ACME:
An Architectural Interchange Language

David Garlan   Robert Monroe   David Wile*

Version 1.01
Draft of 29 November 1995
CMU-CS-95-219

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292

WORKING DRAFT: Do Not Distribute

The research reported here was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grants F33615-93-1-1330 and N66001-95-C-8623; and by National Science Foundation Grant CCR-9357792. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory, the US Department of Defense, the United States Government, or the National Science Foundation. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon.

Keywords: architectural interchange, ACME, architecture description languages, software architecture

Abstract

A number of architectural description languages (ADLs) have been developed to support the formal representation and analysis of software architectures. Each such ADL provides certain complementary capabilities supporting architectural development. Unfortunately, however, each ADL and its supporting toolset typically operates in a stand-alone fashion, making it difficult to combine multiple tools, share architectural descriptions, or leverage previous work in producing new languages and tools. ACME was developed as a joint effort of the architectural research community to provide a common interchange format for architectural design tools. ACME provides a structural framework for characterizing architectural designs together with liberal annotation facilities for including additional ADL-specific auxiliary information. This notational scheme permits subsets of ADL tools to share whatever architectural information is jointly understood by those tools, while tolerating the presence of information that falls outside their common vocabulary. In this report we describe the ACME language: its goals, how it can be used, the tools that are currently under construction, and expected extensions of the language.

1 Introduction

The software architecture of a system defines its high-level structure, exposing its gross organization as a collection of interacting components. A well-defined architecture allows an engineer to reason about system properties at a high level of abstraction. Typical properties of concern include protocols of interaction, bandwidths and latencies, locations of central data stores, and anticipated dimensions of evolution [GP95, GS93, PW92].

Architectural design has always played a strong role in determining the success of complex software-based systems: the choice of an appropriate architecture can lead to a product that satisfies its requirements and is easily modified as new requirements present themselves, while an inappropriate architecture can be disastrous. Moreover, industry (as well as the US DoD) has begun to recognize the important economic role of reusable architectural frameworks and styles, which provide an architectural design for a family of related systems. A new system can usually be built by instantiating an architectural framework at much lower cost than building the system from scratch.

However, despite its importance to software systems engineers, the practice of architectural design has been largely ad hoc, informal, and idiosyncratic. As a result, architectural designs are often poorly understood by developers; architectural choices are based more on default than solid engineering principles; architectural designs cannot be analyzed for consistency or completeness; architectural constraints assumed in the initial design are not enforced as a system evolves; and there are virtually no tools to help the architectural designers with their tasks.

In response to these problems a number of researchers in industry and academia have proposed formal notations for representing and analyzing architectural designs. Generically referred to as "Architectural Description Languages" (ADLs), these notations usually provide both a conceptual framework and a concrete syntax for characterizing system software architectures. They also typically provide tools for parsing, unparsing, displaying, compiling, analyzing, or simulating architectural descriptions written in their associated language.

Examples of ADLs include Aesop, Adage, ArTek, Meta-H, Rapide, UniCon, and Wright [GAO94, CS93, T$^+$94, BV93, LAK$^+$95, SDK$^+$95, AG94]. Although all of these languages are concerned with architectural design, each language provides certain distinctive capabilities: Aesop supports design using architectural styles; Adage supports description of architectural frameworks for avionics navigation and guidance, Meta-H provides specific guidance for designers of real-time avionics control software; Rapide allows architectural designs to be simulated, and has tools for analyzing the results of those simulations; UniCon has a high-level compiler for architectural designs that support a heterogeneous mixture of component and connector abstractions; Wright supports the specification and analysis of interactions between architectural components.

The proliferation of ADLs—and their supporting toolsets—is both a blessing and a curse. On the positive side, different ADLs have explored different facets of the overall architectural design problem. By exposing different features of architectural design and ways to exploit those features, collectively they are helping to deepen our understanding of the concrete roles that architectural description can play in software development. Thus at this early stage in the development of a discipline of software architecture, research exploration of multiple approaches to architectural description is both appropriate and necessary.

On the negative side, however, each ADL system typically operates in a stand-alone fashion, making it difficult to combine facilities of one ADL with those of another. Furthermore, there are many common aspects of architectural design support that are reimplemented afresh for each ADL. Examples include graphical tools for visualizing and manipulating architectural structures, facilities for storing architectural designs, and certain domain-independent forms of analysis (such as checking for cycles, or the existence of dangling connections). Such gratuitous redundancy is clearly a waste of resources for individual researchers as well as the community as a whole. Finally, for many practitioners, deeper semantic differences between different ADLs are a second-order issue. First and foremost they need a way to describe their architectural structures *at all*—any way that allows them to record system structures at an appropriate level of abstraction will do. But currently adopting a specific exiting ADL requires a substantial investment to install the ADL tools and learn to use the ADL effectively. Alternatively, practitioners are forced to develop their own idiosyncratic notations and tools, exacerbating the proliferation problem even further.

As we argue in detail later, one way to ameliorate these problems would be to provide an architectural interchange language for software architecture. Such a language would permit the integration of different tools by providing a common form for interchanging architectural descriptions. It could also serve as a basis for certain kinds of analyses, and even provide a starting point for the design of new ADLs with specific, perhaps domain-specific, features. Further, it could clarify the relationship between different ADLs and their analyses.

ACME is an architectural interchange language with precisely those goals. It was developed as a joint effort to provide a common intermediate representation for a wide variety of architecture tools. ACME is based on the premise that there is sufficient commonality in the requirements and capabilities of ADLs that meaningful ADL-independent information can be shared. ACME attempts to embody those commonalities. However, ACME also allows the incorporation of ADL-specific information, so that auxiliary information can be retained, and so that certain tools can share kinds of architectural information that are not necessarily understood by all other tools. This notational scheme permits subsets of ADL tools to share whatever architectural information is jointly understood by those tools, while tolerating the presence of information that falls outside their common vocabulary.

In the remainder of this report we describe the main features of ACME and illustrate how it can be used to represent and communicate architectural designs. In the next section we elaborate on the rationale behind ACME, and describe its goals. In the ...

2 Language Rationale and Goals

2.1 Rationale

The existence of multiple languages is not unique to the domain of software architecture. A similar situation exists for document formatting, programming, graphical encodings, hypertext, and many others. As with ADLs, such diversity creates problems for users of these languages.

In other domains a number of approaches have been used to cope with problems of language heterogeneity.

1. **Pick one:** Let the community or marketplace decide on a single dominant language, and coerce future tool development to occur around that language.

2. **Design a "union" language:** Design a language that incorporates all of the features of all of the languages, and thereby allow users to express anything that they could have expressed in any of the individual languages.

3. **Design an "intersection" language:** Pick a lowest common denominator language that includes the aspects of architectural description that are shared by all ADLs.

4. **Give up:** Admit that language diversity is simply too large to try to find any coordinated solution at present. This usually results in a large number of pairwise (or sometimes n-way) conversions to handle specific instances of language interoperability.

With respect to ADLs, none of the techniques is particularly appetizing. As we noted earlier, the first alternative is inappropriate: given the relative immaturity of our understanding about architectural modelling and analysis, it would be foolhardy to legislate a single fixed language at this time. Moreover, each of the existing languages can do some things well, but may be weak in other respects. The second alternative—a union language—is likewise unrealistic. Not only are the capabilities of different ADLs significantly different at a semantic level (hence making language synthesis difficult), but it is not yet clear what kinds of capabilities one would ideally want in a such a language. The third alternative—an intersection language—is not likely to be successful either. The range of constructs provided by different ADLs is sufficiently broad that it would be difficult to find a single semantic core to which the capabilities could be translated.

This would suggest that the only alternative is the fourth: give up, and live with a proliferation of specialized inter-ADL solutions. However, all is not lost. To understand why, consider the following two observations about architectural description languages.

First, an examination of ADLs reveals that there is in fact considerable agreement about the role of *structure* in architectural description. One of the results of the First International Workshop on Architectures for Software Systems [Gar95] was that virtually all ADLs take as their starting point the need to express an architectural designs as a hierarchical collection of interacting components. On top of this structural skeleton different ADLs then add various kinds of additional information, such as run-time semantics, code fragments, protocols of interaction, design rationale, resource consumption, topological invariants, processor allocations, etc. In some cases this additional information could in principle be understood and manipulated by tools for some other ADL. (For example several tools could share a common interpretation of the visual information for displaying the architecture.)

Second, although there is little beyond the use of architectural structure about which all ADLs agree, significant subsets of existing ADLs *do* agree on certain kinds of extra-structural information. For example, both Rapide and Wright represent interactions in terms of events.[1] Aesop and the notations used at SRI [MQ94] are both concerned with the expression of stylistic invariants. Aesop, UniCon, and Meta-H both provide capabilities for expressing properties that permit real-time schedulability analysis.

These two observations suggest that a plausible path towards integration of ADL facilities is to design a language that centers on the shared structural core of architectural description, but that in addition, permits the inclusion of other aspects of architectural description that may be relevant to one or more ADL. In this way all ADLs can communicate structural aspects of an architecture in a uniform manner, while permitting variability about other aspects of an architectural design. To the extent that subsets of ADL tools can agree on those additional aspects, they can also take advantage of that shared information. Over time, one can well imagine that as software architecture community develops a better understanding of the value of certain classes of architectural information, conventionalization of its representation will take place—and thereby be exploitable through the interchange language.

This is the essence of ACME. The language provides fixed syntactic forms for representing architectural structure. Additionally it provides an *open semantic framework* in which architectural

[1] The details of their semantic models differ somewhat, Rapide using POSETs and Rapide using CSP's failures-divergences-refusals model. But both are concerned with predicting or analyzing the trace behavior of communication events.

structures can annotated with ADL-specific properties. In this way ACME functions as both an intersection and a union language: the shared structural core represents an intersection of the expressive capabilities of most ADLs, while the use of annotations accommodates the union of ADL-specific concerns.

2.2 Goals and Design Philosophy

The design of any language should reflect its intended purpose. For example, if the primary purpose of a language is to support formal analysis, then minimality of features and semantic simplicity are likely top-level concerns. On the other hand, if the primary purpose of a language is to support a domain-specific design activity (such as for control systems in chemical plants), then closeness of match to engineers' natural vocabulary of design is crucial. It is important, therefore, to be clear about the intended purposes of ACME.

In decreasing order of priority, ACME has five main goals.

1. *To provide an interchange format for architectural development tools and environments.* This implies that the design should make it easy for tools to parse and unparse architectural descriptions, and that the language should be able to represent in some form all of the information that architecture-based tools might need to exchange.

2. *To provide a representational scheme that will permit the development of new tools for analyzing and visualizing architectural structures.* This implies that the language should provide an architectural vocabulary that makes it straightforward for tool writers to map their intuitions about architectural structures into the forms expressible in the language.

3. *To provide a foundation for developing new, possibly domain-specific, ADLs.* This implies that the language should focus on core capabilities of all ADLs, leaving open the possibility of specific variants built on top of the core.

4. *To serve as a vehicle for future conventionalization of representations for architectural information.* This implies that the language should make it possible to incorporate new standards for architectural representation as they emerge.

5. *To provide human-readable and human-writable descriptions of architectural designs.* This implies that the language should be compact and allow direct expression of common architectural structures and idioms.

While these goals are complementary, taken individually they lead to quite different choices in design. In particular, the first goal—ease of tooling—leads one down a quite different path than the last goal—ease of reading and writing for humans. The former argues for a simple, easy-to-parse language, while the latter argues for expressive language features and abstractions that lead to compact architectural descriptions.

To accommodate the different goals, we have adopted a staged approach to the language. As illustrated in Figure 1, at the core is *Kernel-ACME*. The kernel language is small and designed for ease of tool translation, manipulation, and analysis. As we detail in Section refkernel:section, it provides basic structuring constructs for representing instances of system architectures.

Extending Kernel-ACME is *Full ACME* (or simply *ACME*). It adds to Kernel-ACME the ability to define reusable architectural *templates*. As we will see, the use of templates makes architectural

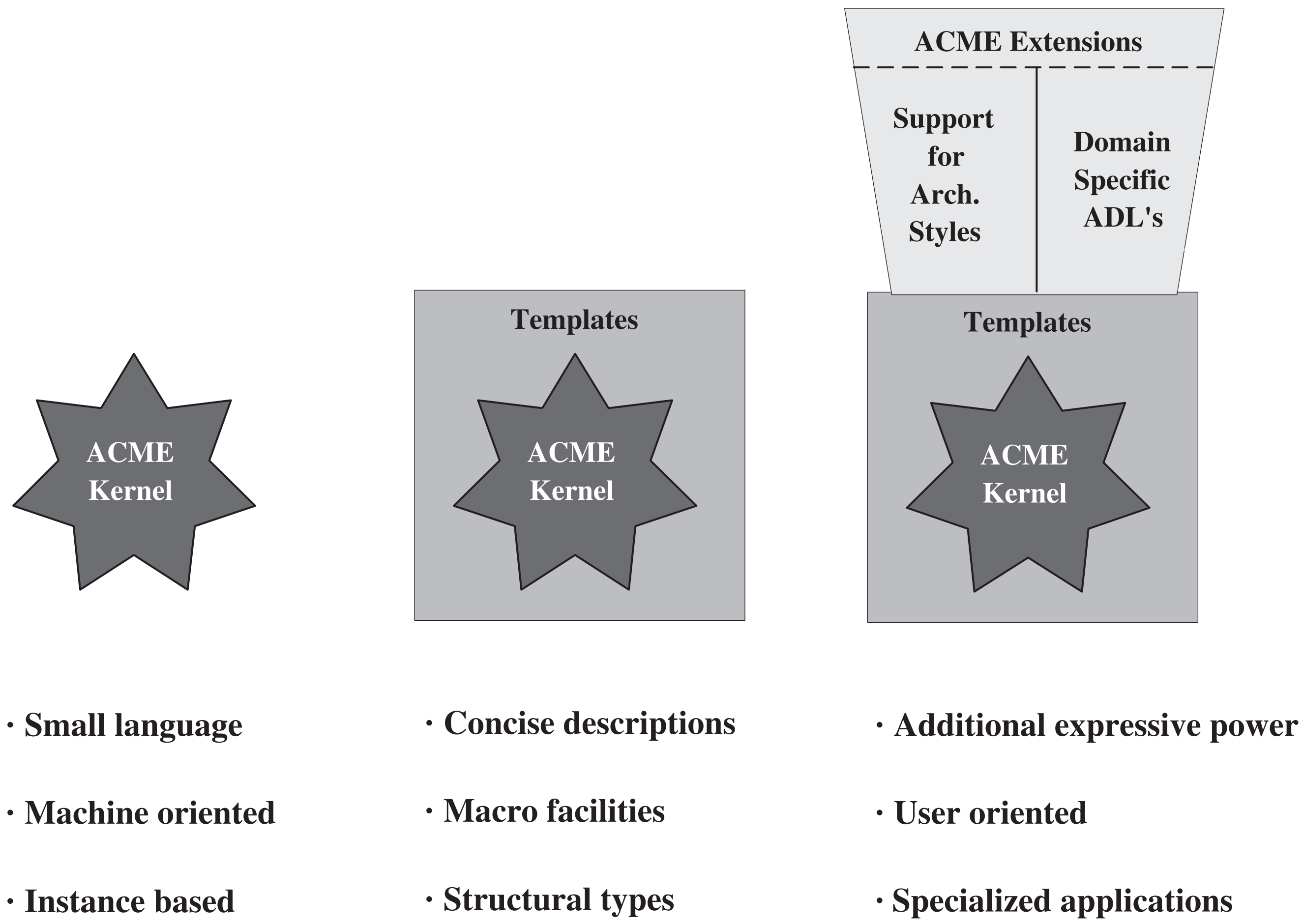

Figure 1: The ACME Family of Languages

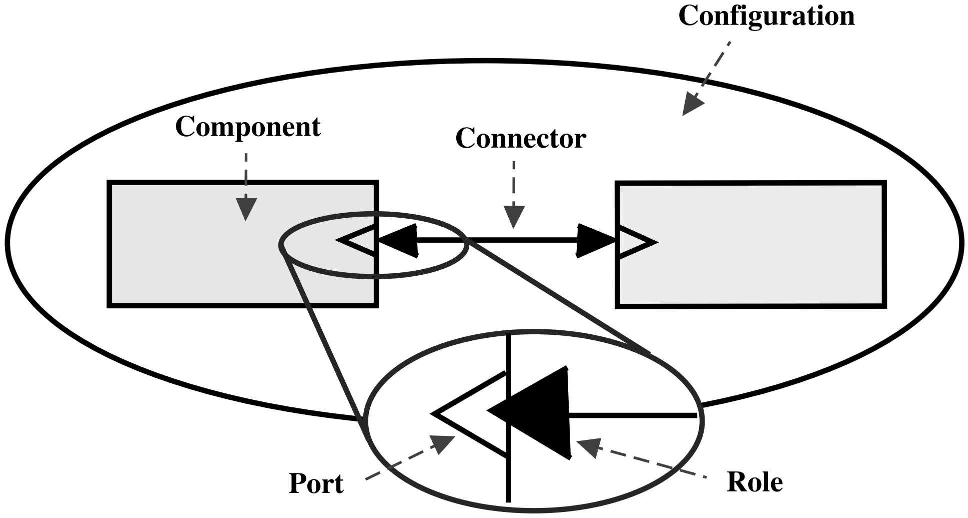

Figure 2: Elements of an ACME Description

descriptions significantly more compact and readable (although templates can complicate tool processing). To bridge the gap between the Full and Kernel languages we have built an "expander" tool that converts from the former to the later (see Section 6).

On top of Full ACME, we envision a variety of more specialized (and expressive) languages. Many of these may be domain-specific. For example, we are currently experimentating with a language that includes syntactic constructs to support pipe-filter architectures. Other extensions may incorporate additional generic support for characterizing architectural styles, specializing the representation of operational semantics for an architecture, etc.

We also envision that over time a certain amount of language specialization will occur around conventionalization of the use of annotations. As we detail in later sections, ACME permits the attribution of architectural structures with extra-structural information. Currently there is no established vocabulary for these properties or their semantics. As people develop tools to exploit architectural properties, we anticipate the emergence of standards for using the annotation facilities to characterize common information. As an example, we present an initial proposal for one such standard (for visualization properties) in Section D.

In the next three sections we will examine Kernel-ACME, ACME, and language extensions.

3 Kernel-ACME

In this section provide a quick overview of the Kernel-ACME language constructs. These provide a fixed vocabulary for the structural aspects of architectural definition, together with an open-ended annotation mechanism. After outlining the language constructs, we discuss the rationale behind their inclusion in the language. Finally, we illustrate the use of kernel with a simple example. The BNF for Kernel-ACME is included in Appendix E.

3.1 ACME Design Elements

ACME represents architectures using a generic ontology of seven entities: components, connectors, configurations, ports, roles, representations, and bindings. These are illustrated in Figures 2 and 3.

Of the seven entities, the most basic elements of architectural description are *components*, *connectors*, and *configurations*.

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

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

- *Configurations* represent topologies of components and connectors.

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

Connectors also have an interface defined by a set of *roles*. Each role of a connector defines a participant of the interaction represented by the connector. Binary connectors will have two roles. Examples include the *caller* and *callee* roles of a procedure call or RPC connector, the *reading* and *writing* roles of a pipe, or the *sender* and *receiver* roles of a message passing connector. Other kinds of connectors may have more than two roles. For example an event broadcast connector might have a single *event-announcer* role and an arbitrary number of *event-receiver* roles.

ACME permits the description of hierarchical architectural descriptions. That is, any component (or connector) can be represented by a one or more detailed or lower-level descriptions. Each such description is termed a *representation* in ACME.

ACME distinguishes between two general kinds of representations. One kind is simply a reference to an external file. These representations can be used to associate a piece of source code, a formal specification, or a system-build file with an architectural element.

The second kind of representation is one that is itself an architectural configuration. These representations are themselves described in ACME. For such representations there must be some way to indicate the correspondence between the internal configuration and the external interface of the component or connector that is being represented. A *binding* defines this correspondence: each binding identifies an internal port with an external port (or, for connectors, an internal role with an external role). Note that bindings are not connectors: connectors define paths of interaction, while bindings identify equivalences between two interface points. Moreover, connectors always associate a roles with a port, while a binding associates a port with another port, or a role with another role.

3.2 ACME Properties

The seven classes of design element outlined above are sufficient for defining the *structure* of an architecture as a hierarchical graph of components and connectors.

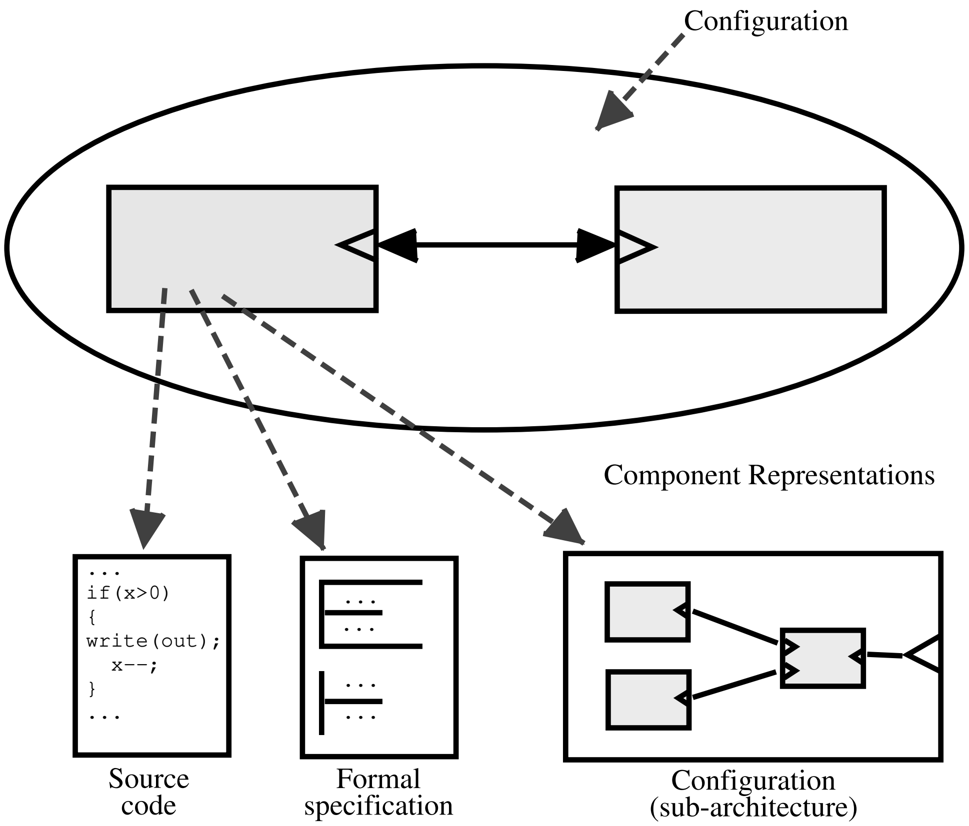

Figure 3: Multiple Representations of a Component

But there is more to architectural description than structure. As discussed earlier, each ADL typically has its own set of auxiliary information that determines such things as the run-time semantics of the system, detailed typing information (such as types of data communicated between components), protocols of interaction, scheduling constraints, and information about resource consumption.

To accommodate this auxiliary information ACME permits the annotation of the structural skeleton with arbitrary lists of properties. Each property has a name, an optional type, and a value. From ACME's point of view the properties are uninterpreted values. Properties become useful only when a tool makes use of them for analysis, translation, and manipulation.

In order for properties to be useful when interchanged between different tools there must be a common understanding of their meaning. ACME does not explicitly define those meanings, but it does allow for the shared use when those meanings exist. The default behavior of a tool that does not understand a specific property should be to ignore it. We anticipate that over time ACME will serve as vehicle for conventionalization of properties that are useful to more than one ADL.

As a preliminary example of such conventionalization, Appendix D contains a proposal for visualization specification standard that can be used by graphical editors to display architectural descriptions. These attributes are currently used by the ACME WEBifier tool, described in Section 6.

3.3 Discussion

To be written: Discussion of reasons for using the language choices, including:

- *the fact that connectors as first class elements in the language*
- *the distinction between connectors and bindings*
- *the potential/pitfalls of property lists*
- *the difference between representations and properties*
- *property space management and conventionalization*

3.4 Example

Let us now illustrate how Kernel-ACME might be used to represent architectures. Figure 4 contains an architectural description of a small signal processing system consisting of three components and two connectors.

This system might be represented in many different ways, depending on the ADL used to characterize it. In the Aesop System, the architecture would appear more or less as pictured in Figure 5 together with auxiliary textual specifications of the parts that could be displayed using the Aesop design browser tool. With the UniCon system the system might be displayed using a textual form shown in Appendix A.[2]

In addition to the structural description of the system, each ADL would typically associate other kinds of information with the architecture. For example, Aesop might classify the overall design as an instance of the PF Style, while UniCon might associate information that helps a compiler produce an executable system (such as the fact that the connectors should be implemented using Unix pipes).

[2]UniCon also has a graphical tool that might show a picture similar to the one illustrated here.

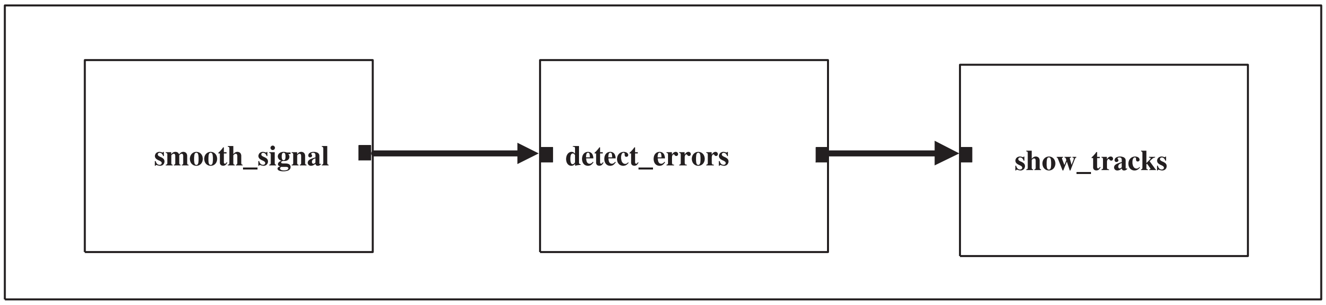

Figure 4: A simple pipe-filter system

To see how the system would be represented when translated to ACME, consider Figure 5. Let us consider the parts of this description separately.

To be written: Detailed discussion of the example.

4 Full ACME: Extending the kernel with templates

4.1 Templates

Kernel-ACME is a very simple, small language that was intended to be easy for tools to parse and unparse, while still giving a rich enough architectural vocabulary that software system architectures can be naturally represented.

But as is clear from the previous example, there is considerable redundancy to most Kernel-ACME descriptions. For example, each connector duplicates the description of "pipe-ness". This makes the expression of architectures verbose, and therefore difficult to read and write.

To address this problem the full ACME language includes *templates*. A template is a typed, parametrized macro facility that allows the specification of recurring patterns. These patterns are used (or instantiated) by applying them to the appropriate types of parameters. Additional properties may be added to an instantiated template using the *augment* construct.

4.2 The Example Revisited

To illustrate how templates simplify the description of an architecture, consider Figure 6, which recreates the earlier example using templates. (See also Appendix F for the BNF of the full language.)

To be written: Detailed discussion of the example.

5 Augmentations of ACME

To be written: Domain specific extensions of ACME, including:

- *The visual specification conventions*
- *The ACME PF specialization*

```
System kernel-radar-pipeline = Configuration {
  Components : {
    smooth_signal = Component {
      Ports : { stdin = Port { Properties : { datatype = char-stream } }
                stdout = Port { Properties : { datatype = char-stream } }}
      Representations : {External file : /lib/kalman.c}
      Properties : { Aesop-style = pf;
                     Aesop-datapath = "stdin to stdout";
                     UniCon-type = filter } }

    detect_errors = Component {
      Ports : { stdin = Port { Properties : { datatype = char-stream } }
                stdout = Port { Properties : { datatype = char-stream } }}
      Representations : {External file : /lib/find_error.c}
      Properties : { Aesop-style = pf;
                     Aesop-datapath = "stdin to stdout";
                     UniCon-type = filter } }

    show_tracks = Component {
      Ports : { stdin = Port { Properties : { datatype = char-stream } }
                stdout = Port { Properties : { datatype = char-stream } }}
      Representations : {External file : /lib/show_tracks.c}
      Properties : { Aesop-style = pf;
                     Aesop-datapath = "stdin to stdout";
                     UniCon-type = filter } }
  } /* end component declarations */


  Connectors : {
    p1 = Connector {
      Roles : { source = Role { Properties : { datatype = char-stream }}
                sink = Role { Properties : { datatype = char-stream }}}
      Properties : { Aesop-style = pf;
                     Aesop-datapath = "source to sink";
                     UniCon-type = pipe } }

    p2 = Connector {
      Roles : { source = Role { Properties : { datatype = char-stream }}
                sink = Role { Properties : { datatype = char-stream }}}
      Properties : { Aesop-style = pf;
                     Aesop-datapath = "source to sink";
                     UniCon-type = pipe } }
   }  /* end connector declarations */

  Attachments : {
     smooth_signal.stdout to p1.source;
     error_detect.stdin to p1.sink;
     error_detect.stdout to p2.source;
     show_tracks.stdin to p2.sink
  }
}
```

Figure 5: An ACME-kernel Description Simple System

```
/* Declare the templates to be used in the system */

Template Filter(reps : Representations) : Component =
  Component {
     Ports : { stdin = Port { Properties : { datatype = char-stream }}
               stdout = Port { Properties : { datatype = char-stream }}}

     Representations : { $reps }

     Properties : { Aesop-style = pf;
                    Aesop-datapath = "source to sink";
                    UniCon-type = filter }
}


Template Pipe() : Connector =
   Connector {
      Roles : { source = Role { Properties : { datatype = char-stream }}
                sink = Role { Properties : { datatype = char-stream }}}

      Properties : { Aesop-style = pf;
                     Aesop-datapath = "source to sink";
                     UniCon-type = pipe }
}


/* Describe the configuration of templates */

System template-radar-pipeline = Configuration {
   Components : {
        smooth_signal = Filter( {External file : /lib/Kalman_filter.c} );
        detect_errors = Filter( {External file : /lib/find_error.c} );
        show_tracks   = Filter( {External file : /lib/show_tracks.c} );
   }

   Connectors : {
        p1 = Pipe();
        p2 = Pipe();
   }

   Attachments : {
        smooth.stdout to p1.source;
        error_detect.stdin to p1.sink;
        error_detect.stdout to p2.source;
        show_tracks.stdin to p2.sink
   }
}
```

Figure 6: Description of System Using Templates

6 ACME Tools

To be written: Discussion of current ACME tools, including:

- *The ACME WEBifier*
- *The ACME Elaborator*
- *The ACME PF Animator*
- *ACME Analysis Tools*

7 Current Status and Plans

To be written: Discussion of expected development path and the issues to be considered in the future, including:

- *Conventionalization of A*
- *The ACME Elaborator*
- *The ACME PF Animator*
- *ACME Analysis Tools*

A UniCon Version of Simple Example

The following describes the example of Section 3.4 in the UniCon textual language.

```
CONNECTOR Unix-pipe
    PROTOCOL IS
        TYPE Pipe
        PIPETYPE (UNNAMED)
        ROLE source IS source
        ROLE sink IS sink
    END PROTOCOL

    IMPLEMENTATION IS
        BUILTIN
    END IMPLEMENTATION
END Unix-pipe


COMPONENT Smooth
    INTERFACE IS
        TYPE Filter
        PLAYER input IS StreamIn
            SIGNATURE ("line")
            PORTBINDING (stdin)
        END input
        PLAYER output IS StreamOut
            SIGNATURE ("line")
            PORTBINDING (stdout)
        END output
        PLAYER error IS StreamOut
            SIGNATURE ("line")
            PORTBINDING (stderr)
        END error
    END INTERFACE

    IMPLEMENTATION IS
        VARIANT c-code IN "/tmp/Kalman_filter.c"
            IMPLTYPE (Source)
        END c-code
    END IMPLEMENTATION
END Smooth


COMPONENT Detect-Errors
    INTERFACE IS
        TYPE Filter
        PLAYER input IS StreamIn
            SIGNATURE ("line")
            PORTBINDING (stdin)
        END input
        PLAYER output IS StreamOut
            SIGNATURE ("line")
            PORTBINDING (stdout)
```

```
        END output
        PLAYER error IS StreamOut
            SIGNATURE ("line")
            PORTBINDING (stderr)
        END error
    END INTERFACE

    IMPLEMENTATION IS
        VARIANT c-code IN "/tmp/find_error.c"
            IMPLTYPE (Source)
        END c-code
    END IMPLEMENTATION
END Detect-Errors


COMPONENT Show-Tracks
    INTERFACE IS
        TYPE Filter
        PLAYER input IS StreamIn
            SIGNATURE ("line")
            PORTBINDING (stdin)
        END input
        PLAYER output IS StreamOut
            SIGNATURE ("line")
            PORTBINDING (stdout)
        END output
        PLAYER error IS StreamOut
            SIGNATURE ("line")
            PORTBINDING (stderr)
        END error
    END INTERFACE

    IMPLEMENTATION IS
        VARIANT c-code IN "/tmp/show_tracks.c"
            IMPLTYPE (Source)
        END c-code
    END IMPLEMENTATION
END Show-Tracks


COMPONENT Pipeline
    INTERFACE IS
         TYPE General
    END INTERFACE

    IMPLEMENTATION IS
         USES smooth INTERFACE Smooth
         USES detect-errors INTERFACE Detect-Errors
         USES show-tracks INTERFACE Show-Tracks
         ESTABLISH Unix-pipe WITH
              smooth.output AS source
              detect-errors.input AS sink
         END Unix-pipe
```

```
        ESTABLISH Unix-pipe WITH
             detect-errors.output AS source
             show-tracks.input AS sink
        END Unix-pipe
    END IMPLEMENTATION
END Pipeline
```

B KWIC – Pipe-and-filter Solution

```
// The following is a pipe-and-filter architecture of a "kwic"
// system, described using the acme generic architecture interchange
// language (version 1.1).


//
// Set of templates to define common design elements
//

Template FilterInputPort () : Port =
   Port {
        Properties : { attaches-only-to = PipeSinkRole;
                       attachment-class = FilterInputPort }
   }


Template FilterOutputPort () : Port =
   Port {
        Properties : { attaches-only-to = PipeSourceRole;
                       attachment-class = FilterOutputPort }
   }


Template PipeSourceRole () : Role =
   Role {
        Properties : {  attaches-only-to = FilterOutputPort;
                        attachment-class = PipeSourceRole;
                        data-format = characters }
   }


Template PipeSinkRole () : Role =
   Role {
        Properties : {  attaches-only-to = FilterInputPort;
                        attachment-class = PipeSinkRole;
                        data-format = characters }
   }


Template Pipe () : Connector =
   Connector {
        Roles : { source = PipeSourceRole() ;
                  sink = PipeSinkRole()  }
        Properties : { blocking-type = non-blocking ;
                       Aesop-style = pipe-and-filter }
   }


Template Filter (port-list : Ports) : Component =
   Component {
```

```
        Ports : { $port-list }
        Properties : { Aesop-style = pipe-and-filter }
   }



// This is a simple definition of UnixFilter that does not use any
// inheritance from the previously defined filter.

Template UnixFilter () : Component =
   Component {
        Ports : { stdin = FilterInputPort() ;
                  stdout = FilterOutputPort() ;
                  stderr = FilterOutputPort()  }
        Properties { Aesop-style = pipe-and-filter ;
                     Aesop-style = unix-pipe-and-filter }
   }




// This second version of UnixFilter gives an idea for how we might be
// able to use some notions of inheritance by specializing existing
// templates.
//
// Both UnixFilter and UnixFilterInherited should return the same
// component.

Template UnixFilterInherited () : Component =
   Filter( stdin = FilterInputPort() ;
           stdout = FilterOutputPort() ;
           stderr = FilterOutputPort()
         )
        augment with {Properties : {Aesop-style = unix-pipe-and-filter}};



// Define the instantiation itself.
// Note that because the templates are standard for this style, they
// probably should have been included through a "#include" or "with"
// type mechanism, so that they can be stored in and used as a
// library.

System kwic_pf = configuration {
  Components : {
        input_medium = Component {
                        Ports : { system_input = Port { }  } } ;

        input = Filter( in = FilterInputPort(),
                        out = FilterOutputPort() ) ;

        circular_shift = Filter( in = FilterInputPort(),
                                 out = FilterOutputPort() ) ;
```

```
        alphabetizer = Filter(in = FilterInputPort(),
                              out = FilterOutputPort() );
          augment alphabetizer with {
                     Representations : { External : ~/alphabetize.c } };

        output = Filter( in = FilterInputPort(),
                         out = FilterOutputPort() ) ;

        output_medium = Component {
                         Ports : { system_output = Port {}  } ;
   }


  Connectors : {
        p1 = Pipe();
        p2 = Pipe();
        p3 = Pipe();

        system_in = Connector {
                        Roles : { source = Role {} ;
                                  sink   = PipeSinkRole()  }
        }

        system_out = Connector {
                        Roles : { source = PipeSourceRole() ;
                                  sink   = Role {}  }
        }
   }

  Attachments : {
        input_medium.system_input to input.in;
        input.out to circular_shift.in;
        circular_shift.out to alphabetizer.in;
        alphabetizer.out to output.in;
        output.out to output_medium.system_output;
   }

}
```

C KWIC – Shared Data Solution

```
// This file contains the "shared-data" architecture of a "kwic"
// system, described using the acme generic architecture interchange
// language (version 1.1).  This example is taken from Garlan and
// Shaw's Introduction to Software Architecture paper.


// Declare the templates for the different species of components,
// connectors, ports, and roles.

//
// Port templates
//
Template SubroutineCallPort () : Port =
   Port {
        Properties : { attaches-only-to = SubroutineCallRole;
                       attachment-class = SubroutineCallPort }
   }


Template SubroutineEntryPort () : Port =
   Port {
        Properties : { attaches-only-to = SubroutineEntryRole;
                       attachment-class = SubroutineEntryPort }
   }


Template DMAPort () : Port =
   Port {
        Properties : { attaches-only-to = DMARole;
                       attachment-class = DMAPort }
   }


Template SystemIOPort () : Port =
   Port {
        Properties : { attaches-only-to = SystemIORole;
                       attachment-class = SystemIOPort }
   }


//
// Role templates
//

Template SubroutineCallRole () : Role =
   Role {
        Properties : {  attaches-only-to = SubroutineCallPort;
                        attachment-class = SubroutineCallRole }
   }
```

```
Template SubroutineEntryRole () : Role =
   Role {
        Properties : {  attaches-only-to = SubroutineEntryPort;
                        attachment-class = SubroutineEntryRole }
   }


Template DMARole () : Role =
   Role {
        Properties : { attaches-only-to = DMAPort;
                       attachment-class = DMARole }
   }


Template SystemIORole () : Role =
   Role {
        Properties : { attaches-only-to = SystemIOPort;
                       attachment-class = SystemIORole }
   }


//
// Connector templates
//

Template SystemIO () : Connector =
   Connector {
        Roles : { source = SystemIORole() ;
                  sink = SystemIORole()  }
        Properties : { blocking-type = non-blocking ;
                       Aesop-style = Subroutine-call }
   }


Template SubroutineCall () : Connector =
   Connector {
        Roles : { caller = SubroutineCallRole() ;
                  callee = SubroutineEntryRole() }
        Properties : { blocking-type = blocking ;
                       Aesop-style = Subroutine-call }
   }


Template DMA () : Connector =
   Connector {
        Roles : { request = DMARole() ;  mem-entry = DMARole() }
        Properties : { blocking-type = blocking ;
                       Aesop-style = Subroutine-call }
   }


//
```

```
// Component Templates
//

Template Subroutine (port-list : Ports) : Component =
   Component {
        Ports : { $port-list  }
        Properties : { Aesop-style = Subroutine-call }
   }


Template DataStructure (dma-port-list : Ports) : Component =
   Component {
        Ports : { $dma-port-list }
   }


Template IOMedium ( io-port-list : Ports ) : Component =
   Component {
        Ports { $io-port-list }
   }


// Use inheritance to make the single-port input and output medium
// component templates.  In practice, this would probably usually be
// done in the declaration of the system because the templates are so
// simple and only used once.  They are done as templates here to show
// the ability to use an inheritance type mechanism for specializing
// the components.

Template SingleInputMedium () : Component =
   IOMedium(input-port = SystemIOPort())
      augment with { Properties : { IO-direction = input } };


Template SingleOutputMedium () : Component =
   IOMedium(output-port = SystemIOPort())
      augment with { Properties : { IO-direction = output } };


// Finished defining templates, now define the instantiation itself.

System kwic_shared_data = configuration {
  Components : {

    // IO based component declarations
        input_medium = SingleInputMedium();
        input = Subroutine( from_medium  = SystemIOPort(),
                            chars        = DMAPort(),
                            invoke       = SubroutineEntryPort() );

        output_meduium = SingleOutputMedium();
        output = Subroutine( to_medium  = SystemIOPort(),
                             chars      = DMAPort(),
```

```
                          alph_index = DMAPort(),
                          invoke     = SubroutineEntryPort() );

  // Manipulation components
      circular_shift = Subroutine( invoke = SubroutineEntryPort(),
                                   chars  = DMAPort(),
                                   index  = DMAPort() );

      alphabetizer =
              Subroutine( invoke = SubroutineEntryPort(),
                          chars  = DMAPort(),
                          index  = DMAPort(),
                          alph_index = DMAPort() );

  // Main controler component
      master_control = component {

              Ports : { call_input        = SubroutineCallPort(),
                        call_circ_shift   = SubroutineCallPort(),
                        call_alphabetizer = SubroutineCallPort(),
                        call_output       = SubroutineCallPort() }

              Reps : { External : "~/kwic_sd.c" }

              Properties :  { Initial-subroutine = true ;
                              Aesop-style = Subroutine-call }
      };


  // Data structure components
      characters = DataStructure(
                      input   = DMAPort(),
                      c_shift = DMAPort(),
                      alph    = DMAPort(),
                      output  = DMAPort() );

      index = DataStructure(
                      c_shift = DMAPort(),
                      alph    = DMAPort() );

      alph_index = DataStructure(
                      alph   = DMAPort(),
                      output = DMAPort() );


 }  // end components


Connectors : {
      // Master control invocations
      sc1 = SubroutineCall();
      sc2 = SubroutineCall();
      sc3 = SubroutineCall();
```

```
      sc4 = SubroutineCall();

      // software to IO medium connectors
      system_input  = SystemIOConn();
      system_output = SystemIOConn();

      // DMA connectors
      dma1 = DMA();
      dma2 = DMA();
      dma3 = DMA();
      dma4 = DMA();
      dma5 = DMA();
      dma6 = DMA();
      dma7 = DMA();
      dma8 = DMA();

 }

Attachments : {

      // subroutine invocations by master controller.
      master_controller.call_input to sc1.caller;
      input.invoke to sc1.callee;
      master_controller.call_circ_shift to sc2.caller;
      circular_shift.invoke to sc2.callee;
      master_controller.call_alphabetizer to sc3.caller;
      alphabetizer.invoke to sc3.callee;
      master_controller.call_output to sc4.caller;
      output.invoke to sc4.callee;

      // IO requests from input and output subroutines to appropriate
      // IO media
      input_medium.input to system_input.source;
      system_input.sink to input.from_medium;
      output_medium.output to system_output.sink;
      output.from_output to system_output.source;

      // Direct Memory Access connections.
      // characters
      input.chars to dma1.request;
      characters.input to dma1.mem_entry;
      circular_shift.chars to dma2.request;
      characters.c_shift to dma2.mem_entry;
      alphabetizer.chars to dma3.request;
      characters.alph to dma3.mem_entry;
      output.chars to dma4.request;
      characters.output to dma4.mem_entry;

      // index
      circular_shift.index to dma5.request;
      index.c_shift to dma5.mem_entry;
      alphabetizer.index to dma6.request;
      index.alph to dma6.mem_entry;
```

```
      // alphabetized index.
      alphabetizer.alph_index to dma7.request;
      alph_index.alph to dma7.mem_entry;
      output.alph_index to dma8.request;
      alph_index.output to dma8.mem_entry;

   } // end attachments

}  // end kwic_shared_data configuration.
```

D ACME Visual Specifications

Overview:

In order to produce graphical depictions of ACME architectures there needs to be a set of standard conventions for embedding graphical layout descriptions in ACME. As a starting point, we propose the following conventions for describing the graphical layout of an architectural diagram using ACME's property list construct.

Basic framework:

Graphical layout descriptions may be provided in an ACME description. Alternatively, they may be omitted, in which case it is the importing tool's responsibility to devise a rendering for the architecture, if it needs to do so. If an ACME description contains insufficient layout information to render graphically, the importing tool can simply display the textual ACME description.

Initially, ACME provides a core set of simple attributes for describing the visualization of a software architecture. These attributes are designed to render "box and line" diagrams. There are, of course, visualization schemas other than lines for representing connectors, (e.g. embedding boxes, adjacency, etc.). Although these other types of visualizations can generally be described with the primitives given below, the set of visualization primitives is clearly geared towards drawing box and line diagrams. Visualization information provided by an ACME description should be though of as a set of "suggestions" about how the importer should render the item. The importing tool can choose to ignore, override, or require specific information to render an architecture.

The basic graphical description hierarchy provides a standard set of visualization properties (enumerated below) for the basic types of design elements. All of these properties have a prefix of "vis-". If a specific tool (e.g. Aesop, Unicon, etc.) wants to encode more visualization information it is free to do so. These tools will be encouraged to use a naming standard that includes the name of the creating tool, followed by "-vis-". For example, if Aesop wants to write out that a thick outline should be provided for a particular element, it would provide that element with the property "Aesop-vis-outline-width : thick". When a tool reads in a description, it can choose to use or discard properties that it doesn't understand (like Aesop-vis-outline-width), and either use or override those that it does understand. It may be necessary to extend the set of generic "vis-*" properties as tools and visualizations evolve.

Standard Properties:

Each of the standard classes of design elements – components, connectors, ports, roles, and bindings – share the following properties.

```
Property    Value type and meaning
--------    ----------------------


vis-x:(int) (x,y) pixel position of the north-west (upper-left)
vis-y:(int) corner of bounding box for components, configurations,
    ports, and roles.  The bounding box surrounds the "knot,"
    or the centroid, in the case of connectors.

vis-width : (int)  width and height of the bounding box of an
vis-height :(int)  object, in pixels.  This specifies an offset "right"
```

```
   and "down" from the vis-xy location.  or the
   south-east (or lower right) corner of the bounding
   box as an offset from the (vis-x,vis-y) vector, in
   pixels.

Note: The vis-{x,y,width,height} properties have been broken up into
      individual properties to make the parsing easier.  It is not
      clear whether they should stay individual properties or be
      broken into two vectors.  I think that we want to at least keep
      the height and width (size) independent of the x,y (location)
      information so that one of the two properties can be defaulted
      without defaulting the other.


vis-shape:  (string)  This string describes the shape that is to be
      drawn within the bounding box.  Initially supported
      shapes will be (acceptable abbreviations follow in
      parentheses): rectangle (rect), oval, triangle (tri),
      rounded-edge rectangle (round-rect), line, arrow.

vis-color:  (string) This is a string describing the color in which the
      outline of the object is to be drawn.  Initially, this
      string will follow the X-Windows color naming
      convention.  If X can understand the color name, you can
      use it.  I'm not sure how this will translate to the
      Mac/PC world, or if there are any systems running on
      Mac/PC that we expect to use ACME.

vis-fill-color: (string) Like vis-color, except it describes what to
  fill the shape with.  Defaults to not filling the
  shape at all.
```

Guidelines for the semantics of these properties, with respect to each of the different types of design elements that can be displayed in an ACME graphic follow. Items listed as the defaults are suggested defaults. As mentioned earlier, importing tools can institute their own defaults for dealing with any of the properties:

Components: The pairs (vis-x,vis-y),(vis-width,vis-height) describe the bounding box in which the component's vis-shape will be drawn. The vis-shape property can take on any legal value specified above except for "line" or "arrow." vis-color and vis-fill-color inherit the general description given above. The default shape for a component is rectangle. The default colors are black outline, no fill. A further default decision is to display the name of the component in the center of the rectangle. Non-standard visualization properties can be specified to over-ride or adjust this behavior.

Configurations: A configuration need only supply the vis properties vis-width and vis-height, which specifies how large the bounding box for displaying the configuration will be. vis-shape defines the shape of the bounding box drawn inside the larger rectangular image, vis-x and vis-y are ignored. vis-color and vis-fill-color inherit the general behavior.

Connectors: As mentioned above, the vis-x, vis-y, vis-width, and vis-height fields describe the bounding box surrounding the "knot" in the center of the connector, *not* the connector's line itself. In the case of binary connectors, these fields are optional, as a line between the attached ports will be drawn without a knot. vis-shape describes the shape to use for the "knot." Specifying a vis-shape of "line" draws a line for the knot (like a bus diagram). vis-shape defaults to no shape for the knot and simply draws lines meeting at a center point.

Ports: The vis-x, vis-y coordinates for a port describe a relative offset from the north-west corner of the bounding box of the component to which the port is attached. As is standard, the port's vis-x, vis-y fields specify the northwest corner of the port's bounding box. Likewise, vis-width-height specifies the southwest corner of the port's bounding box as an offset from the port's vis-x, vis-y coordinates. The location is given as a relative value because a relative position makes it easier to move the component around while keeping its ports attached. In the case where a port is not attached to a component (as in a repository display), the vis-x and vis-y coordinate will be considered an absolute position. The semantics for all other vis-* properties of ports are inherited from the general case described above.

Roles: Unlike ports, the vis-x, vis-y coordinates for a role, if specified, will be an absolute position. If the role is attached to a port then the vis-x,vis-y properties are optional. By default, an attached port will be rendered over the port to which it is attached. If, however, an attached role specifies a vis-x, vis-y coordinate the specification overrides the default. If the role is unattached then it must provide the vis-x, vis-y properties to be rendered properly. All of the other vis-* properties inherit their semantics from the general case described above.

Bindings: Generic binding visualizations behave just like the ports or roles described above, with the exceptions that (1) they are rendered on the perimeter of the bounding box, (2) that a binding connects a role to a role, or a port to a port (visually), instead of connecting a port to a role in the attachment case, and (3) in the case of port-port bindings, a line needs to be drawn between the binding's box on the perimeter of the encompassing configuration's box, and the port to which it is bound. This line will be automatically generated and need not be specified explicitly. Of course, more specialized visualization properties and tools can override this default behavior. In the case of roles that are bound to a binding, the role to be bound will be drawn on the perimeter of the enclosing configuration, on top of the binding's glyph. The line from the bound role to the knot of it's connector will be automatically generated. All other vis-* properties relate to the binding's glyph and its position on the perimeter of the enclosing configuration's bounding box.

E Kernel-ACME BNF

```
 $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

  K E R N E L

 $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
NOTE

======================== ACME language inclusion: ===================
Kernel-ACME BNF, Version 1.2

Creation date: 4/19/95
Last revision: 12/2/95

========================================
The following BNF describes a grammar for specifying architectural
descriptions.

NOTE ARCHITECTURAL-DESCRIPTION is the initial non-terminal for
 describing a valid system. ;

ARCHITECTURAL-DESCRIPTION := { PROPERTY ^ '; } { '; } SYSTEM ;
SYSTEM :=  'System  (COMPONENT | CONNECTOR | PORT | ROLE | CONFIGURATION) ;

NOTE                  C O M P O N E N T S                    ;

COMPONENT := NAME '= 'Component '{
                     'Ports ': '{ PORT ^ '; { '; } '} { '; }
                     { 'Representations ':
                       '{ REPRESENTATION ^ '; { '; } '} { '; } }
                     { 'Properties ': '{ PROPERTY ^ '; { '; } '} { '; } }
            '} ;

NOTE                    C O N N E C T O R S                    ;

CONNECTOR := NAME '= 'Connector '{
                     'Roles ': '{ ROLE ^ '; { '; } '} { '; }
                     { 'Representations ':
                         '{ REPRESENTATION ^ '; { '; } '} { '; } }
                     { 'Properties ': '{ PROPERTY ^ '; { '; } '} { '; } }
                  '} ;

NOTE                       P O R T S                          ;


PORT := NAME '= 'Port '{
```

```
                        { 'Representations ':
                            '{ REPRESENTATION ^ '; { '; } '} { '; } }
                        { 'Properties ': '{ PROPERTY ^ '; { '; } '} { '; } }
                      '} ;

NOTE                     R O L E S                             ;
ROLE := NAME '= 'Role '{
                        { 'Representations ':
                            '{ REPRESENTATION ^ '; { '; } '} { '; } }
                        { 'Properties ': '{ PROPERTY ^ '; { '; } '} { '; } }
                       '} ;

NOTE              C O N F I G U R A T I O N S                          ;


CONFIGURATION := NAME '= 'Configuration '{
                       'Components ': '{ COMPONENT ^ '; { '; } '} { '; }
                       'Connectors ': '{ CONNECTOR ^ '; { '; } '} { '; }
                       'Attachments ': '{ ATTACHMENT ^ '; { '; } '} { '; }
                       { 'Properties ': '{ PROPERTY ^ '; { '; } '} {'; } }
                     '} ;

NOTE              M I S C E L L A N E O U S   P R O D U C T I O N S ;

REPRESENTATION :=  EXTERNAL-REP | CONFIGURATION-BINDINGS || ;
 EXTERNAL-REP := 'External { ': TYPE } '= ATT-VALUE |> VALID-REPRESENTATION ;

  PROPERTY :=  ATTRIBUTE { ': TYPE } '= ATT-VALUE ;

  ATTRIBUTE := NAME || ;
  ATT-VALUE :=  STRING | NAME || ;


ATTACHMENT := PORT-NAME 'to ROLE-NAME ;

NOTE This looks a bit strange;
CONFIGURATION-BINDINGS := CONFIGURATION BINDINGS ;

BINDINGS :=  'Bindings ': '{ BINDING ^ '; '} ;

BINDING :=  ROLE-BINDING | PORT-BINDING || ;
  ROLE-BINDING := ROLE-NAME#N 'to ROLE-NAME#V ;
  PORT-BINDING := PORT-NAME#N 'to PORT-NAME#V ;


ROLE-NAME :=  NAME || |> VALID-ROLE ;
```

```
PORT-NAME :=  NAME || |> VALID-PORT ;

NAME := LEXEME <| architectural-entity ;
STRING := LEXEME <| is-string
```

F Full ACME BNF

```
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

                F U L L      A C M E

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
 NOTE

 ======================== ACME language inclusion: ===================
 ACME BNF, version 1.2

 Last revision date: 12/2/95
  12/2/95 revisions (RTM):
    - Changed the 'Reps to 'Representations in the ELEMENT-LITERAL
       production (old bug)
    - Added properties to bindings.

 ========================================
 The following BNF describes a grammar for specifying architectural
 descriptions.

 Made more amenable to standard parser uses by making template
 invocation and specialization standard option to each constituent
 (role, connector, etc.).  We lose knowing whether the template or the
 specialization is relevant, but we can check that later in a second pass ;

 NOTE ARCHITECTURAL-DESCRIPTION is the initial non-terminal for
  describing a valid system. ;

 ARCHITECTURAL-DESCRIPTION :=  { TEMPLATE ^ '; } { '; }
                               { PROPERTY ^ '; } { '; } SYSTEM ;


 NOTE                    T E M P L A T E S                     ;


 TEMPLATE :=  'Template NAME '( { FORMAL-PARAMETER ^ ', } ')
       ': DEFINITION |> record-template-name ;

   DEFINITION := D-COMPONENT | D-CONNECTOR |
 D-PORT | D-ROLE |
 D-CONFIGURATION || ;
   D-COMPONENT := 'component '= COMPONENT-EXPR || ;
   D-CONNECTOR := 'connector '= CONNECTOR-EXPR || ;
   D-PORT := 'port '= PORT-EXPR || ;
   D-ROLE := 'role '= ROLE-EXPR || ;
```

```
  D-CONFIGURATION := 'configuration '= CONFIGURATION-EXPR || ;


NOTE The FORMAL-PARAMETER-LIST is a list of (identifier : type) pairs
  separated by commas. ;
  FORMAL-PARAMETER := NAME ': SYNTACTIC-CLASS ;

  SYNTACTIC-CLASS := ELEMENT-LITERAL | ELEMENT-LITERALS || ;
    ELEMENT-LITERAL := 'Component
| 'Connector
| 'Port
| 'Role
| 'Configuration ;
    ELEMENT-LITERALS :=  'Ports
| 'Roles
| 'Representations
| 'Properties ;

SYSTEM :=  'System
           (COMPONENT | CONNECTOR | PORT | ROLE | CONFIGURATION)
           { 'Properties ': '{ PROPERTY ^ '; { '; } '} { '; } } ;

NOTE  NAME may be a parameter name or a template name ;
  TEMPLATE-INVOCATION := NAME '( { ACTUAL-PARAMETER ^ ', } ') ;
  ACTUAL-PARAMETER := CONSTITUENT | ACTUAL-LIST || ;
  CONSTITUENT := O-COMPONENT | O-CONNECTOR | O-PORT | O-ROLE |
 O-CONFIGURATION |

 NAME || ;

  ACTUAL-LIST := REPRESENTATION-LIST | PROPERTY-LIST |
 CONSTITUENT-LIST || ;
  CONSTITUENT-LIST := '{ CONSTITUENT ^ '; '} ;

  REPRESENTATION-LIST := '{ REPRESENTATION ^ '; '} ;
  PROPERTY-LIST := '{ PROPERTY ^ '; '} ;
  CONFIG-OP := 'augment | 'override ;

  SPECIALIZED := TEMPLATE-INVOCATION | PARAMETER-REFERENCE | ALIAS || ;
  ALIAS := NAME || ;
  PARAMETER-REFERENCE := '$ NAME ;
  SPECIALIZATION := SPECIALIZATIONS | TEMPLATE-INVOCATION |
    PARAMETER-REFERENCE || ;
  SPECIALIZATIONS := SPECIALIZED SPECIFIC-SPECIALIZATION + ;

  SPECIFIC-SPECIALIZATION  := CONFIG-OP 'with DESCRIPTION ;
```

```
  DESCRIPTION := COMPONENT-DESCRIPTION |
 CONNECTOR-DESCRIPTION |
 PORT-DESCRIPTION |
 ROLE-DESCRIPTION |
 CONFIGURATION-DESCRIPTION || ;


NOTE                   C O M P O N E N T S                     ;



COMPONENT := O-COMPONENT | ALIAS-COMPONENT || ;
ALIAS-COMPONENT := NAME '= NAME#COMPONENT-ID ;
O-COMPONENT := NAME '= COMPONENT-EXPR ;
 COMPONENT-EXPR := R-COMPONENT | SPECIALIZATION || ;

  R-COMPONENT :=  'Component COMPONENT-DESCRIPTION |> HasPorts ;
  COMPONENT-DESCRIPTION := '{
     { 'Ports ': '{ EXPLICIT-PORT ^ '; { '; } '} { '; } }
     { 'Representations ':
'{ REPRESENTATION ^ '; { '; } '} { '; } }
     { 'Properties ': '{ PROPERTY ^ '; { '; } '} { '; } }
  '} ;

NOTE                     C O N N E C T O R S                      ;


CONNECTOR := O-CONNECTOR | ALIAS-CONNECTOR || ;
ALIAS-CONNECTOR := NAME '= NAME#CONNECTOR-ID ;
O-CONNECTOR := NAME '= CONNECTOR-EXPR ;
 CONNECTOR-EXPR := R-CONNECTOR | SPECIALIZATION || ;

  R-CONNECTOR :=  'Connector CONNECTOR-DESCRIPTION |> HasRoles ;
  CONNECTOR-DESCRIPTION :=  '{
     { 'Roles ': '{ EXPLICIT-ROLE ^ '; { '; } '} { '; } }
     { 'Representations ':
  '{ REPRESENTATION ^ '; { '; } '} { '; } }
     { 'Properties ': '{ PROPERTY ^ '; { '; } '} { '; } }
  '} ;

NOTE                        P O R T S                            ;


EXPLICIT-PORT := NAME { '= PORT-EXPR } || ;
PORT := O-PORT | ALIAS-PORT || ;
ALIAS-PORT := NAME '= NAME#PORT-ID ;
O-PORT := NAME '= PORT-EXPR ;
```

```
 PORT-EXPR := R-PORT | SPECIALIZATION || ;

  R-PORT :=  'Port PORT-DESCRIPTION ;
  PORT-DESCRIPTION := '{
{ 'Representations ':
    '{ REPRESENTATION ^ '; { '; } '} { '; } }
{ 'Properties ': '{ PROPERTY ^ '; { '; } '} { '; } }
      '} ;


NOTE                     R O L E S                               ;

EXPLICIT-ROLE := NAME { '= ROLE-EXPR } || ;
ROLE := O-ROLE | ALIAS-ROLE || ;
ALIAS-ROLE := NAME '= NAME#ROLE-ID ;
O-ROLE := NAME '= ROLE-EXPR ;
 ROLE-EXPR := R-ROLE | SPECIALIZATION || ;

  R-ROLE :=  'Role ROLE-DESCRIPTION ;
  ROLE-DESCRIPTION := '{
{ 'Representations ':
    '{ REPRESENTATION ^ '; { '; } '} { '; } }
{ 'Properties ': '{ PROPERTY ^ '; { '; } '} { '; } }
'} ;


NOTE              C O N F I G U R A T I O N S                        ;


CONFIGURATION := O-CONFIGURATION | ALIAS-CONFIGURATION || ;
ALIAS-CONFIGURATION := NAME '= NAME#CONFIGURATION-ID ;
O-CONFIGURATION := NAME '= CONFIGURATION-EXPR ;
 CONFIGURATION-EXPR := R-CONFIGURATION | SPECIALIZATION || ;

  R-CONFIGURATION := 'Configuration CONFIGURATION-DESCRIPTION |> HasCCA ;
  CONFIGURATION-DESCRIPTION := '{
{ 'Components ': '{ COMPONENT ^ '; { '; } '} { '; } }
{ 'Connectors ': '{ CONNECTOR ^ '; { '; } '} { '; } }
{ 'Attachments ': '{ ATTACHMENT ^ '; { '; } '} { '; } }
{ 'Properties ': '{ PROPERTY ^ '; { '; } '} {'; } }
      '} ;


NOTE              M I S C E L L A N E O U S   P R O D U C T I O N S ;


REPRESENTATION :=  EXTERNAL-REP | CONFIGURATION-BINDINGS || ;
```

```
EXTERNAL-REP := 'External { ': TYPE } '= ATT-VALUE |> VALID-REPRESENTATION ;

 PROPERTY :=  ATTRIBUTE { ': TYPE } '= ATT-VALUE ;

  ATTRIBUTE := NAME || ;
  ATT-VALUE :=  STRING | NAME || ;


ATTACHMENT := PORT-NAME 'to ROLE-NAME ;

CONFIGURATION-BINDINGS := CONFIGURATION BINDINGS ;
BINDINGS :=  'Bindings ': '{ BINDING ^ '; '} ;

BINDING :=  ROLE-BINDING | PORT-BINDING || ;
  ROLE-BINDING := ROLE-NAME#N 'to ROLE-NAME#V
           { 'Properties ': '{ PROPERTY ^ '; { '; } '} { '; } } ;
  PORT-BINDING := PORT-NAME#N 'to PORT-NAME#V
           { 'Properties ': '{ PROPERTY ^ '; { '; } '} { '; } } ;

ROLE-NAME :=  NAME || |> VALID-ROLE ;

PORT-NAME :=  NAME || |> VALID-PORT ;

NAME := LEXEME <| architectural-entity ;
STRING := LEXEME <| is-string
```

G Kernel-ACME AST in C++

```
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
{class ARCHITECTURAL-DESCRIPTION;class SYSTEM;class COMPONENT;class CONNECTOR;
 class PORT;class ROLE;class CONFIGURATION;class REPRESENTATION;
 class EXTERNAL-REP;class PROPERTY;class ATTRIBUTE;class ATT-VALUE;
 class ATTACHMENT;class CONFIGURATION-BINDINGS;class BINDINGS;class BINDING;
 class ROLE-BINDING;class PORT-BINDING;class ROLE-NAME;class PORT-NAME;
 class name;class STRING;

 class name:public ATTRIBUTE,public ATT-VALUE,public ROLE-NAME,public PORT-NAME
 {};
 class PORT-BINDING:public BINDING{};
 class ROLE-BINDING:public BINDING{};
 class CONFIGURATION-BINDINGS:public REPRESENTATION{};
 class EXTERNAL-REP:public REPRESENTATION{};
 class ARCHITECTURAL-DESCRIPTION{public:SYSTEM * system;};
 class SYSTEM{public:CONFIGURATION * configuration;ROLE * role;PORT * port;
             CONNECTOR * connector;COMPONENT * component;};
 class COMPONENT{public:RWTPtrSlist<PROPERTY> * property;
                RWTPtrSlist<REPRESENTATION> * representation;
                RWTPtrSlist<PORT> * port;name * name;};
 class CONNECTOR{public:RWTPtrSlist<PROPERTY> * property;
                RWTPtrSlist<REPRESENTATION> * representation;
                RWTPtrSlist<ROLE> * role;name * name;};
 class PORT{public:RWTPtrSlist<PROPERTY> * property;
           RWTPtrSlist<REPRESENTATION> * representation;name * name;};
 class ROLE{public:RWTPtrSlist<PROPERTY> * property;
           RWTPtrSlist<REPRESENTATION> * representation;name * name;};
 class CONFIGURATION{public:RWTPtrSlist<PROPERTY> * property;
                    RWTPtrSlist<ATTACHMENT> * attachment;
                    RWTPtrSlist<CONNECTOR> * connector;
                    RWTPtrSlist<COMPONENT> * component;name * name;};
 class EXTERNAL-REP{public:ATT-VALUE * att-value;};
 class PROPERTY{public:ATT-VALUE * att-value;ATTRIBUTE * attribute;};
 class ATTACHMENT{public:ROLE-NAME * role-name;PORT-NAME * port-name;};
 class CONFIGURATION-BINDINGS{public:BINDINGS * bindings;
                             CONFIGURATION * configuration;};
 class BINDINGS{public:RWTPtrSlist<BINDING> * binding;};
 class ROLE-BINDING{public:ROLE-NAME * v;ROLE-NAME * n;};
 class PORT-BINDING{public:PORT-NAME * v;PORT-NAME * n;};
 class name{public:Token lexeme;};
 class STRING:public ATT-VALUE{public:Token lexeme;};}
```

References

[AG94] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994.

[BV93] Pam Binns and Steve Vestal. Formal real-time architecture specification and analysis. In *Tenth IEEE Workshop on Real-Time Operating Systems and Software*, New York, NY, May 1993.

[CS93] L. Coglianese and R. Szymanski. DSSA-ADAGE: An Environment for Architecture-based Avionics Development. In *Proceedings of AGARD'93*, May 1993.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 179–185. ACM Press, December 1994.

[Gar95] David Garlan, editor. *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, WA, April 1995. Published as CMU Technical Report CMU-CS-95-151, April 1995.

[GP95] David Garlan and Dewayne Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company. Also appears as SCS and SEI technical reports: CMU-CS-94-166, CMU/SEI-94-TR-21, ESC-TR-94-021.

[LAK+95] David C Luckham, Lary M. Augustin, John J. Kenney, James Veera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):336–355, April 1995.

[MQ94] M. Moriconi and X. Qian. Correctness and composition of software architectures. In *Proceedings of ACM SIGSOFT'94: Symposium on Foundations of Software Engineering*, pages 164–174, New Orleans, Louisiana, December 1994.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[SDK+95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):314–335, April 1995.

[T+94] Allan Terry et al. Overview of teknowledge's domain-specific software architecture program. *ACM SIGSOFT Software Engineering Notes*, 19(4):68–76, October 1994.