Adding Maps to Acme

DRAFT

Jianing Hu
August 2000

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 maps between the structural aspects of architecture and for expressing constraints over those maps.*

1. Introduction

A system's software architecture defines its overall structure. In practice, software architectures are usually described in an informal and idiosyncratic way, using box and line diagrams 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 [10, 5, 2, 4, 3, 8, 13, 12, 14, 1]. These 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 [10] and architectural compatibility analysis and deadlock detection supported by Wright [1].

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. For example, Kruchten proposes a model in which five concurrent views of the same software architecture capture different set of concerns [1]. The Meta-H notation requires a hardware view that shows the hardware structure and a software v7ew that shows the software application running on the hardware [2]. Multiple views could also be the result of system evolution, in which case different versions of the system may be represented using different views.

In all of these cases, there are important 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 for doing this.

In general, describing arbitrary relationships – or "maps" – between architectural views is a hard problem, since each architectural view may have its own semantics. The capability of addressing arbitrary kinds of maps requires a deep understanding of the properties of individual views, making a generic means of mapping between views difficult to define. Nonetheless, despite the differences in semantics, most views describe their structures using common structural notations. Medvidovic and Taylor argue that all architectural descriptions must explicitly specify their components, connectors, and architectural configurations[9]. Their survey show that all ADLs provide means for structural specification. The universal use of structural notations makes it possible to describe maps between structural elements of most views.

Furthermore, besides the universal usage of structural notations, many ADLs carry semantic constructs with structural descriptions. For example, Acme and Armani use properties that are attached to structural constructs to carry non-structural information and semantics; Wright attaches protocols to components and connectors to describe their behavior; Armani also uses predicates to impose constraints on structural constructs [10, 5, 6, 1]. A structural map description therefore provides a solid basis for describing more sophisticated maps that address those semantic constructs. In this paper we present an approach for

describing structural maps between different views. We also show how structural maps can capture some relationships between the semantic constructs.

When architecture descriptions are written in ADLs, it is also natural to try to describe maps between architectures using an ADL. Some ADLs have mechanisms for mapping between specific views. For example, Meta-H provides its own mechanism for mapping between software and hardware views [2]; SADL allows one to characterize mapping between refinement levels [12]. However, each of these capabilities is tied to the specific semantics of particular kinds of views and to the semantics of a particular 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 maps (and map types) to Acme, a generic language for characterizing architectural structures. The goal is to permit the view specifier to identify classes of maps, based on the types of structure and constructs, and their relationships. Map types impose constraints on their instances. Maps between specific architectural descriptions are the instances of these classes, and they have to satisfy the constraints imposed by their types. 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.

In Section 2 we go over the requirements of a mapping extension. Section 3 gives an overview of Acme and why we choose Acme as the platform of our extension. We give an overview of the mapping notation using some couple of simple examples in Section 4. The formal syntax of the mapping extension is given in Section 5. Section 6 uses more examples to illustrate further aspects of the mapping extension. Section 7 briefly presents our implementation of mapping tools and points out future research directions.

2. 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 such an extension to satisfy. These are:

- **Expressive power.** The extension should be able to describe a wide variety of maps. In particular, in the context of structural maps, our extension should allow maps between arbitrary number and types of structural elements.
- **Map classification support and reusability.** The extension should support classification of maps. It should also allow the user to build a map by extending an existing map description, thus reusing the existing description.
- **Analyzability.** There are various kinds of constraints that the specifier might want a map class or instance to satisfy. The constraints could be on all instances of a map class or on a single map instance. Our extension should support formal notations for such constraints. It should also support formal analysis on these constraints.

3. Background of Acme

Acme is a simple, generic ADL. As a second-generation ADL, Acme presents a simple but comprehensive core set of concepts for defining system structures. In particular, Acme uses seven constructs to specify an architecture – components, connectors, ports, roles, systems, representations and rep-maps. In contrast to some first-generation ADLs, Acme treats components as first-class entities and provides all the architectural modeling features mentioned in Medvidovic and Taylor's ADL comparison framework [9]. Acme's powerful structural modeling ability makes it a good base for our structural mapping extension.

Besides providing a common infrastructure for describing architectural structures, Acme also provides a flexible annotation mechanism supporting association of non-structural information using virtually any externally-defined languages. Extra-structural information can be annotated in other languages and be attached to structural constructs as properties. The ability to describe properties makes it possible to specify any auxiliary information of a structure. A mapping extension built on top of Acme can use this ability to carry non-structural information or constraints with maps.

Furthermore, Acme is an interchange language. That is, architectural descriptions written in other ADLs can be encoded in Acme. Hence our mapping extension to Acme is not ADL-specific. Any ADL can use our mechanism by translating their descriptions to Acme first.

The designers of Acme claim that Acme plays the following key roles in the software architecture community. [1] One role is as a basis for new architecture design and analysis tools, the other is as the starting point of new domain-specific ADLs[6]. These two roles imply that Acme is a good base for new architectural description mechanisms and our experience of extending it with the mapping mechanism supports these claims.

4. Overview of Approach

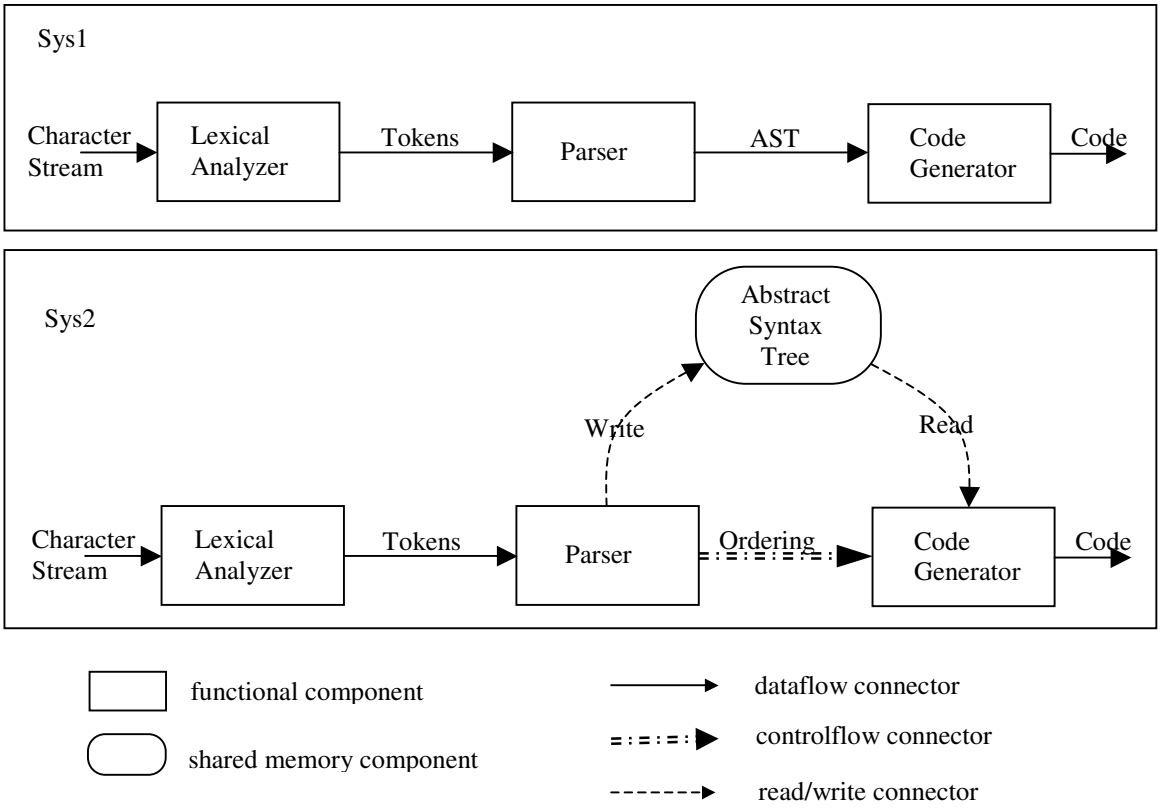

Figure1: Two architectures for a compiler

In this section we give an informal overview of our approach to describing maps between views. A formal syntax of our notation is given in Section 5. Formal semantics of our notation is given in Appendix B and C. More examples are given in Section 6.

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[2] 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*. Complete declarations of *Sys1* and *Sys2* in Acme can be found in Appendix A.

[1] Garlan et. al. claim in [4] that Acme plays four key roles in the software architecture community. Here we list only the two that are relevant to the mapping extension.
[2] The two architectures shown in Figure 1 are simplified versions of the example used in [7].

4.1 Basic Maps

As a refinement of *Sys1*, *Sys2* has much in common with *Sys1*. For example, intuitively we would expect the data-processing components in *Sys2* to be "similar" to their corresponding components in Sys1. The similarity could include similar functionality and performance, etc.

Figure 2 shows how we might indicate this intuitive correspondence more formally as a map. [3] In this declaration "*comp_map*" is the name of this map. In the brackets ("{}") is the map body, which consists of pairs of elements that are associated by this map. Each of these pairs is called a maplet. [4]

```
Map comp_map with {
    Sys1.analyzer to Sys2.analyzer;
    Sys1.parser to Sys2.parser;
    Sys1.generator to Sys2.generator;
  }
```

Figure 2. Declaring a map between components of Sys1 and Sys2

4.2 Map between Element Aggregations

Figure 2 declares a map between data-processing components of *Sys1* and *Sys2*, with each maplet associating two components that directly correspond in each architecture. It is not as straightforward, however, to declare a map between the two architectures based on the similarity of connectors. Though it is not hard to find *Sys1.Token*'s functional correspondence in *Sys2*, there is no single connector or component in *Sys2* that takes the same responsibility as *Sys1.AST* does in *Sys1*. *Sys1.AST*'s responsibility is to transmit data and control flow from *Sys1.Analyzer* to *Sys1.Generator*. In *Sys2*, the same responsibility is carried out collectively by *Sys2.write, Sys2.ast, Sys2.read,* and *Sys2.Ordering*.

To address the aggregation of these elements as a whole, we introduce compound elements. A compound element is an aggregation of other elements, which could be compound elements themselves. There are three kinds of compound elements: 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 denote the aggregation of elements in Sys2 as a record. The map 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. An example of a compound element map

4.3 Map with Sub-Structures

Sometimes maps between structures might indicate maps between substructures. For example, in Figure 1, the "*comp_map*" map 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 maps", we allow hierarchical map declarations. An example of a nested map is shown in Figure 4.

[3] In this paper, bold in declarations indicates keywords unless otherwise stated.
[4] As we will see later, a maplet could consist of more than just a pair of elements.

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

Figure 4

4.4 Map with Constraints

Though we have illustrated several examples of how to declare maps, in each description we merely state that there exists some kind of relationship between the declared pairs but do not describe any semantic relations. One way to elaborate on more detailed aspects of a map is to impose constraints on it. A constraint is a predicate over the structural and extra-structural attributes of the associated elements. The extra-structural attributes are often captured by properties. A constraint can either be an invariant or a heuristic. Both invariants and heuristics are predicates, the difference being that an invariant is a predicate that must hold all the time, while a heuristic represents a "rule of thumb" that is expected but not required to hold. Following the terminology of Armani, we call them *design rules* [10].

Since properties capture the extra-structural information of a structural construct, using constraints in a map declaration provides a means to address extra-structural aspects of the map. For example, the *comp_map* map might declare a constraint that *Sys1.analyzer* and *Sys2.analyzer* have equal throughput. This equality declares a consistency statement that the abstract system *Sys1* and its realization in *Sys2* have to agree on. Constraints are declared using the Armani predicate language, which is based on first order predicate logic (FOPL) [10]. Figure 5 shows how we might declare equal-throughput constraints in map *comp_map*. (The word "throughput" could have 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 data-processing components. See Appendix A for details of the family and system declarations.)

```
Map comp_map 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. Adding invariants to map declarations

4.5 Map Types

In this section we illustrate how map types are declared and instantiated. Map types play two useful roles in the mapping extension: (a) they capture the common properties of a class of maps, hence facilitating classification and reuse of map declarations, and (b) they support a powerful form of analysis, which is type checking. A map type is interpreted as a predicate. The predicate denoted by a map type is the conjunction of its structural and extra-structural constraints. The structural constraints address the structural aspects of a map type, for example the type of elements it associates (referred to as the *signature* of the map type) and

the minimal structure (e.g., nested maps) that instances of the map type must have. Extra-structural constraints address the extra-structural aspects of a map type and are usually declared in the form of design rules. Figure 6 illustrates how to declare and instantiate a map type.

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

Map comp_map : Comp_mapT with {
   Sys1.analyzer to Sys2.analyzer;
   Sys1.parser to Sys2.parser;
   Sys1.generator to Sys2.generator;
}
```

Figure 6. Declaring a map type

Declaring a map instance to be of a map type is the same as asserting that it satisfies the predicate denoted by the type. In the example shown in Figure 6, *comp_map* is declared to be an instance of *Comp_mapT*. Therefore the following assertions must be true: (a) *Sys1.analyzer, Sys1.parser* and *Sys1.generator* must be instances of component type *PF.Filter; Sys2.analyzer, Sys2.parser* and *Sys2.generator* must be instances of component type *Hybrid.Filter*, and (b) *Sys1.analyzer.throughput == Sys2.analyzer.throughput, Sys1.parser.throughput == Sys2.parser.throughput,* and *Sys1.generator.throughput == Sys2.generator.throughput*. Here (a) is the structural constraint imposed by the map type, and (b) is the extra-structural constraint.

We need to explain the keywords **source** and **target** used in Figure 6. When declaring a map type, a specifier does not know what elements the instances of this type are going to associate. Placeholders have to be used to address the associated elements in the type declaration in order to declare design rules over the associated elements. We use the placeholder **source** to denote the first element in a maplet, and **target** to denote the second element. Therefore, the map instance declared in Figure 6 is effectively the same as that declared in Figure 5.

Figure 7 shows another example of a map type declaration in which the map type imposes more structural constraints on its instances, namely, that each maplet of the instance must contain a nested map as declared in the map type. Therefore the map instance declared in Figure 7 is effectively the same as that declared in figure 4.

```
Map type Comp_mapT2 : PF.Filter <-> Hybrid.Filter = {
   Map nested-port-map with {
      source.output to target.output;
   }
}

Map comp_map : Comp_mapT2 with {
   Sys1.analyzer to Sys2.analyzer;
}
```

Figure 7. A map type containing a nested map

One can extend an existing map type by adding constraints to it and/or refining its signature. Figure 8 shows an example of map type extension. More examples can be found in Section 6. In Figure 8, *Comp_mapT3* extends *Comp_mapT* with a new constraint that the two filters must have the same number of ports.

```
Map type Comp_mapT : PF.Filter <-> Hybrid.Filter = {
```

```
    Invariant source.throughput == target.throughput;
  }
  Map type Comp_mapT3 : PF.Filter <-> Hybrid.Filter Extends Comp_mapT with {
    Invariant size(source.ports) == size(target.ports);
  }
```

Figure 8. Extending map types

5. Mapping Syntax

In this section we give the formal syntax of the mapping extension to Acme. We will use the following notational conventions:

- Keywords are specified with **bold text**. Keywords are case-insensitive.
- Non-Terminals are specified with *italics*.
- [...] Indicates an optional production.
- (...)* A sequence of zero or more elements.
- (...)+ A sequence of one or more elements.
- | Seperates alternative choices.

Map-Type-Decl ::= **Map type** *TypeId* : *Signature* "=" *Map-Type-Body* [";"] |
Map type *TypeId* : *Signature* **Extends** *TypeId* ("," *TypeId*)* **with** *Map-Type-Body* [";"]
Map-Type-Body ::= "{"
(
Map-Type-Decl |
DesignRule |
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 *DesignRule* is the same as the non-terminal defined in Armani syntax [10]. We do not elaborate on the above syntax what the definition of element is. An element may be a structural element such as a component or a connector, or may itself be an aggregation of other elements. The syntax of element and element type is given below. [5]

[5] In Acme, aggregation of elements is not addressed. The reason that we do not present the element and element type syntax together with the core mapping syntax is that we believe the compound element is a useful construct not only in the mapping domain but also in the wider domain of software architecture description. We intend 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*)* ">"

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

In Section 6 we present more examples of map declarations.

6. Additional Map Examples

In this section we give more examples of how our notation are used in describing maps to give the reader a better understanding of what the mapping extension is capable of. Examples in this section use the declarations in Appendix A unless otherwise indicated.

6.1. Extending a Map Type

In Section 4 we illustrate how a map type can be extended to impose new constraints. The ability to extend map types allows a hierarchical structure of map types and reuse of map type declarations. To extend a map type is to conjoin new predicates with those that are already in the type. Predicates added to a map type could be either structural or extra-structural. They could constrain the type of elements that instances of this map type can associate, or substructures that instances must have, or design rules that instances must obey. In this section we give some examples of some other means to declare a new map type from existing ones.

Example 6.1.1 Combining existing map types

We can obtain a new map type by combining existing map types. The resulting map type has all the constraints of each super type. In this example we combine the map type *Comp_mapT* declared in Figure 6 and *Comp_mapT2* declared in Figure 7 to produce a new map type *Comp_mapT4*. *Comp_mapT4* has both the design rule declared in *Comp_mapT*, and the structural constraint (nested-map) declared in *Comp_mapT2*. For convenience we re-declare the original types declared in Figures 6 and 7.

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

Map type Comp_mapT2 : PF.Filter <-> Hybrid.Filter = {
  Map nested-port-map with {
    source.output to target.output;
  }
}
```

```
Map type Comp_mapT4 : PF.Filter <-> Hybrid.Filter Extends Comp_mapT, Comp_mapT2 with  {
}
```

Figure 9. Constructing a new map type from multiple map types

Comp_mapT4, though it declares no new predicates, is the conjunction of both *Comp_mapT* and *Comp_mapT2*'s predicates. Therefore an instance of *Comp_mapT4* must satisfy: (a) it associates instances of *PF.Filter* to instances of *Hybrid.Filter*, (b) the associated components in a maplet have the same throughput, and (c) the output ports of associated components in a maplet are associated by a nested map named *nested-port-map*.

Example 6.1.2 Accommodating new element types

When we extend element types, we may introduce new substructures, properties and new type definitions. We can extend map types that associate the old element types to address these new features of the new element types. For example, in Figure 10 we extend the families PF and Hybrid used in previous examples to add a new property, OS, to component types *Filter* and *Processor*.

```
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. Extending families

Now we declare a map type *Comp_mapT_OS* that associates *PF2.Filter2* and *Hybrid2.Filter2*. We want this map type to have all the predicates declared in *Comp_mapT*, and additionally we want to add a predicate that associated components should be running on the same OS (i.e., their *OS* properties are the same). Instead of declaring *Comp_mapT_OS* from the ground, we can extend map type *Comp_mapT* to accommodate the constraint over the new property *OS*.

```
Map Type Comp_mapT_OS : PF2.Filter2 <-> Hybrid2.Filter2 Extends Comp_mapT with {
     Invariant source.OS == target.OS;
}
```

Figure 11. Extending map types to accommodate the extended families

Comp_mapT_OS as described in Figure 11 extends *Comp_mapT* with a new invariant. Hence *Comp_mapT_OS* has both the invarint over filter *throughput* declared in *Comp_mapT*, and the new invariant over *OS.* Note that *Comp_mapT_OS* has a different signature to that of *Comp_mapT_OS*'s. Changing the signature of a map type when extending it is not always allowed. The signature of an extended map type is covariant with its super types. This means that if we have the following map type extension:

Map Type m1: T1<->T2 …

Map Type m2: S1<->S2 **Extends** m1 …

Then *S1* must be a subtype of *T1* and *S2* must be a subtype of *T2*. By keeping this rule satisfied we guarantee that whatever type is visible in the declaration of *m1* is also visible in the declaration of *m2*, which is essential to the correctness of *m2* since *m2* inherits all constraints and type declarations that *m1*

has. In the example in Figure 11, *PF2.Filter2* satisfies *PF.Filter* and *Hybrid2.Filter2* satisfies *Hybrid.Filter*. Therefore the rule is not violated when extending *Comp_mapT* to *Comp_mapT_OS*.

6.2. Extending a Map Instance

When extending or instantiating a map type, one might want to refine a map instance defined in the original map type to add more constraints. This can be done by re-declaring the map instance with the new constraints and unifying the two map instance declarations using the *unifyMaps* algorithm given in Appendix D. What the *unifyMaps* algorithm does is check the compatibility of the old and new map instance declarations and if they are compatible, conjoin their constraints. The following examples show how map instances are extended. In this subsection, a capital letter followed by a number, e.g. *T1*, denotes a type. A type name followed by a ` denotes a subtype, e.g. *T1`* denotes a subtype of *T1*.

Example 6.2.1 Extend a map instance with new maplets

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

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

Figure 12. Extend a map instance with a new maplet

T1`* extends *T1* with a refined instance *inst* of type *T2*. *T1`.inst 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 map instance has to carry all constraints that the one it refines carries. This is called the *implicit re-declaration rule*. Therefore the declaration of T1` above is equivalent to the following declaration (note the under-line).

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

Figure 13. Equivalent declaration of T1` in Figure 12

Because of the implicit re-declaration rule, one has to be careful when refining a map instance. Sometimes a seemingly valid refinement can be wrong. To see how that could happen, consider the following declaration.

```
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;
}

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

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

Figure 14. An incorrect map instance refinement

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 under-line)

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

Figure 15. Equivalent declaration of T1` in Figure 14

The under-lined description contains an error because *source.C*, which is of type *F1.C1*, does not satisfy the signature of map type *T2`*, whose domain type is *F1`.C1`*.

Example 6.2.2 Extend a map instance by refining its maplets

In Figure 12, we refine a map instance by adding a maplet. One can also refine a map instance by refining the existing maplets, as shown in the following example.

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

```
    }
  }
}

Map type 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 16. Extend a map instance by refining one of its maplets

In Figure 16, *T1`.inst* refined *T1.inst* by refining a maplet declared in *T1.inst*. Specifically, it refined that maplet by refining another map instance declared in that maplet, i.e., *source.p2 to target.p2*. One can also refine a maplet by declaring new map instances in it, or using a combination of the two techniques. By the implicit re-declaration rule, the above T1` is equal to the following (note the under-line).

```
Map type 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 17. Equivalent declaration of T1` in Figure 1 in Figure 16

Example 6.2.3 Extend a map instance by changing its type

One can also refine a map instance by just changing its type. The new type(s) does not have to be a subtype of the old type(s). The refined map instance will have both constraints of its old type(s) and those of the new type(s). Figure 18 gives an example.

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

Figure 18. Extend a map instance by changing its type

Because of the implicit re-declaration rule, it is equal to (note the under-line)

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

Figure 19. Equivalent declaration of T1` in Figure 18

7. Implementation and Future Work

We have implemented a parser and checker for the mapping extension based on the Armani parser and checker. The parser and checker can not only be used as standalone applications, but can also be used as libraries providing parsing and checking functions to other applications.

Future work will be along the following directions.

- Refining the implementation. The current parser and checker uses a text interface. A GUI that facilitates building and viewing maps is definitely desirable. Our next step will be integrating the mapping extension with AcmeStudio, which is a GUI Acme description building and viewing tool.
- Illustrating the use of maps in the real world. Our knowledge of maps in real systems is limited. A study of the use of maps in real systems will not only verify the usefulness of the mapping extension, but also enhance our understanding of the mapping problem. Our current research activity along this dimension is a study of consistency between major system views. A mechanism to check the consistency between run-time behavior and constraints on function calls imposed by an underlying software system has been proposed. Consistency problems between other system views have also been explored.
- Extending the scope of map usage. Our research of maps has so far been concentrated on maps between different system views. In general, maps could exist within a system and the mapping mechanism described here can be used for describing either inter- or intra-system maps. In another work we have shown examples of how to declare bindings and attachments using the mapping mechanism. Other intra-system relationships that can be described using the mapping mechanism will be explored in future research.

References:

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology.* July 1997.

[2] P. Binns, M. Engelhart, M. Jackson, and S. Vestal. Domain-Specific Software Architectures for Guidance, Navigation, and Control. *International Journal of Software Engineering and Knowledge Engineering,* January 1994.

[3] L. Coglianese and R. Szymanski. DSSA-ADAGE: An Environment for Architecture-based Avionics Development. In *Proceedings of AGARD'93*, May 1993.

[4] David Garlan, Robert Allen, and John Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineerng*, December 1994.

[5] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Interchange Language. Technical Report, CMU-CS-95-219. Carnegie Mellon University, November 1995.

[6] D. Garlan, R. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. In *Foundations of Component-Based Systems*, edited by G. Leavens and M. Sitaraman. Cambridge University Press, 2000.

[7] P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, November 1995.

[8] N. Medvidovic, R. N. Taylor, and E. J. Whitehead, Jr. Formal Modeling of Software ARchitectures at Multiple Levels of Abstraction. In *Proceedings of the California Software Symposium 1996,* April 1996.

[9] N. Medvidovic and R. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of European Software Engineering Conference 1997*, September 1997.

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

[11] M. Moriconi, X. Qian. Correctness and Composition of Software Architectures. In *Proceedings of the Second ACM SIGSOFT Symposium on foundations of Software Engineering,* Software Engineering Notes, AcM Press, December 1994.

[12] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering,* April 1995.

[13] Rapide Design Team. Rapide 1.0 Language Reference Manual. Program Analysis and Verification Group, Computer Systems Lab, Stanford University, January 1996.

[14] M. Shaw, R. Deline, and G. Zelesnik. Abstractions and Implementations for Architectural Connections. In *proceeding of the Third International Conference on Configurable Distributed Systems,* May 1996.

[15] B. Woo. Software and Hardware Architecture Mapping. Independent study report, not published.

Appendix A: Family and System declaration

This appendix gives the family and system declarations used in all mapping examples.

```
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 Mapping Semantics

We present the semantics of the mapping extension using denotational semantics equations and canonical semantics representations similar to those used in [10] 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 [10] to accommodate the mapping constructs. In this section we use the word "element" in a different way than in other parts of this paper. In this section "element" not only refers to the structural constructs of Acme, but also to design spaces (collections of structural elements) and mapping constructs (map instances, maplets and map types).

A basic element is represented as 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 map 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 map types visible to e.

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

A map instance m extends the basic element with the following additional field

m = (…, l)

Informally :

l = set of maplets declared in m.

A maplet extends the basic 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 map type mt extends the basic 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 canonical 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 field x of context c. A context is simply an element whose fields can be modified as a result of the evaluation of the expression.

Statement composition

Composition of declarations is evaluated from left to right, by the following rule.
M[s1 s2 … sn]e = M[sn]…M[s2]M[s1]e

Map type semantics

M[Map type n : t1<->t2 x] e = if (n ∉ Names(e.s)) and (n ∉ Names(e.p)) and (n ∉ Names(e.t))) then
[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}]

M[heuristics x;] mt = [mt|mt.h←mt.h∪ {x}]

Map instance semantics

M[Map n x] e = if ((n ∉ Names(e.s)) and (n ∉ Names(e.p)) and (n ∉ Names(e.t))) then
if ((n∉ Names(e.m)) then
[e| M[x] [m | m.n ← n, m.t_asserted ← {Map}, m.t ←e.t,
m.m←e.m] e.m←e.m ∪ {m}]
else
[e | m1←lookup(Names(e.m),n),
m←unifyMaps(m1, M[x] [m | m.n ← n, m.t_asserted ← {Map},
m.t←e.t, m.m←e.m], e.m←e.m∪ {m}]

M[: t1, …, tn x] m = if (t1 ∈ Names(m.t_map)) and … and (tn ∈ Names(m.t_map)) then
M[x] [m | m.t_asserted← {t1,..,tn}]
M[extended with {x}] m = M[x] m

M[invariant x;] m = [m|m.i←m.i∪ {x}]

M[heuristics x;] m = [m|m.h←m.h∪ {x}]

M[x1 to x2 = {x}] m = if (forall t ∈ m.t_asserted ((satisfiesType(x1,t.d) and (satisfiesType(x2,t.ra))) then
[m|m.l←m.l∪{ml|ml.sr←x1, ml.t←x2,
ml.i← $\bigcup_{t \in m.t_asserted} substitute(x1,"source",x2,"t\,arg\,et",t.i)$ [8]
ml.h← $\bigcup_{t \in m.t_asserted} substitute(x1,"source",x2,"t\,arg\,et",t.h)$ }]

[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 map types. If there are conflicts, instances in conflict have to be unified using the unifyMap algorithm described in Appendix D.
[8] substitute(x1,y1,x2,y2,s) substitutes x1 for y1 and x2 for y2, in all elements in set s.

Appendix C Alternative Mapping Semantics (in another notation)

In this section we describe the semantics of the mapping extension using evaluation semantics 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 with respect to 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 is used to keep track of the evaluation steps. It is defined as follows.

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 sequences 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 map 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 map 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 ⊢ **Map 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. mt ; S, [y,mapT• ; e] ⊢ ∅ ⇓ A

[e | e.t_map←e.t_map ∪ {mt}] ; S ⊢ y ⇓ A

7. mt ; S, [mapT• ; e] ⊢ **Extends** t1, …, tn **with** {x} y ⇓ A
t1 ∈ e.t_map
…
tn ∈ e.t_map
mt.d <: t1.d and mt.ra <: t1.ra
…
mt.d <: tn.d and mt.ra <: tn.ra

[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] ; S, [y,mapT t1 … tn, e] ⊢ x ⇓ A

8. mt ; S, [y,maptT t1 … tn, e] ⊢ ∅ ⇓ A

[e | e.t_map←e.t_map ∪ unifyMap type(mt, t1, …, tn)] ; S ⊢ y ⇓ A

9. e ; S ⊢ **Map** n x ⇓ A
n ∉ Names(e.s)
n ∉ Names(e.p)
n ∉ Names(e.t)

[m | m.n←n, m.t_asserted←{**Map**}, m.t←e.t, m.m←e.m] ; S, [map• , e] ⊢ x ⇓ A

10. m ; S ⊢ : t1, …, tn x ⇓ A
t1 ∈ Names(m.t_map)
…
tn ∈ Names(m.t_map)

[m | m.t_asserted ←m.t_asserted ∪ {t1,…,tn}, m.t_real←unifyMap types(m.t_real,t1,…,tn),
m.i←m.i ∪ t1.i ∪ … ∪ tn.i, m.h←m.h ∪ t1.h ∪ … ∪ tn.h, m.m←m.m ∪ t1.m ∪ … ∪ tn.m] ;
S ⊢ x ⇓ A

11. m ; S ⊢ **Extended with** {x} y ⇓ A

---

m.t_real ; S, [y, mapEx• , m] ⊢ x ⇓ A

12. m1 ; S, [y, mapEx• , m2] ⊢ ∅ ⇓ A

[m2 | m2.t_real ← m1] ; S ⊢ y ⇓ A

13. m ; S ⊢ x1 **to** x2; y ⇓ 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 ⊢ y ⇓ A

14. m ; S ⊢ x1 **to** x2 = {x} y ⇓ A
satisfiesType(x1, m.t_real.d)
satisfiesType(x2, m.t_real.ra)

[ml | ml | ml.sr←x1, ml.t←x2, ml.i← $\bigcup_{t \in m.t_asserted} substitute(x1, "source", x2, "t\,\arg et", t.i)$,

ml.h← $\bigcup_{t \in m.t_asserted} substitute(x1, "source", x2, "t\,\arg et", t.h)$, ml.t←m.t, ml.m←m.m] ;

S, [y, maplet, m] ⊢ x ⇓ A

15. m ; S, [y, map•, e] ⊢ ∅ ⇓ A
m.n ∉ Names(e.m)

[e | e.m←e.m ∪ {m}] ; S ⊢ y ⇓ A

16. m ; S, [y, map•, e] ⊢ ∅ ⇓ A
m.n ∈ Names(e.m)

[e | e.m←{unifyMaps(m,lookup(e.m, n))} ∪ {e.m - {lookup(e.m, n)}] ; S ⊢ y ⇓ A

19. ml ; S, [y, maplet, m] ⊢ ∅ ⇓ A

[m | m.l←m.l ∪ {ml}] ; S ⊢ y ⇓ A

Appendix D Map Unification

In this appendix we discuss map unification and give the algorithm that is used in Appendix C and D. Map unification is used to resolve name conflicts between map instances. Unlike declaring a map type with the same name multiple times, which is deemed as an error, declaring a map instance with the name of an existing map instance is allowed and is deemed as refining the existing map instance. This feature is very useful when one wants to add more constraints to an existing map instance, or wants to declare a map type that extends multiple map types that declare submaps with the same name. When one declares a map instance with the same name as an existing map instance, the two map instances are unified and further reference to that name refers to the unified map instance.

Now let us see how two map instances are unified. Let the existing map instance be m1, the new declaration be m2, and the unified map instance be m. The type of the unified map instance is the same as that of m2. Since m is a refinement of m1, it must contain all constraints implied by m1.[9] Therefore the type of m2 must accommodate all of m1's constraints. In other words, all of m1's constraints must still be valid under m2's type declaration. For example, if m1 is declared to be of a type that maps a component to another component, then m2 cannot be of a type that maps a connector to another connector. The unified map instance m contains constraints of both m1 and m2. Therefore m1's maplets must satisfy m2's type declaration. For example, if m1 is declared to be of a type that maps a component to another component, and has a maplet that maps C1 to C2, and if m2 is declared to be of a type that maps a filter to another filter (assuming filter is a subtype of component), then C1 and C2 must be filters or m1 and m2 cannot be unified.

Based on the discussions above, we present the unifyMaps algorithm as follows. It takes two map instances, m1 and m2, as input and returns the unification of m1 and m2 if m1 and m2 are unifiable, or throws an error otherwise.

```
Map unifyMaps(m1,m2){
   Map m = new Map();     // the return value
   // maps 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.
   if (forall x in m2.t_asserted  and y in m1.t_asserted, x.d<:y.d and x.ra<:y.ra){
      m.t_asserted = m2.t_asserted;
      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_asserted);
   } else {
      throw error;
   }
   return m;
}

// This function merges two sets of maplets
// s1 and s2 are two sets of maplets to be merged.
// t is a set of types whose signatures have to be satisfied by maplets in s1.
set{Maplet} mergeMaplets(s1,s2,t){
   s = new set of Maplet;
   forall x in s1{
```

[9] The word "constraint" here means both design rules and maplets that are declared in m1.

```
      // Find the maplet in s2 that has the same source and target pair.
      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 and add the unified maplet to the result
         s += 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 signatures of types in t.
         if (forall t_asst in t (satisfiesType(x.sr,d) and satisfiesType(x.t,ra))){
            s += 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 += y;
      }
   }
   return s;
}

// Unify two maplets. This includes combining their design rules and unifying their submaps.
Maplet unifyMaplets(ml1, ml2){
   Maplet ml = new Maplet();
   forall x in ml1.m{
      if (ml2.lookup(x.name) == NULL){
         // If a submap of ml1 is not declared in ml2, directly add it to the result
         ml.m += x;
      } else {
         // If both ml1 and ml2 declared a submap of the samename, unify the submap
         ml.m += (unifyMaps(x, ml2.lookup(x.name));
      }
   }
   forall y in ml2.m{
      if (ml1.lookup(y.name) == NULL){
         ml.m += y;
      }
   }
   ml.i = ml1.i + ml2.i;
   ml.h = ml1.h + ml2.h;
   return ml;
}
```