached[r1, p1] && parent[p1] = c1 &&
    attached[r2, p2] && parent[p2] = c2 &&
    parent[r1] = parent[r2]
}
```

As illustrated above, built-in functions may be overloaded. For example, `attached` can be applied to a role and port, returning true if they are directly attached. But it can also be applied to a component and connector, returning true if some role of the connector is attached to some port of the component. Because such functions have different parameter types, Alloy resolves these name conflicts automatically.

5.3 Translating a Style

An Acme style definition includes architectural element type definitions and constraints associated with those types. For example, in the repository style the `Access` connector type included a constraint dictating that those connectors must not be dangling.

The translation of a style is broken into the translation of individual constituents. The translation results are then packaged as an Alloy module that serves as an enclosure, illustrated schematically below. Note that the common module `cnc_view`, which contains basic models for built-in features of Acme language, is imported using the `open` command.

```
module RepositoryStyle
open cnc_view

// translations of type definitions
...
// translations of constraints
...
```

5.3.1 Translating type definitions

During translation each port and role type becomes a subtype of the built-in type, `Port` or `Role` respectively, either as an immediate subtype or a subtype of

another port or role type. The same is true for each component and connector type with respect to the built-in types `Component` or `Connector`, respectively.

To illustrate, the translation of the ports, roles, components and connectors of the repository style are 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. Thus, a relational join expression `user.attachment` is valid since `attachment` is defined in the supertype `Role`. The local constraints of element types are separately translated into predicates, as we discuss shortly.

One subtle 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 `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 following additional Alloy fact added to the translation above:

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

The Alloy binary operator `<:` denotes domain restriction that can be 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 `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 are similarly included in an Alloy fact statement as above. The translation function automatically generates the fact statement by scanning the type definitions of a style.

When one of the basic elements is declared to be a subtype of another type, translation to Alloy is straightforward, since in both Acme and Alloy subtyping is understood as subsetting and both inherit structure. For example, Suppose connector type `ExclusiveAccess` is declared to be a subtype of `Access`. Then whenever an element of `Access` type is required in a context, it should be still well-typed to use an element of `ExclusiveAccess`. In our Alloy model substitutability is faithfully followed because when a subtype is defined, all the fields defined in its supertypes are made available automatically in the subtype. Thus any reference to an interface defined in a supertype of an element will be well-typed and non-empty.

5.3.2 Translating property types and expressions

Acme supports various primitive property types and compound property type constructors. Primitive property types are: integer, string, character, float, enumeration, and Boolean. Compound type constructors are: set, sequence, and record. Consequently there are various types of expressions in Acme, while there are only three types of expressions in Alloy: integer expressions, Boolean expressions, and object expressions. For the translation it is necessary to map the property types in Acme into the supported types in Alloy. Similarly it is also necessary to transform expressions in Acme into valid Alloy expressions. We do this in a preprocessing step of the parsed Acme AST.

During preprocessing, string types and enumeration types are converted into integer types. String values and enumerated values are converted into integer objects. The preprocessor guarantees that different string values or enumerated values are converted into the same distinct integer objects. Since equality test is the only operator over elements of these types, there is no loss of semantics in this preprocessing step.

Float types are converted into integer types, and all float values are converted into integer values by shifting the decimal place to the right, effectively multiplying them by a power of 10. This multiplication is done consistently to all integer values because integer values and float values might be used together in an expression (such as comparisons between them). There is no negative semantic side-effect of this preprocessing as long as no overflow occurs.

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.

Alloy provides two possible representations for representing integer expressions: integer values and integer objects. An integer object is constructed by encapsulating an integer value into an object, which is used when defining Alloy relations. Integer objects must be explicitly converted back to integer values to be used in arithmetic or comparison expressions. (A similar distinction is found in programming languages like Java, which support both integer values and integer objects, together with methods to convert between the two.) Hence the translation of integer values from Acme to Alloy is context sensitive: using integer objects for relations and integer values for arithmetic expressions. (The translation rules listed in the Appendices define formally what those context conditions are.)

For the Boolean expressions the translator must make a choice between a Boolean expression and integer objects in Alloy. This decision is also context-sensitive for similar reasons.

Acme supports three compound property type constructors: set, sequence, and record types. Set types in Acme are directly translated into set types in Alloy. Sequence types in Acme are translated into a partial function from integer objects in Alloy. Record types in Acme are translated into a new signature definition with all of the fields recursively translated.

5.3.3 Translating constraints

Translation of constraints is straightforward because constraints are modeled as Boolean expressions in both Acme and Alloy. Specifically, the translation of the Boolean expressions of a constraint is as a named predicate definition. As indicated before, local constraints are translated as a separate predicate. This choice is done to support certain patterns of analysis (like checking the equivalence of local and global constraints), because if local constraints were translated as local facts of each signature, they could not be referenced non-locally in the model.

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: Component | declaresType[c, Database]
  some n: Connector | 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 bounded variable. Use of the variable name `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 are translated in a uniform way regardless of whether they are local or global. The translation of constraints includes the references to both local and global constraints. If needed, local or global one can be individually referred to in any analysis of the style.

5.3.4 Limitations of translation

While most of the features provided in a style definition language like Acme can be handled by the translator, there are several limitations that occur because Alloy does not currently support certain forms of modeling. These include:

- Expressions involving certain arithmetic operators such as multiplication and division.
- Higher-order types, such as sets of sets, sequences of sets, and functions whose signatures contain high-order types like the Acme pre-defined functions `superTypes` or `flattern`.
- Recursive definitions.

6 Analyzing Styles

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

The Alloy Analyzer works by looking for an instance of a specified model within a *scope* that indicates the maximum number of elements for each top-

level signature. In the case of architectural styles, the scope indicates the number of architectural elements of each type.

6.1 Checking the Consistency of a Style

A style is consistent if there exists at least one architectural configuration that conforms to the style (i.e., satisfies the style's structure and invariants). Consistency checking is important to make sure that a style's definition is internally consistent. Although consistency errors can arise in a single style definition, more typically they occur when combining other styles, since the other styles may have been written by different people with conflicting assumptions.

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

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

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

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

```
module analysis
  open RepositoryStyle
  run RepositoryStyle_constraints for 10
```

When the command is executed, the Alloy Analyzer reports that no instance can be found within the specified scope number. Therefore the repository style with the mistakenly written constraints above is inconsistent (for the size of model specified). After correcting the mistake and executing the command again, Alloy Analyzer will find an instance, indicating that the repository style is consistent.

While consistency checking formally establishes the existence of *some* system in the style, in many 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 consistent.

To check if a specific architectural configuration is constructible within a given style, it is necessary to translate the architectural configuration into Alloy predicates and facts that require that configuration to be present in a model. These will be in addition to the original constraints of the style. If the Alloy Analyzer can find an instance of the fortified style, the specific architectural configuration is constructible.

As an example, suppose that for the repository style an architect wants to make sure that it is possible for two components to access a database at the same time. In addition to the repository style this specific architectural configuration is described in Acme and automatically translated into the Alloy model shown below. That model is combined with the constraints of the repository style (by opening `RepositoryStyle`).

```
module analysis
  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 }
  pred RepositoryStyle_constructability() {
      RepositoryStyle_constraints
      a1.user.attachment = c1.use
      a2.user.attachment = c2.use
      a1.provider.attachment = d.provide
      a2.provider.attachment = d.provide
  }
  run RepositoryStyle_constructability for 10
```

When the command is executed, the Alloy Analyzer reports that there is an instance that satisfies the newly predicate, indicating that it is possible to have two components connected to a database at the same time.

6.3 Checking Properties of a Style

Frequently it is useful to check whether the systems that conform to a style also satisfy certain additional derivable properties. A property of a style can be checked using assertion and implication. If a property P of a style is valid for the constraints Q, the logical expression $Q => P$ should be true for every instance of the style.

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

```
module analysis
  open RepositoryStyle
  assert RepositoryStyle_property_check {
    RepositoryStyle_constraints[] =>
      some c: Component | some p: c.ports |
        declaresType[p, Use]
  }
  check RepositoryStyle_property_check for 10
```

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

6.4 Global and Local Constraint Equivalence

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

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

The equivalence of local and global constraints can be checked using bi-

implication and assertion. If a conjunction of local constraints L of a style is equivalent to a global constraint G, the Boolean expression $L <=> G$ should be true for every instance of the style.

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

```
module analysis
  open RepositoryStyle
  pred RepositoryStyle_constraints_global_alt[] {
    all p: Port | all r: Role | attached[r,p] =>
      (declaresType[p, Use]     <=> declaresType[r, User]) &&
      (declaresType[p, Provide] <=> declaresType[r, Provider])
  }
  assert equivalence_check {
    RepositoryStyle_constraints_local[] <=>
        RepositoryStyle_constraints_global_alt[]
  }
  check equivalence_check for 10
```

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

```
  all r: Role |
    (declaresType[r, User] || declaresType[r, Provider]) =>
      #(r.attachment) = 1
```

6.5 Compatibility of Styles

It is often the case that real world systems employ multiple styles at the same time. Different styles can be used in different levels of architectural hierarchy or at the same level. Unfortunately, some styles cannot be mixed together because their constraints conflict. It would be helpful if we can check for this condition.

Compatibility of styles can be checked by evaluating the consistency of the merged style. A new merged style is written by importing all the styles to check compatibility. The constraints of the merged style are the conjunction of the constraints of included styles. If the merged style is consistent, the imported styles are compatible.

Consider the previous example of the repository style and a new pipe-and-filter style introduced below. We will test if these two styles are compatible.

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

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

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

pred pipe_and_filter_constraints[] {
  all p:Input  | all r:p.~attachment |
    declaresType[r, Sink]
  all p:Output | all r:p.~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 c:Filter | all p:c.ports |
    declaresType[p, Input] || declaresType[p, Output]
  all n:Pipe   | all r:n.roles |
    declaresType[r, Source] || declaresType[r, Sink]
}
```

The following is the analysis module to check if the repository style and the

pipe-and-filter style are compatible.

```
module analysis
open RepositoryStyle
open pipe_and_filter

pred compatibility_check[] {
  RepositoryStyle_constraints[]
  pipe_and_filter_constraints[]
}
run compatibility_check for 10
```

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

6.6 Checking Overlapping Styles

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

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

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

```
module analysis
open cnc_view
open RepositoryStyle
open pipe_and_filter

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

pred overlapping_check[] {
```

```
  some UserFilter
  RepositoryStyle_constraints[]
  pipe_and_filter_constraints[]
}
run overlapping_check for 10
```

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

6.7 Checking Style Refinement

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

However in some situations there may be no explicitly declared substyle relation. Consider the previous pipe-and-filter style and a new Unix-pipe style. Note that Unix-pipe style uses different element type names such as `StdIn` instead of `Input`. We will check whether the Unix-pipe style is a refinement of the pipe-and-filter style.

```
module unix_pipe
open cnc_view

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

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

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

```
sig ErrOutput  extends Component {ei: ErrIn}

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

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

pred unix_pipe_constraints[] {
  all p:StdIn  | some n:Pipe | attached[n.sink,  p]
  all p:StdOut | some n:Pipe | attached[n.source,p]
  all p:StdErr | all r:p.~attachment | declaresType[r, Source]
  all p:ErrIn  | all r:p.~attachment | declaresType[r, Sink]

  all r:Source | some p:r.attachment |
    declaresType[p, StdOut] || declaresType[p,StdErr]
  all r:Sink | some p:r.attachment |
    declaresType[p, StdIn] || declaresType[p,ErrIn]

  some Filter && some Pipe && lone ErrOutput

  all n:Pipe | some e: StdErr |
    attached[n.source, e] =>
      (some c:CharOutput | attached[n,c]) ||
      (some c:ErrOutput  | attached[n,c])
}
```

The following is the analysis module to check if the Unix-pipe style is a refinement of the pipe-and-filter style. The constrants of the pipe-and-filter was taken and element type names were properly renamed to those of Unix-pipe style.

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

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

```
  some Filter
  some Pipe

  all c:Filter | all p:c.ports |
    declaresType[p, StdIn] || declaresType[p, StdOut] ||
    declaresType[p, ErrIn] || declaresType[p, StdErr]
  all n:Pipe   | all r:n.roles |
    declaresType[r, Source] || declaresType[r, Sink]
}
assert refinement_check {
  unix_pipe_constraints[] => modified_constraints[]
}
check refinement_check for 10
```

When the command is executed, the Alloy Analyzer reports that there is a counter-example to the assertion, indicating that there is no refinement relation between the pipe-and-filter style and the Unix-pipe style. Closer inspection of the modified constraints of the pipe-and-filter style reveals that a filter can only have ports of `StdIn` type, `StdOut` type, `ErrIn` type, or `StdErr` type, which prevents a filter from having other types of ports. This constraint is overly restrictive. After removing the last two constraints in the `modified_constraints`, the Alloy Analyzer reports that there is no counterexample, which indicates that the refinement relation is satisfied.

7 Case Study: MDS Style

To investigate the scalability of our approach to a larger, more realistic example, we formalized and analyzed NASAs Mission Data System (MDS) [13,12,14,31]. 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 22 port types, 22 role types, 9 component types, 12 connector types, and 76 constraints.

The component types included in MDS are as follows:

SchedulerT	A scheduler component is used to coordinate real-time scheduling of tasks with 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.

Figure 5 graphically illustrates a system that can be constructed in the MDS style. It uses 7 component types and 7 connector types defined in the MSD 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`).

The rules in MDS were initially defined in English and had to be hand translated into Acme constraints. Appendix E contains a complete list of 19 rules as specified by NASA engineers. These resulted in 72 Acme invariants for the style.

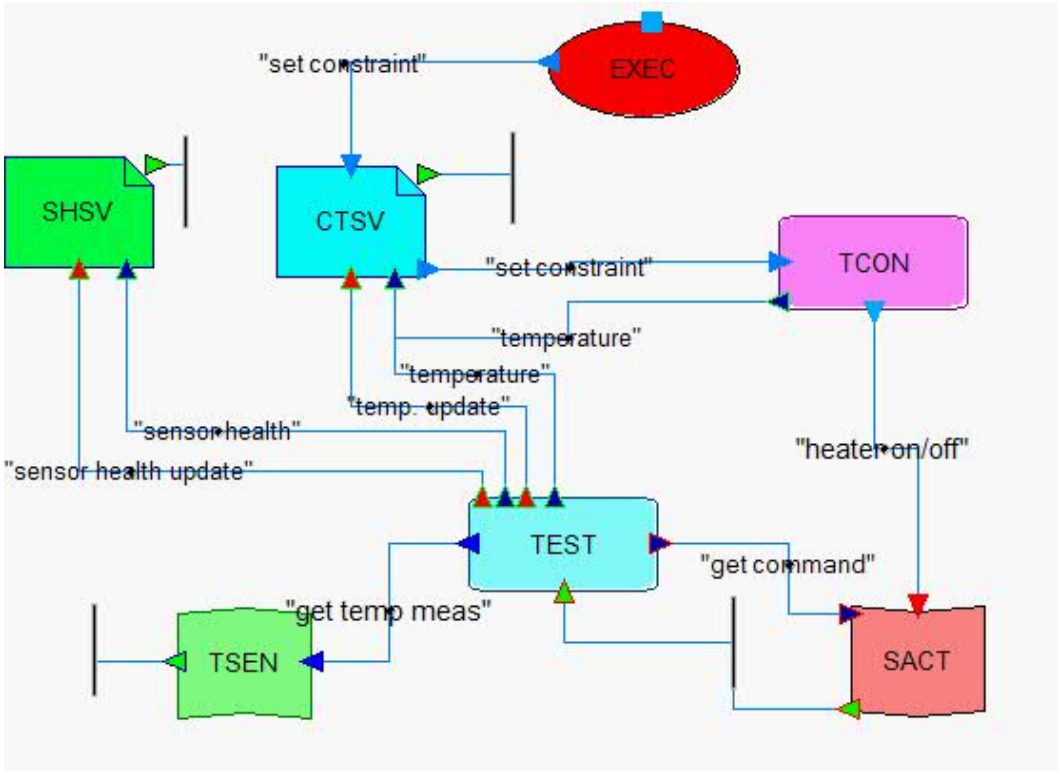

Fig. 5. MDS style example

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)"*, but in general 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 image. It can be more than one if the measurements are informative in the estimation of more than one state variable."

Formulating such complex constraints by hand throughout the description of a style may introduce mistakes and/or unwanted side-effects. Moreover, with such complexity simply understanding the style itself becomes a challenging task. Therefore, the automated analysis support presented in this paper can potentially provide significant benefits in dealing with the complexity of the style and gaining insight into it.

To investigate the scalability of the analysis, we first translated the MDS style described in Acme into an Alloy model. Then we performed a consistency check on the translated Alloy model using bound ranging from 3 to 11 (objects of each top-level type). Using a 2GHz AMD CPU and 2GB memory, we executed the analysis using Alloy Analyzer Version 4 beta3 with MiniSat as the SAT solver. Each consistency check provided an instance for the translated Alloy model, confirming that the MDS style is consistent within the bounded limit.

The following graph shows the time spent for the analysis. As is evident, with a bound of 11 the Analyzer was reaching its limit of tractability for this style .

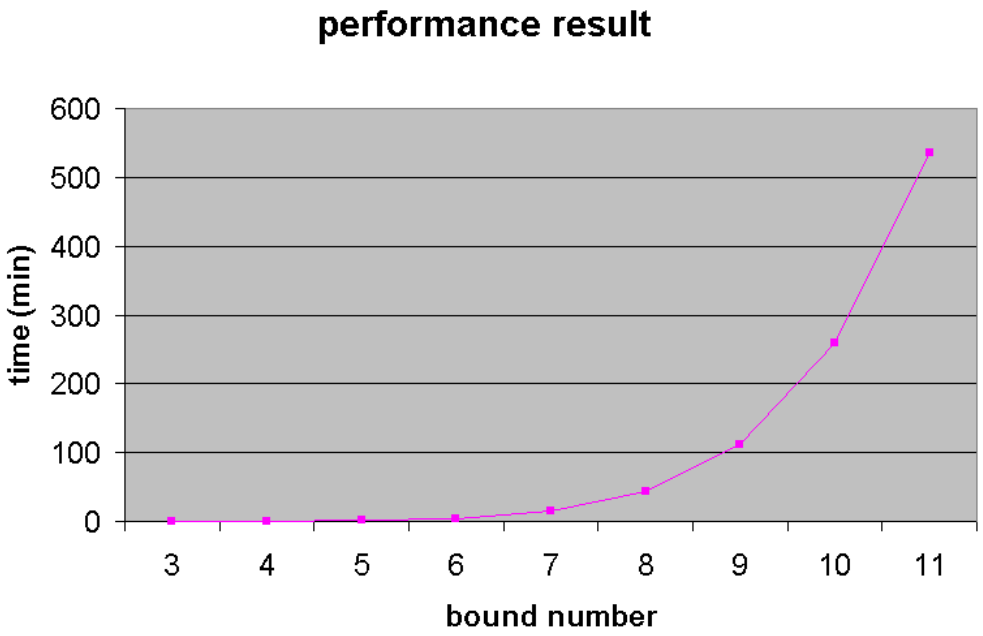

Fig. 6. Performance of the MDS style consistency check

Although formal consistency of our MDS model was easily demonstrated, a more significant check was to investigate candidate architectures that contained certain minimal structures. In the new consistency check we added constraints to require at least one controller, actuator, estimator, sensor, and state variable components. [7]

```
module analysis
  open MDSFam
    run { MDSFam_constraints && some ControllerT
```

[7] To do this we increased the number of ports used to generate the instance, since in general a given component may have several ports. For example, in the MDS style each actuator component or state variable component has a minimum of three ports.A general rule of thumb that we found was that usually twice as many ports as components lead to a better check.

```
&& some ActuatorT && some EstimatorT
&& some SensorT && some StateVarT } for 5 but 10 Port, 10 Role
```

The Alloy Analyzer found the following instance with the new consistency check, taking about 12 minutes to complete.

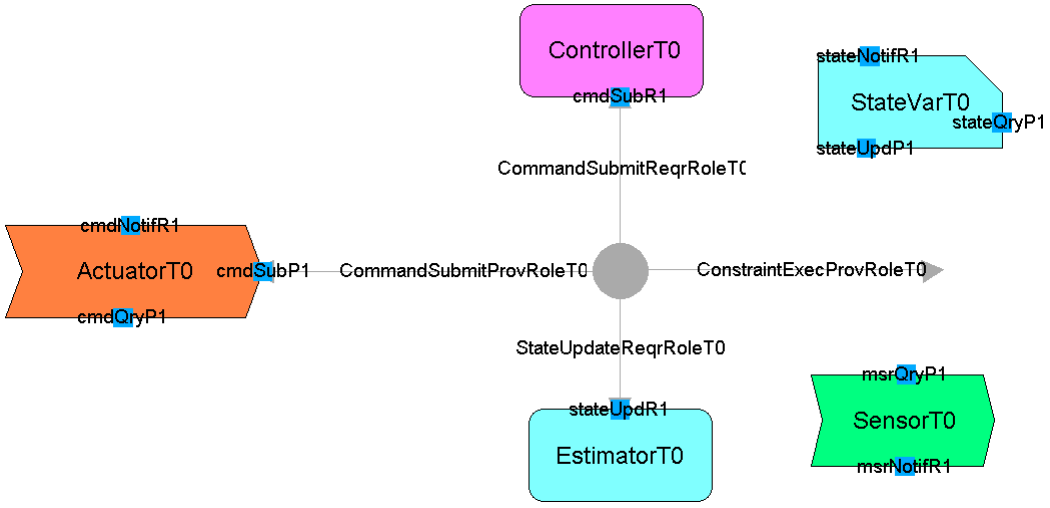

Fig. 7. An MDS system constructed by Alloy analyzer and translated into Acme

One interesting consequence of this process was that while the analysis confirmed the consistency of the requested structure with respect to the style, it yielded a counter-intuitive example. In particular, 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.

8 Conclusion and Future Work

As we have illustrated, the use of a model generator, such as the Alloy Analyzer, can provide substantial benefits to the style designer in terms of checking critical properties of styles. These properties include style consistency, validity of specific structures, implied properties, refinement, equivalence of global and local constraints, and checking for compatibility between styles.

Since specification languages such as Acme have the expressive power of first-order predicate logic, such properties are in general undecidable and typically require mathematical proof. This makes it unlikely that in practice style designers will actually be able to check these properties by hand. Hence, having an semi-automated 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 size, it may be that this holds for all models, or only for those of that finite size. Experience has shown, however, that if a specification has a flaw, it can usually be demonstrated by relatively small counterexample.

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

A third issue is the need to relate counterexamples back to the source specification. While it is straightforward to automate the reverse translation between a generated Alloy model and an instance in Acme, a more tricky issue is understanding what flaw in the design caused the counterexample to be generated in the first place. This problem is common to any model checking approach, and is an area of active research in that community [6].

A fourth area is that of performance. While today's sat solver-based model checkers (like Alloy) can handle a large number of variables, they are still limited in the size of the model that can be checked. As we indicated, our own experience with Alloy is that when the model bound approaches 15 top-level architectural elements, or when the model contains a large number of component and connector types, it may take some time to check a property, if it is indeed tractable at all. Thankfully, most of the flaws we find in practice require relatively simple models to generate.

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

Acknowledgements

This research was sponsored by the US Army Research Office (ARO) under grants DAAD19-01-1-0485 and DAAD19-02-1-0389, and by the National Science Foundation under Grant No. CCR-0113810, by NASA under the High Dependability Computing Program. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARO, NASA, the U.S. government or any other entity. We thank Dan Dvorak, Kirk Reinholtz, Kenny Meyer, and Robert Rasmussen for their help in understanding MDS. 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 for their constructive comments on this research.

References

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

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

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

[4] 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.

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

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

[7] 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.

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

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

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

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

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

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

[23] 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.

[24] 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.

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

[26] 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.

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

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

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

[30] 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.

[31] 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.

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

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

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

[35] 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.

[36] 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.

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

[38] 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.

Appendix

Appendix A: Abstract Syntax of Acme Language

The following abstract syntax is a subset of Acme language that is translatable to Alloy language using the technique presented in this paper. For the parts of Acme language that are not included below please refer to Section 5. Note that the following syntax is desugarized and abstract which may be different from the concrete Acme syntax.

$$
\begin{array}{rll}
\mathit{Style} & := & \texttt{style}\ \mathit{id}\ [\texttt{extends}\ \mathit{id}^{+}]\ \mathit{Declaration}^{*} \\
 & & \texttt{fun}\ \mathit{id} : \mathit{ArbiType}\ \texttt{(}\ (\mathit{id} : \mathit{ArbiType})^{*}\ \texttt{)}\ \mathit{Expr} \\
 & & |\ \texttt{type}\ \mathit{id}\ \texttt{extends}\ \mathit{ElemType}\ \mathit{Declaration}^{*} \\
 & & |\ \texttt{type}\ \mathit{id}\ \texttt{=}\ \mathit{PropType} \\
 & & |\ \texttt{type}\ \mathit{id}\ \texttt{=}\ \texttt{enum}\ \mathit{id}^{+} \\
\mathit{Declaration} & := & |\ \texttt{type}\ \mathit{id}\ \texttt{=}\ \texttt{record}\ (\mathit{id} : \mathit{PropType})^{+} \\
 & & |\ \texttt{element}\ \mathit{id} : \mathit{ElemType} \\
 & & |\ \texttt{property}\ \mathit{id} : \mathit{PropType}\ [\texttt{=}\ \mathit{Value}] \\
 & & |\ \texttt{invariant}\ \mathit{Expr} \\
 & & |\ \mathit{id}.\mathit{id}\ \texttt{to}\ \mathit{id}.\mathit{id} \\
\mathit{Value} & := & \mathit{Expr}\ |\ \texttt{<}\ \mathit{Expr}^{*}\ \texttt{>}\ |\ \texttt{[}\ (\mathit{id}\ \texttt{=}\ \mathit{Expr})^{+}\ \texttt{]} \\
\mathit{ElemType} & := & [\mathit{id}\ \texttt{/}]\ \mathit{id}\ |\ \texttt{element}\ |\ \texttt{component}\ |\ \texttt{connector}\ |\ \texttt{port}\ |\ \texttt{role} \\
\mathit{PropType} & := & \begin{array}{l}[\mathit{id}\ \texttt{/}]\ \mathit{id}\ |\ \texttt{integer}\ |\ \texttt{set}\ [\mathit{id}\ \texttt{/}]\ \mathit{id}\ |\ \texttt{set integer} \\ |\ \texttt{seq}\ [\mathit{id}\ \texttt{/}]\ \mathit{id}\ |\ \texttt{seq integer}\end{array} \\
\mathit{ArbiType} & := & \mathit{ElemType}\ |\ \texttt{set}\ \mathit{ElemType}\ |\ \mathit{PropType} \\
\mathit{id} & := & < identifier >
\end{array}
$$

$$
\begin{array}{rll}
 & & \texttt{!}\ \mathit{Expr} \\
 & | & \mathit{Expr}\ \ \mathit{BinaryOp}\ \ \mathit{Expr} \\
 & | & [\ \mathit{id}\ \texttt{/}\ ]\ \ \mathit{id}\ \ \texttt{(}\ \ \mathit{Expr}^{*}\ \ \texttt{)} \\
 & | & \mathit{id}\,.\,\mathit{Expr} \\
 & | & \mathit{Expr}\,.\,\mathit{Reference} \\
\mathit{Expr} & := & |\ \ \mathit{Quantifier}\ \ \mathit{id} : \mathit{Expr}\ \ \texttt{|}\ \ \mathit{Expr} \\
 & | & \{\ \ \mathit{Expr}^{*}\ \ \} \\
 & | & \texttt{select}\ \ \mathit{id} : \mathit{Expr}\ \ \texttt{|}\ \ \mathit{Expr} \\
 & | & \texttt{collect}\ \ \mathit{id}\,.\,\mathit{id} : \mathit{Expr}\ \ \texttt{|}\ \ \mathit{Expr} \\
 & | & \mathit{Literal} \\
 & | & \mathit{id}
\end{array}
$$

$$
\mathit{BinaryOp} := \texttt{or} \mid \texttt{->} \mid \texttt{<->} \mid \texttt{and} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{+} \mid \texttt{-}
$$

$$
\begin{array}{rll}
\mathit{Reference} & := & \texttt{components} \mid \texttt{connectors} \mid \texttt{ports} \mid \texttt{roles} \\
 & & \mid \texttt{attachedports} \mid \texttt{attachedroles}
\end{array}
$$

$$
\mathit{Quantifier} := \texttt{forall} \mid \texttt{exists} \mid \texttt{exists unique}
$$

$$
\mathit{Literal} := \texttt{true} \mid \texttt{false} \mid < \mathit{integer} >
$$

It is assumed that there are $AcmeSyntax$ and $AlloySyntax$ types, which represent valid Acme and Alloy descriptions respectively. A new type $Kind$ is defined to distinguish different Alloy expressions during translation. Note that $Tx[\![\ \_\ ]\!]$ returns not only the translated expression but also the kind of the translated expression.

$$Kind \texttt{ := } \{\texttt{bex}, \texttt{num}, \texttt{obj}\}$$

$$\begin{aligned}
Ts[\![\ _\ ]\!] &: AcmeSyntax \rightarrow AlloySyntax \\
Td[\![\ _\ ]\!] &: AcmeSyntax \rightarrow AlloySyntax \\
td[\![\ _\ ]\!] &: AcmeSyntax \rightarrow AlloySyntax \\
Tt[\![\ _\ ]\!] &: AcmeSyntax \rightarrow AlloySyntax \\
Ti[\![\ _\ ]\!] &: AcmeSyntax \rightarrow AlloySyntax \\
Tx[\![\ _\ ]\!] &: AcmeSyntax \rightarrow (AlloySyntax \times Kind) \\
Fact[\![\ _\ ]\!] &: AcmeSyntax \rightarrow AlloySyntax \\
fact[\![\ _\ ]\!] &: AcmeSyntax \rightarrow AlloySyntax \\
Cl[\![\ _\ ]\!] &: AcmeSyntax \rightarrow AlloySyntax \\
Cg[\![\ _\ ]\!] &: AcmeSyntax \rightarrow AlloySyntax \\
-_{bex} &: (AlloySyntax \times Kind) \rightarrow AlloySyntax \\
-_{num} &: (AlloySyntax \times Kind) \rightarrow AlloySyntax \\
-_{obj} &: (AlloySyntax \times Kind) \rightarrow AlloySyntax \\
-_{img} &: (AlloySyntax \times Kind) \rightarrow AlloySyntax \\
-_{kind} &: (AlloySyntax \times Kind) \rightarrow Kind \\
kind(\ _\ ) &: AlloySyntax \rightarrow Kind
\end{aligned}$$

A custom notation $S\,[\,i..j/\!/k \mid sep\,]$ is used throughout the definition of the translation functions. Given the Alloy description S it returns a new Alloy description which is the concaternation of sequential $S\,[\,l/k\,]$ by replacing l with the integers between i and j. A separator sep is placed between any $S[\,l/k\,]$ and $S[\,l+1/k\,]$. A recursive definition of the custom notation is shown below.

$$
\begin{aligned}
S\,[\,i..j /\!\!/ k \mid sep\,] &= \bot && (if\ \ i > j)\\
S\,[\,i..j /\!\!/ k \mid sep\,] &= S\,[\,i/k\,] && (if\ \ i = j)\\
S\,[\,i..j /\!\!/ k \mid sep\,] &= S\,[\,i/k\,]\ \ sep\ \ S\,[\,i{+}1..j /\!\!/ k \mid sep\,] && (if\ \ i < j)
\end{aligned}
$$

Appendix C: Translation Function Definitions

$Ts[\![\ \_\ ]\!]$ is applied to a Acme description of a style, thereafter other translation functions are applied to partial Acme descriptions recursively. $Str_1\ \ Str_2$ denotes concaternation of Str_1 and Str_2. $S[i..j/\!/k|sep]$ binds stronger than concaternation, so parentheses are used in the definition of translation functions to override the precedence.

$Ts[\![$ `style` id [`extends` id^{+}] d^{*} $]\!] =$

`module` $Ti[\![\ id\ ]\!]$ `open cnc_view` [(`open` $Ti[\![\ id_k\ ]\!]$)$[1..n/\!/k\ |\ \bot]$]

$Td[\![\ d_k\ ]\!][1..n/\!/k\ |\ \bot]$

`fact { none (+` $Fact[\![\ d_k\ ]\!]$)$[1..n/\!/k\ |\ \bot]$ `in ports + roles }`

`pred` $Ti[\![\ id\ ]\!]$`_constraints_local() {` $Cl[\![\ d_k\ ]\!][1..n/\!/k\ |$ `&&`$]$ `}`

`pred` $Ti[\![\ id\ ]\!]$`_constraints_global() {` $Cg[\![\ d_k\ ]\!][1..n/\!/k\ |$ `&&`$]$ `}`

`pred` $Ti[\![\ id\ ]\!]$`_constraints() {`

$Ti[\![\ id\ ]\!]$`_constraints_local() && ` $Ti[\![\ id\ ]\!]$`_constraints_global() }`

The translation functions are partially defined. For example $Fact[\![\ \_\ ]\!]$ is defined only for element type definitions. It is assumed that translation functions are not applied to Acme descriptions that are not defined with the functions. For example in case of $S\,[\,1..3/\!\!/k \mid sep\,]$ if $S\,[\,2/k\,]$ is not defined, it is equivalent to $S\,[\,1/k\,]\ sep\ S\,[\,3/k\,]$.

$$
\begin{aligned}
Td[\![\,\texttt{type}\ id\ \texttt{extends}\ et\ d^{*}\,]\!] &= \texttt{sig}\ Ti[\![\,id\,]\!]\ \texttt{extends}\ Tt[\![\,et\,]\!]\ \{\ td[\![\,d_k\,]\!]\,[\,1..n/\!\!/k \mid \texttt{,}\,]\ \}\\
Td[\![\,\texttt{type}\ id\ \texttt{=}\ pt\,]\!] &= \texttt{sig}\ Ti[\![\,id\,]\!]\ \texttt{extends}\ Tt[\![\,pt\,]\!]\ \{\ \}\\
Td[\![\,\texttt{type}\ id\ \texttt{=}\ \texttt{enum}\ id^{+}\,]\!] &= \begin{array}{l}\texttt{abstract sig}\ Ti[\![\,id\,]\!]\ \{\ \}\\ \texttt{one sig}\ Ti[\![\,id_k\,]\!]\,[\,1..n/\!\!/k \mid \texttt{,}\,]\ \texttt{extends}\ Ti[\![\,id\,]\!]\ \{\ \}\end{array}\\
Td[\![\,\texttt{type}\ id\ \texttt{=}\ \texttt{record}\ (\,id : pt\,)^{+}\,]\!] &= \texttt{sig}\ Ti[\![\,id\,]\!]\ \{\ Ti[\![\,id_k\,]\!] : Tt[\![\,pt_k\,]\!]\,[\,1..n/\!\!/k \mid \texttt{,}\,]\ \}\\
Td[\![\,\texttt{element}\ id : et\,]\!] &= \texttt{one sig}\ Ti[\![\,id\,]\!]\ \texttt{extends}\ Tt[\![\,et\,]\!]\ \{\ \}\\
Td[\![\,\texttt{property}\ id : pt\ [\texttt{=}\ v\,]\,]\!] &= \texttt{one sig}\ Ti[\![\,id\,]\!]\ \texttt{extends}\ Tt[\![\,pt\,]\!]\ \{\ \}\\
Td[\![\,\texttt{fun}\ id : t\ \texttt{(}\ (\,id : t\,)^{*}\ \texttt{)}\ e\,]\!] &= \begin{array}{l}\texttt{fun}\ Ti[\![\,id\,]\!]\ \texttt{(}\ Ti[\![\,id_k\,]\!] : Tt[\![\,t_k\,]\!]\,[\,1..n/\!\!/k \mid \texttt{,}\,]\ \texttt{)}\\ \texttt{:}\ Tt[\![\,t\,]\!]\ \{\ Tx[\![\,e\,]\!]_{obj}\ \}\end{array}\\
td[\![\,\texttt{element}\ id : et\,]\!] &= Ti[\![\,id\,]\!] : Tt[\![\,et\,]\!]\\
td[\![\,\texttt{property}\ id : pt\ [\texttt{=}\ v\,]\,]\!] &= Ti[\![\,id\,]\!] : Tt[\![\,pt\,]\!]\\
Tt[\![\,\texttt{set}\ t\,]\!] &= \texttt{set}\ Tt[\![\,t\,]\!]\\
Tt[\![\,\texttt{seq}\ t\,]\!] &= \texttt{Int -> lone}\ Tt[\![\,t\,]\!]\\
Tt[\![\,[\,id_1\ \texttt{/}\,]\ id_2\,]\!] &= [\,Ti[\![\,id_1\,]\!]\ \texttt{/}\,]\ Ti[\![\,id_2\,]\!]\\
Tt[\![\,\texttt{element}\,]\!] &= \texttt{Element}\\
Tt[\![\,\texttt{component}\,]\!] &= \texttt{Component}\\
Tt[\![\,\texttt{connector}\,]\!] &= \texttt{Connector}\\
Tt[\![\,\texttt{port}\,]\!] &= \texttt{Port}\\
Tt[\![\,\texttt{role}\,]\!] &= \texttt{Role}\\
Tt[\![\,\texttt{integer}\,]\!] &= \texttt{Int}\\
Ti[\![\,\texttt{size}\,]\!] &= \texttt{\#}\\
Ti[\![\,<identifier>\,]\!] &= <identifier>
\end{aligned}
$$

$$
\begin{aligned}
Fact[\![\,\texttt{type}\ id\ \texttt{extends}\ et\ d^{*}\,]\!] &= (Ti[\![\,id\,]\!]\ \texttt{<:}\ fact[\![\,d_k\,]\!])\,[\,1..n/\!/k \mid \texttt{+}\,] \\
fact[\![\,\texttt{element}\ id : et\,]\!] &= Ti[\![\,id\,]\!] \\
\\
Cg[\![\,\texttt{property}\ id : pt\ \texttt{=}\ e\,]\!] &= Ti[\![\,id\,]\!]\ \texttt{=}\ Tx[\![\,e\,]\!]_{obj} \\
Cg[\![\,\texttt{property}\ id : pt\ \texttt{=}\ \texttt{<}\ e^{*}\ \texttt{>}\,]\!] &= Ti[\![\,id\,]\!]\ \texttt{=}\ (\ \texttt{Int}\ k\ \texttt{->}\ Tx[\![\,e_k\,]\!]_{obj}\,)\,[\,1..n/\!/k \mid \texttt{+}\,] \\
Cg[\![\,\texttt{property}\ id : pt\ \texttt{=}\ \texttt{[}\ (\ id\ \texttt{=}\ e)^{\texttt{+}}\ \texttt{]}\,]\!] &= (Ti[\![\,id\,]\!].Ti[\![\,id_k\,]\!]\ \texttt{=}\ Tx[\![\,e_k\,]\!]_{obj}\,)\,[\,1..n/\!/k \mid \texttt{\&\&}\,] \\
Cg[\![\,id_c\,.\,id_p\ \texttt{to}\ id_n\,.\,id_r\,]\!] &= Ti[\![\,id_c\,]\!].Ti[\![\,id_p\,]\!]\ \texttt{=}\ Ti[\![\,id_n\,]\!].Ti[\![\,id_r\,]\!].\texttt{attachment} \\
Cg[\![\,\texttt{invariant}\ e\,]\!] &= Tx[\![\,e\,]\!]_{bex} \\
\\
Cl[\![\,\texttt{type}\ id\ \texttt{extends}\ et\ d^{*}\,]\!] &= \texttt{all self :}\ Tt[\![\,et\,]\!]\ \texttt{|}\ Cl[\![\,d_k\,]\!]\,[\,1..n/\!/k \mid \texttt{\&\&}\,] \\
Cl[\![\,\texttt{property}\ id : pt\ \texttt{=}\ e\,]\!] &= \texttt{self}.Cg[\![\,\texttt{property}\ id : pt\ \texttt{=}\ e\,]\!] \\
Cl[\![\,\texttt{property}\ id : pt\ \texttt{=}\ \texttt{<}\ e^{*}\ \texttt{>}\,]\!] &= \texttt{self}.Cg[\![\,\texttt{property}\ id : pt\ \texttt{=}\ \texttt{<}\ e^{*}\ \texttt{>}\,]\!] \\
Cl[\![\,\texttt{property}\ id : pt\ \texttt{=}\ \texttt{[}\ (\ id\ \texttt{=}\ e)^{\texttt{+}}\ \texttt{]}\,]\!] &= \texttt{self}.Cg[\![\,\texttt{property}\ id : pt\ \texttt{=}\ \texttt{[}\ (\ id\ \texttt{=}\ e)^{\texttt{+}}\ \texttt{]}\,]\!] \\
Cl[\![\,\texttt{invariant}\ e\,]\!] &= Cg[\![\,\texttt{invariant}\ e\,]\!]
\end{aligned}
$$

$$
\begin{aligned}
(\ as\ ,\ \texttt{obj}\ )_{bex} &= \texttt{int}\ as\ \texttt{= 1} \\
(\ as\ ,\ \texttt{bex}\ )_{bex} &= as \\
(\ as\ ,\ \texttt{obj}\ )_{num} &= \texttt{int}\ as \\
(\ as\ ,\ \texttt{num}\ )_{num} &= as \\
(\ as\ ,\ \texttt{bex}\ )_{obj} &= \texttt{if}\ as\ \texttt{then Int 1 else Int 0} \\
(\ as\ ,\ \texttt{num}\ )_{obj} &= \texttt{Int}\ as \\
(\ as\ ,\ \texttt{obj}\ )_{obj} &= as \\
(\ as\ ,\ k\ )_{img} &= as \\
(\ as\ ,\ k\ )_{kind} &= k \\
kind(\texttt{declaresType}) &= \texttt{bex} \\
kind(\texttt{declSubtype}) &= \texttt{bex} \\
kind(\texttt{attached}) &= \texttt{bex} \\
kind(\texttt{connected}) &= \texttt{bex} \\
kind(\texttt{reachable}) &= \texttt{bex} \\
kind(\texttt{contains}) &= \texttt{bex} \\
kind(\texttt{isSubset}) &= \texttt{bex} \\
kind(\texttt{\#}) &= \texttt{num} \\
kind(\texttt{parent}) &= \texttt{obj} \\
kind(\texttt{union}) &= \texttt{obj} \\
kind(\texttt{intersection}) &= \texttt{obj} \\
kind(\texttt{setdiff}) &= \texttt{obj} \\
kind(< identifier >) &= \texttt{obj}
\end{aligned}
$$

$$
\begin{aligned}
Tx[\![\ \texttt{!}\ e\ ]\!] &= (\ \texttt{!}\ Tx[\![\ e\ ]\!]_{bex}\ ,\ \texttt{bex}\ ) \\
Tx[\![\ e_1\ \texttt{or}\ e_2\ ]\!] &= (\ Tx[\![\ e_1\ ]\!]_{bex}\ \texttt{||}\ Tx[\![\ e_2\ ]\!]_{bex}\ ,\ \texttt{bex}\ ) \\
Tx[\![\ e_1\ \texttt{->}\ e_2\ ]\!] &= (\ Tx[\![\ e_1\ ]\!]_{bex}\ \texttt{=>}\ Tx[\![\ e_2\ ]\!]_{bex}\ ,\ \texttt{bex}\ ) \\
Tx[\![\ e_1\ \texttt{<->}\ e_2\ ]\!] &= (\ Tx[\![\ e_1\ ]\!]_{bex}\ \texttt{<=>}\ Tx[\![\ e_2\ ]\!]_{bex}\ ,\ \texttt{bex}\ ) \\
Tx[\![\ e_1\ \texttt{and}\ e_2\ ]\!] &= (\ Tx[\![\ e_1\ ]\!]_{bex}\ \texttt{\&\&}\ Tx[\![\ e_2\ ]\!]_{bex}\ ,\ \texttt{bex}\ ) \\
Tx[\![\ e_1\ \texttt{==}\ e_2\ ]\!] &= \begin{array}{l}(\ Tx[\![\ e_1\ ]\!]_{bex}\ \texttt{<=>}\ Tx[\![\ e_2\ ]\!]_{bex}\ ,\ \texttt{bex}\ ) \\ \quad if\ \ Tx[\![\ e_1\ ]\!]_{kind} = \texttt{bex}\ \ \vee\ \ Tx[\![\ e_2\ ]\!]_{kind} = \texttt{bex}\end{array} \\
Tx[\![\ e_1\ \texttt{==}\ e_2\ ]\!] &= \begin{array}{l}(\ Tx[\![\ e_1\ ]\!]_{num}\ \texttt{=}\ Tx[\![\ e_2\ ]\!]_{num}\ ,\ \texttt{bex}\ ) \\ \quad if\ \ Tx[\![\ e_1\ ]\!]_{kind} = \texttt{num}\ \ \vee\ \ Tx[\![\ e_2\ ]\!]_{kind} = \texttt{num}\end{array} \\
Tx[\![\ e_1\ \texttt{==}\ e_2\ ]\!] &= \begin{array}{l}(\ Tx[\![\ e_1\ ]\!]_{img}\ \texttt{=}\ Tx[\![\ e_2\ ]\!]_{img}\ ,\ \texttt{bex}\ ) \\ \quad otherwise\end{array} \\
Tx[\![\ e_1\ \texttt{!=}\ e_2\ ]\!] &= \begin{array}{l}(\ Tx[\![\ e_1\ ]\!]_{bex}\ \texttt{!<=>}\ Tx[\![\ e_2\ ]\!]_{bex}\ ,\ \texttt{bex}\ ) \\ \quad if\ \ Tx[\![\ e_1\ ]\!]_{kind} = \texttt{bex}\ \ \vee\ \ Tx[\![\ e_2\ ]\!]_{kind} = \texttt{bex}\end{array} \\
Tx[\![\ e_1\ \texttt{!=}\ e_2\ ]\!] &= \begin{array}{l}(\ Tx[\![\ e_1\ ]\!]_{num}\ \texttt{!=}\ Tx[\![\ e_2\ ]\!]_{num}\ ,\ \texttt{bex}\ ) \\ \quad if\ \ Tx[\![\ e_1\ ]\!]_{kind} = \texttt{num}\ \ \vee\ \ Tx[\![\ e_2\ ]\!]_{kind} = \texttt{num}\end{array} \\
Tx[\![\ e_1\ \texttt{!=}\ e_2\ ]\!] &= \begin{array}{l}(\ Tx[\![\ e_1\ ]\!]_{img}\ \texttt{!=}\ Tx[\![\ e_2\ ]\!]_{img}\ ,\ \texttt{bex}\ ) \\ \quad otherwise\end{array} \\
Tx[\![\ e_1\ \texttt{<}\ e_2\ ]\!] &= (\ Tx[\![\ e_1\ ]\!]_{num}\ \texttt{<}\ Tx[\![\ e_2\ ]\!]_{num}\ ,\ \texttt{bex}\ ) \\
Tx[\![\ e_1\ \texttt{>}\ e_2\ ]\!] &= (\ Tx[\![\ e_1\ ]\!]_{num}\ \texttt{>}\ Tx[\![\ e_2\ ]\!]_{num}\ ,\ \texttt{bex}\ ) \\
Tx[\![\ e_1\ \texttt{<=}\ e_2\ ]\!] &= (\ Tx[\![\ e_1\ ]\!]_{num}\ \texttt{=<}\ Tx[\![\ e_2\ ]\!]_{num}\ ,\ \texttt{bex}\ ) \\
Tx[\![\ e_1\ \texttt{>=}\ e_2\ ]\!] &= (\ Tx[\![\ e_1\ ]\!]_{num}\ \texttt{>=}\ Tx[\![\ e_2\ ]\!]_{num}\ ,\ \texttt{bex}\ ) \\
Tx[\![\ e_1\ \texttt{+}\ e_2\ ]\!] &= (\ Tx[\![\ e_1\ ]\!]_{num}\ \texttt{+}\ Tx[\![\ e_2\ ]\!]_{num}\ ,\ \texttt{num}\ ) \\
Tx[\![\ e_1\ \texttt{-}\ e_2\ ]\!] &= (\ Tx[\![\ e_1\ ]\!]_{num}\ \texttt{-}\ Tx[\![\ e_2\ ]\!]_{num}\ ,\ \texttt{num}\ )
\end{aligned}
$$

$$
\begin{aligned}
Tx[\![\,[\,id_1\ \texttt{/}\,]\ id_2\ \texttt{(}\ e^{\texttt{*}}\ \texttt{)}\,]\!] &= \begin{array}{l}(\ [\,Ti[\![\,id_1\,]\!]\ \texttt{/}\,]\ Ti[\![\,id_2\,]\!]\ \texttt{(}\ Tx[\![\,e_k\,]\!]_{obj}\,[\,i..j /\!/ k \mid \texttt{,}\,]\ \texttt{)} \\ ,\ kind(Ti[\![\,id_2\,]\!])\ )\end{array} \\
Tx[\![\,id\,.\,e\,]\!] &= (\ Ti[\![\,id\,]\!].Tx[\![\,e\,]\!]_{obj}\,,\ \texttt{obj}\ ) \\
Tx[\![\,e\,.\texttt{components}\,]\!] &= (\ \texttt{Component}\,,\ \texttt{obj}\ ) \\
Tx[\![\,e\,.\texttt{connectors}\,]\!] &= (\ \texttt{Connector}\,,\ \texttt{obj}\ ) \\
Tx[\![\,e\,.\texttt{ports}\,]\!] &= (\ Tx[\![\,e\,]\!]_{obj}\,.\texttt{ports}\,,\ \texttt{obj}\ ) \\
Tx[\![\,e\,.\texttt{roles}\,]\!] &= (\ Tx[\![\,e\,]\!]_{obj}\,.\texttt{roles}\,,\ \texttt{obj}\ ) \\
Tx[\![\,e\,.\texttt{attachedports}\,]\!] &= (\ Tx[\![\,e\,]\!]_{obj}\,.\texttt{attachment}\,,\ \texttt{obj}\ ) \\
Tx[\![\,e\,.\texttt{attachedroles}\,]\!] &= (\ Tx[\![\,e\,]\!]_{obj}\,.\sim\texttt{attachment}\,,\ \texttt{obj}\ ) \\
Tx[\![\,\texttt{forall}\ id : e_t\ \texttt{|}\ e\,]\!] &= (\ \texttt{all}\ Ti[\![\,id\,]\!] : Tx[\![\,e_t\,]\!]_{obj}\ \texttt{|}\ Tx[\![\,e\,]\!]_{bex}\,,\ \texttt{bex}\ ) \\
Tx[\![\,\texttt{exists}\ id : e_t\ \texttt{|}\ e\,]\!] &= (\ \texttt{some}\ Ti[\![\,id\,]\!] : Tx[\![\,e_t\,]\!]_{obj}\ \texttt{|}\ Tx[\![\,e\,]\!]_{bex}\,,\ \texttt{bex}\ ) \\
Tx[\![\,\texttt{exists unique}\ id : e_t\ \texttt{|}\ e\,]\!] &= (\ \texttt{one}\ Ti[\![\,id\,]\!] : Tx[\![\,e_t\,]\!]_{obj}\ \texttt{|}\ Tx[\![\,e\,]\!]_{bex}\,,\ \texttt{bex}\ ) \\
Tx[\![\,\{\ e^{\texttt{*}}\ \}\,]\!] &= (\ \texttt{none}\ (\texttt{+}\ Tx[\![\,e_k\,]\!]_{obj})\,[\,1..n /\!/ k \mid \bot\,]\,,\ \texttt{obj}\ ) \\
Tx[\![\,\texttt{select}\ id : e_t\ \texttt{|}\ e\,]\!] &= (\ \{\ Ti[\![\,id\,]\!] : Tx[\![\,e_t\,]\!]_{obj}\ \texttt{|}\ Tx[\![\,e\,]\!]_{bex}\ \}\,,\ \texttt{obj}\ ) \\
Tx[\![\,\texttt{collect}\ id_1\,.\,id_2 : e_t\ \texttt{|}\ e\,]\!] &= (\ \{\ Ti[\![\,id_1\,]\!] : Tx[\![\,e_t\,]\!]_{obj}\ \texttt{|}\ Tx[\![\,e\,]\!]_{bex}\ \}\,.Ti[\![\,id_2\,]\!]\,,\ \texttt{obj}\ ) \\
Tx[\![\,\texttt{true}\,]\!] &= (\ \texttt{Int 1}\,,\ \texttt{obj}\ ) \\
Tx[\![\,\texttt{false}\,]\!] &= (\ \texttt{Int 0}\,,\ \texttt{obj}\ ) \\
Tx[\![\,<integer>\,]\!] &= (\ <integer>\,,\ \texttt{num}\ ) \\
Tx[\![\,id\,]\!] &= (\ Ti[\![\,id\,]\!]\,,\ \texttt{obj}\ )
\end{aligned}
$$

```
////////////////////////////////////////////////////////////////////////
// This module models the built-in features (types and functions) of the
// Acme language. Therefore any translated module of an Acme style should
// import it. This module additionally contains definitions of several
// predefined funtions for convenience.
////////////////////////////////////////////////////////////////////////

module cnc_view

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

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

fact {~ports = component && ~roles = connector}

abstract sig System {components: set Component, connectors: set Connector}
one sig self extends System {}
fact{ self.components = Component && self.connectors = Connector }

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

// Returns true if 'element' declares that it satisfies 'type'.
pred declaresType [element: univ, type: set univ] {
  element in type
}

// Returns true if 'subtype' declares that it is a subtype of 'supertype'.
pred declaredSubtype [subtype: set univ, supertype: set univ] {
  subtype in supertype
}

// "satisfiesType" should be translated into a type predicate (tentative).
// "typesDeclared" cannot be modeled because it returns high-order value.
// "superTypes" cannot be modeled because it returns high-order value.

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

```
// Returns true if role 'r' is attached to port 'p'.
pred attached [r: Role, p: Port] {
  r -> p in attachment
}

// Returns true if connector 'n' is attached to component 'c'.
pred attached [n: Connector, c: Component] {
  n -> c in roles.attachment.component
}

// Returns true if component 'c1' is directly connected to component 'c2'
// by at least one connector.
pred connected [c1: Component, c2: Component] {
  some r1,r2: Role |
  some p1,p2: Port |
    disj[r1, r2] &&
    attached[r1, p1] && parent[p1] = c1 &&
    attached[r2, p2] && parent[p2] = c2 &&
    parent[r1] = parent[r2]
}

// Returns true if port 'p1' is directly connected to port 'p2'
// by at least one connector.
pred connected [p1: Port, p2: Port] {
  some r1,r2: Role |
    disj[r1, r2] &&
    attached[r1, p1] &&
    attached[r2, p2] &&
    parent[r1] = parent[r2]
}

// Returns true if component 'c2' is reachable from component 'c1'.
pred reachable [c1: Component, c2: Component] {
  some cs: set Component-c1-c2 |
    ( one c: cs+c2 | connected[c1, c] ) &&
    ( one c: cs+c1 | connected[c2, c] ) &&
    all c: cs |
      some c3, c4: cs+c1+c2-c |
        disj[c3, c4] &&
        connected[c, c3] &&
        connected[c, c4] &&
        no c5: cs+c1+c2-c-c3-c4 | connected[c, c5]
}

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

```
//.....................................................................

// Returns the component in which port 'p' is instantiated.
fun parent [p: Port]: Component {
  p.component
}

// Returns the connector in which role 'r' is instantiated.
fun parent [r: Role]: Connector {
  r.connector
}

// "parant" is implemented only for ports and roles.

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

// Returns true if set 'b' contains object 'a'.
pred contains [a: univ, b: set univ] {
  a in b
}

// Returns true if set 'a' is a subset of set 'b'.
pred isSubset [a: set univ, b: set univ] {
  a in b
}

// Returns the union of set 'a' and set 'b'.
fun union [a: set univ, b: set univ]: univ {
  a + b
}

// Returns the intersection of set 'a' and set 'b'.
fun intersection [a: set univ, b: set univ]: univ {
  a & b
}

// Returns the difference of set 'a' and set 'b'.
fun setdiff [a: set univ, b: set univ]: univ {
  a - b
}

// "size" should be translated into the cardinality operator #
//   because Alloy functions cannot return integer value directly.
// "flattern" cannot be directly modeled because the input is high-order value.
// "sum" and "product" cannot be directly modeled because they require recursion.
```

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

// Returns true if components in 'cs' form star structure.
pred predefined_topology_star [cs: set Component] {
  one core: cs |
    ( all node: cs-core | connected[core, node] ) &&
    no node1,node2: cs-core | disj[node1,node2] && connected[node1, node2]
}

// Returns true if components in 'cs' form linear structure.
pred predefined_topology_linear [cs: set Component] {
  some c1, c2: cs |
    disj[c1, c2] &&
    ( one c: cs-c1 | connected[c1, c] ) &&
    ( one c: cs-c2 | connected[c2, c] ) &&
    all c: cs-c1-c2 |
      some c3, c4: cs-c |
        disj[c3, c4] &&
        connected[c, c3] &&
        connected[c, c4] &&
        no c5: cs-c-c3-c4 | connected[c, c5]
}

// Returns true if components in 'cs' form ring structure.
pred predefined_topology_ring [cs: set Component] {
  #cs != 0
  all c: cs | predefined_topology_linear[cs-c]
}

// Returns true if components in 'cs' do not form a cycle.
pred predefined_no_cycle [cs: set Component] {
  no cs': set cs | predefined_topology_ring[cs']
}

// Returns true if each component in 'cs' is connected to at least one component.
pred predefined_no_unconnected_component [cs: set Component] {
  all c: cs | some c': Component | connected[c, c']
}

// Returns true if each component in 'cs' is attached to at least one connector.
pred predefined_no_unattached_component [cs: set Component] {
  all c: cs | some roles.attachment.component.c
}
```

```
// Returns true if each connector in 'ns' is attached to at least one component.
pred predefined_no_unattached_connector [ns: set Connector] {
  all n: ns | some n.roles.attachment.component
}

// Returns true if all ports of each component in 'cs' is attached.
pred predefined_no_unattached_port [cs: set Component] {
  all p: component.cs | some attachment.p
}

// Returns true if all roles of each connector in 'ns' is attached.
pred predefined_no_unattached_role [ns: set Connector] {
  all r: connector.ns | some r.attachment
}
```

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.