# Capturing Software Architecture Design Expertise with Armani

Version 1.0

Robert T. Monroe

October, 1998
CMU-CS-98-163

Carnegie Mellon

# Capturing Software Architecture Design Expertise with Armani

Version 1.0

Robert T. Monroe

October, 1998
CMU-CS-98-163

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Abstract:**

*Armani* is a language for capturing software architecture design expertise and specifying software architecture designs. This document describes the Armani language in detail with specifications for the language syntax and semantics, as well as examples illustrating common usage.

The research reported here was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grants F33615-93-1-1330 and N66001-95-C-8623; and by National Science Foundation under Grant CCR-9357792 and a Graduate Research Fellowship. Support was also provided by the Kodak Corporation in the form of a Graduate Research Fellowship. 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 Wright Laboratory, the US Department of Defense, the United States Government, or the National Science Foundation. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon.

**Keywords**: Software architecture, software design, architecture description languages, design rules, design constraints, software architecture design expertise, software design expertise

Chapter

1

# Armani Project Overview

The Armani design language provides software architects with a rich language for describing software architecture designs, constraints on the evolution of those designs, and *architectural design expertise*. The types of architectural design expertise that can be captured with the Armani language include *design vocabulary*, *design rules*, and *architectural styles*. In addition to capturing software architecture design expertise, the Armani design language is also a full-fledged software architecture description language in its own right.

The Armani design language is also used as the configuration language for the Armani software architecture design environment. This environment can be rapidly and incrementally configured to use the design expertise captured with the Armani design language. This captured design expertise guides software architects using the Armani environment in the creation of appropriate software system architectures and provides an analytical base to confirm that the system designs produced obey the design rules specified.

This document provides a detailed description of the syntax and the semantics of the Armani design language and illustrates its use as a design specification language for capturing both architectural design expertise and the specifications of individual software architectures. The design environment, and details on how the design language can be used to customize the environment, are not discussed in this document. They will be described in detail in a subsequent technical report.

This document is the first public draft of the Armani Language Reference Manual. It is intended as a detailed proposal for the language that will allow people to begin experimenting with the design language and provide feedback on ways in which it may be improved. Please read the manual with these goals in mind.

## Language Design Fundamentals

As with any design language, Armani[1] must make tradeoffs between competing goals and constituencies. This section enumerates the fundamental principles and design decisions underlying Armani, and discusses the tradeoffs made in defining the language.

[1] For the remainder of this document the single term *Armani* will refer to the Armani design language. The design environment associated with the language will be referred to as the *Armani design environment*.

## Language Design Principles

The following principles have guided the design of the Armani language.

- Armani focuses on the description of system structure and how that system structure may evolve over time rather than the dynamic run-time behavior of systems.

- Armani is a fundamentally declarative language. Armani specifications describe architectural structure and constraints on that structure; they do not describe operations for modifying or deleting architectural structures. At an informal level, an Armani specification describes the structure of a system and the constraints that must hold on that system. The Armani description itself describes neither how to create the structure of a system, nor how to check that the constraints are satisfied. These operations are embedded in the language processing tools, rather than the language itself.

- Armani uses a single integrated language for describing instances of system architectures and capturing abstract architectural design expertise.

- Armani is built as an extension to the Acme architecture description and interchange language to encourage easy translation to and from Acme. This interchangability should allow Armani users to take advantage of existing and future Acme-based tools and infrastructure.

- The Armani constraint language allows a style or system designer to differentiate between *invariant constraints* (fundamental design constraints) and *heuristics* (rules of thumb).

- The Armani constraint language uses a first-order predicate logic-based language to express and package design constraints. This language stays decidable by allowing variables to be quantified only over finite sets.

- Armani uses the following seven core constructs for describing instances of architectural designs: *components*, *connectors*, *ports*, *roles*, *systems*, *representations*, and *properties*. These constructs are described in greater detail in the following section.

- Armani supports the capture of reusable abstract design expertise with the following core constructs: *design element types*, *property types*, *design invariants*, *heuristics*, and *analyses*, and *architectural styles*. These constructs are also described in greater detail in the following section.

## Design Tradeoffs

In designing the Armani language, there were two critical classes of tradeoffs to be made. The first class of tradeoff concerns the conflicting goals of making the language well suited for use as a stand-alone textual design language for humans vs. making a machine-manipulable storage format for architectural designs and design expertise. In general, most decisions favored making the language flexible, expressive, and concise for human language users. These decisions were, however, tempered by the need to make everything that can be expressed in the language readily usable by automated tools.

The second critical tradeoff in designing the Armani language deals with the need to make the language rich and intuitive for the software architects and environment designers that make up Armani's target audience, while keeping the language simple, elegant, and tractable enough to support automated tooling and some degree of semantic formalization. Armani walks a fine line between these two goals, tending to favor richness and naturalness of expression only when doing so does not severely compromise the semantic tractability of the language.

## Language Structure Overview

Armani provides seven core constructs for describing instances of architectural designs:

**Components** represent the primary computational units of a system.

**Connectors** represent and mediate interactions between components.

**Ports** represent components' external interfaces.

**Roles** represent connectors' external interfaces.

**Systems** are collections of components, connectors, and a description of the topology of the components and connectors.

**Properties** are annotations that store additional information about elements (components, connectors, ports, roles, and systems).

**Representations** allow a component, connector, port, or role to describe its design in greater detail by specifying a sub-architecture that refines the parent element.

Chapter 2 provides a detailed overview of these constructs, including informal discussion and examples of their their usage, an informal syntax specification, and a detailed semantic specification of their meanings.

To support the capture of reusable abstract design expertise, Armani provides the following core language constructs:

**Design Element Types** are predicates that describe the fundamental structure and constraints of design vocabulary elements.

**Property Types** are predicates that describe the type and structure of properties.

**Design Invariants** capture design constraints that must hold in a design.

**Design Heuristics** capture suggestions for creating effective designs.

**Design Analyses** are functions computed over instances of architectural designs. Design analyses may be specified in the Armani predicate language or as external functions linked directly into the environment.

**Architectural Styles** aggregate collections of the previous five constructs for capturing design expertise to form a coherent collection of reusable design expertise.

Chapter 3 describes the constructs for capturing abstract architectural design expertise in detail, including informal discussion and examples of their usage, an informal syntax specification, and a more rigorous specification of their semantics.

The full Armani predicate language used for specifying design invariants, heuristics, and analyses is presented in Chapter 4. Chapter 5 walks through a set of examples that exercise the full Armani language. Finally, a detailed specification of the Armani design language syntax is given in Appendix A.

Chapter

2

# Specifying Architectural Structure

## Overview of the Structural Instance Language

Armani uses seven core constructs to specify a system's architectural structure – components, connectors, ports, roles, systems, representations, and properties. Each of these constructs is described below in detail.

### Components and Ports

*Components* represent the primary computational elements and data stores of a system. Intuitively, they correspond to the boxes in box-and-line descriptions of software architectures. Typical examples of components include clients, servers, filters, objects, blackboards, and databases.

Components' interfaces are defined by a set of *ports*. Each port identifies a point of interaction between the component and its environment. A component may provide multiple interfaces by using multiple ports. A port can represent an interface as simple as a single procedure signature or more complex interfaces such as a collection of procedure calls that must be invoked in a specified order or an event multi-cast interface point.

### Connectors and Roles

*Connectors* represent interactions among components. Computationally speaking, connectors mediate the communication and coordination activities among components. Informally they provide the "glue" for architectural designs and correspond to the lines in box-and-line descriptions. Examples include simple forms of interaction, such as pipes, procedure call, and event broadcast. But connectors may also represent more complex interactions, such as a client-server protocol or a SQL link between a database and an application.

Like components, connectors have explicitly specifiable interfaces that are defined by a set of *roles*. Each role of a connector defines a participant of the interaction represented by the connector. Binary connectors have two roles such as the *caller* and *callee* roles of an RPC connector, the *reading* and *writing* roles of a pipe, or the *sender* and *receiver* roles of a message passing connector. Other kinds of connectors may have more than two roles. For example, an event broadcast connector might have a single *event-announcer* role and an arbitrary number of event-receiver roles.

## Systems

*Systems* represent configurations of components and connectors. A system includes (among other things) a set of components, a set of connectors, and a set of attachments that describe the topology of the system. An attachment describes the relationship between a connector and a component by associating a port interface on a component with a role interface on a connector.

### Example

To illustrate the simple structure language, Example 2.1 describes a trivial architectural specification of a system with two components – a *client* and a *server* -- connected by an *rpc* connector. The *client* component is declared to have a single *send-request* port, and the server has a single *receive-request* port. The connector has two roles designated *caller* and *callee*. The topology of this system is declared by the set of *Attachments*.

```
System simple_cs = {
    Component client = { Port send-request }
    Component server = { Port receive-request }
    Connector rpc = { Roles {caller, callee} }
    Attachments {
                client.send-request to rpc.caller ;
                server.receive-request to rpc.callee }
}
```

**Example 2.1: Simple client-server specification in Armani**

## Representations

Complex architectural designs generally require hierarchical descriptions to provide a specification that is both sufficiently detailed and tractable for individual designers to understand. Recognizing this, Armani supports hierarchical decomposition of architectures. Specifically, any component or connector can be represented by one or more detailed, lower-level descriptions. Each such description is termed a *representation.* The use of multiple representations allows Armani to encode multiple views of architectural entities (although there is nothing built into Armani that supports resolution of inter-view correspondences). It also supports the description of encapsulation boundaries, as well as multiple refinement levels.

## Properties

There is clearly more of interest in an architectural description of a system than the topology of its components and connectors. The *property* construct in Armani provides a mechanism for annotating designs and design elements with detailed, generally non-structural, information. All of the architectural entities described to this point (components, connectors, ports, roles, systems, and representations) can be annotated with properties.

An Armani property is a name, type, value triple. The Armani type system is described in detail in section 3.

## Examples

The following example describes an extension to the simple client-server system given in Example 2.1. This example has been annotated with properties that describe characteristics of the structural elements of the system.

```
System simple_cs = {
    Component client = {
        Port send-request;
        Properties {  request-rate : float = 17.0;
                      source-code : external-file = "CODE-LIB/client.c" }}

    Component server = {
        Port receive-request;
        Properties { idempotent : boolean = true;
                     max-concurrent-clients : integer = 1;
                     source-code : external-file =  "CODE-LIB/server.c" }}

    Connector rpc  = {
        Role caller;
     Role callee;
     Properties { synchronous : boolean = true;
                  max-roles : integer = 2;
                  protocol : Wright = "..." }}

    Attachments {   client.send-request to rpc.caller ;
                    server.receive-request to rpc.callee }
}
```

**Example 2.2: Simple-client-server system with properties**

## Armani structural language syntax

The full Armani BNF is included in Appendix A (along with a description of the BNF notation used here). A short BNF for the core Armani structural language is provided here. The simplicity of the core structural language is reflected in the language's short and simple BNF. This syntax description will be extended and improved throughout this document as new language features are introduced.

*System* ::= *System Name* = { *EntityDecl** } ;

*EntityDecl* ::= *ComponentDecl*
| *Connector-Decl*
| *Port-Decl*
| *Role-Decl*
| *Property-Decl*
| *Rep-Decl*
| *Attachments-Decl*

*GenericDecl* ::= *PropertyDecl* | *RepDecl*;

*ComponentDecl* ::= ***Component*** *Name* = { (*PortDecl* | *GenericDecl*)* } ;

*ConnectorDecl* ::= ***Connector*** *Name* = { (*RoleDecl* | *GenericDecl*)* } ;

*PortDecl* ::= ***Port*** *Name* = { *GenericDecl* * } ;

*RoleDecl* ::= ***Role*** *Name* = { *GenericDecl* * } ;

*AttachmentDecl* ::= ***Attachments*** { (*PortName* ***To*** *RoleName*)* } ;

*Name* ::= [a-zA-Z][a-zA-Z0-9_\-+]*

**Properties**

*PropertyDecl* ::= ***Property*** *Name* [ **:** *Type* ] = *Value*
| ***Properties*** { (*Name* [ : *Type* ] = *Value* ; ) * } ;

**Representations**

*RepDecl* ::= ***Representation*** [ *Name* ]= *System*
***Bindings*** = { (*Name* ***To*** *Name*)* } ;

## Armani Semantics

Throughout this document, sections providing semi-formal specifications of the semantics of each language construct follow an informal discussion of that construct's use and a presentation of the construct's syntax. Using this approach, the semantic structure of the language is incrementally presented in small, digestable pieces. To properly read the semantic specifications it is important to understand both the goals underlying, and the audience for, the presentation of the Armani language semantics. The primary and overriding goal of specifying the Armani design language semantics is to provide an unambiguous description of the meaning of Armani specifications that the target audience is able to understand and use. Elegance and conciseness of the specification are secondary goals.

The intended audience for the semantic specification and, in fact, the entire Armani Language Reference Manual, consists of the following three groups, listed in descending order of importance.

1. **Architectural Style Developers** are the people who use the Armani design language to capture and package reusable architectural design expertise. This includes people who only need to capture architectural design expertise with the Armani language in an architectural style, as well as custom environment builders who use architectural style specifications to customize an Armani design environment.

2. **Software Architects** use architectural styles and their associated environments to design, specify, and evaluate architectures for specific software systems. They may also be style developers, but this need not be the case.

3. **Armani Tool Developers** write tools to operate on Armani descriptions. It is not yet clear if there will be a significant market for third-party Armani tools, but a clean specification of the language syntax and semantics will reduce the barriers to building such tools, encouraging the development of a third-party tool market.

It is important to note that most members of the groups specified above are unlikely to be experts in programming language semantics or formal methods. The schema used for specifying the semantics of the language must, however, be comprehensible to the target audience. Thererfore, a semi-formal specification schema is used to specify the semantics of the Armani design language. Formalization is used where it is deemed either necessary or particularly helpful in clarifying the meaning of a language construct. Extensive use of sophisticated semantic notations is avoided where the use of such notations is more likely to confuse the target audience than to clarify semantic issues for them. The level of mathematical sophistication required to understand the specification is targeted at experienced systems engineers and designers with the equivalent of a bachelors degree in computer science or engineering who do not necessarily possess any extensive knowledge of formal language specification techniques.

## Overview of semantic evaluation

The remainder of this section presents an overview of the steps taken in processing an Armani design description, the semantic domain into which the Armani syntax is translated, and the notation and form of the equations used to specify the translation from abstract syntax to the semantic domain.

### Language processing steps

The processing of an Armani specification occurs as a series of discrete steps, each step building on the previous step. The four primary processing steps are described below.

1. Parse Armani file to create initial Abstract Syntax Tree (AST).

2. Resolve the fully qualified names and scope of styles, types, elements, properties, representations, and design rules.

3. Translate the AST to canonical Armani semantic representation.

4. Perform type-correctness and consistency checks on the canonical representation (against both explicitly defined type definitions and implicit type specifications given with instance-level elements).

The well-formedness of expressions is verified in steps one and two. Well-formedness equations are supplied to flag ill-formed expressions before they are considered for semantic analysis in step three. The meaning of an ill-formed expression is undefined and can not be analyzed semantically. The semantic equations given throughout this document primarily assist in the third step, conversion from abstract syntax to a canonical Armani semantic representation. Type-correctness and consistency checks are performed in the semantic domain on the canonical representation. The meaning of type-correctness and internal design consistency is specified with algorithms that operate on canonical semantic

representations to determine whether a given instance satisfies a given type, and whether a given instance is internally consistent. The concepts of both type-correctness and internal consistency will be introduced later in this document.

### Canonical semantic representations

The canonical Armani semantic representation is simply a collection of tuples that represent the structure of Armani element instances, properties, types, design rules, and styles. The exact structure of the tuples varies for each of these classes of Armani constructs. As the semantics are presented for each Armani construct, the structure of that construct's tuple in the semantic domain is presented. The values stored in the fields of the tuples may be scalar values, other tuples, or sets of tuples, depending on the type of entity being represented. Because tuples can store other tuples (as well as references to other tuples), the semantic representation of an Armani structure emerges as a forest of trees, where each node of each tree is a tuple that may contain other tuples as children.

Denotational semantic equations are the preferred form for describing the translation of abstract Armani syntax into its canonical semantic representation. Consider the denotational equation:

$$\mathcal{M}[\ \langle\text{Armani Expression}\rangle\ ]\ c = [\ c \mid\ c.x \leftarrow 4\ ]$$

The above expression is read "The meaning of <Armani Expression> (which is specified in Armani's abstract syntax), in the context c is the context c with the new value '4' assigned to the tuple field x of context c". All semantic equations are evaluated within a context. A context is simply a tuple whose values can be modified as a result of the evaluation of the syntactic expression. It is also possible for an equation to return a new context that may or may not be linked in some way to the original context, as the following example shows:

$$\mathcal{M}[\ \langle\text{Armani Expression}\rangle\ ]\ c = [\ c' \mid\ c.s \leftarrow c',\ c'.x \leftarrow 104\ ]$$

In this example, the meaning of the expression on the left hand side of the equation is the new context c', such that c' is linked to context c by the field c.s of context c, and field c'.x has the value '104'.

In most cases, the context of evaluation will be an element tuple or a design space tuple (discussed in subsequent sections or chapters), but other contexts are used occasionally as needed. When the context of evaluation is an element, the context will be annotated with a variable named *e*. When the context of evaluation is unknown, but may be any kind of context (e.g. design space, element, or something else), the context is indicated with a *c*.

The result of the semantic evaluation of an entire Armani description will generally be the global design space tuple that serves as the initial context. After semantic evaluation, this global design space tuple will hold the forest of trees that represents the structures of the Armani design that was evaluated.

## Semantics of the Structural Instance Language

### Abstract syntax

Before providing a semantic specification for the instance language the concrete syntax specified in the beginning of this chapter needs to be simplified and represented with the following abstract syntax. The abstract syntax simply strips some of the syntactic sugar from the concrete syntax. Semantic equations operate over the abstract syntax, rather than the concrete syntax.

| | | |
|---|---|---|
| *Element* | ::= | *Category Name = ElementContents* |
| *Category* | ::= | ***System*** *|* ***Component*** *|* ***Connector*** *|* ***Port*** *|* ***Role*** |
| *ElementContents* | ::= | *(Element | Property | Representation | Attachment)** |
| *Attachment* | ::= | ***Attachment*** *( Element, Element )* |
| *Property* | ::= | ***Property*** *Name PropValue* |
| *Representation* | ::= | ***Representation*** *Name = Element Bindings* |
| *Bindings* | ::= | ***Bindings*** *( Element, Element )** |
| *Name* | ::= | *<valid Armani identifier >* |

### Instance semantics

As an introduction to the semantics of the Armani design language, this section presents some simple equations for converting Armani expressions in the language's abstract syntax into the expressions' underlying semantic representations. This semantic treatment assumes that all names have been fully resolved and include their full qualification for purposes of name comparison. In this approach, denotational semantic equations describe how tuples of values, sets, and other tuples are built from Armani declarations. All evaluations of the simple instance-based expression language described in this section are done in the context of an element description $e = (n, c, s, p, r, a)$. The notation $\mathcal{M}[\,s\,]\,e$ is read "the meaning of statement s in the context of element $e$". To bootstrap the context for this simple introduction, semantic analysis begins in the context of an empty, unnamed element in which the toplevel system is declared. Additional types of scopes (and their associated rules) will be added as more sophisticated language expressions are added.

**Attachments**. An Armani attachment is represented as a triple of the form $(c, n_1, n_2)$ where $c$ is a category (always Attachment) and each $n_i$ is an identifier that specifies the name of an element.

Well-formedness rules:

$n_i$ : Identifier
c : Category

((Port($n_1$) **and** Role($n_2$)) **or** (Role($n_1$) **and** Port($n_2$)))
**and** Parent(Parent($n_1$)) = Parent(Parent($n_2$))

Meaning: $\mathcal{M}$[ Attachment $n_1$ $n_2$ ] e = [ e | e.a ← e.a ∪ { (Attachment, $n_1$, $n_2$) } ]

**Bindings**. An Armani binding is a triple of the form (*c, $n_1$, $n_2$*) where each $n_i$ is an identifier that specifies the name of an element and *c* stores the category of the relation (always Binding). The context in which a binding is evaluated is a representation tuple *e.r*, rather than an element *e*.

Well-formedness rules:

$n_i$ : Identifier
c : Identifier
pr : Representation = Parent(b)
ps : System = Parent(b).system
pe : Element = Parent(pr)
$N_{ps}$ : Namespace = Namespace(ps)
$N_{pe}$ : Namespace = Namespace(pe)

$n_1 \in N_{ps}$ **and** $n_2 \in N_{pe}$
**and** ((Port($n_1$) **and** Port($n_2$)) **or** (Role($n_1$) **and** Role($n_2$)))

Meaning: $\mathcal{M}$[Binding $n_1$ $n_2$] e.r = [ e.r | e.r.b ← e.r.b ∪ { (Binding, $n_1$, $n_2$) } ]

**Properties**. An Armani property is a tuple of the form (*c, n, t, v*), where *n* represents the fully qualified name of the property, *t* specifies the type of the property, *v* specifies the value of the property, and *c* stores the category of the tuple (always Property). The semantics of property types and the visibility of type names are described in detail in chapter 3, which describes Armani's type system. For the purposes of this section, a type *t* is a fully resolved reference to a property type. If the name can't be fully resolved this is an error that should be caught before the semantic analysis stage. The property well-formedness rules and semantic equation follow:

Well-formedness rules:

n : Identifier
t : PropertyTypeReference
v : PropertyValue

(n != "")
**and** (not IsArmaniKeyword(n))
**and** (ResolvedPropertyType(t))
**and** (SatisfiesType(v, t))

Meaning:

$\mathcal{M}$[ Property n : t = v ] e =
**if** (n ∉ Names(e.r)) **and** (n ∉ Names(e.s)) **and** (n ∉ Names(e.p)) **then**
[ e | e.p ← e.p ∪ { (Property, n, t, v) } ] **else** *Error*

**Representations.** An Armani representation is a tuple of the form (*c, n, e', b*).

Well-formedness rules:

c : Category
n : Identifier
e' : Element
b : Set{Binding}

System(e')

Meaning:

$\mathcal{M}$[ Representation n = e' b ] e =
**if** (n ∉ Names(e.r)) **and** (n ∉ Names(e.s)) **and** (n ∉ Names(e.p)) **then**
[ e | e.r ← e.r ∪ { (Representation, n, $\mathcal{M}$[e']e.r, $\mathcal{M}$[b]e.r) } ] **else** *Error*

**Declaration statement sequencing.** An instance declaration statement $s_i$ specifies an element, property, attachment, representation, or binding. Composition of declaration statements is handled by the following rule:

Meaning: $\mathcal{M}[ s_1; \ldots; s_n ]\, e = \mathcal{M}[ s_n ]\, \mathcal{M}[ s_{n-1} ] \ldots \mathcal{M}[ s_1 ]\, e$

**Empty statement sequence.** The meaning of an empty sequence of declaration statements is handled by the following rule:

Meaning: $\mathcal{M}[ \{\} ]\, e = e$

**Elements.** An Armani element *e* is a tuple of the form *e* = (*n, c, s, p, r, a*).

Informally:

*n* = name of the element
*c* = category of the element
*s* = set of elements that define *e's* substructure (e.g. ports, roles, etc.)
*p* = set of properties that define the properties of *e.*
*r* = set of representations that define *e's* subarchitectures
*a* = set of attachments that define *e's* topology, iff e is a system.

The well-formedness rule for elements uses the function *Names : Set → Set* that creates a new set of name identifiers that is a projection of the "n" field (the name field) of all tuples in the original set of tuples (e.g. Names({(n=foo, ...), (n=bar,...)}) returns the set {foo,bar}).

This well-formedness rule declares the legal substructure for various categories of elements, and no entities declared as part of this element can share the same identifier.

Well-formedness rules:

n : Identifier
c : Category
s : Set{Element}
p : Set{Property}
r : Set{Representation}
a : Set{Attachment}

$N_e$ : Namespace = Namespace(e)

Port(e) → (s = { })
**and** Role(e) → (s = { })
**and** Component(e) → (forall e' in s | Port(e'))
**and** Connector(e) → (forall e' in s | Role(e'))
**and** System(e) → (forall e' in s | Component(e') **or** Connector(e'))
**and** (a != {}) → System(e)

**and** forall x, y in p | (x.n = y.n) → (x = y)
**and** forall x, y in r | (x.n = y.n) → (x = y)

**and** (Names(s) ∩ Names(p) = { })
**and** (Names(r) ∩ Names(p) = { })
**and** (Names(s) ∩ Names(r) = { })

Meaning:

*Category* is a meta-variable that represents the category declaration of the element.

$\mathcal{M}$[ *Category* n = { $s_1$ ;...; $s_n$ } ] e =
**if** (n ∉ Names(e.r)) **and** (n ∉ Names(e.s)) **and** (n ∉ Names(e.p)) **then**
$\mathcal{M}$[ { $s_1$ ;...; $s_n$ } ] [ e' | e.s ← e.s ∪ { e' }, e'.n ← n, e'.c ← *Category*,
e'.s ← {}, e',p ← {}, e'.r ← {}, e'.a ← {} ]

## Extending the simple instance language

The structural language described thus far is a subset of the complete Armani structural instance language. The constructs that need to be added to this description include typed properties, meta-properties, {...} extended with {...} clauses, and instance-based invariants and heuristics. Instance-based invariants and heuristics simply define a local subtype that the instance must satisfy, rather than defining instance-based constraints. As such, the semantics of instance-scoped constraints are specified as part of the typed-language semantic specification.

**Extended with clauses**

Syntactically, it is possible to assign a value to an element by "unifying" multiple element value specifications to form a single value for that element. This is not a particularly useful construct when used with the instance language alone, but it is quite useful for extending a minimal type instance with greater detail or composing multiple fragments to form an instance. The abstract syntax for such an operation has the form:

*Element* ::= *Category Name = EltContents*

*Category* ::= ***System*** | ***Component*** | ***Connector*** | ***Port*** | ***Role***

*EltContents* ::= (*Element* | *Property* | *Representation* | *Attachment*)*
| (*Element* | *Property* | *Representation* | *Attachment*)+
**(extended with** *EltContents*)*

A straightforward example of using this notation is:

***Component*** *c = { **Port** p; }* ***extended with*** *{ **Property** rate : int = 100; };*

Which is semantically the same as specifying the following component:

***Component*** *c = { **Port** p; **Property** rate : int = 100; };*

**Unifying properties.** The following, slightly more problematic, example shows the unification of a property type and value:

***Component*** *d = { **Property** rate : int; }* ***extended with*** *{ **Property** rate : int = 100; };*

Which is semantically the same as specifying the following component:

***Component*** *c = { **Property** rate : int = 100; };*

The algorithm for unifying properties p = (n, v, t) and p' = (n', v', t') within the scope of an element e is fairly straightforward. The following algorithm unifies a property *p* with a set of properties $p_{set}$ that may, or may not, already contain a property with the unique name *p.n*:

**UnifyProperties algorithm**

**Function signatures:**
LookupPropByName : (name, Set{Property}) → Property
UnifyProperties : (Property, Set{Property}) → Set{Property}

**Algorithm:**
```
UnifyProperties(p : Property, pset : Set{Property}) returns Set{Property} = {
    If p.n ∉ Names(pset)
        return pset ∪ {p}
    else {
        Property targetProp = LookupPropByName(p.n, pset)

        // unify the type specifications
        if (targetProp.t is undefined) then targetProp.t = p.t
```

```
            else if (p.t is defined) and (targetProp.t != p.t) then
                throw Error        // the properties' types can't be unified

            // unify the value specifications
            if (targetProp.v is undefined) then targetProp.v = p.v
            else if (p.v is defined) and (targetProp.v != p.v) then
                throw Error        // the properties' values can't be unified
        }

        // replace the previous property tuple for p.n with the property unified as
        // targetProp and return the revised set of properties
        return { p_set - GetPropByName(p.n, p_set) } ∪ { targetProp }
    }
```

We can use the UnifyProperties function to define the meaning of the extending an existing element *e* with a property *p*.

$$\mathcal{M}[\text{ extended with Property } n : t = v ]\ e = [\ e \mid e.p \leftarrow \text{UnifyProperties}((n, t, v), e.p)\ ]$$

**Unifying attachments**. Unifying an attachment *a* with an existing element structure *e* is trivial. The unified set of attachments is simply the union of the set of attachments in e and the singleton set containing the attachment a. That is:

$$\mathcal{M}[\text{extended with Attachment}(n_1, n_2)]e = [\ e \mid e.a \leftarrow e.a \cup \{\ (\text{Attachment}, n_1, n_2)\ \}\ ]$$

**Unifying substructure**. The meaning of extended with clauses becomes significantly more complex when we need to unify elements and representations. The following example illustrates the unification of substructure elements.

```
System s = {
    Component c = {
            Port p1 = {  Property x;
                         Property y : float = 7.1;
            };
            Port p2 = { Property s : string = "foo"; };
    } extended with {
            Port p1 = {  Property x : int = 100;
                         Property y : float = 7.1;
                         Property z : int=100;
            };
    };
}
```

The system description above is semantically equivalent to the following system description done in the canonical form.

```
System s = {
    Component c = {
        Port p1 = { Property x : int = 7;
                    Property y : float = 7.1;
                    Property z: int = 100 };
        Port p2 = { Property s : string = "foo"; };
    };
}
```

The following system description, on the other hand, has an *extended with* clause that can not be unified because it redefines the type of property s in port p2. This declaration can not be represented in a valid canonical form.

```
System s = {
    Component c = {
        Port p1 = { Property x; Property y : float = 7.1; };
        Port p2 = { Property s : string = "foo"; };
    } extended with {
        Port p2 = { Property s : int = 100; };    // ERROR -- string != int
    }
}
```

The algorithm for unifying element substructure e' = (n', c', s', p', r', a') with an existing element e = (n, c, s, p, r, a) is presented next. This algorithm returns the new set of substructure elements for element *e* (*e.s*). It is important to note that the element *e'* being unified with element *e* has the relationship that *e'* represents substructure of *e*. That is, if the unification is successful then a postcondition of this algorithm is that $e' \in e.s$

**UnifyElements algorithm**

**Function signatures:**

LookupEltByName : (name, Set{Element}) → Element
UnifyElements : (Element, Element) → set{Element}

**Algorithm:**

```
UnifyElements(e_parent, e_child : Element) returns Set{Element} = {
    If (e_child.n ∉ Names(e_parent.s) )
        return e_parent.s ∪ { e_child }

    Element targetElt = LookupEltByName(e_child.n, e_parent.s)
    If (targetElt.c != e_child.c)
        throw Error
    else {
        // else we have a matching name and category, try to unify properties
        // if an error is thrown in unifyProperties() it propogates up and returns an
        // error from this function
        forall properties p' in e_child.p {
            targetElt.p ← unifyProperties(p', targetElt.p)
        }

        // repeat for attachments (if applicable)
        forall attachments a' in e_child.a {
```

```
            targetElt.a ← { a' } ∪ targetElt.a)
        }

        // repeat for representations (if applicable)
        forall representations r' in e_child.r {
            targetElt.r ← unifyRepresentations(r', targetElt.r)
        }

        // repeat recursively for substructure of e_child
        forall elements e_grandchild in e_child.s {
            targetElt.s ← unifyElements(e_grandchild, e_child)
        }

    } { pass any errors that were thrown by unify* functions on to caller }

    // replace the previous element tuple named by e_child.n in e_parent.s
    // with targetElt, which has now been unified with the extension in e_child.
    // and return the revised set of substructure
    return { e_parent.s - LookupEltByName(e_child.n, e_parent.s) } ∪ { targetElt }
}
```

We can use the UnifyElements function to define the meaning of the extending an existing element $e_{parent}$ with a substructure element $e_{child}$.

$$\mathcal{M}[\text{ extended with } e_{child}\ ]\ e_{parent} = [\ e_{parent} \mid e_{parent}.s \leftarrow \text{UnifyElements}(e_{parent}.s,\ e_{child})\ ]$$

**Unifying representations.** Like properties, attachments, and element substructure, unification of representations must be supported in *extended with* clauses. The semantics for representation unification are expressed with an algorithmic function and a denotational equation, as the previous types of extensions have been. Consider the following examples of extending existing structure with an additional representation.

```
Component c = {
    Port outer;
    Representation r1 = {
        System subSys = {
            Component subComp = { Port inner; Property y : float = 7.1; };
        }
        Bindings { c.outer to c.r.subSys.subComp.inner; };
    };
} extended with { Representation r2 = {...}; Representation r3 = {...}; };
```

This component description is semantically equivalent to the following component description specified with the canonical form. Unification in this case is simple because the representations all have different names.

```
Component c = {
    Port outer;
    Representation r1 = {
        System subSys = {
            Component subComp = { Port inner; Property y : float = 7.1; };
        }
        Bindings { c.outer to c.r.subSys.subComp.inner; };
    };
    Representation r2 = {...};
    Representation r3 = {...};
};
```

The following description, on the other hand, has a more complex extended with clause where the name of a representation is repeated in the *extended with* clause. This specification can be unified because the substructure of the representations can be unified.

```
Component c = {
    Port outer;
    Representation r1 = {
        System subSys = {
            Component subComp = { Port inner; Property y : float = 7.1; };
        }
        Bindings { c.outer to c.r.subSys.subComp.inner; };
    }
} extended with {
    Representation r1 = {
        System subSys = {
            Component subComp = {
                Port inner; Port end;
                Property y : float = 7.1; Property x :int = 1;
            };
            Component subComp2 = { Port innerEnd; }
        };
        Bindings { c.outer to c.r.subSys.subComp.end;};
    }
};
```

This component description is semantically equivalent to the following component description done in the canonical form. The system *subSys* of the representation $r_1$ is unified with the system of the same name in the extended with clause, as are the bindings.

```
Component c = {
    Port outer;
    Representation r1 = {
        System subSys = {
            Component subComp = {
                Port inner; Port outer;
                Property y : float = 7.1; Property x :int = 1;
            };
            Component subComp2 = { Port innerEnd; }
        }
        Bindings { c.outer to c.r.subSys.subComp.inner;
                   c.outer to c.r.subSys.subComp.end;
        };
    }
}
```

Two representations can not be unified if their systems can not be unified. The following example illustrates an illegal *extended with* statement because the type of a property of representation $r_1$'s system (subSys) is redefined in the extended with clause.

```
Component c = {
    Port outer;
    Representation r1 = {
        System subSys = { Property x : int = 7; }
        Bindings { };
    }
} extended with {
    Representation r1 = {
        System subSys = {
            Property x : string = "illegal";     -- ERROR! Type of x redefined
        }
        Bindings { c.outer to c.r.subSys.subComp.end;};
    }
};
```

The algorithm for unifying an "extension representation" $r_{extension}$ = (n', e', b') with a set of existing representations $r_{set}$ is presented next. If it is able to perform the unification, this algorithm returns a new set of representations containing the unification of $r_{extension}$ with $r_{set}$. If it is unable to unify the representation $r_{extension}$ with $r_{set}$ it throws an error and the meaning of the encapsulating *extended with* statement is undefined.

**UnifyRepresentations algorithm**

**Function signatures:**

LookupRepByName : (name, Set{Representations}) → Representation

UnifyRepresentations :

(Representation, Set{Representation}) → set{Representation}

**Algorithm:**

UnifyRepresentations ($r_{extension}$ : Representation, $r_{set}$ : Set {Representation} )

returns Set{Representations}

{

```
    If (r_extension.n ∉ Names(r_set) )
        return r_set ∪ { r_extension }

    // else we have a matching name for the rep, attempt to unify the system
    // if an error is thrown in unifyElements() it propogates up and returns an
    // error from this function
    Representation targetRep = LookupRepByName(r_extension.n, r_set)

    If (r_extension.e.n != targetRep.s.n)  throw error;
                                        // Error: different system names

    // try to recursively unify all of the substructure of r_extension's system with
    // targetRep

    forall properties p' in r_extension.e.p {
        targetRep.e.p ← unifyProperties(p', targetRep.e.p)
    }

    // repeat for attachments (if applicable)
    forall attachments a' in r_extension.e.a {
        targetRep.a ← { a' } ∪ targetRep.a)
    }

    // repeat for representations (if applicable)
    forall representations r' in r_extension.e.r {
        targetRep.r ← unifyRepresentations(r', targetRep.r)
    }

    // repeat recursively for substructure of r_extension.e
    forall elements e_grandchild in r_extension.e.s {
        targetRep.e ← unifyElements(e_grandchild, r_extension.e)
    }

    // unify the bindings sets with a simple unioning
    targetRep.b ← r_extension.b ∪ targetRep.b

    // pass any errors that were thrown by unify* functions on to caller

    // replace the previous representation tuple named by r_extension.n in the
    // passed-in set  of representations with targetRep, which has now been
    // unified with the extension in r_extension and return the revised set of Reps
    return { r_set - LookupRepByName(r_extension.n, r_set) } ∪ { targetRep }
}
```

**Semantics of multiple *Extended with* statements**. The description of the semantics of the *extended with* construct to this point has specified the meaning of an element description extended with a single-statement declaration. The language also supports extending an element description with another element description. That is, specifying a list of declaration statements as an extension to an existing element description. Semantically, this is the same as repeatedly extending an element description with each statement in the list. The semantic equation for using multiple statements in an *extended with* clause follows:

Let $s_1, ..., s_n$ be declaration statements

$\mathcal{M}$[ extended with { $s_1$; ... ; $s_n$ } ] e =
$\mathcal{M}$[ extended with $s_n$] $\mathcal{M}$[ extended with $s_{n-1}$ ] ... $\mathcal{M}$[ extended with $s_1$ ] e

Semantic equations are given earlier in this section for each kind of valid declaration statement that can follow *extended with*.

Multiple *extended with* clauses can be strung together. The meaning of the composition of multiple extended with clauses is expressed in the following semantic equations.

Let $s_1, ..., s_n, s_{n+1}, ..., s_m$ be declaration statements

$\mathcal{M}$[ extended with { $s_1$; ... ; $s_n$ } extended with { $s_{n+1}$; ... ; $s_m$ }] e =
$\mathcal{M}$[ extended with { $s_{n+1}$; ... ; $s_m$ } ] $\mathcal{M}$[ extended with { $s_1$; ... ; $s_n$ } ] e

### Meta-properties

Meta-properties are ignored for purposes of semantic evaluation. At an informal level, a meta-property is a property tuple that applies to a property, rather than an element. Meta-properties are allowed as annotations on properties for the purposes of tooling, but they have no effect on the underlying semantic meaning of an Armani expression. Specifically, meta-properties are not considered for type-checking or consistency checking. It is possible that a future extension of the semantics will support meta-properties by allowing property tuples to recursively include other property tuples (which represent that property's meta-properties) but this step is not taken in this draft of the Armani semantics specification.

### Case sensitivity

Throughout the full Armani language, identifiers for all entities are case sensitive. This includes user-defined types, instances, design rules, and design analyses. Keywords, on the other hand, are not case sensitive.

Chapter

3

# Types, Design Rules, and Architectural Styles

## Capturing Design Expertise with Armani

The Armani design language provides constructs for capturing three fundamental classes of architectural design expertise – *design vocabulary*, *design rules*, and *architecural styles*. A brief overview of each of these follows.

- **Design vocabulary** is the most basic form of design expertise that can be captured with Armani, and possibly the most valuable. The design vocabulary available to a software architect specifies the basic building blocks for system design. Design vocabulary describes the selection of components, connectors, and interfaces (ports and roles) that can be used in system design. As an example, the design vocabulary available for a naïve client-server style of design might include client and server components and an HTTP connector. Armani provides a rich predicate-based type system that environment designers can use to specify the design vocabulary, the properties of vocabulary elements, and the design invariants and heuristics that describe how the vocabulary elements can be used.

- **Design rules** specify heuristics, invariants, composition constraints, and contextual cues to assist architects with the design and analysis of software architectures. Armani makes the following aspects of a design rule independently modifiable: the specification of the rule itself, the policy for dealing with violations of the rule, and the scope over which the rule is enforced. Armani allows the association of design rules with a complete style, a collection of related design elements (such as all of the components in a system), a type of design element, or an individual instance of a component or connector. By making the scoping of design rules highly flexible and specifying their policy independent of the rule itself, Armani allows an architect to add, remove, modify, or temporarily ignore design rules as appropriate for various stages and types of design.

- **Architectural styles** provide a mechanism for packaging and aggregating related design vocabulary, rules, and analyses. An Armani style specification consists of the declaration of a set of design vocabulary that can be used for designing in the style, and a set of design rules that guide and constrain the composition and instantiation of the design vocabulary.

# Types

The Armani design language's type system provides a mechanism that designers can use to capture abstract design vocabulary specifications. These specifications can be used both to create instances of design elements and to verify that an instance of a design element satisfies the design constraints specified by the type.

The Armani type system serves a significantly different purpose than the type systems typically provided by programming languages. Programming language type systems are generally designed to provide statically-checkable guarantees of run-time program behavior (e.g. to insure that a function will not accidentally attempt to add a floating point value to an array of strings). Armani's type system, on the other hand, provides a form of checkable redundancy that assures the design constraints for a given type of design vocabulary are satisfied where that vocabulary is used. The type system provides a mechanism for ensuring that the system's fundamental design constraints are not violated as a design evolves over time (e.g. through system maintenance, upgrades, etc.).

To achieve these goals, Armani uses a predicate-based type system that supports the expression of complex type constraints and invariants. Type expressions are predicates that elements can satisfy. A type definition determines a set of design elements—those that satisfy the type's predicate. An element that satisfies type *T*'s predicate is said to satisfy type *T*. A computationally-decidable predicate language (described in detail in chapter 4) is used to ensure that complex type constraints can be mechanically checked.

The Armani type system supports two broad categories of type expressions—design element types (component, connector, port and role types) and property types (primitive, compound, and aliased property types). The type system used for design elements supports both a subtyping facility and the specification of rich constraints on the structure and properties of design elements. The property type system, on the other hand, is significantly simpler than the design element type system. The property type system allows only simple predicates whose primary purpose is to specify the structure used for storing property values. It does not support subtyping or rich constraints on property values.

This section provides an overview of the syntax, common usage, and semantics for the design element type system. The following section provides a similar overview of the property type system.

## Design Element Types:

### Declaring a design element type

A design element type can specify two broad kinds of constraints on design elements. First, it can specify required structure and properties, possibly with default values. Second, it can specify explicit invariants and heuristics (predicates) that describe legal property and structure values of an element of that type. This section describes the syntax and semantics of *component, connector, port* and *role* types (referred to as design element types or just element types). Armani *system* types are referred to as *architectural styles* (or simply *styles*), and are discussed in a later section. Styles extend the capabilities of the design element types described in this section.

A detailed and rigorous specification of the syntax and semantics of element type declarations is given later in this chapter. For purposes of immediate discussion, the following informal description of the syntax and semantics of the Armani element type system is provided.

The informal syntax for declaring a design element type is:

*<Category>* **Type** *<TypeName> = {*
    *<Sequence of:* *required structure and values*
    *| properties*
    *| explicit invariants*
    *| explicit heuristics >*
*}*

In the informal syntax given above, *<Category>* can be any of the literals *Component, Connector, Port,* or *Role*, and *<TypeName>* specifies a valid identifier. The body of the type declaration consists of a sequence of constraints by which instances of this type must abide. Informally, the meaning of the four kinds of constraint declarations that can be made within a type declaration are described below:

- **Required Structure.** The structural declarations in a type description *T* define the substructure that an element *e* of type *T* (written *e : T*) must have. Informally, for every port, role, or representation defined in *T*, an instance *e : T* must have a corresponding port, role, or representation. The port, role, or representation defined in the instance must be defined with at least as much detail as its corresponding port, role, or representation in the type declaration. A more detailed specification of the semantics of required structure statements is given in table 3.1.

- **Required Properties.** A property $p_i$ declared in a type declaration T specifies that an element *e : T* must define the property $p_i$. Further, if $p_i$ is declared to have a type and/or a value in *T*, $p_i$ declared in *e : T* must also have the same type and/or value. As with required structure, a more detailed specification of the semantics of property declarations is given in table 3.1.

- **Explicit Invariants.** In addition to the required structure and properties of a type, additional invariant constraints can be specified using Armani's Predicate Language (described in chapter 4). These invariants can specify ranges of valid values for properties, constraints on the types and number of substructure elements that an element of type *T* can have, and any other constraint that can be specified with the Armani Predicate Language. An element *e : T* must satisfy all of the invariant constraints defined in *T* in order to satisfy *T*'s predicate (and thus satisfy type *T*).

- **Explicit Heuristics** use the same predicate specification language as explicit invariants. Unlike invariants, though, heuristics are not considered in determining whether an element *e* satisfies a type *T*. Violations of type heuristics can be flagged during constraint analysis or analyzed by external tools, if desired, but the heuristics themselves are not part of a type's predicate. The heuristics construct provides architects and designers with a way to capture design "rules of thumb" that are less strict than invariants.

An element *e : T* satisfies the type *T*'s predicate if *e* contains all of the required structure and properties specified in *T*, and *e* satisfies all of the invariant predicates defined in *T*.

The following example shows a type specification that declares constraints that must be satisfied by all instances of the type in the form of required minimal structure and predicates that must be maintained. Keywords are indicated in boldface type, comments in greyed-text.

```
Component Type Client = {

 // Declare the minimal structure that must exist. In this case, it says that an instance
 // of this type must have a port called request, and that port must have the protocol
 // rpc-client.
 Port Request = { Property protocol : CSProtocolT = rpc-client };

 // The next declaration says that a client must have a property of type "float" called
 // "request-rate." It also provides a default value for that property, which can be
 // changed when an instance of this type is created.
 Property request-rate : float << default = 0.0 >>;

 // Now specify the invariants that all elements that claim to satisfy this type must possess.

 // all ports must support the rpc-client protocol
 Invariant forall p in self.Ports • p.protocol = rpc-client;

 // there may be no more than 5 ports on a client
 Invariant  size(self.Ports) <= 5;

 // The request rate must be a non-negative value less than 100
 Invariant request-rate >= 0;

 // Specify a heuristic indicating the request rate should not exceed 100
 Heuristic request-rate < 100;
}
```

***Example 3.1: Declaring component type "Client"***

The *Client* type specification imposes the following structural and invariant constraints on component instance *C : Client*:

Structural constraints:

- A *Client* instance must have a port called *request*, with a property called *protocol*. The protocol property must be of type CSProtocolT and have a value of *rpc-client*.

- A *Client* instance must have a property called *request-rate* of type *float*. The default value of 0.0 can be overriden with an *extended with {...}* clause, but the initial value for this property on all Client instances created with the *new* operator will be 0.0.

Invariant constraints:

- All ports of a client must have a property named *protocol*, which has a value of *rpc-client*.
- There may be no more than 5 ports on a *Client* instance.
- The request-rate property of a *Client* component must have a value greater than 0.

The heuristic constraint that the request-rate property of an instance of a *Client* component have a value less than 100 is not considered in determining whether that instance satisfies the *Client* type.

## Creating a simple instance of a typed architectural element

Instances of the four basic architectural elements – components, connectors, ports, and roles, can be created with the following (informal) syntax:

*<Category> <InstanceName> [ : <TypeName> ] = <Value> ;*

*where*
*<value> ::= ( { <sequence of property and structure specs.> } | **new** <TypeName> )*
*( **extended with** <value> )**

Specifying an explicit type for an instance is optional. If no type is explicitly declared for an individual instance, then the type of that instance defaults to <Category>. Consider the following example of a component declared without an explicit type declaration:

***Component** C = { **Port** input; } ;*

In this instance, the value of component *C* is *{ **Port** input }* which satisfies the constraints of the *Component* type, so this instance declaration is valid.

When an instance is explicitly typed, as in the following example, the value on the right hand side of the "=" token must satisfy the predicate defined by the declared type. Consider the following example:

***Component** C : Client = **new** Client;*

In this example, a component *C* is declared to satisfy type *Client*. The value of *C* is defined using the Armani ***new*** operator. The expression ***new** <TypeName>* creates a value expression consisting of the minimal structure declared in the declaration of *<TypeName>* with default values applied to properties as specified in the type specification. Properties with no default value provided in the type declaration have undefined values in the instance generated.

Using the Client type defined in example 3.1, the previous example creates a component with the following canonical structure:

```
Component C : Client = {
        Port Request = { Property protocol : CSProtocolT = rpc-client }
        Property request-rate : float = 0.0;
}
```

This default Client component satisfies the invariants and heuristics declared in the Client type definition.

It is possible to associate non-default values with an element created from a given type using the ***extended with*** *<value>* construct. The following example illustrates a client with an additional port and an additional property.

```
Component C' : Client = new Client extended with {
        Port ExtraPort = {  Property protocol : CSProtocolT = rpc-client;
                            Property primary-port = true };
        Property request-rate : float = 5.0;
}
```

This declaration would result in the creation of a new component *C'* with the following structure:

```
Component C' : Client = {
        Port Request = { Property protocol : CSProtocolT = rpc-client } ;
        Port ExtraPort = {  Property protocol : CSProtocolT = rpc-client};
                            Property primary-port = true;};
        Property request-rate : float = 5.0;
}
```

In this example, the default constructor is extended with new property values that either add new structure and values or override the default structure and value of the type. The value that is assigned to *C'* in this case is the unification of the structure declared with the *extended with {... }* clause and the structure that is created with the *new <TypeName>* constructor. The detailed algorithm for unifying substructure of an element using the *extended with {...}* construct is given in Chapter 2 where the semantics of *extended with* are specified.

### Informal Element type Semantics

A type specification defines the minimal set of structure and property fields that elements of a given type have, along with a set of invariants that must hold for all instances that satisfy the type. Every type T can be converted to a predicate function $F_t$ that takes a single element E as an argument. If the function $F_t(E)$ evaluates to true for element E, then element E satisfies type T (written T(E)).

Detailed descriptions of the semantics of required structure and invariant specifications of a type declaration follow:

### Required Structure:

The semantics of structural declarations in an element type specification are described in table 3.1.

| Declaration Type | Example | Meaning |
|---|---|---|
| Structural element C with no type or value declaration | `Port C;` | Forall elements E s.t. E declares type T (written E:T), T(E) implies E has the element named C as a child. |
| Structural element C with a type but no value declaration | `Port C : t';` | Forall elements E s.t. E:T, T(E) implies E has the element named C as a child, and that C satisfies t' (t'(C)) |
| Structural element C with a type and a value declaration | `Port C : t' = { Property j:t'' = bar};` | Forall elements E s.t. E:T, T(E) implies E has the element named C as a child, and t'(C) and C has the property j:t'' with a value of bar. |
| Property named P with no type or value given | `Property P;` | Forall elements E st E:T, T(E) implies E has the property P of type "Property." |
| Property named P with a type t' specified, but no value given | `Property P : t';` | Forall elements E st E:T, T(E) implies E has the property P of type t'. P's value is unconstrained beyond the requirement that the value of P satisfy type t'. |
| Property named P with a type t' specified and a default value v given. | `Property P : t' <<default=v>>;` | Forall elements E st E:T, T(E) implies E has the property P of type t'. P's value defaults to v when a new instance of type T is created but the <<default = v>> clause is simply a convenience that the type has no obligation to maintain. The << ... >> notation specifies that "default = v" is a meta-property. |
| Property named P with a type t' specified and a value v assigned directly to the property | `Property P:t' = v;` | Forall elements E st E:T, T(E) implies E has the property P of type t' and P's value must be v. This statement declares a constant valued property for the type. |

**Table 3.1 Structural Specification Semantics**

### Invariant Predicates:

The invariant declarations of a type T define a set of predicates that must hold for all instances of type T. These invariants are specified using a subset of the predicate language described in chapter 4. The primary restriction that this subset of the language imposes is that the scope of names (and entities) visible from within a predicate in the type declaration is limited to those entities (properties, ports, roles, etc.) defined in the type definition or the definition of any of its supertypes. In order to improve modularity, the type

predicates are limited to operating over values of an instance of the type. The design rule mechanism used with systems (described later in this chapter) supports constraints spanning multiple types and instances.

One implication of this design decision is that invariants placed in element type declarations are most appropriate for local constraints on all elements of the type, such as valid ranges of property values. Constraints involving the relationship between elements, such as valid system topologies or valid port/role pairs, are best put in a system or style specification.

Details of the scoping constraints for type specifications follow:

- **Scope of names within a type declaration:** Names are lexically (statically) scoped and the namespace for a type spans both the structural constraints and invariants of the type specification. No two property and/or structural elements of a type declaration may share names. Structural elements (e.g. ports, roles, and representations) share the name space with properties. As a result, names used in the structure section may be unambiguously dereferenced by invariant predicates, allowing the unambiguous use of dot notation to refer to substructure and properties, while reducing the complexity of dereferencing names. The root of the type namespace that is visible to invariant predicates is the identifier *self* which is a reference to the instance of this type that is being checked for type-compliance. Invariant predicates can reference only *self* and entities that are descendents of *self* in the AST.

- **Scope of predicates:** Predicates declared within a type specification may only reference properties or structural elements named within the structural specification of the type, including substructure affiliated with those elements and properties. They may not refer to anything outside of the scope of the type declaration. This strict limitation on referencing entities outside of the element improves modularity and reusability of type specifications. Rules limiting the interactions of and relationships between design elements can be expressed with style-wide or system-wide design rules (discussed later in this chapter) instead of type specifications.

The keyword *self* is used to refer to the instance of an element that is being type-checked. Unless otherwise explicitly fully qualified, property names in an invariant predicate have an implicit *self.* preceding them and refer to the instance being checked.

Heuristic predicates are specified with exactly the same language as invariants and are governed by the same scoping rules when declared within a type definition. Heuristics differ from invariants only in that they are not considered for type-checking purposes.

### Scope of type declarations

A type may be declared within the global design space or within a style specification. Type names are lexically scoped. Types declared outside of all style declarations have global scope. Types with global scope are visible within all systems or styles declared in that global scope. A type defined within a style specification is visible to all other declarations in the style, all of that style's substyles, and all system that claim to be built in that style.

## Subtypes

Armani supports a strict form of subtyping that ensures substitutability between subtypes and supertypes. That is, if type $T'$ is a subtype of type $T$ (written $T' \leq T$), then an element that satisfies $T'$ may be used wherever an element of type $T$ is required. The following informal syntax describes Armani's subtyping construct.

*<Category>* ***Type*** *<SubTypeName>* ***extends*** *<SuperTypeName>*$^{+}$ ***with*** *{*
    *<Sequence of: required structure and values*
    *| properties*
    *| explicit invariants*
    *| explicit heuristics >*
*}*

The semantics of this construct are straightforward. The new (sub)type *<SubTypeName>* consists of the unification of the structural requirements of all supertypes with the new structural declarations, and the union of the invariant and heuristic predicates of all supertypes with the new invariant and heuristic declarations. The unification operation for type structure is the same as the unification operation on instances is described in chapter 2. Because types are reduced to prototypical elements for semantic evaluation, the same unification operation is applicable to both element types and instances. All instances of the subtype are also instances of the supertype, and satisfy the constraints of both the supertype and the constraints listed in the *extends ... with {...}* clause.

Consider the following example:

***Component Type*** *BlockingClient* ***extends*** *Client* ***with*** *{*
    ***Port*** *BlockingRequest = {****Property*** *protocol = rpc-client};*
    ***Property*** *blocking : boolean = true;*
    ***Property*** *timeout-sec : float << default = 30.0 >>;*

    ***Invariant*** *timeout-sec < 60.0;*
*}*

An instance of a BlockingClient type component would then have all of the structure and rules to maintain that a Client type component would have, plus the additional properties and rules given in this specification. The previous type declaration is equivalent to declaring the BlockingClient type without subclassing as done below (using the Client type definition from the previous section):

```
Component Type BlockingClient = {
    Port Request = {Property protocol = rpc-client};
    Port BlockingRequest = {Property protocol = rpc-client};
    Property request-rate : float << default = 0 >>;
    Property blocking : boolean = true;

    Invariants {
        Forall p in self.Ports | p.protocol = rpc-client;
        Size(Ports) <= 5;
        request-rate >= 0;
        timeout-sec < 60.0;
    };

    Heuristic request-rate < 100;
}
```

## Detailed Element Type Semantics

The mechanism used for specifying the semantics of element type declarations is an extension to the mechanism used for specifying element instances in chapter 2. A slightly modified semantic interpretation of an Armani element instance is adopted for specifying the semantics of an element type. Using this approach, an element type declaration is converted into a tuple representing the structure and constraints of that type. An algorithm is provided that determines whether a tuple representing an element instance satisfies the predicates of the type(s) it claims to satisfy.

The semantic equations specified for instances in chapter 2 regarding properties, attachments, representations, bindings, extended with clauses, and declaration statement sequencing remain unchanged in this revised semantic specification. In order to extend the tuple representation to support element types, though, the following changes must be made to the element semantic tuple and equations:

- Two additional tuple fields must be introduced to store predicates. *i* stores the invariant predicates specified by the element type and *h* stores the heuristic predicates specified by the element type.
- The new tuple field $t_{asserted}$ stores the names of all types that this element claims to satisfy.
- The new tuple field $t_{super}$ stores the names of an element type's supertypes.

The primary distinction between $t_{asserted}$ and $t_{super}$ is that the typechecking algorithm tests whether the types stored in $t_{asserted}$ (and, by transitivity the supertypes of those types) are satisfied by the element it is typechecking. The $t_{super}$ field, on the other hand, only stores the supertypes of a declared type. A subtype satisfies its supertype by definition, so there is no need to verify type-compliance with the typechecker. Exactly one of the two sets $t_{super}$ and $t_{asserted}$ should be empty. The $t_{super}$ field should be empty if the declaration is an instance. Likewise, the $t_{asserted}$ field should be empty if the declaration is a type.

In addition to the extended element tuple, the concept of a *design space* is introduced to support the context used for global type declarations and scoping. A design space is a tuple $d = (t_{elt}, t_{prop}, d_a, s)$ where $t_{elt}$ is the set containing the element types defined in the

context of *d*, $t_{prop}$ is the set of property types defined in the context of *d*, *s* is the set of systems defined in the context of *d*, and $d_a$ is the set of design analyses defined in *d*. All toplevel systems in an Armani description must now be defined in a design space, rather than the empty element specified in chapter 2.

A design space can be used as the context in which a type or system is evaluated. When the context of evaluation is a design space rather than an element, the context will be annotated with a variable named *d*. When the context of evaluation is an element, the context will be annotated with a variable named *e*. When the context of evaluation is unknown, but may be any kind of context (e.g. design space, element, or something else), the context is indicated with a *c*.

**Revised element specification.**

An Armani element (or element type) *e* is a tuple of the form

$$e = (n, c, s, p, r, a, i, h, t_{super}, t_{asserted}).$$

Informally:

*n* = name of the element.
*c* = category of the element.
*s* = set of elements that define *e's* substructure (e.g. ports, roles, etc.).
*p* = set of properties that define the properties of *e*.
*r* = set of representations that define *e's* subarchitectures.
*a* = set of attachments that define *e's* topology, empty unless e is a system.
*i* = set of invariant predicates defined for *e*.
*h* = set of heuristics defined for *e*.
$t_{super}$ = set of names of supertypes of this element type.
$t_{asserted}$ = set of names of types this element claims to satisfy.

The well-formedness rule for elements uses the function *Names : Set → Set* that creates a new set of name identifiers that is a projection of the "n" field (the name field) of all tuples in the original set of tuples (e.g. Names({(n=foo, ...), (n=bar,...)}) returns the set {foo,bar}).

This well-formedness rule declares the legal substructure for various categories of elements, and that no two distinct entities declared as part of this element can share the same identifier.

Well-formedness rules:

n : Identifier
c : Category
s : Set{Element}
p : Set{Property}
r : Set{Representation}
a : Set{Attachment}
i : Set{InvariantPredicate}
h : Set{HeuristicPredicate}
$t_{super}$: Set{TypeName}
$t_{asserted}$ : Set{TypeName}

Port(e) → (s = { })
**and** Role(e) → (s = { })
**and** Component(e) → (forall e' in s | Port(e'))
**and** Connector(e) → (forall e' in s | Role(e'))
**and** System(e) → (forall e' in s | Component(e') **or** Connector(e'))
**and** (a != {}) → System(e)

**and** forall x, y in p | (x.n = y.n) → (x = y)
**and** forall x, y in r | (x.n = y.n) → (x = y)

**and** (Names(s) ∩ Names(p) = { })
**and** (Names(r) ∩ Names(p) = { })
**and** (Names(s) ∩ Names(r) = { })

**and** (($t_{super}$ = { }) **xor** ($t_{asserted}$ = { } ))

Meaning (element instance declarations):

An element instance declaration that declares no type and occurs in the context of an existing element has the following meaning:

$\mathcal{M}$[ *Category* n = { $s_1$ ;...; $s_n$ } ] e =
**if** (n ∉ Names(e.r)) **and** (n ∉ Names(e.s)) **and** (n ∉ Names(e.p)) **then**
$\mathcal{M}$[ { $s_1$ ;...; $s_n$ } ] [ e' |e.s ← e.s ∪ { e' }, e'.n ← n, e'.c ← *Category*,
e'.s ← {}, e'.p ← {}, e'.r ← {}, e'.a ← {},
e'.$t_{asserted}$ ← { *Category* }, e'.$t_{super}$ ← {},
e'.i ← {}, e'.h ← {} ]

A system instance declaration that claims to satisfy no type and occurs in the context of a design space has the following meaning:

$\mathcal{M}$[ System n = { $s_1$ ;...; $s_n$ } ] d =
**if** (n ∉ Names(d.f)) **and** (n ∉ Names(d.s)) **and** (n ∉ Names(d.$t_{prop}$)
**and** (n ∉ Names(d.$t_{elt}$)) **then**
$\mathcal{M}$[ { $s_1$ ;...; $s_n$ } ] [ e | d.s ← d.s ∪ { e }, e.n ← n, e.c ← System,
e.s ← {}, e.p ← {}, e.r ← {}, e.a ← {},
e.$t_{asserted}$ ← { System }, e.$t_{super}$ ← {},
e.i ← {}, e.h ← {} ]

An element instance declaration that claims to satisfy one or more types and occurs in the context of an existing element has the following meaning:

$\mathcal{M}$[ *Category* n : $t_1$, ..., $t_n$ = { $s_1$ ;...; $s_n$ } ] e =
**if** (n ∉ Names(e.r)) **and** (n ∉ Names(e.s)) **and** (n ∉ Names(e.p)) **then**
$\mathcal{M}$[ { $s_1$ ;...; $s_n$ } ] [ e' |e.s ← e.s ∪ { e' }, e'.n ← n, e'.c ← *Category*,
e'.s ← {}, e',p ← {}, e'.r ← {}, e'.a ← {},
e'.$t_{asserted}$ ← { *Category* } ∪ { $t_1$, ..., $t_n$ },
e'.$t_{super}$ ← {}, e'.i ← {}, e'.h ← {} ]

Meaning (element type declarations):

It is not possible to specify a *system* type using the syntax System Type n = {...}. A system type is a family/style, which is described later in this chapter. Therefore, the only valid values for *Category* in the expression *Category* Type n = {...} are *Component, Connector, Port,* or *Role.*

An element type declaration that claims no supertypes and occurs in the context of a design space has the following meaning:

$\mathcal{M}$[ *Category* Type n = { $s_1$ ;...; $s_n$ } ] d =
  **if** (n ∉ Names(d.f)) **and** (n ∉ Names(d.s)) **and** (n ∉ Names(d.$t_{prop}$)
    **and** (n ∉ Names(d.$t_{elt}$)) **then**
    $\mathcal{M}$[ { $s_1$ ;...; $s_n$ } ] [ e | d.s ← d.s ∪ { e }, e.n ← n, e.c ← *Category*,
      e.s ← {}, e.p ← {}, e.r ← {}, e.a ← {},
      e.$t_{asserted}$ ← {}, e.$t_{super}$ ← { *Category* },
      e.i ← {}, e.h ← {} ]

An element type declaration that claims one or more supertypes and occurs in the context of a design space has the following meaning:

$\mathcal{M}$[ *Category* Type n extends $t_1$, ..., $t_n$ with { $s_1$ ;...; $s_n$ } ] d =
  **if** (n ∉ Names(d.f)) **and** (n ∉ Names(d.s)) **and** (n ∉ Names(d.$t_{prop}$)
    **and** (n ∉ Names(d.$t_{elt}$)) **then**
    $\mathcal{M}$[ { $s_1$ ;...; $s_n$ } ] [ e | d.s ← d.s ∪ { e }, e.n ← n, e.c ← *Category*,
      e.s ← {}, e.p ← {}, e.r ← {}, e.a ← {},
      e.$t_{asserted}$ ← {}, e.$t_{super}$ ← {*Category*} ∪ { $t_1$,..., $t_n$},
      e.i ← {}, e.h ← {} ]

### Instantiating a type

Armani supports the instantiation of types by providing a construct that returns an element structure containing the minimal structure specified by the type – properties, representations, attachments, and substructure. The meaning of a *new <TypeName>* statement follows:

$\mathcal{M}$[ new *TypeName* ] *e* =
  [ *e* | *e.p* ← InstantiateProperties(*TypeName*, *e*, *e.p* ),
    *e.r* ← InstantiateReps(*TypeName*, *e*, *e.r*),
    *e.s* ← InstantiateSubstructure(*TypeName*, *e*, *e.s* ),
    *e.a* ← InstantiateAttachments(*TypeName*, *e*, *e.p* ) ]

The Instantiate*(...) functions used in the previous semantic equation all take a typename, a scope in which that typename is visible, and the appropriate set of entities to which the instantiated structure is to be added. The functions use the appropriate extension/unification algorithms to unify the declarations produced by the type instantiation with the set of entities to which the instantiation is being added. The functions then return an appropriately unified set.

```
function InstantiateProperties: (Name, Scope, Set{Property}) → Set{Property}
InstantiateProperties( name_type : Name, e : Scope, p_set : Set{Property})
    returns Set{Property}
{
    ElementType t = lookupTypeByName(name_type, e)
    forall properties p in t.p {
        p_set ← UnifyProperties(p, p_set)
    }
    return p_set
}

function InstantiateReps: (Name, Scope, Set{Representation}) → Set{Representation}
InstantiateProperties( name_type : Name, e : Scope, r_set : Set{Representation})
    returns Set{Representation}
{
    ElementType t = lookupTypeByName(name_type, e)
    forall Representations r in t.r {
        r_set ← UnifyRepresentations(r, r_set)
    }
    return r_set
}

function InstantiateSubstructure: (Name, Scope, Set{Element}) → Set{Element}
InstantiateSubstructure( name_type : Name, e : Scope, e_set : Set{Element})
    returns Set{Element}
{
    ElementType t = lookupTypeByName(name_type, e)
    forall Elements e_child in t.s {
        e_set ← UnifyElements(e_child, Parent(e_set))
    }
    return e_set
}

function InstantiateAttachments: (Name, Scope, Set{Attachment}) →
                                                    Set{Attachment}
InstantiateAttachments(name_type : Name, e : Scope, a_set : Set{Attachment})
    returns Set{Representation}
{
    ElementType t = lookupTypeByName(name_type, e)
    forall Attachments a in t.a {
        a_set ← a_set ∪ { a }
    }
    return a_set
}
```

## Typechecking semantics for design elements

The semantic specifications given to this point have shown how to reduce syntactic Armani instance and type specifications to their canonical forms in the semantic domain—tuples and sets of tuples—for the purposes of semantic evaluation. The semantic specifications

have not, however, explained what it means for an Armani system description to be type-correct. To determine the type correctness (and general consistency) of an Armani specification, the following algorithm, *TypecheckElement(e : Element)*, is applied to the canonical semantic representation of the toplevel Armani system. This algorithm performs a depth-first traversal of the canonical representation's tree structure (the semantic encoding of elements produces a tree structure) rooted at *e* ensuring that all substructure of the system is type-correct. If all of the substructure of the subtree rooted at *e* is type correct the algorithm returns true, indicating that element *e* is type correct. If the algorithm finds any substructure that does not typecheck, it returns false, indicating a type-inconsistency.

The *TypecheckElement(e : Element)* algorithm makes use of the boolean helper functions *CompareToPrototype($e_{prototype}$, $e_{instance}$ : Element), LookupName(n : Identifier, s : Scope)*, and *TypecheckProperty(p : Property)*. *CompareToPrototype* uses the prototype element $e_{prototype}$ as a type specification for $e_{instance}$ and returns true if $e_{instance}$ satisfies the type $e_{prototype}$. The *CompareToPrototype* function is used to recursively typechecks type specifications with multiple layers of substructure. *LookupName* returns the tuple named by *n* in scope *s*, or nil if *n* is undefined in scope *s*. *TypeCheckProperty* verifies that a property *p* is internally type-consistent. That is, it returns true if the value of *p* satisfies the type predicate of *p*.

The signatures of each of the functions used to compute type-correctness are provided below:

**TypecheckElement:** Element → boolean
**LookupName:** (Name, Scope) → Tuple
**CompareToPrototype:** (Element, Element) → boolean
**TypecheckProperty:** Property → boolean

The detailed algorithms that implement these functions follow:

```
TypecheckElement: Element → boolean
TypecheckElement: (ei : Element) returns boolean
{
  // Before checking that the element instance satisfies its declared types, make
  // sure that all of its substructure, properties and representations are internally
  // type-correct.
  // begin by recursing through all substructure to make sure that it typechecks
  foreach element esub in ei.s
    if TypecheckElement(esub) == false then return false

  // then check that all properties in the instance ei are internally consistent
  foreach property p in ei.p
    if TypecheckProperty(p) == false then return false

  // and that all the representations of this element have a system that typechecks
  foreach representation r in ei.r
    if TypecheckElement(r.e) == false then return false

  // and that ei satisfies all of the invariants it declares. At the semantic level, an
  // invariant describes a predicate that is evaluated over an element instance.
  foreach invariant i in ei.i
    if i(ei) == false then return false
```

```
// proceed to make sure that the structure required by e_i's asserted types exists
foreach ElementTypeName n in e_i.t_asserted {
   ElementType e_t = LookupName(n, e_i)
   if (e_t = nil) return false// typename n not visible in this element's scope

   repeat until e_t = nil  {
        // this loop continues until e_i's and all of e_i's supertypes have been tested.

        if (e_t = nil) return false// typename n not visible in this element's scope
        if (e_i.c != e_t.c) return false       // type and element have different categories

        // check that the substructure of e_t exist in e_i
        foreach ElementTuple e_sub in e_t.s {
             Let e_i = LookupName (e_sub.n, e_i.s)
             if (e_i == ({})) return false   // then e_i has no substructure named e_t.n
             else if CompareToPrototype(e_sub, e_i) == false
                  return false               // then substructure doesn't match
        }

        // check that properties of e_t properly exist in e_i
        foreach PropertyTuple p_t in e_t.p {
             Let p_i = LookupName (p_t.n, e_i.p)
             if (p_i == nil) return false    // then e_i does not have a property named p.n
             if (p_t.t is not undefined) then
                  if (p_t.t != p_i.t) return false
             if (p_t.v is not undefined) then
                  if (p_t.v != p_i.v) return false
        }

        // check that the representations of e_i exist in e_i
        foreach RepresentationTuple r in e_t.r {
             Let r_i = LookupName (r.n, e_i.r)
             if (r_i == ({})) return false          // then e_i does not have a rep named r.n
             if (r.s is not undefined) then
                  if (CompareToPrototype(r.s, r_i.s) == false then return false
             if (r.b is not undefined) then
                  if (r.b != r_i.b) return false
        }

        // check that each invariant defined in the type e_t is satisfied in instance e_i.
        foreach invariant i in e_t.i
             if i(e_i) == false then return false

        // repeat for e_t's supertype, if there are no more supertypes then
        // e_t will be set to nil and this loop will be exited
        Let e_t = e_t.t_super

   } // end repeat until supertype is nil

} // end foreach type asserted

// if we get to here then we can't show that the element does not typecheck,
// so return that it does typecheck.
```

```
return true;
}
```

Having defined the primary typechecking algorithm, the typechecking helper functions are defined below:

**TypecheckProperty:** Property → boolean

```
TypecheckProperty(p : property) returns boolean
{
  // if the value of property p (p.v) satisfies the predicate defined by property
  // p's type (expressed here as the function p.t) then return true else return false
  return p.t(p.v);
}
```

**CompareToPrototype:** (Element, Element) → boolean

```
CompareToPrototype(e_prototype, e_instance) returns boolean
{
    // put in algorithm to do prototype-based typechecking. Return false if we
    // can prove that e_instance does not satisfy e_prototype, else return true.
    if (e_instance.c != e_prototype.c)  then return false

    // walk over the structure of the prototype, making sure the prototype's
    // substructure exists in the instance.
    foreach ElementTuple e_sub in e_prototype.s {
        Let e_i = LookupName (e_sub.n, e_instance.s)
        if (e_i == ()) return false      // then e_i has no substructure named e_t.n
            else if CompareToPrototype(e_sub, e_i) == false
                return false             // then substructure doesn't match
    }

    // check that properties of e_prototype properly exist in e_instance
        foreach PropertyTuple p_t in e_prototype.p {
            Let p_i = LookupName (p_t.n, e_instance.p)
            if (p_i == nil) return false     // then e_i does not have a property named p.n
            if (p_t.t is not undefined) then
                if (p_t.t != p_i.t) return false
        if (p_t.v is not undefined) then
                if (p_t.v != p_i.v) return false
    }

    // check that the representations of e_t exist in e_i
    foreach RepresentationTuple r in e_prototype.r {
                Let r_i = LookupName (r.n, e_instance.r)
        if (r_i == ({})) return false          // then e_i does not have a rep named r.n
            if (r.s is not undefined) then
        if (CompareToPrototype(r.s, r_i.s) == false then return false
        if (r.b is not undefined) then
                if (r.b != r_i.b) return false
    }

    // if we get to here then we haven't been able to prove that it does not typecheck
    // so return true.
    return true;
}
```

## Property Types

The discussion of the type system to this point has described the type system used for design vocabulary elements. Properties of these design elements can also be typed. The type system used for element properties uses a syntax and semantics similar to the design element type system's, but the constraints that can be imposed on properties are much simpler than those that can be imposed on design elements.

A property of a design element is simply a scoped name with which a value and a type can be associated. The purpose of a property type is to define the range and structure of values that can be applied to the named property.

A property type can be either an atomic type, an enumerated type, a compound type (set, sequence or record), or a type renaming.

**Atomic property types.** An *atomic* property type is one of Armani's basic built in type primitives – *int*, *float*, *boolean*, or *string*. Atomic types do not need to be defined by the user, as they are built into the Armani language. An example of two properties declared to have atomic types follows:

***Property*** *rate : float = 7.5;*
***Property*** *purpose-description : string = "This component...";*

The declaration of a type can, but does not need to, be separated from the use of that type in specific properties. Explicitly named types are declared with the following (informal) syntax:

***Property Type*** *<TypeName> = <TypeStructure>;*

where <TypeName> is an identifier to be associated with <TypeStructure>. <TypeStructure> can define an enumerated type, a compound type, or rename a previously defined type. Semantically, <TypeStructure> specifies a predicate that defines the set of valid values for the type and in doing so defines the structure that values of the type must posses. A more detailed discussion of the semantics of property types is included at the end of this section.

Instances of properties are declared using the following syntax.

***Property*** *<PropertyName> : <TypeName> = <PropertyValue>;*

The property named <PropertyName> is associated with the element in which it is declared. The type of <PropertyName> is explicitly specified using the ": <Typename>" notation.

**Enumerated property types.** An *enumerated* type defines a set of valid values that a property of that type may hold. The following example defines a type "color" that can have any of the values white, red, blue, green, or black.

***Property Type*** *color =* ***enum*** *{white, red, blue, green, black};*

A *compound* property type is a type that provides a property with structure to store multiple values. Compound types are either sets, sequences, or records of other types. Compound typed properties may either use named compound types or explicitly create a new type in the type signature of the property. Alternatively, if a property does not declare a type but uses the syntax for specifying the value as a record, set, or sequence, then Armani will use simple type inferencing to store the value appropriately.

Examples of compound type declarations and usage follow:

**Records**. A record type contains multiple typed fields that store distinct but related values. Values of the fields of a record property can be referenced by the name of the field.

The syntax for record type declarations is:

***Property Type*** *<TypeName> =* ***Record*** *"[" <FieldDefs>* "]";*

where <FieldDefs> is a sequence of (name, type) pairs. Examples of record type declarations and the use of record types in property declarations follow:

***Property Type*** *visualization =* ***Record*** *[ x,y : int; fill-color : color];*

***Property*** *point :* ***Record*** *[x,y : int] = [x = 4; y = 10];*
***Property*** *vis : visualization = [x = 10; y = 20; fill-color = blue];*

**Sequences**. A sequence type is an ordered list of elements, separated by commas. A sequence instance may have repeated values. The items stored in a sequence must be of a single homogeneous type. The following syntax is used for specifying a type that stores a sequence of <TypeName>'s :

***Property Type*** *<TypeName> =* ***Sequence*** *< <TypeName> > ;*

Examples of sequence property type and sequence property instance declarations follow:

***Property Type*** *string-list =* ***Sequence****<string>;*
***Property Type*** *vis-list =* ***Sequence*** *<visualization>;*

***Property*** *int-seq :* ***Sequence****<int> = <1, 2, 3, 4, 2, 3, 4>;*
***Property*** *string-seq : string-list = <"one", "two", "three", "one">;*
***Property*** *comp-vis : vis-list = < [x = 10; y = 20; fill-color = black ],*
*[x = 50; y = 300; fill-color = white ] >;*

**Sets**. A set type defines an unordered set of elements, separated by commas, with no duplicate values. Like a sequence type, a set type must be homogeneous. That is, all of its elements must be of a specific type. The following syntax is used for specifying a set type:

***Property Type*** *<TypeName> =* ***Set*** *{ <TypeName> } ;*

The following examples illustrate the declaration of set property types and their use:

***Property Type*** *int-set =* ***Set****{int};*
***Property Type*** *color-set =* ***Set*** *{color};*

***Property*** *set-of-floats :* ***Set****{float} = {1.2, 3.4, 5.6, 7.8, 9.0};*
***Property*** *set-of-ints : int-set = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};*
***Property*** *set-of-colors : color-set = {blue, red, green, white};*

**Type renaming**. A *renamed* property type allows users to separate the logical meaning of a type from its underlying storage structure. A renamed type is comparable to a "typedef" in the C language. For example, both a URL and a Java method declaration can be specified as properties of a component using a string. The semantics that tools should use to interpret the content of those two strings, however, are significantly different. The following example shows how these types could be renamed to be more descriptive:

***Property Type*** *java-method = string;*
***Property Type*** *url = string;*

***Property*** *some-java-method : java-method = "foo(x,y:int){...}";*
***Property*** *some-url : url = "http://www.codeland.com";*

## Property Type Semantics

A property type, like a design element type, specifies a predicate that defines a set of valid values for instances of that type. A property is type correct if its value is an element of the set described by its type. The range of type predicates that can be defined for property types is more limited than those that can be defined for element types. Specifically, in this first iteration of the Armani language it is not possible to associate arbitrary invariants with a property type and the language provides no support for property subtypes. A property type defines only structural predicates. This limitation is made in the interest of keeping the property type system relatively simple. Extending the property type language to include support for arbitrary invariants should, however, be reasonably straightforward.

In order to specify the semantics of property types and property type satisfaction, a set of equations describing how property type declarations are transformed into predicates in the semantic domain is provided. A property is type-correct if the property's value satisfies the property's type predicate (in the semantic domain). Because property types can be either explicitly named (and then referenced by that name) or used anonymously in a property instance specification, the semantic specifications for the meaning of a property's structural predicate is defined separately from the meaning of naming a specific property type.

The abstract syntax for property type declarations follows (keywords are in boldface):

*PropertyTypeDecl* ::= ***Property Type*** *Name = PropertyTypeDesc*

*PropertyTypeDesc* ::= ***int*** | ***float*** | ***string*** | ***boolean***
| ***set*** { *PropertyTypeDesc* }
| ***sequence*** < *PropertyTypeDesc* >
| ***enum*** { *Name* ("," *Name*)* }
| ***record*** [ $Name_1$ : $PropertyTypeDesc_1$ ;
... $Name_n$ : $PropertyTypeDesc_n$ ; ]
| *Name*

*Name* ::= *<valid-identifier>*

**Property type meaning equations.** An Armani property type is translated into the semantic domain as a predicate that takes a single property value (of unknown type) as an argument. If the predicate evaluates to true then the value satisfies the type that the predicate represents, otherwise the value does not satisfy the type of the property.

Meaning equations:

$\mathcal{M}$[ *PropertyTypeDesc* ] c : PropertyValue → Boolean

$\mathcal{M}$[ Property Type n = *PropertyTypeDesc* ] d =
[ d | $d.t_{prop} \leftarrow d.t_{prop} \cup$ (n, $\mathcal{M}$[ *PropertyTypeDesc* ] d) ]

$\mathcal{M}$[ int ] c = λv. v ∈ { 32-bit Integers }
$\mathcal{M}$[ float ] c = λv. v ∈ { 32-bit FloatingPointValues }
$\mathcal{M}$[ string ] c = λv. v ∈ { ValidArmaniStrings }
$\mathcal{M}$[ boolean ] c = λv. v ∈ { true, false }

$\mathcal{M}$[ enum { $n_1$, ..., $n_m$ } ] c = λv. v ∈ { $n_1$, ..., $n_m$ }

$\mathcal{M}$[ Sequence <PropertyTypeDesc> ] c = λv. Sequence(v) and
forall $e_i$ in v • ($\mathcal{M}$[ PropertyTypeDesc ] c) $e_i$
$\mathcal{M}$[ Set <PropertyTypeDesc> ] c = λv. Set(v) and
forall $e_i$ in v • ($\mathcal{M}$[ PropertyTypeDesc ] c) $e_i$

$\mathcal{M}$[ record [ $n_1$ : $PropertyTypeDesc_1$ ;... $n_m$ : $PropertyTypeDesc_n$ ; ] ] c =
λv. Record(v) and forall i in { 1..n } • $\mathcal{M}$[ $PropertyTypeDesc_i$ ] c v.$n_i$.value

**Anonymous property types.** An instance of a property (e.g. the property *rate* on component instance *foo*) must be typed, but the property does not need to use a previously defined type. A property instance may declare an anonymous compound type, as the following example illustrates:

```
Component foo = {
    Property rate : Record [ speed : int; units : string ] =
                        [ speed : int = 100; units : string = "kb/s" ];
};
```

In this example property *foo.rate* has declared a new anonymous type—a record with the fields *speed* (of type *int*) and *units* (of type *string*). This new type is not visible to any other property or element (hence the term anonymous) but it specifies the structure that the value of the property must possess. Semantically, an anonymous type declaration in the context of a property instance specifies a predicate that the value of that property instance must satisfy. The addition of anonymous types does not require any modification to the semantic equations for property instances given in chapter 2.

## Architectural Styles

Vocabulary types and design rules provide mechanisms for capturing and encapsulating design expertise in the form of design vocabulary and constraints. Although individual types and design rules can be useful by themselves, expertise of this sort tends to be more useful when packaged as part of a coherent collection of related vocabulary and constraints. Armani's architectural style construct provides the ability to aggregate and package related vocabulary and constraints.

### Syntax and semantics of a style declaration

An Armani architectural style specification consists of a set of vocabulary type definitions, a set of design rules, a set of design analyses, and a set of minimal required structure. Any or all of these sets may be empty. A style is fundamentally a system type that also defines a design space. Styles obey all of the rules and semantics of types presented thus far, with some additional syntax and semantics to support the style's design space function.

The informal syntax for defining a style is:

***Style*** *<style-name> = { <style-element>* } ;*

*or*

***Style*** *<style-name>* ***extends*** *<super-style-name>+*
    ***with*** *{ <style-element>* } ;*

*<style-element> ::= <Sequence of:* *required structure and values*
*| required properties*
*| explicit invariants*
*| explicit heuristics*
*| design analyses*
*| type definitions >*

The syntax and semantics for declaring individual type specifications, design rules, and design analyses have been described earlier in the chapter. In its role as a design space,

a style is a named collection (or a package) of such constructs. In its role as a system type, a style constrains the design of systems defined in that style.

System instances may make use of the design expertise packaged in a style by declaring that the system satisfies a style. The syntax for declaring that a system instance satisfies a style is:

***System*** *sample-system : sample-style = { <system-decl-body> } ;*

When a system instance declares that it is designed in a specific style the names of all of the types and design analyses declared in that style are visible within the system instance. Further, all of the design rules contained in the style definition must hold over the system instance. That is, the design rules in the style definition take effect in the scope of the system instance, binding the concrete elements in the system instance to the appropriate abstract design rules of the style.

Declaring that a system is designed in a specific style indicates that the constraints specified by that style in the form of design rules must be maintained in the system instance. Failure to satisfy these constraints constitutes a type error.

The set of type specifications given in a style declaration provide a set of vocabulary types that can be used within a system specification done in that style. The system definition is not, however, limited to using only the types provided by the style (unless there is a design rule that explicitly limits the types of vocabulary that can be used). Design elements within the system instance that claim to satisfy a type defined in the style must, however, satisfy the type predicate given in the style definition.

**Name visibility within a style.** In order to insure that styles can be used as independent, modular packages, abstract design rules and types defined within a style have limited visibility to names defined outside of the scope of the style definition. Type definitions within a style may only be subclasses of types defined in a superstyle. Likewise, ports and roles defined within a component or connector type definition may only claim to satisfy types defined within the style itself or one of the style's superstyles.

A design rule defined in a style may refer to design analyses defined within that style, design rules defined in an explicitly included library, the types defined within that style, and, of course, all primitive Armani predicates.

### Substyles

A style can extend an existing style to make use of the types and design rules defined in the existing style. The following example illustrates such an extension:

```
Style super = { ... };

Style sub extends super with {
        Component type new-component = { ... };
        Invariant forall x in self.components • foo(x));
};
```

This example creates a new style called *sub* that extends an existing style called *super.* The new style sub consists of the union of the types, design rules, design analyses, and structure defined in both the style "super" and those defined directly in the definition of style

"sub." The new style "sub" is a *substyle* of the style "super." Because the substyling operation only allows additional types, design rules, and structure to be added to a style, any system that satisfies the constraints of sub will also satisfy the constraints of super.

A substyle may not redefine types or design rules named in the superstyle. It may, however, create new types that extend the types defined in a superstyle.

Multiple inheritance is supported in creating substyles. To create a substyle of multiple superstyles, a style definer simply adds multiple superstyles to the "extends with" statement, as the example below indicates.

```
Style sub extends super-1, super-2, super-3 with {
        Component type new-component = { ... };
        Invariant forall x in self.components | foo(x));
};
```

The semantics for using multiple superstyles are effectively the same as using a single superstyle. The new style "sub" will consist of the union of the types, abstract design rules, and binding statements defined in all of its "style-n" superstyles and those defined directly in the definition of style "sub."

### System instances can use multiple styles

A system can declare that it satisfies multiple styles. In specifying that it satisfies multiple styles it also claims to satisfy the constraints of all of the styles used. The syntax for declaring that a system uses (satisfies) multiple styles is:

```
System sample-sys : style-1, style-2, ..., style-n = {...} ;
```

The semantics of a such a declaration say that the vocabulary of styles 1 through n are all available for use in the system, design rules from styles 1 to n are bound to sample-sys. As a result, sample-sys needs to satisfy all of the design rules of each style. The style constraints of system sample-sys are the unification of the vocabulary and quantified design rules of styles 1 to n.

The "new" operator defined earlier in this chapter for creating minimal instances of simple element types can also be used with styles to create systems with the minimal required structure to satisfy the style specification. As with simple element types, the "extended with" construct can be used to extend the minimal structure provided by "new" and customize the created system. The basic syntax for creating a new minimal instance of a style follows:

```
System <sys-name> : <style-name> = new <style-name> [ extended with {...} ] ;
```

The semantics for using the new operator with systems are the same as the semantics for using new with simple element types.

### Dealing with multi-style conflicts

Using multiple styles in a single system can cause various kinds of conflicts. The two primary kinds of conflicts are name conflicts (for vocabulary and design rules) and conceptual conflicts where the styles are fundamentally incompatible.

**Naming conflicts** are fairly easy to deal with. The use of an ambiguous name within a system specification is an error. Ambiguous naming can be avoided by qualifying types and design rules specified within style definitions with the name of the desired style from which the type or design rule should be used. For example:

```
Style generic-cs = { ... Component Type client = {...} ... };
Style special-cs = { ... Component Type client = {...} ... };

System sample : generic-cs, special-cs = {
  Component generic-client : generic-cs.client = {...};
  Component special-client : special-cs.client = {...};

  Invariant special-client.no-peers(...);
}
```

In this example, the client type is defined in both the generic-cs style and the special-cs style. Each of the *.client type components in the system *sample* explicitly specify which of the client types (special-cs.client or generic-cs.client) they claim to satisfy. The components in system sample cannot simply claim to be of type client, as the client type is provided by multiple styles used by the system, making the identifier "client" ambiguous. Type and design rule names that appear in only one style do not need to be qualified in a system declaration. Qualification with a style name is required only where the lack of a qualifying identifier leads to ambiguity. References from within a system specification to abstract design rules that are provided by styles follow the same set of disambiguating and qualifying rules as those used for disambiguating types defined by multiple styles.

Style designers do not need to explicitly qualify references to types or design analyses that are defined within the same style specification. All references to types or abstract design rules from within a style definition are implicitly qualified and, in fact, only able to, refer to types and abstract design rules of the same style. Consider the following (revised) example:

```
Style generic-cs = { ...
        Component Type client = {...}
        Component Type server = {...}
        ...
};

Style special-cs = { ...
        Component Type client = {...}
        Component Type server = {...}
        Design Analysis only-cs-conns(c : client, s:server) : boolean = {...};
        Invariant Forall c,s in {select self.Components | SatisfiesType(c, Client) and
                                        SatisfiesType(s, Server) } | only-cs-conns(c,s);
}

System sample : generic-cs, special-cs = { ... }
```

In this example, the design rules and the types have been automatically (and implicitly) qualified to refer to their "native" style when evaluated in the context of the system instance.

**Fundamental conceptual mismatch.** The second important type of conflict that can occur when using multiple styles within a single system are fundamental conceptual

mismatches. These conflicts occur because the styles being used are fundamentally incompatible with each other. An example of such a conflict is a system that merges a pipe-filter style, which requires that all components are filters and all connectors are pipes, with a client-server style that requires all components to be clients or servers and all connectors to be HTTP streams. Unless the required types are (accidentally) compatible with each other (e.g. instances of Filters satisfy the constraints of Client) it is unlikely that any non-empty system instances can be created that satisfy the constraints of both styles.

It is up to Armani users to detect and avoid such conceptual conflicts. Using multiple styles in a single system instance expands the vocabulary available for use in that system but generally constrains the design of the system further by introducing additional design constraints. As the previous example indicates, it is possible to overly constrain a design by using multiple styles. Tools can be developed to detect obvious style incompatibilities, but they will not eliminate the need to be careful when using multiple styles for a single system instance.

## Style Semantics

The specification of the formal semantics of architectural styles is a straightforward extension to the formal semantics of simple elements (components, connectors, ports, and roles) and simple element types (their corresponding types). Logically, an architectural style is a system type. In addition to its role as a system type, though, an architectural style also defines a design space for declaring property and element type specifications, abstract design rules, and design analyses. The types, design rules, and design analyses defined in a style *s* are available for use within system instances defined in the style *s*. Declaring that a system instance is a member of a style (e.g. System s : SomeStyle = {...} ) makes the type, design rule, and design analyses declared in the style visible within the system instance specification but does not limit the use of types, design rules, and analyses defined elsewhere (unless explicitly prohibited by style-wide invariants).

To maintain the semantic parallel between the structure of an element instance tuple and an element type tuple introduced with simple elements, system instances, like styles, will define a design space, allowing element types, property types, and design analyses to be defined within the scope of a system instance. The design spaces defined by styles and system instances do not, however, support the recursive definition of systems or styles within themselves.

As with the other Armani constructs, the semantics of styles is given by providing a set of semantic equations that translate an Armani style description into a tuple that can be analyzed algorithmically. To remain consistent with the format used for defining simple element instances and types, a tuple type *s* is introduced that can be used to represent either systems or styles. A tuple *s* has all of the substructure of an element tuple *e*, with the addition of a field *d*, which represents the new design space defined by the style or system.

The revised tuple *s* representing an Armani system or style in the semantic domain is a tuple of the form:

$$s = (n, c, s, p, a, i, h, t_{super}, t_{asserted}, d).$$

Informally, the fields of the tuple represent:

*n* = name of the family.
*c* = category of the tuple (always family).
*s* = set of elements that define an instance of *f*'s default structure
*p* = set of properties that define the properties of *f*.
*a* = set of attachments that define the default topology of a system instance
*i* = set of invariant predicates defined for *f*.
*h* = set of heuristics defined for *f*.
$t_{super}$ = set of names of super-styles that this style definition extends.
$t_{asserted}$ = set of names of styles this system instance claims to satisfy.
*d* = design space tuple that stores the design space defined by *s*.

Well-formedness rules:

The well-formedness rules given here describe the legal constructs that can be added to a style definition. The primary purpose of the well-formedness rules is to insure that no two distinct entities declared as part of this system or style share the same identifier.

s = (n, c, s, p, a, i, h, $t_{super}$, $t_{asserted}$, d).

n : Identifier
c : Category
s : Set{Element}
p : Set{Property}
a : Set{Attachment}
i : Set{InvariantPredicate}
h : Set{HeuristicPredicate}
$t_{super}$: Set{StyleName}
$t_{asserted}$ : Set{StyleName}
d : DesignSpace

Style(s)
**and** (($t_{super}$ == {}) **xor** ($t_{asserted}$ == {}))
**and** d.s == {}
**and** forall t in $d.t_{elements}$ • t.c != System

Meaning (system instance declarations):

A system instance declaration that claims to satisfy no style/type and occurs in the context of a design space has the following meaning:

$\mathcal{M}$[ System n = { $s_1$ ;...; $s_n$ } ] d =
**if** (n ∉ Names(d.f)) **and** (n ∉ Names(d.s)) **and** (n ∉ Names($d.t_{prop}$)
**and** (n ∉ Names($d.t_{elt}$)) **then**
$\mathcal{M}$[ { $s_1$ ;...; $s_n$ } ] [ e | d.s ← d.s ∪ { e }, e.n ← n, e.c ← System,
e.s ← {}, e.p ← {}, e.r ← {}, e.a ← {},
$e.t_{asserted}$ ← { System }, $e.t_{super}$ ← {},
e.i ← {}, e.h ← {},
$e.d.t_{elements}$ ← {}, $e.d.t_{properties}$ ← {},
$e.d.s$ ← {}, $e.d.d_a$ ← {} ]

Meaning (style declarations):

A system type (style) declaration that claims no super-styles and occurs in the context of a design space has the following meaning:

$\mathcal{M}$[ Style n = { $s_1$ ;...; $s_n$ } ] d =

**if** (n ∉ Names(d.f)) **and** (n ∉ Names(d.s)) **and** (n ∉ Names($d.t_{prop}$)
**and** (n ∉ Names($d.t_{elt}$)) **then**

$\mathcal{M}$[ { $s_1$ ;...; $s_n$ } ] [ e | d.s ← d.s ∪ { e }, e.n ← n, e.c ← *Category*,
e.s ← {}, e.p ← {}, e.r ← {}, e.a ← {},
$e.t_{asserted}$ ← {}, $e.t_{super}$ ← { System },
e.i ← {}, e.h ← {}, e.d ← d,
$e.d.t_{elements}$ ← {}, $e.d.t_{properties}$ ← {},
e.d.s ← {}, $e.d.d_a$ ← {} ]

A style declaration that claims one or more superstyles and occurs in the context of a design space has the following meaning:

$\mathcal{M}$[ Style n extends $t_1$, ..., $t_n$ with { $s_1$ ;...; $s_n$ } ] d =

**if** (n ∉ Names(d.f)) **and** (n ∉ Names(d.s)) **and** (n ∉ Names($d.t_{prop}$)
**and** (n ∉ Names($d.t_{elt}$)) **then**

$\mathcal{M}$[ { $s_1$ ;...; $s_n$ } ] [ e | d.s ← d.s ∪ { e }, e.n ← n, e.c ← *Category*,
e.s ← {}, e.p ← {}, e.r ← {}, e.a ← {},
$e.t_{asserted}$ ← {}, $e.t_{super}$ ← {System} ∪ { $t_1$,..., $t_n$},
e.i ← {}, e.h ← {},
$e.d.t_{elements}$ ← {}, $e.d.t_{properties}$ ← {},
e.d.s ← {}, $e.d.d_a$ ← {} ]

Chapter

4

# The Armani Predicate Language

The Armani predicate language is the portion of the Armani design language used for specifying the predicates of design invariants and design heuristics. The predicate language is based on first-order predicate logic with composable terms, user-definable functions, and limited quantification capabilities. To keep the language decidable, variables may be quantified only over finite sets.

This chapter presents the core constructs of the predicate language in four sections: primitive functions, operators, quantification, and design analyses. Each of these sections provides a description of the construct, examples of its use, and a semi-formal description of its syntax and semantics.

The semantics given for the predicate logic language are less formal than those given for the Armani structural and type languages because the predicate language consists primarily of predicate logic expressions, which is a well understood semantic domain in and of itself. As such, discussion of language construct semantics focuses on the portions of the predicate language that differ from traditional predicate logic, or require a tricky integration with the rest of the Armani language. For primitive functions and operators a simple intuitive description of the item's meaning should suffice. Tricky semantic issues, on the other hand, include how to evaluate a predicate expression in the semantic domain (e.g. for determining type satisfaction), how parameters are passed to functions, and the scope and visibility of identifiers in expressions. These issues are dealt with in Chapter 3's detailed discussion of the semantics of the type system.

## Primitive Functions

This section describes the primitive functions built into the Armani predicate language for use in design invariants, heuristics, and analyses. The general form used for specifying function signatures is:

*FunctionName(FormalParamName : FormalParamType ,*) : ReturnType*

Followed by a description of the behavior of the function. *Object* is the most general type that can be used as a ReturnType or a formal parameter type. Design elements, design element types, properties, and representations are all subtypes of *object*. Primitive property values (integers, floats, booleans, and strings) are not objects and are not valid arguments to functions that take *objects* as arguments.

## Type Functions

*declaresType(e : Element, t : ElementType) : boolean*

Returns true if element *e* declares that it satisfies type *t*, else returns false.

*satisfiesType(e : Element, t : ElementType) : boolean*

Returns true if element *e* satisfies the predicate defined by type *t*, independent of whether *e* declares to satisfy type *t*, else returns false.

*typesDeclared(e : Element) : set{Type}*

Returns the set of identifiers of the types that element *e* declares that it satisfies.

*declaredSubtype(subType,superType : ElementType) : boolean*

Returns True if *subType* declares that it is a subtype of *superType*, else returns False.

*superTypes(t : ElementType) : set{Type}*

Returns the set of all types that are declared as supertypes of element type *t*. This function is applied recursively to *t*'s supertypes so that the returned set contains all of *t*'s ancestors in the type hierarchy.

## Graph Connectivity Functions

*attached(conn : Connector, comp : Component) : boolean*

Returns True if connector *conn* is attached to component *comp*, else False.

*attached(r : Role, p : Port) : boolean* Returns True if role r is attached to port p, else False.

*connected($c_1$, $c_2$ : Component) : boolean*

Returns True if component $c_1$ is directly connected to component $c_2$ by at least one connector, else False.

*reachable($c_1$, $c_2$ : Component) : boolean*

*Reachable(...)* is the transitive closure of *Connected(...)*. It returns True if component $c_2$ is reachable from component $c_1$, else False. This function examines only undirected graph connectivity. It does not take into account connector directionality or valid port to port paths through components.

## Parent-Child Functions

| | |
|---|---|
| *parent(c : Component) : System* | Returns the System in which Component *c* is instantiated, or nil if *c* is not a child of anything. |
| *parent(c : Connector) : System* | Returns the System in which Connector *c* is instantiated, or nil if *c* is not a child of anything. |
| *parent(p : Port) : Component* | Returns the Component in which Port *p* is instantiated, or nil if *p* is not a child of anything. |
| *parent(r : Role) : Connector* | Returns the Connector in which Role *r* is instantiated, or nil if *r* is not a child of anything. |
| *parent(s : System) : Representation* | Returns the Representation in which System *s* is declared, or nil if *s* is a toplevel System and thus not declared in a representation. |
| *parent(p : Property) : Element* | Returns the Element of which *p* is a Property, or nil if *p* is not a property of anything. |
| *parent(r : Representation) : Element* | Returns the Element of which *r* is a Representation, or nil if r is not a Representation of anything. |

## Set Functions

| | |
|---|---|
| *union* ($s_1$, $s_2$ : *Set{α}*) : *Set{α}* | Returns the union of sets $s_1$ and $s_2$. |
| *intersection* ($s_1$, $s_2$ : *Set{α}*) : *Set{α}* | Returns the intersection of sets $s_1$ and $s_2$. |
| *contains(x : object, s : set) : boolean* | Returns true if set *s* contains object *x*, else false. |
| *flatten( sets : Set{Set{α}}) : Set{α}* | Returns the union of the elements of all sets contained in the argument *sets*. The signature for this function is Flatten : Set(Set(α)) → Set(α) |
| *setDifference(lhs, rhs : Set{α}}) : Set{a}* | Returns the set difference of sets *lhs* and *rhs*. That is: *rhs – lhs*. The signature of this function is SetDifference : (Set(α) x Set(α)) → Set(α) |
| *isSubset(subset, superset : Set{α}) : boolean* | Returns true if set *subset* is a subset of set *superset*, else false |
| *size(s : set) : integer* | Returns the cardinality of the set *s*. |
| *sum(s : set{number}) : number* | Returns the sum of all of the numbers in the set *s*. |
| *product(s : set{number}) : number* | Returns the product of the numbers in the set *s*. |

The *select(...)* and *collect(...)* set constructors also operate on sets, but with a slightly different syntax from other functions. Set constructors are described in detail in this chapter's section on quantification.

### Identifiers and Literal Constants

Literal constants may be used for comparison, as parameters, or as functions that return their own value. Examples of literal constants include: *true*, *124*, *"string"*, and *"zanzibar"*. Identifiers are names that can be resolved as references to an object.

Appropriately typed literal constants can be used as arguments to functions. Likewise, appropriately typed identifiers may be used as arguments to functions. The meaning of an identifier passed as an actual parameter to a function varies depending on the type specified by the function's formal parameter. If the formal parameter specifies an object type then a reference to the object referred to by the identifier is passed as the function argument. This is pass-by-reference semantics. If, on the other hand, the function's formal parameter specifies a primitive property type (int, float, boolean, string, record[...], set{...}, or sequence<...>) the identifier passed as an actual parameter is dereferenced and the value of the property referred to by the identifier is passed as the actual parameter (pass by value semantics).

None of the primitive functions specified in the Armani predicate language have side effects or change the state of the objects passed as arguments. Therefore, the parameter passing semantics can always be thought of as pass-by-value, with all of the dereferencing of identifiers done before passing the value of the object.

## Operators

The Armani predicate language includes primitive operators for comparison, logic, arithmetic, and set operations, along with a small collection of miscellaneous operations. Details for each category of operator are given in the following tables.

### Comparison operators

The predicate language includes the following operators for comparing two values:

| Operator | Example | Meaning |
|---|---|---|
| == | x == y | true if x equals y, else false |
| != | x != y | true if x does not equal y, else false |
| > | x > y | true if x is greater than y else false |
| >= | x >= y | true if x is greater than or equal to y, else false |

| | | |
|---|---|---|
| < | x < y | true if x is less than y, else false |
| <= | x <= y | true if x is less or equal to y, else false |

Comparison operator notes:

- Operands must be numeric for the operators >, <, >=, and <=.
- The equality and inequality operators (== and !=) perform object (or reference) equality tests for design elements and references to design elements. That is, the statement *element e == element e'* is true iff *element e is element e'*. The equality operators perform value equality tests on properties and literal values. That is, if **property** *x* == 4 and **property** *y* == 4 then *x* = *y*.

## Logic operators

The predicate language includes the following operators for building logical expressions:

| Operator | Example | Meaning |
|---|---|---|
| **and** | x **and** y | true if both x *and* y are true, else false |
| **or** | x **or** y | true if either x *or* y are true, else false |
| **xor** | x **xor** y | true if (x = true and y = false) or (x = false and y = true), else false |
| -> | x -> y | Implication (if). If x is true then y must be true. If x is false, the value is true. |
| <-> | x <-> y | two-way implication (iff). true if x and y are either both true or both false. false if they have different values. |
| ! | !x | not x. True if x is false, else false |

For all logical operators, both the left-hand-side (lhs) and the right-hand-side (rhs) operands must evaluate to boolean typed values. Likewise, all logical operators return a boolean typed value.

## Arithmetic operators

The predicate language includes the following operators for doing simple arithmetic operations:

| Operator | Example | Meaning |
|---|---|---|

| | | |
|---|---|---|
| + | x + y | Addition. Sum of the values of x and y |
| - | x – y | Subtraction. Difference of x – y |
| * | x * y | Multiplication. Product of x and y |
| / | x / y | Integer division if both x and y are integers, else floating point division. |
| **mod** | x **mod** y | Modular division. |

For all of the arithmetic operators, the operands need to be numeric. Modular division requires that both operands be integers.

## Miscellaneous operators

The following miscellaneous operators are also be primitive operations in the Armani predicate language:

| Operator | Example | Meaning |
|---|---|---|
| () | (z or (x < y)) | Precedence operator. all expressions in the innermost parentheses are evaluated before moving to the outer level of parentheses. Standard C interpretation used. |
| . | object.property<br><br>comp.portName | Qualifier and dereferencer. The period is used to qualify names and dereference substructure within an object. The value of the referenced entity is returned by the operation. |

## Using Operators to Build Expressions

It is, of course, possible to build up expressions using the primitive predicates and the supplied operators. The grammar for doing so looks is:

*Expression* *::= Primitive-Function(ActualParams)*
*| Literal-Constant*
*| Identifier*
*| DesignAnalysisIdentifier(ActualParams)*
*| QuantifedPredicate*
*| Unary-Operator Expression*
*| Expression Binary-Operator Expression;*

*Unary-Operator* *::= Armani primitive unary-operator ;*

*Binary-Operator* *::= Armani primitive binary-operator ;*

*Primitive-Function* ::= *Primitive Armani Predicate Language function ;*

*Identifier* ::= *Valid reference to a variable or property*

*DesignAnalysisIdentifier* ::= *Valid reference to a user-defined Design Analysis*

*ActualParams* ::= *List of values for parameters to pass to function*

The production for *QuantifedPredicate* is defined in this chapter's Quantification section. A *PredicateExpression* is an *Expression* that evaluates to a boolean value.

## Quantification

In addition to primitive functions and operators, the Armani predicate language provides a limited quantification mechanism. Both universal and existential quantification are supported, with the limitation that all quantification must be done over finite sets. This limitation insures that the satisfaction of any predicate expression is computationally decidable. Although limiting quantification to finite sets reduces the range of expressions that can be specified in the predicate language, the guarantee of computational decidability more than offsets this limitation. Further, a sufficient range (and possibly the bulk) of the predicate expressions of interest to software architects and environment designers can be expressed using quantification over finite sets.

Quantified predicates use the following syntax in the predicate language.

*QuantifiedPredicate ::= ( **forall** | **exists** ) Identifier ( , Identifier)* : TypeExpression*
*in SetExpression ( , Identifier ( , Identifier)*
*: TypeExpression **in** SetExpression )**
*| PredicateExpression ;*

Quantified predicates are boolean functions that evaluate a predicate expression for all members of a finite set of values. The keyword *forall* indicates a universal quantification and the keyword *exists* indicates an existential quantification. The following examples illustrate the use of simple quantified predicates.

```
Forall comp : aCompType in sys.Components | comp.secure = True;
Exists conn : connector  in sys.Connectors |DeclaresType (conn, EventSystemType);
```

Quantified predicates may be embedded within other quantified predicates. For example:

```
Forall comp : aCompType in sys.Components |
        Forall port : aPortType in comp.Ports | port.protocol = RPC;
Forall comp : aCompType in sys.Components |
        Exists conn : aConnType in sys.Connectors |
            Connected(comp,conn);
```

As the syntax specification indicates, quantified predicates are not limited to single variables. Multi-variable quantifications are supported. The following predicate is true in systems where all components are connected to all other components by at least one

connector, forming a fully connected graph. Admittedly, this example probably better illustrates complex quantifications than clean system design.

***Forall** c1, c2 : component **in** sys.Components |*
*  **Exists** conn : connector **in** sys.Connectors |*
*  Attached (c1, conn) **and** Attached (c2, conn)*

**Quantification Semantics.**

The following three semantic equations describe the meaning of simple quantified predicates whose quantification includes just a single variable. The more complex semantic equations described subsequently are just generalizations of these basic equations. The first equation indicates that a *PredicateExpression* in the Armani predicate language is a function from some context (a mapping of variable names to values) to a boolean value. The meanings of the individual functions and operators that can be composed to form a valid PredicateExpression are given in the previous two sections of this chapter.

$$\mathcal{M}[\ \textit{PredicateExpression}\ ] : \text{Context} \rightarrow \text{Boolean}$$

$$\mathcal{M}[\ \textbf{Forall}\ v\ \text{in}\ \{ e_1 ; \ldots ; e_m \} \mid \textit{PredicateExpression}\ ]\ c =$$
$$\mathcal{M}[\ \textit{PredicateExpression}\ ][\ c \mid v \leftarrow e_1\ ]$$
$$\textbf{and}\ \mathcal{M}[\ \textit{PredicateExpression}\ ][\ c \mid v \leftarrow e_2\ ]$$
$$\textbf{and} \ldots \textbf{and}\ \mathcal{M}[\ \textit{PredicateExpression}\ ][\ c \mid v \leftarrow e_m\ ]$$

$$\mathcal{M}[\ \textbf{Exists}\ v\ \text{in}\ \{ e_1 ; \ldots ; e_m \} \mid \textit{PredicateExpression}\ ]\ c =$$
$$\mathcal{M}[\ \textit{PredicateExpression}\ ][\ c \mid v \leftarrow e_1\ ]$$
$$\textbf{or}\ \mathcal{M}[\ \textit{PredicateExpression}\ ][\ c \mid v \leftarrow e_2\ ]$$
$$\textbf{or} \ldots \textbf{or}\ \mathcal{M}[\ \textit{PredicateExpression}\ ][\ c \mid v \leftarrow e_m\ ]$$

The following three pairs of equations specify the meaning for quantifications involving more than one variable. The first pair of equations simply define the meaning of a shorthand for quantifying multiple variables over a single set. They say that quantifying multiple variables over a single set is the same operation, semantically, as quantifying each of the variables individually over different sets that all contain the same values. The second pair of equations provide a generalization of the first equations that describe how complex combinations of variables quantified in a single quantification expression with multiple variables and multiple sets are translated into the canonical form of a series of simple quantification expressions. The final pair of equations describes the meaning of quantified predicates whose quantification expression are expressed in this canonical form.

The intuitive explanation of these semantic equations is that in a predicate expression that is universally quantified, the value of that expression is true if and only if the expression holds for all combinations of values of the variables specified in the quantification expression. The value of an existentially quantified predicate expression is true if and only if there exists at least one combination of variable bindings that makes the predicate expression true.

Semantic equations for translating quantifications of multiple variables over a single set into canonical semantic form.

$\mathcal{M}$[ **Forall** $v_1, v_2, \ldots, v_n$ in $\{ e_1 ; \ldots ; e_m \}$ | *PredicateExpression* ] c =
$\mathcal{M}$[ **Forall** $v_1$, in $\{ e_1 ; \ldots ; e_m \}$, $v_2$, in $\{ e_1 ; \ldots ; e_m \}$, ... , $v_n$, in $\{ e_1 ; \ldots ; e_m \}$,
| *PredicateExpression* ] c

$\mathcal{M}$[ **Exists** $v_1, v_2, \ldots, v_n$ in $\{ e_1 ; \ldots ; e_m \}$ | *PredicateExpression* ] c =
$\mathcal{M}$[ **Exists** $v_1$, in $\{ e_1 ; \ldots ; e_m \}$, $v_2$, in $\{ e_1 ; \ldots ; e_m \}$, ... , $v_n$, in $\{ e_1 ; \ldots ; e_m \}$,
| *PredicateExpression* ] c

Generalization of prior equations for quantifying multiple variables over multiple sets:

$\mathcal{M}$[ **Forall** $v_1, v_2, \ldots, v_n$ in $\{ e_1 ; \ldots ; e_{m1} \}$, $v_1', v_2', \ldots, v_n'$ in $\{ e_1' ; \ldots ; e_{m2}' \}$, ...,
$v_1^{p'}, v_2^{p'}, \ldots, v_n^{p'}$ in $\{ e_1^{p'} ; \ldots ; e_{mn}^{p'} \}$ | *PredicateExpression* ] c =
$\mathcal{M}$[ **Forall** $v_1$, in $\{ e_1 ; \ldots ; e_m \}$, $v_2$, in $\{ e_1 ; \ldots ; e_m \}$, ... , $v_n$, in $\{ e_1 ; \ldots ; e_m \}$,
$v'_1$, in $\{ e'_1 ; \ldots ; e'_{m1} \}$, $v'_2$, in $\{ e'_1 ; \ldots ; e'_{m1} \}$, ..., $v'_n$, in $\{ e'_1 ; \ldots ; e'_{m1} \}$, ...,
$v^{p'}_1$, in $\{ e^{p'}_1 ; \ldots ; e^{p'}_{mn} \}$, $v^{p'}_2$ in $\{ e^{p'}_1 ; \ldots ; e^{p'}_{mn} \}$, ..., $v^{p'}_n$, in $\{ e^{p'}_1 ; \ldots ; e^{p'}_{mn} \}$
| *PredicateExpression* ] c

$\mathcal{M}$[ **Exists** $v_1, v_2, \ldots, v_n$ in $\{ e_1 ; \ldots ; e_{m1} \}$, $v_1', v_2', \ldots, v_n'$ in $\{ e_1' ; \ldots ; e_{m2}' \}$, ...,
$v_1^{p'}, v_2^{p'}, \ldots, v_n^{p'}$ in $\{ e_1^{p'} ; \ldots ; e_{mn}^{p'} \}$ | *PredicateExpression* ] c =
$\mathcal{M}$[ **Exists** $v_1$, in $\{ e_1 ; \ldots ; e_m \}$, $v_2$, in $\{ e_1 ; \ldots ; e_m \}$, ... , $v_n$, in $\{ e_1 ; \ldots ; e_m \}$,
$v'_1$, in $\{ e'_1 ; \ldots ; e'_{m1} \}$, $v'_2$, in $\{ e'_1 ; \ldots ; e'_{m1} \}$, ..., $v'_n$, in $\{ e'_1 ; \ldots ; e'_{m1} \}$, ...,
$v^{p'}_1$, in $\{ e^{p'}_1 ; \ldots ; e^{p'}_{mn} \}$, $v^{p'}_2$ in $\{ e^{p'}_1 ; \ldots ; e^{p'}_{mn} \}$, ..., $v^{p'}_n$, in $\{ e^{p'}_1 ; \ldots ; e^{p'}_{mn} \}$
| *PredicateExpression* ] c

Semantic equations for translating canonical quantifications of multiple variables over multiple sets into predicates in the semantic domain.

$\mathcal{M}$[ **Forall** $v_1$, in $\{ e_1 ; \ldots ; e_{m1} \}$, $v_2$ in $\{ e'_1 ; \ldots ; e'_{m2} \}$,
... $v_n$ in $\{ e^{p'}_1 ; \ldots ; e^{p'}_{mn} \}$ | *PredicateExpression* ] c =
$\mathcal{M}$[ *PredicateExpression* ] [ c | $v_1 \leftarrow e_1, v_2 \leftarrow e'_1, \ldots, v_n \leftarrow e^{p'}_1$ ]
**and** $\mathcal{M}$[ *PredicateExpression* ] [ c | $v_1 \leftarrow e_2, v_2 \leftarrow e'_1, \ldots, v_n \leftarrow e^{p'}_1$ ]
**and** ... **and** $\mathcal{M}$[ *PredicateExpression* ] [ c | $v_1 \leftarrow e_{m1}, \ldots, v_n \leftarrow e^{p'}_1$ ]
**and** $\mathcal{M}$[ *PredicateExpression* ] [ c | $v_1 \leftarrow e_1, v_2 \leftarrow e'_2, \ldots, v_n \leftarrow e^{p'}_1$, ]
**and** ... **and** $\mathcal{M}$[ *PredicateExpression* ] [ c | $v_1 \leftarrow e_{m1}, v2 \leftarrow e'_2, \ldots, v_n \leftarrow e^{p'}_1$ ]
**and** ... **and** $\mathcal{M}$[ *PredicateExpression* ] [ c | $v_1 \leftarrow e_{m1}, v_1 \leftarrow e'_{m2}, \ldots, v_n \leftarrow e^{p'}_{mn}$]

$\mathcal{M}$[ **Exists** $v_1$, in $\{ e_1 ; \ldots ; e_{m1} \}$, $v_2$ in $\{ e'_1 ; \ldots ; e'_{m2} \}$,
... $v_n$ in $\{ e^{p'}_1 ; \ldots ; e^{p'}_{mn} \}$ | *PredicateExpression* ] c =
$\mathcal{M}$[ *PredicateExpression* ] [ c | $v_1 \leftarrow e_1, v_2 \leftarrow e'_1, \ldots, v_n \leftarrow e^{p'}_1$ ]
**or** $\mathcal{M}$[ *PredicateExpression* ] [ c | $v_1 \leftarrow e_2, v_2 \leftarrow e'_1, \ldots, v_n \leftarrow e^{p'}_1$ ]
**or** ... **or** $\mathcal{M}$[ *PredicateExpression* ] [ c | $v_1 \leftarrow e_{m1}, \ldots, v_n \leftarrow e^{p'}_1$ ]
**or** $\mathcal{M}$[ *PredicateExpression* ] [ c | $v_1 \leftarrow e_1, v_2 \leftarrow e'_2, \ldots, v_n \leftarrow e^{p'}_1$, ]
**or** ... **or** $\mathcal{M}$[ *PredicateExpression* ] [ c | $v_1 \leftarrow e_{m1}, v2 \leftarrow e'_2, \ldots, v_n \leftarrow e^{p'}_1$ ]
**or** ... **or** $\mathcal{M}$[ *PredicateExpression* ] [ c | $v_1 \leftarrow e_{m1}, v_1 \leftarrow e'_{m2}, \ldots, v_n \leftarrow e^{p'}_{mn}$]

## Set expressions and set constructors

As mentioned earlier in this section, variables can only be quantified over finite sets. The Armani predicate language provides a number of set operators and constructors for

creating an appropriate set for quantification. The syntax for the *SetExpression* nonterminal used in the syntax production of *QuantifiedPredicate* follows:

*SetExpression* ::= *SetLiteral* | *Identifier* | *SetFunction* | *SetConstructor*

*SetLiteral* ::= { } | { (*Literal* | *Identifier*) (,*Literal* | , *Identifier*)* }

*SetFunction* ::= *Primitive-Function* ( *ActualParams* )
| *DesignAnalysisIdentifier* ( *ActualParams* )

*SetConstructor* ::= { ***select*** *variable* ***in*** *SetExpression* | *PredicateExpression* }
| { ***collect*** *variable.substructure* ***in*** *SetExpression* | *PredicateExpression* }

The standard ways for specifying and/or creating a set over which a variable will be quantified include (1) explicitly specifying a literal set, (2) referring to an existing set, (3) calling a function that returns a set type, or (4) constructing a set using the *select* or *collect* set constructor operators. The following examples illustrate each of these four methods.

(1) ***Forall*** *x : int* ***in*** { *1, 2, 3, 4, 5* } | *P(x)*
(2) ***Forall*** *x : port* ***in*** *self.ports* | *P(x)*
(3) ***Exists*** *x : port* ***in*** *union(foo.ports, bar.ports)* | *P(x)*
(4) ***Forall*** *x : component* ***in***
{ *select c : aCompType* ***in*** *self.Components* | *c.rate > 100* } | *P(x)*

These techniques can be combined as appropriate wherever a *SetExpression* can be used. The following example illustrates mixing and matching *SetExpressions.*

***Forall*** *x : int* ***in*** { *select y in { 1, 2, 3, 4, 5 } | y > 2* } | *P(x)*

**Semantics of *select* and *collect*.** The set expression syntax introduces the functions *select* and *collect*. These functions are both set constructors that create a new set from the members of an existing set.[2] The select function acts as a filter that takes a set *s* and a predicate expression *p* as arguments and returns a new set *s'* that contains the members of the set *s* that satisfy the predicate *p*. The returned set *s'* is a subset of the original set *s*, as the following example illustrates:

{ ***select*** *v : int* ***in*** { 1, 2, 3, 4, 5 } | *v* > 2 } returns the set { 3, 4, 5 }.

Like the *select* function, the *collect* function acts as a filter on a set *s* and creates a new set *s'* from the elements of *s* that satisfy the predicate *p*. The new set *s'*, however, is not a subset of the original set *s*. Rather, the new set consists of the values of the specified substructure of the original set's elements. The following examples should help to illustrate:

{ ***collect*** *v.x : int* ***in*** { [x=4; y=6], [x=5; y=7]; [x=5; y=8] } | *v.y* – *v.x* = 2 }
returns the set { 4, 5 }

{ ***collect*** *c.ports : set{port}* ***in*** self.Components | *satisfiesType(c, Filter)* }
returns the set { all sets of ports on self's components that satisfy type Filter }

[2] The *select* and *collect* constructs were derived from similar constructs in UML's Object Constraint Language (OCL). Further information on OCL is available at www.omg.org and at www.rational.com/uml/html/ocl.

**Standard sets.** All design elements consist of one or more finite sets that can be dereferenced using the "." (dot) operator. A list of the standard sets available for various categories of design elements follows. The basic notation used here is that anything in angle brackets (e.g. <Component> ) refers to a type or category of design element or property. The item in angle brackets needs to be replaced by a reference to an instance of that category in order to retrieve the desired set. The standard sets include:

*<System>.Components* Refers to the set of components that are instantiated in <System>. Only components whose parent is <System> are returned. It does not traverse into any representations.

*<System>.Connectors* Refers to the set of connectors that are instantiated in <System>. Only connectors whose parent is <System> are returned. It does not traverse into any representations.

*<Component>.Ports* Refers to the set of ports defined as interfaces to <Component>.

*<Connector>.Roles* Refers to the set of roles defined as interfaces to <Connector>.

*<Element>.Representations*
Refers to the set of representations of <Element>

*<Element>.Properties* Refers to the set of properties of a design element

In addition to these standard sets, the dot notation can be used on elements with set-typed properties to dereference the sets stored in those properties for use in quantification.

## Design Analyses

In addition to the primitive functions, the Armani predicate language supports user-defined functions with the *design analysis* construct. A design analysis is simply a user-defined function that can be invoked from design invariants or design heuristics. Design Analyses can be defined by composing primitive functions from the Armani Predicate Language and/or other Design Analyses. Alternatively, design analyses can be externally defined functions or tools that are accessed through the Java interpreter. External design analyses are performed out of the scope of the Armani system, leaving the semantics for such functions undefined. It is the responsibility of the user defining the external design analyses to insure that the function returns the type of entity published in the design analysis signature.

The syntax for the design analysis construct adds the following productions to the Armani predicate language grammar presented thus far:

*DesignAnalysis* ::= ***analysis*** *Identifier ( FormalParams ) : ReturnTypeIdentifier = Expression*
| ***external analysis*** *Identifier( FormalParams ) : ReturnTypeIdentifier = ExternalReference ;*

An *ExternalReference* is a reference to a function defined outside of the Armani language. In general, this will be a reference to a method available in the Java interpreter that is running Armani. The general form of an external reference that calls the Java interpreter will be *package.class.method(ActualParams).*

The following examples illustrate the declaration of design analyses.

```
analysis goesFast(fil : Filter) : boolean = (fil.rate > 1000);

analysis goesFastSecurely(fil : Filter) : boolean =  (goesFast(fil) and fil.secure = true) ;

external analysis sourceCompiles(fil : Filter) : rate = java.tools.checkSrc(fil);
```

Once declared, these design analyses are available for use in design invariants and design heuristics. As an example, the following style description makes use of these design analyses.

```
Style sample = {
    Component Type Filter = { Property rate : int; ... };

    // invariant that says all filters are fast and secure.
    Invariant forall f : component in self.Components |
                satisfiesType(f, Filter) → goesFastSecurely(f);

    // invariant that says the source code compiles for all filters
    Invariant forall f : component in self.Components | sourceCompiles(f);

    // heuristic that limits fan-in and fan-out
    Heuristic forall c : component in self.Components | size(connectedTo(c)) <= 2;
}
```

## Complete Predicate Language Syntax

*Expression* ::= *Primitive-Function(ActualParams)*
| *Literal-Constant*
| *Identifier*
| *DesignAnalysisIdentifier(ActualParams)*
| *QuantifedPredicate*
| *Unary-Operator Expression*
| *Expression Binary-Operator Expression;*

*Unary-Operator* ::= *Armani primitive unary-operator ;*

*Binary-Operator* ::= *Armani primitive binary-operator ;*

*Primitive-Function* ::= *Primitive Armani Predicate Language function ;*

*Identifier* ::= *Valid reference to a variable or property*

*DesignAnalysisIdentifier* ::= *Valid reference to a user-defined Design Analysis*

*ActualParams* ::= *List of values for parameters to pass to function*

*QuantifiedPredicate* ::= ( ***forall*** | ***exists*** ) *Identifier ( , Identifier)** ***in*** *SetExpression*
*( , Identifier ( , Identifier)** ***in*** *SetExpression )**
| *PredicateExpression ;*

*SetExpression* ::= *SetLiteral | Identifier | SetFunction | SetConstructor*

*SetLiteral* ::= { } | { (*Literal* | *Identifier*) (,*Literal* | , *Identifier*)* }

*SetFunction* ::= *Primitive-Function ( ActualParams )*
| *DesignAnalysisIdentifier ( ActualParams )*

*SetConstructor* ::= {***select*** *variable* ***in*** *SetExpression | PredicateExpression* }
| { ***collect*** *variable.substructure*
***in*** *SetExpression | PredicateExpression* }

*DesignAnalysis* ::= ***analysis*** *Identifier ( FormalParams ) : ReturnTypeIdentifier =*
*Expression*
| ***external analysis*** *Identifier( FormalParams ) :*
*ReturnTypeIdentifier = ExternalReference ;*

**Syntax Notes:**

A *PredicateExpression* is an *Expression* that evaluates to a boolean value. An *ExternalReference* is a reference to a function defined outside of the Armani language. In general, this will be a reference to a method available in the Java interpreter that is running Armani. The general form of an external reference that calls the Java interpreter will be *package.class.method(ActualParams).*

Chapter

5

# Armani Examples

This chapter ties together all of the constructs and concepts presented in the previous four chapters by presenting two styles—a Pipe-and-Filter style and a Client-Server style—and two example systems that use the styles.

## Pipe-and-Filter Style and System

The Pipe-and-Filter style is used for systems that can be modeled as a sequence of transformations on a stream of data. The classes of systems that can be modeled in this style range from Unix-based text processing systems to Digital Signal Processing (DSP) pipelines. All components in the Pipe-and-Filter style are *Filters* that read a stream of input data, perform some computation, and then write out the results. All connectors in the Pipe-and-Filter style are *Pipes*, which asynchronously transport a stream of typed data from a source input to a sink output, maintaining the ordering of data elements.

### Pipe-and-Filter style

```
Style Pipe-and-Filter = {

    // First define the design vocabulary

    // Define the flowpaths type
    Property Type flowpathRecT = Record [ fromPt : string; toPt : string; ];

    // Define port and role types
    Port Type inputT = { Property protocol : string = "char input"; };

    Port Type outputT = { Property protocol : string = "char output"; };

    Role Type sourceT = { Property protocol : string = "char source"; };

    Role Type sinkT = { Property protocol : string = "char sink"; };

    // Define component types
    Component Type Filter = {
        Port input : inputT;
        Port output : outputT;
        Property function : string;
        Property flowPaths : set{flowpathRecT}
            << default : set{flowpathRecT} =
                [ fromPt : string = "input"; toPt : string = "output"]; >>;

        // constraint that limits the addition of other ports to input or output ports
```

```
        Invariant  forall p : port in self.Ports |
                satisfiesType(p, inputT) or satisfiesType(p,outputT);
};

// Define component types
Connector Type Pipe = {
        Role source : sourceT;
        Role sink : sinkT;
        Property bufferSize : int;
        Property flowPaths : set{flowpathRecT} =
                [ from : string = "source"; to : string = "sink" ];

        // invariants require that a Pipe have exactly 2 roles, and a buffer with
        // positive capacity.
        Invariant size(self.Roles) == 2;
        Invariant bufferSize >= 0;
 };

//
// Define abstract style-wide design analyses
//

// define an abstract design analysis that checks for cycles in the system graph.
Design Analysis hasCycle(sys :System) : boolean =
        forall c1 : Component  in sys.Components | reachable(c1,c1);


// define an external design analysis that computes the throughput rate for a
// component
External Analysis throughputRate(comp :Component) : int =
        armani.tools.rateAnalyses.throughputRate(comp);


//
// Specify the design invariants and heuristics for systems built in this style.
//

// only attach inputs to sinks and outputs to sources
Invariant forall comp : Component in self.Components |
        Forall conn : Connector in self.Connectors |
                Forall p : Port in comp.Ports |
                        Forall r : Role in conn.Roles |
                                attached(p,r) ->
                                  ((satisfiesType(p,inputT) and satisfiesType(r,sinkT)) or
                                   (satisfiesType(p,outputT) and satisfiesType(r,sourceT)));



// no dangling roles
Invariant forall conn : Connector in self.Connectors | forall r : Role  in conn.Roles |
        exists comp : Component in self.Components | exists p : Port in comp.Ports |
                attached(p,r);

// flag unattached ports
```

```
    Heuristic forall comp : Component in self.Components |
        forall p : Port  in comp.Ports | exists conn : Connector in self.Connectors |
            exists r : Role in conn.Roles |
                attached(p,r);

    // a system in the pipe-and-filter style can have no cycles
    Invariant  !hasCycle(self);

    // all components should have a throughput of at least 100 (units?)
    Heuristic forall comp : Component in self.Components |
        throughputRate(comp) >= 100;

}; // end pipeFilterFam style/family definition.
```

**Pipe-and-Filter system instance**

Having specified the Pipe-and-Filter style, the following example illustrates the definition of a system instance specified in the Pipe-and-Filter style. This is a trivial system that has three components strung together with a pair of pipes. It obeys all of the design rules specified in the style definition, and also specifies a set of invariants that constrain the evolution of this instance of the system (and some of the system's components and connectors as well).

```
System sample : Pipe-and-Filter = new Pipe-and-Filter extended with {

    // declare the components
    Component reduce-noise : Filter = new Filter extended with {
        Property function : string = "reduce-noise.exe";

        // add an invariant that says throughput of this particular filter has to be
        // higher than the minimum required by the style (which is 100).
        Invariant throughputRate(self) >= 150;
    };

    Component find-errors : Filter = new Filter extended with {
        Port error : outputT = new outputT;
        Property function : string = "find-errors.exe";
    };

    Component compute-trajectories : Filter = new Filter extended with {
        Property function : string = "compute-trajectories.exe";
    };

    // declare the connectors
    Connector p1 : Pipe = new Pipe extended with {
        Property bufferSize : int = 1024;
    };

    Connector p2 : Pipe = new Pipe extended with {
        Property bufferSize : int = 1024;
    };

    // hook the filters up with the pipe.
```

```
Attachments = {   reduce-noise.output to p1.source;
                  find-errors.input to p1.sink;
                  find-errors.output to p2.source;
                  compute-trajectories.input to p2.sink;
};

// instance-specific invariants that specify that all components are filters and all
// connectors are pipes.
Invariant forall conn : Connector in self.Connectors |
          declaresType(conn, Pipe) and satisfiesType(conn, Pipe);
Invariant forall comp : Component in self.Components |
          declaresType(comp, Filter) and satisfiesType(comp, Filter);
};
```

## Client-Server Style and System

The Client-Server style is a popular generic style used frequently to construct Distributed Information Systems. Components in the Client-Server style consist of *Clients* and *Servers*. Clients send requests for data or processing to Servers, which perform the requested processing or retrieve data. Connectors in the Client-Server style are either *Procedure-Calls* (PC's) or *Remote Procedure-Calls* (RPC's). RPC's and PC's can be either blocking or non-blocking calls.

```
Style naïve-client-server-style = {

    // define the style's vocabulary
    Component Type naïveClientT = {
        Port makeCall;
    };

    Component Type naïveServerT = {
        Port receiveCall;
        Property max-concurrent-requests : int;
        Design Invariant  max-concurrent-requests <= 5;
    };

    Connector Type pcT = {
        Roles { caller;  callee; };
        Property blocking : boolean << default : boolean = true>>;
        Design Invariant size(self.roles) = 2;
    };

    Connector Type rpcT extends pcT with  {
        Property callerAddress : string;
        Property calleeAddress : string;
    }


    // Define the design analyses that can be used within this style

    // specify topological attachment constraints.
```

```
// only allow client-server connections.
// client-client and server-server connections are invalid.
Design Analysis no-peer-connections(c1,c2:Component) : boolean =
   ( Connected(c1,c2) =>
        ~(DeclaresType (c1,naiveClientT) AND DeclaresType (c2,naiveClientT))
        AND ~(DeclaresType (c1,naiveServerT)
              AND DeclaresType (c2,serverClientT)) );


// Define the design rules for this style.
// limit the vocabulary types used in this style to  naive-clients,
// naive-servers and rpc's.
Forall comp in self.component |
        (DeclaresType(comp, naiveClientT)
              AND SatisfiesType(comp, naiveClientT)
        OR (DeclaresType(comp, naiveServerT)
              AND SatisfiesType(comp, naiveServerT) ;

Forall conn in self.connectors |
        (DeclaresType(conn, pcT)
              AND SatisfiesType(conn, pcT)
        OR (DeclaresType(conn, rpcT)
              AND SatisfiesType(conn, rpcT) ;


Design Invariant forall c1, c2 in self.components | No-peer-connections(c1,c2);

}; // end naive-client-server-style definition.
```

We can now extend this example style to create a substyle that introduces a new subtype of server called a database server. The substyle also introduces an additional design rule that insures a system will always have precisely one primary server. All of the structure and constraints of the previous style definition will be included in the substyle. As a result any system developed in the substyle should also satisfy all of the constraints of the superstyle and work with any tools that are designed to work with the superstyle.

```
Style db-cs-style extends naive-client-server-style with {

        // define the substyle's new database server component type
        Component Type databaseServerT extends naiveServerT with {
              // By redefining rpc-callee, this type definition adds a new property
              // to the type's port rpc-callee.  It also adds a new constraint to maintain.
              Port receiveCall = {
                Property query-language : protocol  << default : protocol = RPC>>;
              }
              Property primary-server : boolean << default : boolean = False >>;
              Design Invariant rpc-callee.query-language == RPC;
        };

        // Add a design invariant that says "There must be exactly one server in
        // a system that has the primary-server property set to "true."
        Design Invariant size({select c in self.components |
                    DeclaresType(c, naiveServerT) → c.primary-server} ) == 1;
```

```
};
```

To complete the client-server example, the following system uses the db-cs-style to describe a simple library information system that keeps library records and makes them available via clients. The server also provides a local in-process client for doing server administration.

```
System library-information-system : db-cs-style = {

    // the primary server
    Component lisDB : databaseServerT = new databaseServerT extended with {
        Port adminPort;
        Property max-concurrent-requests : int = 2;
        Property primary-server : boolean = true;
    };

    // a collection of browsing clients that look up library info.
    Component publicBrowser1 : naiveClientT;
    Component publicBrowser2 : naiveClientT;
    Component refLibrarianBrowser : naiveClientT;

    // an admin client to handle server administration.  This lives in the same
    // process as the server.
    Component adminClient : naiveClientT = new naiveClientT extended with {
        Property access-clearance : acl = high;
    };

    // declare the connectors that plumb the system.
    Connector pub1Conn : rpcT = new rpcT extended with {
        Property blocking : boolean = true;
        Property callerAddress : string = "browser1-machineName";
        Property calleeAddress : string = "lis-db-machineName";
    };

    Connector pub2Conn : rpcT = new rpcT extended with {
        Property blocking : boolean = true;
        Property callerAddress : string = "browser2-machineName";
        Property calleeAddress : string = "lis-db-machineName";
    };

    Connector refConn : rpcT = new rpcT extended with {
        Property blocking : boolean = true;
        Property callerAddress : string = "refLibrarianBrowser-machineName";
        Property calleeAddress : string = "lis-db-machineName";
    };

    Connector adminRequests : pcT;

    // hook the components and connectors together to plumb the system.
    Attachments {
        publicBrowser1.makeCall to pub1Conn.caller;
        lisDB.receiveCall to pub1Conn.callee;
        publicBrowser2. makeCall to pub2Conn.caller;
        lisDB. receiveCall to pub1Conn.callee;
```

```
        refLibrarianBrowser. makeCall to refConn.caller;
        lisDB. receiveCall to refConn.callee;
        adminClient.makeCall to adminRequests.caller;
        lisDB.adminPort to adminRequests.callee;
    };

    // make sure that the lisDB remains the primary server.
    Design invariant lisDB.primary-server == true;
};
```

Appendix

A

# Armani BNF

## BNF Meta-Syntax

**Keyword** Keywords are specified with bold text. Keywords are case-insensitive

*Non-Terminal* Non-Terminals are specified with italics

(...) Parentheses group tokens and productions

[...] Indicates an optional production

(...)? Indicates a sequence of zero or one elements (synonymous with [])

(...)+ Sequence of one or more elements

(...)* Sequence of zero or more elements

| Seperates alternative choices

## Armani Grammar

*ArmaniDesign* ::= ( *TypeDeclaration* | *FamilyDeclaration*
| *DesignAnalysisDeclaration* )*
[ *SystemDeclaration* ]
**<EOF>**

### Design Element Types:

*FamilyDeclaration* ::= ***Family*** *Identifier* [ "(" ")" ] "=" *FamilyBody* [ ";" ]

*FamilyBody* ::= "**{**" ( *TypeDeclaration* )* "**}**"

*TypeDeclaration* ::= *ElementTypeDeclaration* | *PropertyTypeDeclaration*

*ElementTypeDeclaration* ::= *ComponentTypeDeclaration*
| *ConnectorTypeDeclaration*

| *PortTypeDeclaration*
| *RoleTypeDeclaration*

*ComponentTypeDeclaration* ::= ***Component Type*** *Identifier* "=" *parse_ComponentDescription* [ ";" ]
|
***Component Type*** *Identifier* ***Extends*** *Identifier* ( "," *Identifier* )*
***With*** *parse_ComponentDescription* [ ";" ]

*ConnectorTypeDeclaration* ::= ***Connector Type*** *Identifier* "=" *parse_ConnectorDescription* [ ";" ]
|
***Connector Type*** *Identifier* ***Extends*** *Identifier* ( "," *Identifier* )*
***With*** *parse_ConnectorDescription* [ ";" ]

*PortTypeDeclaration* ::= Port Type Identifier "=" parse_PortDescription [ ";" ]
|
***Port Type*** *Identifier* ***Extends*** *Identifier* ( "," *Identifier* )*
***With*** *parse_PortDescription* [ ";" ]

*RoleTypeDeclaration* ::= ***Role Type*** *Identifier* "=" *parse_RoleDescription* [ ";" ]
|
***Role Type*** *Identifier* ***Extends*** *Identifier* ( "," *Identifier* )*
***with*** *parse_RoleDescription* [ ";" ]

*lookup_ComponentTypeByName* ::= *Identifier*

*lookup_ConnectorTypeByName* ::= *Identifier*

*lookup_PortTypeByName* ::= *Identifier*

*lookup_RoleTypeByName* ::= *Identifier*

*lookup_PropertyTypeByName* ::= *Identifier*

## Design Elements:

*SystemDeclaration* ::= ***System*** *Identifier* ( ":" *Identifier* )? "=" *systemBody* [ ";" ]

*SystemBody* ::= ( ***New*** *lookup_ComponentTypeByName* |
"{"
( *ComponentDeclaration* | *ComponentsBlock*
| *ConnectorDeclaration* | *ConnectorsBlock*
| *PortDeclaration* | *PortsBlock* | *RoleDeclaration*
| *RolesBlock* | *PropertyDeclaration* | *PropertiesBlock*
| *AttachmentsDeclaration* | *RepresentationDeclaration*
| *DesignRule*
)*
"}"

```
                    )
                    [ Extended With SystemBody ]

ComponentDeclaration ::= Component Identifier
                    [ ":" lookup_ComponentTypeByName ]
                    ( "=" parse_ComponentDescription ";" | ";" )

ComponentsBlock      ::= Components "{"
                    ( Identifier
                      [ ":" lookup_ComponentTypeByName ]
                      ( "=" parse_ComponentDescription ";" | ";" )
                    )*
                    "}" [ ";" ]

parse_ComponentDescription  ::= ( New lookup_ComponentTypeByName
                                |
                                "{" ( PortDeclaration | PortsBlock
                                      | PropertyDeclaration
                                      | PropertiesBlock
                                      | RepresentationDeclaration
                                      | DesignRule )*
                                "}"
                                )
                                [ Extended With parse_ComponentDescription ]

ConnectorDeclaration ::= Connector Identifier
                    [ ":" lookup_ConnectorTypeByName ]
                    ( "=" parse_ConnectorDescription ";" | ";" )

ConnectorsBlock      ::= Connectors "{"
                    ( Identifier
                      [ ":" lookup_ConnectorTypeByName ]
                      ( "=" parse_ConnectorDescription ";" | ";" ) )*
                    "}" [ ";" ]

parse_ConnectorDescription ::= ( New lookup_ConnectorTypeByName
                               |
                               "{" ( RoleDeclaration | RolesBlock |
                                     | PropertyDeclaration
                                     | PropertiesBlock
                                     | RepresentationDeclaration
                                     | DesignRule )*
                               "}"
                               )
                               [ Extended With parse_ConnectorDescription ]

PortDeclaration      ::= Port Identifier
                    [ ":" lookup_PortTypeByName ]
                    ( "=" parse_PortDescription ";" | ";" )

PortsBlock           ::= Ports "{"
                    ( Identifier
                      [ ":" lookup_PortTypeByName ]
                      ( "=" parse_PortDescription ";" | ";" ) )*
                    "}" [ ";" ]
```

```
parse_PortDescription  ::= ( New lookup_PortTypeByName
                             |
                              "{" ( PropertyDeclaration | PropertiesBlock
                                  | RepresentationDeclaration | DesignRule )*
                              "}"
                             )
                             [ Extended With parse_PortDescription ]

RoleDeclaration        ::= Role Identifier
                           [ ":" lookup_RoleTypeByName ]
                           ( "=" parse_RoleDescription ";" | ";" )

RolesBlock             ::= Roles "{"
                           ( Identifier
                             [ ":" lookup_RoleTypeByName ]
                             ( "=" parse_RoleDescription ";" | ";" ) )*
                           "}" [ ";" ]

parse_RoleDescription  ::= ( New lookup_RoleTypeByName
                             | "{" ( PropertyDeclaration | PropertiesBlock |
                                   RepresentationDeclaration | DesignRule )*
                               "}" )
                           [ Extended with parse_RoleDescription ]

AttachmentsDeclaration ::= [ Identifier "=" ]
                           Attachments "{"
                           ( Identifier "." Identifier to Identifier "." Identifier
                           [ "{" ( PropertyDeclaration | PropertiesBlock )* "}" ]
                           ";" )*
                           "}" ";"
```

## Properties:

```
PropertyDeclaration    ::= Property parse_PropertyDescription ";"

PropertiesBlock        ::= Properties "{"
                           [ parse_PropertyDescription
                             ( ";" parse_PropertyDescription | ";" )*
                           ]
                           "}" [ ";" ]

parse_PropertyDescription ::= [ Property ] Identifier
                           ":" PropertyTypeDescription
                           [ "=" PropertyValueDeclaration ]
                           [  "<<" parse_PropertyDescription
                              ( ";" parse_PropertyDescription | ";" )*
                             ">>"
                            |
                             "<<" ">>"
                           ]

PropertyTypeDeclaration ::= Property Type Identifier
                           ( ";"
```

```
                    |
                     "=" ( Int ";" | Long ";" | Double ";" | Float ";"
                       |String ";" | Boolean ";" | Any ";"
                       |Enum [ "{" Identifier ( "," Identifier )* "}" ] ";"
                       |Set [ "{" "}" ] ";"
                       |Set "{" PropertyTypeDescription "}" ";"
                       |Sequence [ "<" ">" ] ";"
                       |Sequence "<" PropertyTypeDescription ">" ";"
                       |Record "[" parse_RecordFieldDescription
                         ( ";" parse_RecordFieldDescription | ";" )* "]" ";"
                       |Record [ "[" "]" ] ";"
                       |Identifier ";"
                         )
                    )

PropertyTypeDescription    ::= Int | Long | Float | Double | String
                             | Boolean | Any
                             | Set [ "{" [ PropertyTypeDescription ] "}" ]
                             | Sequence [ "<" [ PropertyTypeDescription ] ">" ]
                             | Record "[" parse_RecordFieldDescription
                                ( ";" parse_RecordFieldDescription | ";" )* "]"
                             | Record [ "[" "]" ]
                             | Enum [ "{" Identifier ( "," Identifier )* "}" ]
                             | Enum [ "{" "}" ]
                             | Identifier

parse_RecordFieldDescription  ::= Identifier ( "," Identifier )*
                                  [ ":" PropertyTypeDescription ]

PropertyValueDeclaration   ::= Integer_Literal | Floating_Point_Literal | String_Literal
                             | False | True | AcmeSetValue | AcmeSequenceValue |
                             AcmeRecordValue | Identifier

AcmeSetValue          ::=   "{" "}"
                            | "{" PropertyValueDeclaration
                                ( "," PropertyValueDeclaration )* "}"

AcmeSequenceValue     ::= "<" ">"|
                            "<" PropertyValueDeclaration
                                ( "," PropertyValueDeclaration )* ">"

AcmeRecordValue       ::= "[" RecordFieldValue ( ";" RecordFieldValue | ";" )* "]"

RecordFieldValue      ::= Identifier ":" PropertyTypeDescription "="
                          PropertyValueDeclaration
```

## Representations and Bindings:

```
RepresentationDeclaration  ::= Representation "{"
                               SystemDeclaration
                               [ BindingsMapDeclaration ]
                               "}" [ ";" ]
```

*BindingsMapDeclaration*::= ***Bindings*** "=" "{" ( *BindingDeclaration* )* "}" [ ";" ]

*BindingDeclaration* ::= [ *Identifier* "." ] *Identifier* ***to***
[ *Identifier* "." ] *Identifier*
[ "{" ( *PropertyDeclaration* | *PropertiesBlock* )* "}" ] ";"

## Design Rules and Analyses:

*DesignRule* ::= ( ***Design***)? ( ***Invariant*** | ***Heuristic*** )
*DesignRuleExpression* ";"

*DesignRuleExpression* ::= *QuantifiedExpression* | *BooleanExpression*

QuantifiedExpression ::= ( ***forall*** | ***exists***) *Identifier* ":"
*lookup_arbitraryTypeByName* ***in***
*SetExpression* "|" *DesignRuleExpression*

BooleanExpression ::= *OrExpression* ( ***and*** *OrExpression* )*

OrExpression ::= *ImpliesExpression* ( ***or*** *ImpliesExpression* )*

ImpliesExpression ::= *IffExpression* ( "->" *IffExpression* )*

IffExpression ::= *EqualityExpression* ( "<->" *EqualityExpression* )*

EqualityExpression ::= *RelationalExpression* ( "==" *RelationalExpression*
| "!=" *RelationalExpression* )*

RelationalExpression ::= *AdditiveExpression*
( "<" *AdditiveExpression* | ">" *AdditiveExpression*
| "<=" *AdditiveExpression* | "=>" *AdditiveExpression* )*

AdditiveExpression ::= *MultiplicativeExpression*
( "+" *MultiplicativeExpression*
| "-" *MultiplicativeExpression* )*

MultiplicativeExpression ::= *UnaryExpression*
( "*" *UnaryExpression*
| "/" *UnaryExpression*
| "%" *UnaryExpression* )*

UnaryExpression ::= "!" *UnaryExpression*
| "-" *UnaryExpression*
| *PrimitiveExpression*

PrimitiveExpression ::= "(" *DesignRuleExpression* ")"
| *LiteralConstant* | *DesignAnalysisCall* | *Id*

Id ::= *Identifier* ( "." *Identifier* )*

DesignAnalysisCall ::= *Id* "(" *ActualParams* ")"

LiteralConstant ::= *IntegerLiteral* | *FloatingPointLiteral* | *StringLiteral*
| ***true*** | ***false***

ActualParams ::= ( *ActualParam* ( "," *ActualParam* )* )?

FormalParams ::= ( *FormalParam* ( "," *FormalParam* )* )?

ActualParam ::= *LiteralConstant* | *DesignAnalysisCall* | *Id*

FormalParam ::= *Identifier* ( "," *Identifier* )* ":"
(*Identifier* | ***Component*** | ***Connector*** | ***Port*** | ***Role***
| ***Int*** | ***Float*** | ***String*** | ***Boolean*** )

SetExpression ::= ( *SetReference* | *SetFunction* | *LiteralSet*
| *SetConstructor* )

SetReference ::= *Identifier* ( ( "." *Identifier* ) | ( "." *Components* )
| ( "." *Connectors* ) | ( "." *Ports* ) | ( "." *Roles* )
| ( "." *Representations* ) | ( "." *Properties* ) )+

SetFunction ::= ( ***Union*** | ***Intersection*** | ***Setdiff*** )
"(" *SetExpression* "," *SetExpression* ")"

LiteralSet ::= ( "{" "}" |
"{" ( *LiteralConstant* | *Id* ) ( "," ( *LiteralConstant* | *Id* ) )* "}")

SetConstructor ::= "{" ***Select*** *Identifier* ":" *lookup_arbitraryTypeByName* ***in***
*SetExpression* "|" *DesignRuleExpression* "}"