AML: an Architecture Meta-Language

David Wile
University of Southern California / Information Sciences Institute
wile@isi.edu

Abstract

The language AML is used to specify the semantics of architecture description languages, ADLs. It is a very primitive language, having declarations for only three constructs: elements, kinds, and relationships. Each of these constructs may be constrained via predicates in temporal logic. The essence of AML is the ability to specify structure and to constrain the dynamic evolution of that structure. Dynamic evolution concerns arise with considerable variation in time scale. One may constrain how a system can evolve by monitoring its development lifecycle. Another approach to such concerns involves limiting systems' construction primitives to those from appropriate styles. One may wish to constrain what implementations are appropriate; concerns for interface compatibility are then germane. And finally, one may want to constrain the ability of the architecture to be modified as it is running. AML attempts to provide specification constructs that can be used to express all of these constraints without committing to which time scale will be used to enforce them.

1 Architecture semantics

Considerable confusion reigns in the architecture description community about what exactly constitutes an Architecture Description Language (ADL). The spectrum of languages runs from something as "obviously" relevant as Rapide [8] or Wright [1] to languages only loosely construed as ADLs, such as StateMate or UML[2]. An intention of this report is to set straight precisely what I mean by an architecture description and to provide a somewhat rigorous foundation for describing and reasoning about architecture descriptions and architecture description languages. In effect, all ADLs should be specializations of AML, semantically. Naturally, syntactic variations will abound.

I make no pretense to justifying my definition of architecture as fundamentally related to the natural language definition or even to definitions within the ADL community, although I certainly hope to have captured the intent of both. To me an appeal to "the *architecture*" of *some thing* is an appeal to properties related to its form rather than its functionality. One might not even be able to perceive what it does, but one might understand very well how it works or what basic elements are involved in accomplishing the activity and how they are related. Imagine coming upon a piece of machinery you have never seen. You may very well understand much of how the machine works, from the arrangement of gears, pulleys, levers, etc. You may even be able to look at it and see that it could not possibly work (in the obvious way) or that it may be a Rube Goldberg device. Very often, considerable understanding of its form arises from understanding the architectural *style* used to describe it. For example, you might recognize it as a piece of farm equipment – it's big and green! The reason such understanding arises quickly is that properties of the style pertain from instance to instance and need not be reinvented afresh. A surprising amount of leverage comes from reasoning about what is *not* present. This is very uncharacteristic of most approaches to specification and abstraction in software engineering.

The role of semantics of architectures is key in the design of AML. The development of the language is based on what each construct brings to the semantics – each introducing a form of shorthand for a few basic concepts on which the language is founded. At the base, we have objects that we identify called *architectural elements*, or just "elements," for short.

One states predicates that are *assumed* to hold on the elements. Architectural descriptions are always ultimately describing artifacts in the real world whose properties they are trying to model; these assumptions must be shown to hold there. It is fundamental to the model that the elements can be *identified* in the artifact and that the properties of those elements upon which further specification will rely can be *assured* of the artifact itself.

This work was sponsored by the Defense Advanced Research Projects Agency under contract nos. MDA903-87-C-0641, DABT63-91-K-0006, and F3062-96-2-0224.

For example, say we have:

elements *Controller, Main, Helper***;**

describing parts of a computer system. First of all, we must identify these elements in the system we are modeling. In AML it is assumed that *no two uniquely named elements refer to the same element in the artifact*. This is very unlike variables in programming languages, but is quite analogous to unique nullary constructors in an abstract data type. The **element**(**s**) declaration is really a primitive form of *assumption* for our logical theory about the particular architecture we are describing. We can state such assumptions in AML:

assume all *x* ***: id*** *x* ***== exists*** *y : identified-with(x,y)*
assume id *Main ^ (****id*** *Controller |* ***id*** *Helper);*

Logically, the elements are Skolem constants for which we could substitute nullary functions. The unary predicate indicating that an element has been identified is represented using the prefix operator, ***id***

Now we may want to constrain the structure of the Controller to be as follows:

assume *has-part(Controller, Main)*
^ has-part (Controller, Helper);

That is, we assume that Controller has-part Main and Controller has-part Helper, whatever "has part" means. It will be common to need to constrain such *topological relationships* between architectural elements.

Furthermore, a distinguishing feature of architecture descriptions is the preponderance of "closedness axioms" of the form:

assume all *x : has-part(Controller, x)*
=> x =Helper | x = Main

That is, Helper and Main are Controller's only parts. These are used to guarantee properties where completeness of information affects the assumptions, such as with security and fairness issues. Much of the syntactic leverage of AML arises from shortcut ways of stating these last two assumptions.

After convincing oneself that elements and assumptions hold in an artifact, there will be *derivative* propositions about the specification that follow logically from the assumptions. For example, we can specify:

derive id *Controller* **||** ***id*** *Helper*

Naturally, these must be proved within some logical framework.

We have assumed that architectural elements are a distinct type that we want to reason about identifying within an artifact. The sets of relationships that might be of interest to identify of these architectural elements and the types of elements themselves are not so clear-cut. Each ADL seems to have a slightly different vocabulary, concerning connectors, ports, components, interfaces, etc. Hence, two abstraction constructs are introduced into AML. The ***kind*** declaration allows one to declare new architectural element types; the ***relationship*** declaration allows one to characterize relationships whose identification within the artifact will be of interest. Above, we assumed has-part was such a relationship.

The remainder of the paper will lay out the various language constructs that are tuned to the architecture specification concerns mentioned above. Then examples of semantics for a well-known style and ADL will be given. Finally, issues of dynamism and a variety of uses for the semantics are indicated – including a surprisingly practical one involving monitoring running systems for architectural compliance.

2 The AML language

To summarize: software architectures are to be represented as a set of *elements* among which distinguished topological relationships are carefully *described* and *constrained*. These relationships may be *time varying*, or event-based. In effect, AML is designed to facilitate specification of these concepts only: elements, topological relationships, domain-specific relationships on the elements, and temporal constraints, along with facilities for organizing and describing these concepts more concisely.

AML semantics require several different validation and verification activities on the part of its users and/or support tools. Having proved the derivations, one must:

- *Identify* the elements of the model with items in the artifact. Again, it is assumed that no two differently named items be identified with the same artifact.
- Ensure that these identified elements satisfy appropriate *topological* relationships, again in the artifact itself. Specifically, if an element is identified, this usually requires that other topologically related elements be identified as well.
- Ensure that certain *closure* properties hold in the artifact. This involves establishing that e.g. all of the parts, and only the parts are accounted for that participate in part relationships.
- Establish the non-topological properties. This is a purely domain-specific activity, and is actually the major source of leverage of ADLs.

The semantics of AML use only a very small part of the predicate calculus along with some elementary set theory. It is my intention that AML be adapted to different logics for different analysis purposes. In what follows the predicate language will include simple temporal predicates, ***sometimes*** and ***always***, conventional quantifiers (with typed variables), and the usual connectives and propositions, as illustrated in the examples above.

It must be emphasized that AML is a framework. Any extension of it consistent with the underlying logic is encouraged. Hence, I would expect e.g. someone describing Wright semantics to introduce CSP expressions into the specification, perhaps along with a special purpose reasoning engine that understands those semantics in

conjunction with the topological semantics of AML.

3 Relationships

The structural building block in AML is the relationship. Over the past 25 years in the Software Sciences Division at ISI we have built many languages based on "relational abstraction" wherein we model all data access as manipulations of abstract relationships [5]. Some of the conventions adopted there are brought over into AML. In AML the ***relationship*** declaration is used to describe *topological relationships* among elements and *domain relationships* between elements and other external types, such as integers or strings, or even modules in a programming language, for example.

A relationship declaration may simply describe a relation name and its parameters. Assumptions about which tuples are in or out of the relationship may be described. For example,

```
relationship has-part(b,a)
    assume ~ exists p : has-part(p,a) ^ p  <> b
```

This relationship declaration declares has-part and says that nothing is part of two different parents. The assumption is logically equivalent to having written:

```
assume all a,b : has-part(b,a)
        => ~ exists p : has-part(p,a) ^ p <> b
```

Relationships may also be defined to be equivalent to other predicates as follows:

```
relationship attached-to(a,b)
relationship lower(a,b)
  == attached-to(a,b)
    | exists c : attached-to(a,c) ^ lower(c,b)
```

This declares the lower relationship to be the transitive closure of the attached-to relationship.

Some special shortcuts have been found to be useful when relationships form a basis for a language [5, 18]. Of course, relationships are used as propositions in predicates in the normal way. However, a particularly useful shortcut is to use the wildcard symbol, ***$,*** to indicate an unnamed existential variable, to be bound in the immediately enclosing scope. For example,

```
attached-to(Controller,$)
  ^ attached-to($,Controller.Helper)
```

is equivalent to:

```
(exists x1 :attached-to(Controller, x1))
    ^ (exists x2: attached-to (x2,Controller.Helper))
```

There are two symbols used to select values from a relationship in expressions:***?*** and ***??.*** The former selects an *arbitrary element* from the corresponding slot of the tuples that are in the relationship, consistent with the rest of the parameters; the latter refers to the *set of all such elements*. For example,

```
attached-to(?,a)
```

refers to any element which is attached to a. On the other hand,

```
attached-to(a,??)
```

refers to the set of all elements to which a is attached. There must be at most one occurrence of ***?*** or ***??*** in any particular relation access.

Relationships' slots may have multiplicity constraints attached to them. Although these seem at first a bit tricky, they are extremely useful for characterizing common constraints rather concisely, and they form the foundation for extensions to the ***element*** declaration already introduced. The multiplicity of any slot qualifies the range of elements that can be in that slot for any random selection of other elements in the other slots.

The multiplicity is specified using a standard mathematical range notation, viz. ***N .. M***. If ***M*** is omitted, an indefinite number of elements (>= ***N***) may be related.

```
relationship attached-to(0 .. n bottom, top);
```

means:

```
relationship attached-to(bottom, top);
    assume all y: size attached-to(??,y) in 0 .. n
```

where the ***size*** operator is the cardinality of its argument set, the ***in*** operator is just an element test, and ***0..n*** refers to the set of integers in the indicated range. When multiplicity is unspecified, there is no constraint on the slot; that is, several elements can be related to the same set of elements in the other slots. Hence, the default multiplicity of a parameter is "***0 ..***"

Another notational convenience, called "mapping," specifies that a binary relationship holds between an object and several others, simultaneously. The "trick" is to use the relation name as an infix operator whose first argument is a single object and the rest of which are specified in set braces, viz.

```
Controller has-part {Main,Helper}
  == has-part(Controller, Main)
    ^ has-part(Controller, Helper)
```

The final concept needed to specify architectures concisely is that of "closedness"[1] of relationships. Consider the mapped relationship above. It is often desirable to say, "and those are all the parts." Logically, this translates into an assumption:

```
has-part(Controller,??) = {Main,Helper}
```

written

```
Controller has-part ! {Main,Helper}
```

Obviously, this is not any more concise than the previous line, but its utility will become apparent, presently.

4 Elements revisited

Above, the ***element*** was introduced as the fundamental structural building block of AML, the mechanism for introducing Skolem constants into the semantics. The ***element*** declaration is extended here to introduce further elements related to the one being declared, to allow rather concise specification of topological relationships between

[1] This rather awkward word is used instead of "closure," which is technically quite different.

the element and those introduced, and to specify closedness properties over those relationships. Generally, indentation is used to introduce assumptions about the element, leaving the element implicit (much as in object-oriented programming). Given a topological relation declaration for has-part,

relationship *has-part(**1..1** whole, part)*

the indented declaration of has-part beneath an element declaration can use set notation to specify several parts. The notation is an extension of the relation mapping notation above. For example,

element *Controller*
*has-part { **element** Main; **element** Helper}*

(The number of parts must be consistent with the multiplicity declaration for has-part.)

This declaration introduces three assumptions:

(1) That the three element instances, Controller, Controller.Main, and Controller.Helper, are unique names. The latter qualified names occur because the declarations are inside the scope of the Controller declaration.[2][3] It is important to emphasize that these instances are to be created afresh. To relate already-introduced elements to Controller, one would use the names without the element declaration.

(2) That the parts must be identified whenever the controller is, viz.

assume id *Controller =>*
id *Controller.Main ^ **id** Controller.Helper*

(3) That topological relations hold:

assume id *Controller*
=> has-part(Controller, Controller.Main)
^ has-part(Controller, Controller.Helper)

The number of parts (in this case) allowed and the elements' closedness properties can be constrained using the same notation as with relationships, viz.

element *Controller*
*has-part **!** { **element** Main*
*has-part { **element** Surfaces;*
*0 .. 1 **element***
Instruments};
*0 .. 1 **element** Helper*
*has-part { 1 .. 2 **element***
Instruments }}

means that Controller has one element called Main and an optional element, Helper. Unlike with the slots of relations, an element's multiplicity defaults to "1 .. 1," i.e. there is exactly one instance of it. Main itself has a required element Surfaces and an optional element, Instruments. And although Helper itself is optional, when present, it must have at least one part, called Instruments[1], and may have a second, called Instruments[2]. Notice the cascading of assumptions that must occur, regarding conditional identification of the elements and statement of the part relationships. It is important to emphasize that elements are not generic specifications; they specify a single instance of an artifact. The ***kind*** declaration is used to specify generic elements and will be described shortly.

Notice that the ***"!"*** after the has-part restriction indicates closedness as in the relationship mapping construct above. Hence, the only parts Controller can have are Main and Helper. Main itself may have parts in addition to Surfaces and (possibly) Instruments. Similarly, Helper may have elements other than Instruments.

5 Constraints: assumptions and derivations

Again, elements may be constrained in two ways: through constraints that are *assumed* to be true of the element and through constraints that should be able to be logically *derived* as holding. The constraints that are assumed to hold must be validated in the artifact being modeled. So far, we have only considered topological relationships, those that relate different elements of the architecture.

Most "interesting relationships" are non-topological, i.e. relationships between architecture elements and other models. So most assumptions and derivations will not be predicates in terms of purely topological relationships, but in terms of more domain-specific relationships. Almost all of the power of an ADL stems from its ability to model some aspect of the application of the architecture that can be analyzed and reasoned about *a priori*. Although ADLs are normally used to express non-functional aspects, the framework indeed supports the expression of functional constraints, such as detailed requirement specifications on components and system invariants. The derivation of axioms involving the non-topological relations in fact constitutes *the (a) semantics* of an architectural description.

The following example attempts to put some semantics on the simple controller above. The idea is that there are potentially two processors within it that can run in parallel. Here it is possible that the Helper processor will fail, in which case it will not be identified. The state of the system in which this occurs is called "Emergency;" otherwise things are "Normal." In the former case, a subprocess of Main, called Instruments, will take over for the missing Helper process. These concepts are entirely outside of the architecture domain itself, but in a modeling domain for the dynamics of the system itself.

[2] If relations are declared within the scope of an element, they too must be qualified if used outside of the element's scope.

[3] To use the dot notation, the name of the related subordinate element must uniquely belong to only one topological relationship.

Consider:

```
element Controller
    has-part! {element Main
                has-part { element Surfaces;
                           0 .. 1 element Instruments}
              0 .. 1 element Helper
              has-part {1 .. 2 element Instruments }}
    assume mode(Controller) = Normal == id Helper;
    assume mode(Controller) = Emergency
          => id Main.Instruments
    assume mode(Controller)= Emergency
          => ~ id Main.Surfaces.Secondary;
    *   *   *
    assume swap-time(Main) < 1.5 seconds;
    assume sometimes id Helper;

    derive id Main.Surfaces.Secondary
        || id Helper.Instruments[$];
    derive start-time id Main.Instruments
        - start-time ~ id Helper
        < 2 seconds;
```

The first set of assumptions refers to the states the system might be in, and what processes must be identified. With a few more such domain-specific assumptions – necessarily checked within the artifact – derivations such as the first could possibly be proved.

The final assumption, that the Helper processor must be provided for, both constrains further development and refinement of the specification as well as has dynamic implications. Notice that the final ***derive*** statement refers to dynamic event times when the Helper cuts out and Main.Instruments takes up the slack. Even with the assumption that the process swap-time is less than 1.5 seconds, there is not enough information in the specification to prove this – nor have we introduced a powerful enough logic to deal with such events in this report –, but it is indicative of the kind of dynamic constraint one wishes to impose on architectures. It is possible that the proofs of such predicates will not be amenable to automated reasoning approaches, but perhaps a kernel of constraints to be checked dynamically during the running of the system can be derived from them (see Real Use section).

Notice just how open this specification still remains. Surfaces, Instruments (in both processes), and even Main and Helper may have additional parts. In fact, just this openness may cause trouble in trying to prove the first derivative fact (which might be aided by knowing that the only processes in Helper were the Instruments).

6 Kinds

In AML ***kinds*** play the role of both architecture types and architectural styles, introducing and restricting new element types. The word was chosen for its lack of associations in modern programming or specification languages. A ***kind*** declaration looks like an ***element*** declaration, but the meaning of *closed* ***(!)*** is extended somewhat. Consider the kind declaration below:

```
kind component
  has-part ! {0..1 element top;
              0..1 element bottom};
```

This introduces a new type of ***element*** into the specification and a unary predicate, component, to test whether an element is of this type. One can now use ***component***, in boldface (stylistically), to introduce a new kind of element, viz.

```
component StackADT;
```

This is *the only way* to have an element of type component introduced, unlike in Acme, e.g., where types are predicates [13]. Because the has-part relation was closed in the ***kind*** declaration, this component cannot have any parts other than possibly a top and possibly a bottom. One could be more specific and specify:

```
component StackADT
  has-part ! {element bottom};
```

Notice that it is necessary to close off the specification if one does not intend to leave open the possibility that a top element be introduced later.

The next obvious thing to do is extend the ***relationship*** declaration to allow the slots to be restricted to certain kinds. In fact, more generally, it is desirable to restrict them to elements satisfying arbitrary unary predicates. Hence, type specifications may be added to formal parameter specifications. For example,

```
relationship attached-to (b: n-bottom, t: n-top)
  assume above(t,b)
```

The signatures are treated as patterns in the polymorphic programming sense. This specification in and of itself does not require that all instances of the attached-to relationship require n-bottom and n-top arguments. Rather, when such arguments are given, the stated assumptions must hold when the relationship holds. Hence, that declaration entails the following assumption:

```
assume all b,t : n-bottom(b) ^ n-top(t)
                 => ( attached-to(b,t)  => above(t,b))
```

(Notice that == would replace the second implication if used in the relationship declaration instead of ***assume***.)

The *entire set* of declarations for the *attachToCom* relation is used to determine the type restrictions on the relationship. Hence, if the following are also declared:

```
relationship
      attached-to (0 ..1 b: n-bottom, t: m-top);
relationship
      attached-to (b: m-bottom, 0 ..1 t: n-top);
```

and these are all the declarations of the relationship, the following assumption holds:

```
assume all b,t : attached-to (b,t) =>
                   n-bottom(b) ^ m-top(t)
                 | m-bottom(b) ^ n-top(t)
                 | n-bottom(b) ^ n-top(t)
```

The semantics of multiplicity restrictions are complicated slightly by the typing.

relationship *attached-to(**0 .. 1** b:n-bottom, t:m-top);*

means:

relationship *attached-to(b:n-bottom, t:m-top);*
assume all *y: m-top(y) =>*
size *{ x | n-bottom(x)*
*^ attached-to(x,y)} **in** 0 ..1*

where standard "set comprehension" is indicated with the "|" inside the braces.

In AML, the effect of styles in other ADLs is obtained through the use of the ***element*** declaration and constraining constructs from a specific ***kind***. The C2 system [17] will be used as an example style, since it is well-known to the architecture community. C2 has components and connectors, both of which have (potential) attachment points at the top and bottom. The attachments are subject to the following explicit *design rules*:

- The top of a component may be attached to the bottom of a single connector.
- The bottom of a component may be attached to the top of a single connector.
- There is no bound on the number of components or connectors that may be attached to a single connector.
- When two connectors are attached to each other, it must be from the bottom of one to the top of the other.

The idea is to build a *C2-system* ***kind***. AML overloads ***kinds*** to be used as styles using the following observation. If we *limit the declarative facilities* that one can use to describe an element *to those introduced by a kind declaration*, we have effectively captured the goal mentioned early on for ADLs: limiting the use of element declarations to what constitutes an architectural style. Consider the kind declaration below:

kind *C2-system*
kinds *m-top, m-bottom, n-top, n-bottom**;***
relationship *C2-element(x)*
== *component(x) | connector(x);*
relationship *part-of(1..1 C2-system, C2-element);*
relationship
*attaches-at **(**t: C2-element**,** b: C2-element);*
kind *component*
*attaches-at **! {0..1 m-top** t;*
0..1 m-bottom** b**};
kind *connector*
*attaches-at **! { n-top** t;*
n-bottom** b**};
relationship *attaches-at **(**element**,** p:connector);*

The C2-element relationship has been introduced as a type (unary predicate) to restrict the attaches-at relationships to those whose range is either a ***component*** or ***connector*****,** when a declaration of a ***C2-system*** is made.

We can restrict an element to a style by using the closed symbol ***(!)*** after the ***kind*** name in a declaration.

C2-system ! *StackVisualizationSystem*
part-of ***{component*** *StackADT*
*attaches-at **! {bottom** b**};***
components *StackVisualization1,*
StackVisualization2
*attaches-at **! {bottom** b,**top** t**};***
component *GraphicsServer*
*attaches-at **! {top** t**};***
connectors *TopConnector,*
*BottomConnector**}***

In this example, the StackVisualizationSystem is restricted to being a C2-system. Here, only components and connectors can be used. When a ***kind*** has named elements, the required elements are *implicitly* declared to be elements of each instance of the kind. Hence, both TopConnector and BottomConnector have *top* and *bottom* parts. When the elements in the kind declaration are optional, then they must explicitly be declared with the instance of the kind, or identified later, perhaps even at run-time.

In order to describe how the various tops and bottoms can be legally attached, the following set of ***relationships*** should be added to the C2-System ***kind*** declaration so that they are enforced as assumptions in elements of that kind.

relationship *attached-to**(0 ..1** b:n-bottom, t:m-top);*
relationship *attached-to(b:m-bottom, **0 ..1** t:n-top);*
relationship *attached-to(b:n-bottom, t:n-top);*

Notice that these relationships convey exactly the same constraints as required by the C2 design rules: the first two requirements are conveyed by the first two relationship declarations; the third constraint is conveyed by the lack of a restriction on how many m-tops and m-bottoms (respectively) can be connected in the first two relationship restrictions; and the final restriction is conveyed by the last relationship declaration.

A UML model of the C2 style is presented in [15] using constraints on stereotypes. The AML model is more concise, in part, because it is designed to capture these kinds of constraints. The polymorphism of the single *attached-to* relationship gains leverage as well. Unfortunately, the analogous facilities of the UML model constraining multiplicities could not be used in their model; they had to resort to the reflection capabilities, making the descriptions quite clumsy.

(Other example ADLs and styles specified in AML will be available at the author's web-site [19].)

7 Uses for AML semantics

Frequently in the above discussion, when a construct was introduced without closedness assumptions, references were made to further constraining the closedness later in the specification or implementation process. One may wish to constrain how a system can

evolve in various parts of its development lifecycle. *Laws* such as Minsky proposes [12] or *constraints* as in Monroe's Armani system [13] address such evolution concerns during system specification. Another approach to such concerns involves limiting systems' construction primitives to those from appropriate styles, such as in Wright [1] and UniCon [16], or embodied by the choice to use C2 [17]. One may wish to constrain what implementations are appropriate; concerns for interface compatibility such as evidenced in SADL [11] are then germane. And finally, one may want to constrain the ability of the architecture to be modified as it is running; languages such as Rapide [8] and Darwin [9] emphasize these issues. Using AML's specification constructs allows one to express all of these constraints without committing to which time scale will be used to enforce them. Each of the above approaches should map readily into AML. Naturally, different logical systems may be necessary. That is consistent with the philosophy of AML. If more restrictive constraints require more "reflective" capabilities of an ADL than are present here, one should seek to regularize them and introduce them into AML.

Normally, the existence of a formal "semantics" for a language is simply a confidence-building device for its designers. An interesting consequence of AML semantics is that they can actually be useful in realistic settings. If the topmost element of an architectural description is presumed to be identified, there will generally be several other elements that must be identified as well. If these are furthermore assumed to hold, there will still be a residue of assumptions about the identification of optional elements and replicated elements that cannot be assumed.

This residue can be manipulated and used in testing the running architecture for conformance, for example, by instrumenting it with probes that detect the identification and perhaps subsequent non-identification of elements. A "shadow architecture" specification – the residue – can then be monitored for compliance. The Flea system [3,14] has been used for such purposes, although the residue was concocted in an *ad hoc* fashion.

I expect variants of AML to be developed, tuned for specific logic and event languages and perhaps specific theorem prover aids. In such cases, it may be useful to use predicate completion techniques or circumscription in place of the closedness axioms.

8 Previous work

In addition to the dynamic specification techniques of ADLs already discussed, AML is related to three areas: architecture description languages, programming languages, and specification languages. Acme [4] has goals similar to AML but has been unable to achieve community-wide consensus because of some awkwardness with modeling a few prominent languages, namely Rapide and C2, and because of a lack of support for dynamic architectures. AML was inspired by an attempt to add dynamism to Acme.

It is worth repeating the ADL-specific concerns:

1. To specify instances, made up of examinable elements;
2. To specify non-functional properties of the instances;
3. To reasoning based on knowing the extent of instances – what is not present;
4. And, to restrict the style or styles used to construct them.

If we look to programming languages for hints as to how to satisfy these concerns we come up surprisingly short. Indeed, they are all capable of describing a single instance, but the constructs introduced are generally based on abstractions. Activities such as specifying a variable whose only instance has a specific structure (1) is usually very awkward, requiring, for example, the introduction of a nonsense intermediate type declaration. With programming languages one always describes functionality, and only rarely, functionality properties; never, non-functional properties (2). Restricting the reasoning process to specific programs is notoriously difficult (3); witness global variables, aliasing problems, gotos and exception handling. Indeed, applicative languages are designed to make this more perspicuous. In fact, there is a proposal to use Haskell [7] to describe architecture evolution, based on a reflective capability on the architecture's construction primitives. That is a more prescriptive presentation than AML. Finally, we cannot generally limit program construction (4) even to the use of a specific set of types.

Specification languages come closer to providing for some of these, but they are generally clumsy at describing a single instance of a thing (1). Abstraction is emphasized in languages such as Larch [6] and other algebraic approaches, where instances are often nullary constructors of a data type. Indeed, they can be used to describe non-functional properties well (2), based on equational reasoning generally, but sometimes on richer logical foundations (that AML presumes will be used). Specification of the limitations of what can be used is generally difficult (3); in some ways, it is really the same as "the frame problem." Notice that it is a much harder problem for programming languages, where the effects of operations are characterized. Restricting topological relationships is a much easier task. Finally, specification languages also lack the ability to restrict specifications to the use of a subset of all language constructs (4).

Object-oriented languages help somewhat; they have some aspects of both programming languages and specification languages, especially the modern tool suites for OO design, e.g. based on UML. However, they miss the mark on (3) and (4) especially, where constraints using facilities as powerful as *reflection* are needed to provide the necessary limitation to particular styles. Some of the systems do allow constraining instances (1) as easily as

classes. However, protests to the contrary, they are not good for specifying non-functional properties; they have made too many representation-like commitments in the language to think of them as describing anything but imperative programs.

9 Conclusions

Most architecture description languages emphasize particular domain relationships and leave the semantics of the topological relationships and the identification process implicit. AML is an attempt to tease out just that aspect of ADL design and formalize it.

Only a small part of AML has been implemented: a translator into constraints from nested element part declarations. Moreover, some semantic issues remain open, particularly around how ***kinds*** should be defined. Although it has been used to specify portions of C2 and Acme, more examples must be tried to see if it can represent ADLs, such as Rapide and Darwin.

Architecture description languages have not been formulated on a strong semantic base in the past; AML is an attempt to provide such a base. Indeed, it is fundamentally very simple: elements are described, their number and relationships with one another are constrained, domain-specific properties are assumed to pertain, and derivations from these assumptions are stated. In the artifact described, once the elements are identified, their part relationships verified, and their properties checked, any formally established derivations can be trusted to hold.

It is this minimal semantic baggage that probably represents the appeal of ADLs – one can make them mean just about anything. But not just anything.

Acknowledgements

I wish to thank David Garlan, Neil Goldman, Jose Messeguer, Bob Monroe, Peter Pepper, and Carolyn Talcott for influential conversations on this topic.

References

[1] R. Allen. *A Formal Approach to Software Architecture*. Ph.D. Thesis. Carnegie Mellon University CMU Tech. Report CMU-CS-97-144, May, 1997.

[2] G. Booch, I Jacobson, and J. Rumbaugh *The Unified Modeling Language for Object-0Oriented Development*. Documentation Set. Rational Software Corporation. 1997.

[3] S. Fickas and M. Feather. Requirements monitoring in dynamic environments. Proceedings of the Second IEEE International Symposium on Requirements Engineering, York, England, March, 1995, Pp. 140-147.

[4] D. Garlan, R. Monroe, and D. Wile. Acme: an architecture description interchange language. In Proceedings of GASCON '97. (See also: http//www.cs.cmu.edu/~acme/)

[5] N. Goldman and K. Naryanaswamy. Software evolution through iterative prototyping. ICSE 1992. Pp. 158-172

[6] John Guttag and James Horning. Report on the Larch shared language. *Science of Computer Programming*,(6):1984, Pp. 103-134.

[7] Paul Hudak et al. Acme HOT. Draft Report. Yale Computer Science, 1998.

[8] D. Luckham, et al. Specification and analysis of system architecture using Rapide. IEEE Transactions on Software Engineereing 21(4) April, 1995. Pp. 336-355. (See also: http://anna.stanford.edu/rapide/).

[9] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceeding of the ACM SIGSOFT '96: Fourth Symposium on the Foundations of Software Engineering*. San Francisco, CA Oct., 1996. Pp. 24-32.

[10] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions of Software Engineering.* To appear.

[11] M. Moriconi, X. Qian, and R.A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering* 21(4) Apr. 1995. Pp. 356-372.

[12] N. Minsky. Independent on-line monitoring of evolving systems. *ICSE* 96, Berlin, Germany. Pp. 134-143.

[13] R. Monroe Capturing software architecture design expertise with Armani. TR CMU-CS-98-163, Carnegie Mellon University. 1998.

[14] K. Narayanaswamy. http://www.darpa.mil/ito/Summaries97/D931_0.html

[15] J. Robbins, N. Medvidovic, D. Redmiles, and D. Rosenblum. Integrating Architecture Description Languages with a Standard Design Method *Proceedings of the Twentieth International Conference on Software Engineering* (ICSE Ô98, Kyoto, Japan), IEEE Computer Society Press, April 19-25, 1998, pp. 209-218.

[16] M. Shaw, et al. Abstractions for software architecture and tools to support them. *IEEE Transactions of Software Engineering* 21(4) Apr., 1995. pp. 314-335.

[17] Richard N. Taylor, et al. "A Component- and Message-Based Architectural Style for GUI Software", *IEEE Transactions on Software Engineering*, June 1996.

[18] David Wile. Adding relational abstraction to programming languages. In *Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development*. Napa, CA. 1990. Pp. 128-139.

[19] David Wile. Web site. 1999. http://www.isi.edu/software-sciences/wile/home-page.html.