Adding Mappings to Acme

DRAFT

Jianing Hu
November, 1999

Abstract

Multiple architectural views of a system are often helpful in describing a system's design. While there are numerous languages for describing the contents of an individual *architectural view, to date those languages have not supported the description of relationships* between *views. In this report we describe an extension to Acme for expressing mappings between the structural aspects of architecture and for expressing constraints over those mappings.*

Introduction

Software architecture is the overall structure of a software system. In practice, software architectures are usually described in an informal and idiosyncratic way using box and line diagram with associated prose. Recognizing the need for more precise descriptions of software architecture, various Architecture Description Languages (ADLs) have been introduced. Examples of ADLs include Acme, Armani, Aesop, Adage, Meta-H, C2, Rapide, SADL, Unicon and Wright. Those ADLs are able to describe the structure of a software system in terms of components and connectors. Additionally, most of them support certain kinds of analysis over the topology and properties of the structure, such as design rule analysis supported by Armani [2] and architectural compatibility analysis and deadlock detection supported by Wright [6].

In practice, while describing a software architecture precisely is an important step towards making architecture design more rigorous, it often helps to have multiple views of the same system. Kruchten has proposed a model in which five concurrent views of the same software architecture capture different set of concerns [1]. The Meta-H system requires a hardware view that shows the hardware structure and a software view that shows the software application running on the hardware [5]. Multiple views could also be the result of system evolution, in which case different versions of the system are represented different views of the system.

In all of these cases, there are relationships between these views. Those relationships might show, for example, which part of the software architecture runs on which processor, or which part of one architectural design has the same functionality as that of an alternative design. Knowledge of those relationships helps designers to better understand the system and to reason about it. In this paper we discuss the problem of capturing relationships between different system views and present an approach of doing this.

In general, describing arbitrary relationships – or "mappings" – between architectural views is a hard problem, since each architectural view may have its own semantics. The capability of addressing arbitrary kinds of mappings requires understanding these properties of individual views, making a generic means of mapping describing difficult to define. The first step to attack the mapping problem, therefore, is to find the similarities of different architectural views and start with a mechanism that can describe mappings between such similar parts. Despite differences in semantics, most architecture descriptions have some means to describe architectural structures. In this paper we present an approach to describe structural mappings between different views.

When architecture descriptions are written in ADLs, it is natural to try to describe mappings between architectures in ADLs, too. Some ADLs have mechanisms for mapping between specific views (e.g., Meta-H provides its own mechanism for mapping between software and hardware views; SADL allows one to characterize mapping between refinement levels). However, each of these capabilities is specifically tied to the specific semantics of particular kinds of views and to the semantics of the ADL.

In this paper we propose a different approach. Instead of ADL specific views, we consider the general problem of mapping between architectural structures. Specifically, we show how to add an extension of structural mapping (and mapping types) to ACME, a generic language for characterizing architectural structures.

The goal is to permit the view specifier to identify classes of mapping, based on the types of structure and constructs and their relationships. Mappings between specific architecture descriptions are the instances of these classes and can be checked for constraining. This work can be viewed as a first step towards supporting multiple views in software architecture by focusing primarily on flexible relationships between structural elements.

Requirements

If we treat the problem of describing architectural views as one in which we extend existing ADLs with mapping constructs, then there are some natural requirements that we would like the extension to satisfy. These are:

- **Expressive power**; The extension should be able to describe a wide variety of mappings. In particular, in the context of structural mappings, our extension should allow mappings between arbitrary number and types of structural elements.
- **Support of mapping classification and reuse**. The extension should support classification of mappings. It should also allow the user to build a mapping on top of an existing mapping description, thus reuse the existing description.
- **Analyzability**; There are various kinds of constraints that the specifier might want a mapping to satisfy. The constraints could be a certain type that the mapping should satisfy, or relationships between view properties, etc. Our extension should support formal notations for such constraints, which can therefore be formally checked for validity.

Background of Acme

Acme is a simple, generic ADL that was designed to be an interchange format of architectural description. To support interchanging, ACME describes an architectural design as a hierarchical collection of interacting components, which is agreed by all ADLs. In particular, ACME uses seven constructs to specify an architecture – components, connectors, ports, roles, systems, representations, and rep-maps. Besides providing a common infrastructure for describing architectural structures, ACME also provides a flexible annotation mechanism supporting association of non-structural information using externally defined languages. One of such languages is Armani, a predicate language that can be used to describe invariants and heuristics. Armani's ability of declaring invariants and heuristics is especially useful for our extension, since it can be used to declare the constraints or consistency statement that we want mappings to satisfy.

There are several advantages using ACME as the basis of our extension. First, ACME was designed as an interchange language, therefore it focus on the essential issues of architectural description like structure. Also, as an interchange language, it is in some sense the least common denominator of a large family of ADLs. Therefore the extension of Acme can be effectively employed by other ADLs.

Overview of Approach

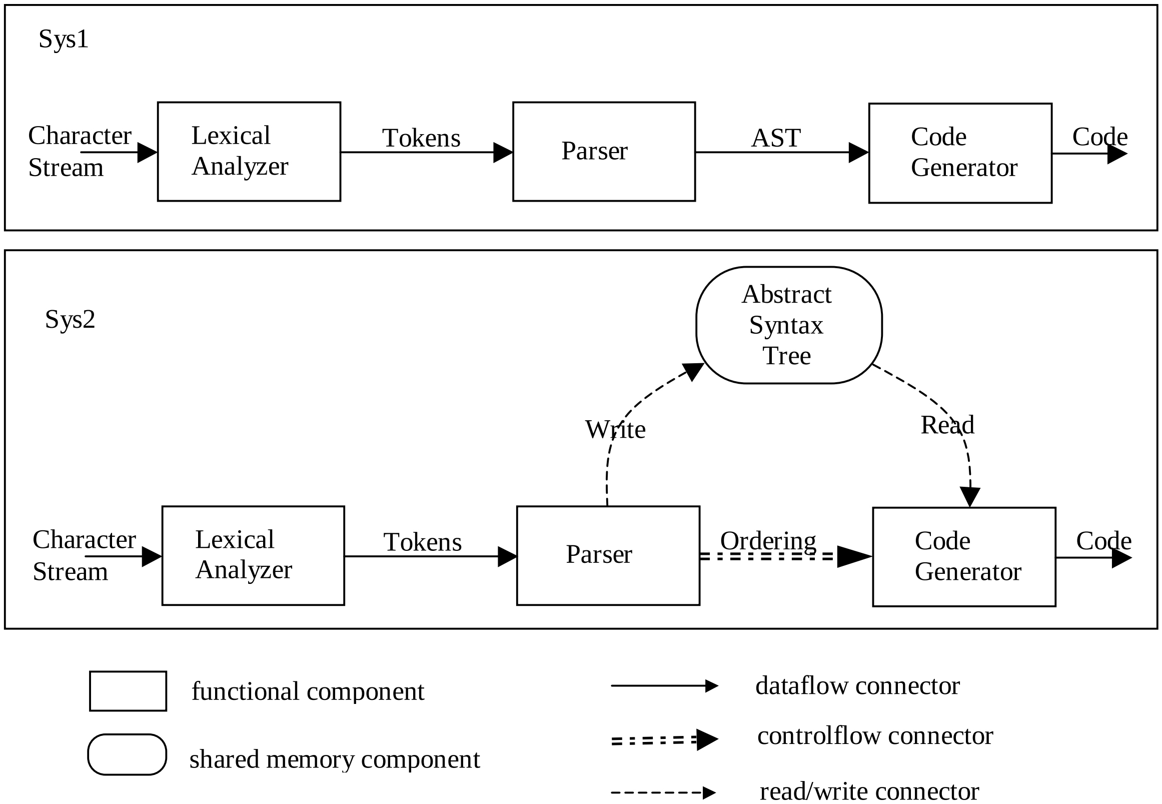

Figure1: Two architectures for a compiler

To illustrate our approach, consider two views of a design of the architecture for a compiler. An abstract pipe-and-filter architecture is depicted at the top of Figure 1[1] as *Sys1*. A more concrete (i.e., implementation-oriented) architecture that is a hybrid of pipe-and-filter and shared-memory styles is shown at the bottom of Figure 1 as *Sys2*. A complete declaration of *Sys1* and *Sys2* in Acme can be found in appendix A.

As a refinement of *Sys1*, *Sys2* has much in common with *Sys1*. For example, intuitively we would expect the functional components in *Sys2* to be the "same" as their corresponding components in Sys1. The similarity could include comparable functionality and equal throughput, etc.

Figure 2 shows how we might indicate this intuitive correspondence more formally as a mapping[2]. In this declaration "*same*" is the name of this mapping. In the brackets ("{}") is the mapping body, which consists of pairs of elements that are associated by this mapping. Each of these pairs is called a maplet[3].

Map same **with** {
    Sys1.analyzer **to** Sys2.analyzer;
    Sys1.parser **to** Sys2.parser;
    Sys1.generator **to** Sys2.generator;
  }

Figure 2

---

[1] The two architectures shown in Figure 1 are a simplified version of the example used in [7].
[2] In this paper, bold in declarations indicates keywords unless otherwise stated.
[3] As we will see later, a maplet could consist of more than just the pair of elements.

Figure 2 declares a mapping between functional components of *Sys1* and *Sys2*, with each maplet connecting two components that have the "same" functionality. It is not as straightforward, however, to declare a mapping between connectors of the two architectures on the same basis. Though it is not hard to find *Sys1.Token*'s functional correspondence in *Sys2*, there is no single component or connector in *Sys2* that takes the same responsibility as *Sys1.AST* does. *Sys1.AST*'s responsibility is to transmit data and control flow from *Sys1.Analyzer* to *Sys1.Generator*. In *Sys2*, the same responsibility is carried by *Sys2.write*, *Sys2.ast*, *Sys2.read*, and *Sys2.Ordering* together. To address the aggregation of these elements as a whole, we introduced compound types. There are three kinds of compound types, set, sequence, and record. A set is an unordered list of elements. A sequence is an ordered list of elements. A record is a tuple of elements, each of which has a name. In this case we can denote the aggregation of elements as a record. The mapping declaration is shown in figure 3.

```
Map pipe2channel with {
  Sys1.ast to [ from = Sys2.write,
                through = Sys2.ast,
                dest = Sys2.read
                control = Sys2.Ordering];
}
```

Figure 3

Sometimes mappings between structures might indicate mappings between sub-structures. For example, in Figure 1, the "*same*" mapping between *Sys1.analyzer* and *Sys2.analyzer* might indicate a certain relationship between the ports *Sys1.analyzer.output* and *Sys2.analyzer.output*. To support the declaration of such "sub relations", hereafter referred to as "nested mappings", we allow hierarchical mapping declarations. An example of nested mapping is shown in figure 4.

```
Map same with {
  Sys1.analyzer to Sys2.analyzer with {
    Map nested-port-map with {
      Sys1.analyzer.output to Sys2.analyzer.output;
    }
  }
}
```

Figure 4

Though we have shown several examples on how to declare mappings, we have done little to show how to express the "meaning" of mappings. For example, in the mapping declared in figure 2, we do not show the "meaning" of this mapping, i.e., in what sense the associated components are the "same". It merely states that there exists some kind of relationship between the declared pairs. Thought the name of the mapping exposes a vague picture of the meaning, it is far from enough for a reader to grasp exactly what this mapping means, let alone automated analysis. To give a formal declaration of the meaning of a mapping, we can add constraints to it. For example, the *same* mapping might declare that *Sys1.analyzer* and *Sys2.analyzer* have equal throughput, making it very clear what "same" means here. This equality declares a consistency statement that the abstract system *Sys1* and its realization *Sys2* have to agree with. Such constraints are common between abstract and concrete architectures. These constraints can be captured using predicates and be used as a guide in architecture refinement. Following the terminology of Armani, we call them *design rules*. In general, two associated architectures often have to agree on some constraints over their properties. These constraints can be captured by mappings in the same manner.

Figure 5 shows how we add constraints to declare equal-throughput constraints in mapping *same*. (The word "throughput" could has different meanings for different components. For the analyzer, it could denote how many characters the analyzer can process per second; for the parser, it could denote the number of tokens the parser consumes per second; for the generator, it could denote the amount of code generated per second. For simplicity, in our declaration we denote the throughput of each component as a float and make it a common property of all functional components. See Appendix A for details of family and system declarations.) We use the Armani predicate language to declare design rules. The use of formal notations in the declaration of design rules makes it possible to use an automated checker to check the consistency of mappings.

```
Map same with {
  Sys1.analyzer to Sys2.analyzer;
  Sys1.parser to Sys2.parser;
  Sys1.generator to Sys2.generator;
  Invariant Sys1.analyzer.throughput == Sys2.analyzer.throughput;
  Invariant Sys1.parser.throughput == Sys2.parser.throughput;
  Invariant Sys1.generator.throughput == Sys2.generator.throughput;
}
```

Figure 5

Mappings are classified by their properties, which includes (not exclusively) the type of the elements they are associating (referred to as *signature* of a mapping type), the constraints they cast on those elements, and their structures (e.g., containment of nested mappings). Figure 6 shows how to declare and instantiate a mapping type.

```
Mapping type SameT : PF.Filter <-> Hybrid.Filter = {
  Invariant source.throughput == target.throughput;
}

Map same : SameT with {
  Sys1.analyzer to Sys2.analyzer;
  Sys1.parser to Sys2.parser;
  Sys1.generator to Sys2.generator;
}
```

Figure 6

A mapping type declares constraints that all of its instances have to satisfy. In the example shown in figure 6, every instance of mapping type *SameT* has to a) associates instances of *PF.Filter* to instances of *Hybrid.Filter*, and b) satisfies the design rule declared in *SameT*. Note that in order to declare design rules in mapping types, we use place holders **source** and **target** to denote the associated elements in mapping instances. Therefore, the mapping instance declared in figure 6 is effectively the same as that declared in figure 5.

Figure 7 shows another example of mapping type delcaration in which the mapping type has a structural constraint on its instances, namely each maplet of the instance has to contain a nested mapping as declared in the mapping type. Therefore the mapping instance declared in Figure 7 is effectively the same as that declared in figure 4.

```
Mapping type SameT2 : PF.Filter <-> Hybrid.Filter = {
  Map nested-port-map with {
    source.output to target.output;
  }
}
```

```
Map same : SameT2 with {
  Sys1.analyzer to Sys2.analyzer;
}
```

Figure 7

One can refine an existing mapping type, adding constraints to it and/or refining its signature, thus reuse the existing mapping type. This is called sub-typeing. Figure 8 shows an example of sub-typing. More examples could be found in Appendix B. In Figure 8, *SameT3* extends *SameT* with a new constraint that the two Filters have to have the same number of ports.

```
Mapping type SameT : PF.Filter <-> Hybrid.Filter = {
  Invariant source.throughput == target.throughput;
}
Mapping type SameT3 : PF.Filter <-> Hybrid.Filter Extends SameT with {
  Invariant size(source.ports) == size(target.ports);
}
```

Figure 8

Mapping Syntax

In this section we give the formal syntax of mapping, according to the discussions in the last section.

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

Non-Terminals are specified with *italics*.

[...] indicates an optional production.

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

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

| Seperates alternative choices.

Map-Type-Decl ::= **Mapping type** *TypeId* : *Signature* "=" *Map-Type-Body* [";"] |
        Mapping type *TypeId* : *Signature* **Extends** *TypeId* (","*TypeId*)* **with** *Map-Type-Body* [";"]
Map-Type-Body ::= "{" (*Map-Type-Decl* |
                DesignRule[4] |
                Map_Instance_Decl)* "}"
Signature ::= *Type* "<->" *Type*
TypeId ::= *Identifier*
Map-Instance-Decl ::= **Map** *Identifier* [":" *TypeId* ("," *TypeId*)* [**Extended with** *Map-Type-Body*]](
                        (**with** *Map-Body-Decl* [";"])
                        | ";"
)
Map-Body-Decl ::= "{" (*Maplet* | *DesignRule*)* "}"
Maplet ::= *Element* **to** *Element* ((**with** "{" (*Map-Instance-Decl* | *DesignRule*)* "}"[";"]) | ";")

In the above syntax we do not explain what *Element* is. It is not necessarily a single element. As said in the last section, it could be an aggregation of elements. However, in Acme, such aggregation of elements is not addressed. We apply the following syntax to address the element aggregation issue in mapping extension. The reason that we do not present it together with the above core mapping syntax is that we believe this

[4] *DesignRule* is the same as the non-terminal defined in Armani syntax. Refer to [2] for details.

element aggregation syntax is not only useful in the mapping context but in a wider domain. We tend to integrate it into the core Acme syntax later.

Type ::= *Primitive-Type* | *Compound-Type* | *Identifier*
Primitive-Type ::= **Component** | **Connector** | **Port** | **Role** | **System**
Compound-Type-Decl ::=**Compound Type** *Identifier* "=" *Compound-Type* ";"
Compound-Type ::= *Record-Type* | *Set-Type* | *Sequence-Type*
Record-Type ::= **Record** "[" *Record-Item* ("," *Record-Item*)* "]"
Record-Item ::= *Identifier* ":" *Type*
Set-Type ::= **Set** "{" *Type* "}"
Sequence-Type ::= **Sequence** "<" *Type* ">"
Element ::= *Identifier* | *Compound-Element*
Compound-Element ::= *Set* | *Record* | *Sequence*
Set ::="{" *Identifier* (","*Identifier*)* "}"
Record ::= "[" *Identifier* "=" *Identifier* (","*Identifier* "=" *Identifier*)* "]"
Sequence ::= "<" *Identifier* (","*Identifier*)* ">"

Mapping types can be defined wherever element or family types can be defined. Map instances can be declared wherever element or system can be declared. Mapping types and instances have the same scoping rules as element types and instances, respectively. Specifically, a mapping type declared in the global scope is visible to all map instances in that global scope, but not to any mapping types in the same scope. A mapping type declared in another mapping type is visible to its parent mapping type. All mapping types visible to a parent mapping type are visible to its children mapping types. If a mapping type is visible to another mapping type, then it is also visible to the other mapping type's instances.

In appendix B we present more examples of mapping declaration.

Mapping Semantics

We present the semantics of the mapping extension using denotational semantics equations and canonical semantics representations similar to those used in [2] to describe Armani semantics. The canonical semantics representation is a collection of tuples that represent the structure of element. In this section we extend the element tuple used in [2] to accommodate the mapping constructs. In this section we use the word "element" in a different than in other parts of this paper. In this section "element" does not only refer to the structural constructs of Acme, it also refers to design spaces and mapping constructs, namely mapping instances, maplets and mapping types.

An element is a tuple of the form
e = (n, c, s, p, r, a, i, h, m, mt, t_elt, t_prop, t_map, t_super, t_asserted).
Informally :
n = name of the element.
c = category of the element.
s = set of elements that define e's substructure.
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.
m = set of mapping instances visible to e.
t_elt = set of element types visible to e.
t_prop = set of property types visible to e.
t_map = set of mapping types visible to e.

We also use t to denote t_elt $\cup$ t_prop $\cup$ t_map for short.
t_super = set of names of supertypes of this element type.
t_asserted = set of names types this element claims to satisfy.

An element does not have to have all these fields. For example a mapping instance does not have any representation.

A mapping instance m is an element tuple with the following additional fields
m = (…, t_real, l)
Informally :
t_real = the real mapping type of m's, which might not have a name.[5]
l = set of maplets declared in m.

A maplet is an element tuple with the following additional properties
ml = (…, sr, t)
Informally:
sr = the name of the source of ml.
t = the name of the target of ml.
source and target form the element pair that a maplet addresses.

A mapping type mt is an element tuple with the following additional fields
mt = (…, d, ra)
Informally :
d = domain type of mt.
ra = range type of mt.

We use denotational semantics equations to describe the translation of abstract syntax to its cananical semantics representation. Consider the denotational equation:
M[<Expression>] c = [c | c.x←4]
The above expression is read "The meaning of <Expression>, in the context c is the context c with "4" assigned to the tuple field x of context c. A context is simply a tuple whose values can be modified as a result of the evaluation of the syntactic expression.

Statement composition
Composition of declaration statements is handled by the following rule.
M[s1 s2 … sn]e = M[sn]…M[s2]M[s1]e

Mapping type semantics
M[Mapping type n : t1<->t2 x ; y] e = if (n ∉ Names(e.s)) and (n ∉ Names(e.p)) and (n ∉ Names(e.t))) then
M[y][e | M[x] [mt | mt.n←n, mt.d←t1, mt.ra←t2, mt.t←e.t, [6]
mt.m←e.m, mt.t_asserted←{Map}, mt.t_super←{Map}],
e.t_map←e.t_map∪ {mt}]
M[={x}]mt = M[x]mt
M[extends t1,…,tn with {x}]mt =
if ((t1∈mt.t_map) and … and (tn∈mt.t_map) and
(mt.d <:t1.d and mt.ra <:t1.ra) and … and (mt.d <:tn.d and mt.ra <:tn.ra)) then
M[x][mt | mt.t_super←mt.t_super∪ {t1,…,tn}∪ t1.t_super∪ …∪ tn.t_super,
mt.i←mt.i∪ t1.i∪ …∪ tn.i, mt.h←mt.h∪ t1.h∪ …∪ tn.h,
mt.t←mt.t∪t1.t∪ …∪ tn.t, mt.m←mt.m∪ t1.m∪…∪ tn.m[7]]
M[invariant x;] mt = [mt|mt.i←mt.i∪ {x}]

---

[5] It could be anonymous because we can extend a mapping type when instantiating it without giving the new type a name. (Actually we can't give the new type a name in this case.)
[6] "mt.t←e.t" is a shorthand for "mt.t_elt←e.t_elt, mt.t_prop←e.t_prop, mt.t_map←e.t_map"
[7] Here we assume that there is no name conflict between mapping instances declared in these mapping types. If there are conflicts, instances in conflict have to be unified using the unifyMap algorithm first.

M[heuristics x;] mt = [mt|mt.h $\leftarrow$ mt.h $\cup$ {x}]

Mapping instance semantics

M[Map n x ; y] e = if ((n $\notin$ Names(e.s)) and (n $\notin$ Names(e.p)) and (n $\notin$ Names(e.t))) then

if ((n $\notin$ Names(e.m)) then
M[y][e| M[x] [m | m.n $\leftarrow$ n, m.t_asserted $\leftarrow$ {Map}, m.t $\leftarrow$ e.t,
m.m $\leftarrow$ e.m] e.m $\leftarrow$ e.m $\cup$ {m}]
else
M[y][e | m1 $\leftarrow$ lookup(Names(e.m),n),
m $\leftarrow$ unifyMaps(m1, M[x] [m | m.n $\leftarrow$ n, m.t_asserted $\leftarrow$ {Map},
m.t $\leftarrow$ e.t, m.m $\leftarrow$ e.m], e.m $\leftarrow$ e.m $\cup$ {m}]

M[: t1, …, tn x] m = if (t1 $\in$ Names(m.t_map)) and … and (tn $\in$ Names(m.t_map)) then
M[x] [m | m.t_asserted $\leftarrow$ {t1,..,tn}, m]

M[extended with {x}y] m = M[y][m|m.t_real $\leftarrow$ M[x] m.t_real]

M[contains{x}]m = M[x]m

M[invariant x;] m = [m|m.i $\leftarrow$ m.i $\cup$ {x}]

M[heuristics x;] m = [m|m.h $\leftarrow$ m.h $\cup$ {x}]

M[x1 to x2 = {x}] m = if ((satisfiesType(x1,m.t_real.d) and (satisfiesType(x2,m.t_real.ra)) then
[m|m.l $\leftarrow$ m.l $\cup$ {ml|ml.sr $\leftarrow$ x1, ml.t $\leftarrow$ x2,
ml.i $\leftarrow$ substitute(x1, "source", x2, "target", m.t_real.i),
ml.h $\leftarrow$ substitute(x1, "source", x2, "target", m.t_real.h),M[x]ml}]

Alternative Mapping Semantics (in another notation)

In this section we describe the semantics of the mapping extension using evaluation semanticss notation.

The basic notation is like e ⇓ V, which means that declaration e evaluates to value V. A value is a design space, which is an element as defined in the previous section. In our notation each declaration is evaluated under an environment. An environment is a pair of a context and a stack. A context is an element as defined in the previous section. A stack, defined as the following, is used to keep trace of the evaluation steps.

Stack ::=∅ | Stack, Frame
Frame ::=[State, Element] | [Declaration, State, Element]
State ::=mapT• | mapT• | mapT typeId…typeId | mapEx•

Thus a notation that reads

C ; S ⊢ e ⇓ V

means that under the environment consisting of element C and stack S, declaration e evaluates to V.

In the following derivation rules, V denotes a value; e denotes an element; x and y denote arbitrary sequence of characters, with the restriction that when used with y, x must contain equal numbers of left and right parentheses (or brackets, etc.) and they have to be correctly paired; n denotes an Identifier which is either a mapping type or map instance name; t1...tn denotes type names, when the usage will not cause confusion, they also denote the types themselves; m denotes a map instance; mt denotes a mapping type; x1 and x2 denote element names.

1. ———————

   V ; ∅⊢∅⇓ V

2. e ; S ⊢ DesignRule y ⇓ A
   isInvariant(DesignRule)
   ————————————————————

   [e | e.i←e.i ∪ {DesignRule}]; S ⊢ y ⇓ A

3. e ; S ⊢ DesignRule y ⇓ A
   isHeuristic(DesignRule)
   ————————————————————

   [e | e.h←e.h ∪ {DesignRule}]; S ⊢ y ⇓ A

4. e ; S ⊢ **Mapping type** n : t1 <-> t2 x ⇓ A
   n ∉ Names(e.s)
   n ∉ Names(e.p)
   n ∉ Names(e.t)
   ————————————————————

   [mt | mt.n←n, mt.d←t1, mt.ra←t2, mt.t←e.t, mt.m ←e.m, mt.t_super ←{**Map**}];
   S , [mapT• ; e] ⊢ x ⇓ A

5. mt ; S, [mapT• ; e] ⊢ ={x} y ⇓ A
   ————————————————————

   mt ; S, [y,mapT• ; e] ⊢ x ⇓ A

6. $$\frac{mt ; S, [y, mapT\bullet ; e] \vdash \emptyset \Downarrow A}{[e \mid e.t\_map \leftarrow e.t\_map \cup \{mt\}] ; S \vdash y \Downarrow A}$$

7. $$\frac{\begin{array}{l} mt ; S, [mapT\bullet ; e] \vdash \textbf{Extends}\ t1, \ldots, tn\ \textbf{with}\ \{x\}\ y \Downarrow A \\ t1 \in e.t\_map \\ \ldots \\ tn \in e.t\_map \\ mt.d <: t1.d \text{ and } mt.ra <: t1.ra \\ \ldots \\ mt.d <: tn.d \text{ and } mt.ra <: tn.ra \end{array}}{\begin{array}{l} [mt \mid mt.t\_super \leftarrow mt.t\_super \cup \{t1,\ldots,tn\} \cup t1.t\_super \cup \ldots \cup tn.t\_super, \\ mt.i \leftarrow mt.i \cup t1.i \cup \ldots \cup tn.i, , mt.h \leftarrow mt.h \cup t1.h \cup \ldots \cup tn.h, \\ mt.t \leftarrow mt.t \cup t1.t \cup \ldots \cup tn.t] ; S, [y, mapT\ t1 \ldots tn, e] \vdash x \Downarrow A \end{array}}$$

8. $$\frac{mt ; S, [y, maptT\ t1 \ldots tn, e] \vdash \emptyset \Downarrow A}{[e \mid e.t\_map \leftarrow e.t\_map \cup unifyMapType(mt, t1, \ldots, tn)] ; S \vdash y \Downarrow A}$$

9. $$\frac{\begin{array}{l} e ; S \vdash \textbf{Map}\ n\ x \Downarrow A \\ n \notin Names(e.s) \\ n \notin Names(e.p) \\ n \notin Names(e.t) \end{array}}{[m \mid m.n \leftarrow n, m.t\_asserted \leftarrow \{\textbf{Map}\}, m.t \leftarrow e.t, m.m \leftarrow e.m] ; S, [map\bullet , e] \vdash x \Downarrow A}$$

10. $$\frac{\begin{array}{l} m ; S \vdash : t1, \ldots, tn\ x \Downarrow A \\ t1 \in Names(m.t\_map) \\ \ldots \\ tn \in Names(m.t\_map) \end{array}}{\begin{array}{l} [m \mid m.t\_asserted \leftarrow m.t\_asserted \cup \{t1,\ldots,tn\}, m.t\_real \leftarrow unifyMapTypes(m.t\_real,t1,\ldots,tn), \\ m.i \leftarrow m.i \cup t1.i \cup \ldots \cup tn.i, m.h \leftarrow m.h \cup t1.h \cup \ldots \cup tn.h, m.m \leftarrow m.m \cup t1.m \cup \ldots \cup tn.m] ; \\ S \vdash x \Downarrow A \end{array}}$$

11. $$\frac{m ; S \vdash \textbf{Extended with}\ \{x\}\ y \Downarrow A}{m.t\_real ; S, [y, mapEx\bullet , m] \vdash x \Downarrow A}$$

12. $$\frac{m1 ; S, [y, mapEx\bullet , m2] \vdash \emptyset \Downarrow A}{[m2 \mid m2.t\_real \leftarrow m1] ; S \vdash y \Downarrow A}$$

13. $m \, ; S, [map\bullet \, , e] \vdash \textbf{Contains} \; \{x\} \; y \Downarrow A$

$m \, ; S, [y, map\bullet, e] \vdash x \Downarrow A$

14. $m \, ; S \vdash x1 \; \textbf{to} \; x2; \, y \Downarrow A$
satisfiesType(x1, m.t_real.d)
satisfiesType(x2, m.t_real.ra)

[m | m.l←m.l ∪ {ml | ml.sr←x1, ml.t←x2, ml.i←substitute(x1, "source", x2, "target", m.t_real.i), ml.h←substitute(x1, "source", x2, "target", m.t_real.h), ml.t←m.t, ml.m←m.m}] ; $S \vdash y \Downarrow A$

15. $m \, ; S \vdash x1 \; \textbf{to} \; x2 = \{x\} \; y \Downarrow A$
satisfiesType(x1, m.t_real.d)
satisfiesType(x2, m.t_real.ra)

[ml | ml | ml.sr←x1, ml.t←x2, ml.i←substitute(x1, "source", x2, "target", m.t_real.i),
ml.h←substitute(x1, "source", x2, "target", m.t_real.h), ml.t←m.t, ml.m←m.m] ;
$S, [y, maplet, m] \vdash x \Downarrow A$

16. $m \, ; S, [y, map\bullet, e] \vdash \emptyset \Downarrow A$
$m.n \notin Names(e.m)$

$[e \mid e.m \leftarrow e.m \cup \{m\}] \, ; S \vdash y \Downarrow A$

17. $m \, ; S, [y, map\bullet, e] \vdash \emptyset \Downarrow A$
$m.n \in Names(e.m)$

[e | e.m←{unifyMaps(m,lookup(e.m, n))} ∪ {e.m - {lookup(e.m, n)}] ; $S \vdash y \Downarrow A$

18. $ml \, ; S, [y, maplet, m] \vdash \emptyset \Downarrow A$

$[m \mid m.l \leftarrow m.l \cup \{ml\}] \, ; S \vdash y \Downarrow A$

Algorithm for mapping unification

In this section we present the algorithm for mapping unification. This algorithm is used in the last two sections.

It takes two mapping instances, m1 and m2, as input. It returns the unification of m1 and m2 if it successfully unifies m1 and m2, throws an error otherwise.

```
Map unifyMaps(m1,m2){
   Map m = new Map();     // the return value
   // mapping to be unified have to have the same
   // name
   if (m1.name != m2.name){
      throw error;
   }
   m.name = m1.name;
   // Check the compatibility of types.
   // m2 has to be of a sub type of m1's
   if (m2.t_real <: m1.t_real){
      m.t_real = m2.t_real;
      m.t = m1.t + m2.t;
      m.i = m1.i + m2.i;
      m.h = m1.h + m2.h;
      m.l = mergeMaplets(m1.l,
               m2.l, m2.t_real.d, m2.t_real.ra);
   } else {
      throw error;
   }
   return m;
}

// This function merges two sets of maplets
// d and ra are the domain and range types that
// maplets in s1 have to satisfy.
set{Maplet} mergeMaplets(s1,s2,d,ra){
   s = new set of Maplet;
   Forall x in s1{
      // Find the corresponding maplet in s2.
      y = s2.lookupMaplet(x.sr, x.t);
      // Unify the corresponding maplets and add
      // the unified maplet to s.
      if (y != NULL) {
         // Unify maplets with the same
         // source/target pair.
         s.l += unifyMaplet(x,y);
      } else {
         // If no corresponding maplet of x is found
         // in s2, x will be added to s directly. We
         // have to check if x satisfies the signature
         if ((satisfiesType(x.sr,d) and
            satisfiesType(x.t,ra)){
            s.l += x;
         } else {
            throw error;
         }
      }
   }
   // Add maplets in s2 that do not have
   // corresponding maplet in s1 to s.
   Forall y in s2{
      if (s1.lookupMaplet(y.sr,y.t) ==
          NULL){
         s.l += y;
      }
   }
   return s;
}

Maplet unifyMaplets(ml1, ml2){
   Maplet ml = new Maplet();
   Forall x in ml1.m{
      if (ml2.lookup(x.name) == NULL){
         ml.m += x;
      } else {
         ml.m += (unifyMaps(x,
                     ml2.lookup(x.name));
      }
   }
   Forall y in ml2.m{
      if (ml1.lookup(y.name) == NULL){
         ml.m += y;
      }
   }
   return ml;
}
```

Reference:

[1] P. Kruchten, The 4+1 View Model of Architecture

[2] R. Monroe, Capturing Software Architecture Design Expertise with Armani, CMU-CS-98-163.

[3] R. Monroe, D. Garlan, D. Wile, Acme Straw Manual.

[4] B. Woo, Software and Hardware Architecture Mapping.

[5] Honeywell, MetaH Language and Tools
[6] R. Allen, D. Garlan, A Formal Basis for Architectural Connection
[7] M. Moriconi, X. qian, Correctness and Composition of Software Architectures

Appendix A: Family and System declaration

```
Family PF = {
  Role Type InputRole = {}
  Role Type OutputRole = {}
  Port Type InputPort = {}
  Port Type OutputPort = {}
  Component Type Filter = {
    Port input : InputPort;
    Port output : OutputPort;
    Property throughput : float;
  }
  Connector Type Pipe = {
    Roles {
      input : InputRole;
      output : OutputRole;
    }
  }
}

Family Hybrid Extends PF with {
  Role Type ReadRole = {}
  Role Type WriteRole = {}
  Port Type ReadPort = {}
  Port Type WritePort = {}
  Component Type ParserT Extends Filter
with {
    Port write : WritePort;
  }
  Component Type GeneratorT Extends Filter
with {
    Port read : ReadPort;
  }
  Component Type SharedData = {
    Ports {
      input : ReadPort;
      output : WritePort;
    }
  }
  Connector Type ReadWrite = {
    Roles {
      read : ReadRole;
      write : WriteRole;
    }
  }
}

System Sys1 : PF = {
  Component analyzer : Filter = new Filter
     extended with {property throughput : float
         = 500;};
  Component parser : Filter = new Filter
     extended with {property throughput : float
         = 200;};
  Component generator : Filter = new Filter
     extended with {property throughput : float
         = 100;};
  Connector tokens : Pipe = new Pipe;
  Connector ast : Pipe = new Pipe;
  Attachments{
    analyzer.output to tokens.input;
    parser.input to tokens.output;
    parser.output to ast.input;
    generator.input to ast.output;
  }
}

System Sys2 : Hybrid = {
  Component analyzer : Filter = new Filter
     extended with {property throughput : float
         = 500;};
  Component parser : ParserT = new ParserT
     extended with {property throughput : float
         = 200;};
  Component generator : GeneratorT = new
GeneratorT extended with {property throughput :
  float = 100;};
  Component ast : SharedData = new
SharedData;
  Connector read : ReadWrite = new
ReadWrite;
  Connector write : ReadWrite = new
ReadWrite;
  Connector tokens : Pipe = new Pipe;
  Connector ordering : Pipe = new Pipe;
  Attachments {
    analyzer.output to tokens.input;
    parser.input to tokens.output;
    parser.output to ordering.input;
    generator.input to ordering.output;
    parser.write to write.read;
    ast.input to write.write;
    ast.output to read.read;
    generator.read to read.write;
  }
}
```

Appendix B: More Examples

Examples in this section are based on the declarations in appendix A unless otherwise mentioned.

Example 1 extending a mapping type

We can add to the constraints of a mapping type by sub-typing. Sub-types of existing mapping types are declared using the **extends** ... **with** key words. Examples follow.

Example 1.1 combining existing mapping types

We can get a new mapping type by combining existing mapping types. The resulting mapping type have all constraints that each parent type has. In this example we combine the mapping type *SameT* declared in figure 6 and *SameT2* declared in figure 7. For convenience we re-declare the original types in figure 6 and figure 7 below, respectively.

```
  Mapping type SameT : PF.Filter <-> Hybrid.Filter = {
    Invariant source.throughput == target.throughput;
  }
…
```

Figure 6

```
  Mapping type SameT2 : PF.Filter <-> Hybrid.Filter = {
    Map nested-port-map with {
      source.output to target.output;
    }
  }
```

Figure 7

Assume both *SameT* and *SameT2* are declared and visible in the current scope, one can combine them as shown in figure 9. The resulting mapping type has both constraints declared in *SameT* and *SameT2*.

```
  Mapping type SameT4 : PF.Filter <-> Hybrid.Filter Extends SameT, SameT2 with  {
  }
```

Figure 9

E.g. 1.2 accommodating new element types

When we sub-type element types, we may introduce new sub-structures, properties and new type definitions. We can sub-type mapping types that associate the old element types to address these new features of the new element types. For example, if we extend the families as the following:

```
Family PF2 extends PF with = {
  Component Type Filter2 extends Filter with {
    Property OS : string;
  }
}

Family Hybrid2 extends Hybrid with {
  Connector Type Processor2 extends Processor with {
    Property OS : string;
  }
}
```

Figure 10

and declare *Sys1* and *Sys2* as of *PF2* and *Hybrid2*, respectively, and change the analyzers in *Sys1* and *Sys2* to be of type *Filter2*, respectively, we can extend the mapping type of *SameT* to accommodate constraints on the new property *OS*.

```
Mapping type SameT_OS : PF2.Filter2 <-> Hybrid2.Filter2 Extends SameT with {
    Invariant source.OS == target.OS;
}
```

Figure 11

In figure 11 we changed the signature of the extended mapping type. The general rule for changing the signature of an extended mapping type is: if you declare "**mapping type** foo: typeA2<->typeB2 **extends** bar...", where bar's signature is "typeA1<->typeB1", then *typeA2* must satisfy *typeA1* and *typeB2* must satisfy *typeB1*. By keeping this rule satisfied we guarantee that whatever type visible in the declaration of bar is visible in the declaration of foo, which is essential to the correctness of foo since foo inherits all constraints and type declarations that bar has. In this particular example, *PF2.Filter2* satisfies *PF.Filter* and *Hybrid2.Filter2* satisfies *Hybrid.Filter*. Therefore the rule is not violated when extending *SameT* to *SameT2*.

Example 2. Mapping Instance Extension

One might want to refine a mapping instance defined in a mapping type in its sub types. This is done by re-declaring the map instance in the sub types and unifying the two map instance declarations using the unifyMaps algorithm. The following examples show how it is done. In this sub-section, without further notice, a capital letter followed by a number, e.g. *T1*, denotes a type name. A type name followed by a ', e.g. *T1'*, denotes a sub type name.

Example 2.1

```
MapType T1 : F1<->F2 = {
  MapType T2 : F1.C1 <-> F2.C1 = {...}
  Map inst : T2 with {
    source.C to target.C;
  }
}

MapType T1' : F1<->F2 extends T1 with {
  Map inst : T2 with {
    source.S to target.S;
  }
}
```

Figure 12

T1' extends *T1* with a refined instance *inst* of type *T2*. *T1'.inst* is refined in that it has one more maplet than *T1.inst*. Note that though *T1'.inst* did not re-declare the maplet declared in *T1.inst*. It was regarded as being implicitly re-declared since a refined mapping instance has to consist of equal or more maplets than the one it refines, otherwise it is an error. Therefore the declaration of T1' above is equal to the following declaration (note the underscored).

```
MapType T1' : F1<->F2 extends T1 with {
  Map inst : T2 with {
    source.C to target.C;
    source.S to target.S;
  }
}
```

Figure 13

This implicit re-declaration of maplets might raise some interesting issues, which will be discussed in example 2.4.
A refined mapping instance does not necessarily have to have more maplets than the one it refines, though. It could rather refine the existing maplets, as shown in the following example.

Example 2.2

```
MapType T1 : F1<->F2 = {
  MapType T2 : F1.C1 <-> F2.C1 = {
    MapType T3 : F1.analyzer <-> F2.analyzer = {...}
    ...
  }
  Map inst : T2 with {
    source.C to target.C with {
      Map portmap : T3 with {
        source.p1 to target.p1;
      }
    }
  }
}

MapType T1' : F1<->F2 extends T1 with {
  Map inst : T2 with {
    source.C to target.C with {
      Map portmap : T3 with {
        source.p2 to target.p2;
      }
    }
  }
}
```

Figure 14

In Figure 14, *T1'.inst* refined *T1.inst* by refining a maplet declared in *T1.inst*. Specifically, it refined that maplet by refining another mapping instance declared in that maplet. One can also refine a maplet by declaring new map instances in it, or using a combination of the two. By our implicit re-declaration rule, the above T1' is equal to the following (note the underscored).

```
MapType T1‘ : F1<->F2 extends T1 with {
  Map inst : T2 with {
    source.C to target.C with {
      Map portmap : T3 with {
        source.p1 to target.p1;
        source.p2 to target.p2;
      }
    }
  }
}
```

Figure 15

Example 2.3

A mapping instance can also be refined by only changing its type, as the following.

```
MapType T1 : F1<->F2 = {
  MapType T2 : F1.C1 <-> F2.C1 = {
    MapType T3 : F1.analyzer <-> F2.analyzer = {...}
    ...
  }
  Map inst : T2 with {
    source.C to target.C;
  }
}
MapType T1‘ : F1<->F2 extends T1 with {
  Mapping type T2‘ : F1.C1 <-> F2.C1 extends T2
        with{...}
  Map inst : T2‘;
}
```

Figure 16

Due to the implicit re-declaration rule, it is equal to (note the underscored)

```
MapType T1‘ : F1<->F2 extends T1 with {
  Mapping type T2‘ : F1.C1 <-> F2.C1 extends T2
        with{...}
  Map inst : T2‘ with {
    source.C to target.C;
  }
}
```

Figure 17

Example 2.4

Because of the implicit re-declaration rule, one has to be careful when refining a mapping instance. Sometimes a seemingly OK refinement could be actually wrong, like the following. To make it clearer, we enclose the family declarations here.

```
Family F1 = {
  Component Type C1 = {...}
  Component C : C1 = new C1;
}

Family F1‘ extends F1 with {
  Component Type C1 extends C1 with {...}
  Component S : C1‘ = new C1‘;
}

Family F2 = {
  Component Type C1 = {...}
  Component C : C1 = new C1;
}

MapType T1 : F1<->F2 = {
  MapType T2 : F1.C1 <-> F2.C1 = {...}
  Map inst : T2 with {
    source.C to target.C;
  }
}

MapType T1‘ : F1‘<->F2‘ extends T1 with {
  Mapping type T2‘ : F1‘.C1‘ <-> F2‘.C1
        extends T2 with{...}
  Map inst : T2‘ with {
    source.S to target.C;
  }
}
```

Figure 18

Though the declaration of *T1‘* might seem to be correct, it is actually wrong. After we apply the implicit re-declaration rule to it, it becomes (note the underscored)

```
MapType T1‘ : F1‘<->F2‘ extends T1 with {
  Mapping type T2‘ : F1‘.C1‘ <-> F2‘.C1
        extends T2 with{...}
  Map inst : T2‘ with {
    source.C to target.C;
    source.S to target.C;
  }
}
```

Figure 19

The underscored line contains an error because *source.C*, which is of type *F1.C1*, does not satisfy the signature of mapping type *T2‘*, which requires a component of type *F1‘.C1‘*.

Example 3. Use mappings to declare attachments

Consider the mapping type declared in figure 21.

```
Mapping type Attachments : Port<->Role = {

  invariant parent(parent(source)) == parent(parent(target));

}
```

Figure 20

This mapping type definition defined a mapping type for attachments. Its constraints says that the component in which the port is declared and the connector in which the role is declared must be in the same system. This constraint captures the semantics of attachments in Acme. The advantage of using mapping to declare attachments is that we can extend the basic attachments type to enrich its semantics. For example we could have a mapping type that checks the compatibility of the port and role, like the following:

Port Type Pport = {

  property protocol : Wright;

}

Role Type Prole = {

  property protocol : Wright;

}

Mapping type Attachments-Check-Compatibility : Pport->Prole **extends** Attachments **with** {

  invariant compatible(source.protocol, dest.protocol);

}

Figure 21

Assuming we already have an external analysis named "compatible" that checks the compatibility of two protocols. Now we can declare the attachments of a simple client and server system like the following:

...

Map attach: Attachments-Check-Compatibility **with** {

  client.send-request **to** rpc.caller;

  server.receive-request **to** rpc.callee;

}

...

Figure 22

It looks quite similar to the attachments declaration in Acme. When the user invokes the analyzer to check the mappings, it will check if the port and role attached together are compatible. Here the richer semantics of mappings makes it a more useful tool than the simple attachments declaration.

Example 4. Use mapping to declare bindings

We can also use mappings to declare bindings. The mapping type declaration follows.

Mapping type binding : **Element<->Element** = {

  invariant (satisfiesType(**source**, **Port**) **and** satisfiesType(**target**, **Port**)) **or**

    (satisfiesType(**source**, **Role**) **and** satisfiesType(**target**, **Role**));

  invariant contains(parent(**target**).Representations, parent(parent(parent(**source**))));

}

Figure 23

The first constraint states that binding can only be declared between two ports or two roles. The second constraint states that the source element is declared in a representation of the parent of the dest element. We can also extend the basic binding mapping and enrich its semantics as we did with the attachments mapping.