Acme: An Architecture Description Interchange Language

David Garlan* Robert Monroe* David Wile**

Computer Science Department* USC/Inf. Sciences Institute**
Carnegie Mellon University 4676 Admiralty Way
Pittsburgh, PA 15217 USA Marina del Rey, CA 90292 USA

Proceedings of CASCON'97, November, 1997.

Abstract

Numerous architectural description languages (ADLs) have been developed, each providing complementary capabilities for architectural development and analysis. Unfortunately, each ADL and supporting toolset operates in isolation, making it difficult to integrate those tools and share architectural descriptions. Acme is being developed as a joint effort of the software architecture research community as a common interchange format for architecture design tools. Acme provides a structural framework for characterizing architectures, together with annotation facilities for additional ADL-specific information. This scheme permits subsets of ADL tools to share architectural information that is jointly understood, while tolerating the presence of information that falls outside their common vocabulary. In this paper we describe Acme's key features, rationale, and technical innovations.

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 [7, 8, 13].

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.

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 "Architecture Description Languages" (ADLs), these notations usually provide both a conceptual framework and a concrete syntax for characterizing 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, Meta-H, C2, Rapide, SADL, UniCon, and Wright [5, 3, 2, 11, 10, 12, 14, 1]. Although all of these languages are concerned with architectural design, each provides certain distinctive capabilities: Aesop supports the use of architectural styles; Adage supports the description of architectural frameworks for avionics navigation and guidance; Meta-H provides specific guidance for designers of real-time avionics control software; C2 supports the description of user interface systems using a message-based style; Rapide allows architectural designs to be simulated, and has tools for analyzing the results of those simulations; SADL provides a formal basis for architectural refinement; UniCon has a high-level compiler for architectural designs that support a mixture of heterogeneous component and connector types; Wright supports the specification and analysis of interactions between architectural components.

The proliferation of ADLs and their supporting toolsets[1] 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 roles that architectural description can play in software development. 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 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. Currently, however, adopting an existing ADL requires a substantial investment to install the ADL tools and learn to use them effectively, along with a significant "lock-in" to the selected ADL.

One way to ameliorate these problems would be to provide an *interchange* language for software architecture. Ideally, 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 generic, ADL-neutral structural analyses, allowing tool writers to develop architectural analysis tools that are compatible with multiple ADL's. Further, it could clarify the relationship between different ADLs and the analyses that they provide.

Acme is an architecture description language with precisely those goals. It is being developed as a joint effort of the software architecture research community 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 while also allowing the incorporation of ADL-specific information, so that auxiliary information can be retained. This 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 paper we describe the main features of Acme, its rationale, and technical innovations. While Acme is still too new to tell whether it will succeed as a community-wide tool for architectural interchange, we believe it is important to expose its language design and philos-

[1] In the remainder of this paper we will simply use the term "ADL" to refer to both the language and its supporting toolset.

ophy to the broader software engineering community at this stage for feedback and critical discussion. To do this we will focus primarily on the key design choices made by the language.

2 Language Rationale

2.1 Goals

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

The primary purpose of Acme is to provide an interchange format for architectural development tools and environments. As such, the language should make it possible to integrate a broad variety of separately-developed ADL tools by providing an intermediate form for exchanging architectural information.

In addition to its primary goal of interchange, Acme was designed with the following secondary goals in mind. These goals are listed in decreasing order of importance.

- *To provide a representational scheme that will permit the development of new tools for analyzing and visualizing architectural structures.* 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.
- *To provide a foundation for developing new, possibly domain-specific, ADLs.* The language should not preempt the ability to build on its core capabilities with additional constructs and semantics.
- *To serve as a vehicle for creating conventions and standards for architectural information.* The language should make it easy for groups of ADL developers to standardize aspects of architectural specification that are not explicitly included in Acme
- *To provide expressive descriptions that are easy for humans to read and write.* The language should allow compact, direct expression of architectural structures and idioms.

While these goals are complementary, taken individually they lead to quite different choices in design. In particular, the primary goal of supporting interchange of software architecture descriptions between different ADLs argues for a simple, easy-to-parse language, while the secondary goal of ease of reading and writing for humans argues for expressive language features. In the remainder of this paper we will see how Acme attempts to achieve the main goal of architectural description interchange while accommodating the secondary goals.

2.2 Reconciling Standardization and Diversity

The existence of multiple languages arises in numerous other domains including document formatting, programming, graphical encodings, and hypertext. As with ADLs, such diversity creates problems for users of these languages. 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 least 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 appealing. 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 architecture description languages.

First, an examination of existing 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 [4] was that virtually all ADLs take as their starting point the need to express an architectural design 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, and processor allocations. 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. Both Aesop and SADL are concerned with the expression of stylistic invariants. Aesop, UniCon, and Meta-H all 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 also 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 the software architecture community develops a better understanding of the value of certain classes of architectural information, representation conventions will emerge that can be used by the interchange language.

This is the essence of Acme. The language provides a fixed vocabulary (or ontology) for representing architectural structure. Additionally it provides an *open semantic framework* in which architectural structures can be annotated with ADL-specific properties. In this way Acme achieves the benefits of 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.

3 Acme

We now describe Acme, highlighting its key features with a small illustrative example.[2] These

[2] We will stress the main ideas behind the language design. Additional details and examples can be found in [6].

key features are:

1. an *architectural ontology* consisting of seven basic architectural design elements;
2. a flexible *annotation mechanism* supporting association of non-structural information using externally defined sublanguages;
3. a *template mechanism* for abstracting common, reusable architectural idioms and styles; and
4. an *open semantic framework* for reasoning about architectural descriptions.

3.1 Acme Architectural Design Element Types

Acme is built on a core ontology of seven types of entities for architectural representation: *components, connectors, systems, ports, roles, representations, and rep-maps.* These are illustrated in Figures 3 and 4.

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

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

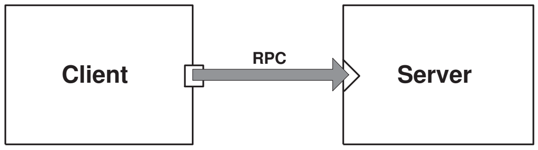

Figure 1: Simple Client-Server Diagram

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

Figure 2: Simple Client-Server System in Acme

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

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

As a simple illustrative example, Figure 1 shows a trivial architectural drawing containing a client and server component, connected by an RPC connector. Figure 2 contains its Acme description. The *client* component is declared to have a single *send-request* port, and the server has a single *receive-request* port. The connector has two roles designated *caller* and *callee*. The topology of this system is declared by listing a set of *attachments*.

Acme supports the hierarchical description of architectures. Specifically, any component or connector can be represented by one or more detailed, lower-level descriptions. (See Figure 4.) Each such description is termed a *representation* in Acme. The use of multiple representations allows Acme to encode multiple views of architectural entities (although there is nothing built into Acme that supports resolution of inter-view correspondences). It also supports the description of encapsulation boundaries, as well as multiple refinement levels.

When a component or connector has an architectural representation there must be some way to indicate the correspondence between the internal system representation and the external interface of the component or connector that is being represented. A *rep-map* (short for "representation map") defines this correspondence. In the simplest case a rep-map provides only an association between internal ports and external ports (or, for connectors, internal roles and external roles).[3] In other cases the map may be considerably more complex. For those cases the rep-map is essentially a tool-interpretable placeholder—similar to the use of properties described in the following section.

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.

But there is clearly more to architectural description than structure. As discussed earlier, currently there is little consensus about exactly what should be added to the structural information: 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 the wide variety of auxiliary information Acme supports annotation of architectural structure with lists of properties. Each property has a name, an optional type, and a value. Any of the seven kinds of Acme architectural design entities can be annotated. Figure 4 shows several properties attached to a hypothetical architecture.

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 Acme the "type" of a property indicates a "sublanguage" with which the property is specified. Acme itself predefines simple types such as integer, string, and boolean. Other types must be interpreted by tools: these tools use the "name" and "type" indicator to figure out whether the value is one that they can process. The default behavior of a tool that does not understand a specific property or property type should be to leave it uninterpreted but preserve it for use by other tools. This is facilitated by requiring standard property delimiter syntax so that a tool can know the extent of a property without having to interpret its contents.

Figure 5 shows the simple client-server system elaborated with several properties. For example, several of the properties indicate how the elements relate to constructs in target ADLs—such as Aesop and UniCon styles. Likewise, the "protocol" property of the RPC connector is declared to be in the "Wright" language and would only be meaningful to a tool that knows how to process that language. (For simplicity we have elided the specification: see [1] for details.)

Of course, in order for properties to be useful when interchanged between different ADLs, there must be a common understanding of their meaning. As we have noted, Acme does not explicitly define those meanings, but it does allow for the shared use of properties when those meanings do exist. We anticipate that over time Acme will serve as a vehicle for conventionalization of properties that are useful to more than one ADL.

[3]Note that rep-maps are not connectors: connectors define paths of interaction, while rep-maps identify an abstraction relationship between sets of interface points.

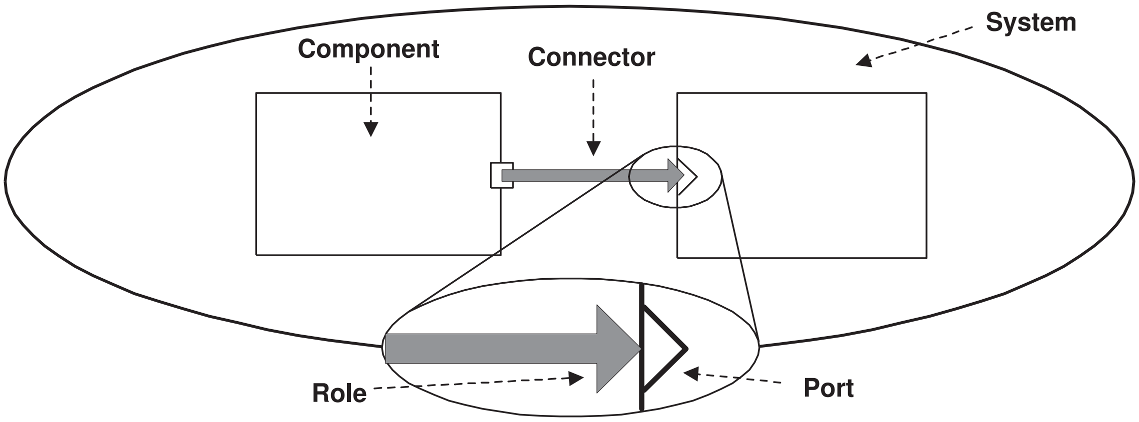

Figure 3: Elements of an Acme Description

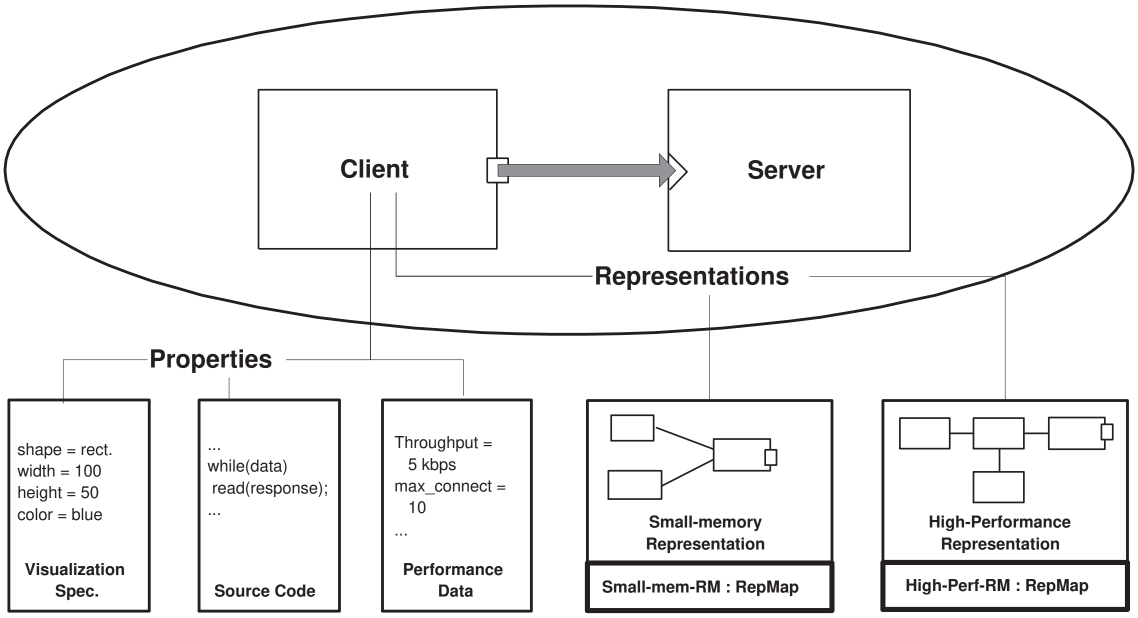

Figure 4: Representations and Properties of a Component

```
System simple_cs = {
  Component client = {
        Port send-request;
        Properties { Aesop-style : style-id = client-server;
                     UniCon-style : style-id = cs;
                     source-code : external = "CODE-LIB/client.c" }}

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

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

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

Figure 5: Client-Server System with Properties in Acme

Several property sublanguages are currently being developed. One is a standard for specifying visualization properties to be used by graphical editors to display architectural descriptions. Another sublanguage is being developed to describe temporal constraints on an architectural description. Details of these sublanguages are beyond the scope of this report, but can be found in [6].

3.3 Acme Templates and Style Definition

The Acme features described thus far are sufficient to define an architectural instance, and, in fact, form the basis for the core capabilities of Acme parsing and unparsing tools. As a representation that is good for humans to read and write, however, these features leave much to be desired. Specifically, they provide no facilities for abstracting architectural structure. As a result, common structures in complex system descriptions need to be repeatedly specified. Consider, for example, extending the simple client-server system described in Figure 5 to include multiple clients and multiple servers. Although there is significant common structure underlying each of the clients and servers in the design, the language facilities presented thus far would require the architect to explicitly specify this structure for each design element.

To address this problem the Acme language includes *templates*, a typed, parametrized macro facility for specification of recurring patterns. These patterns are used (or instantiated) by applying them to the appropriate types of parameters. Templates define syntactic structures that can be expanded in place to produce new declarations. They are quite flexible, permitting the definition of attachments as well as individual components and connectors.

The utility of templates is further extended when they are grouped into collections of architectural *styles*. In Acme, a style defines a set of related templates that make up the common vocabulary of a family of systems. Styles provide a mechanism for capturing and reusing common structures and idioms in architectural design.

Figure 6 illustrates the use of a *client-server style*, which defines *client*, *server* and *rpc* templates. This example extends the simple client-server example of Figure 5 by turning the client and server specifications into templates, and declaring a system instance with multiple clients and multiple servers. As illustrated, the client and server templates are straightforward constructs that create a new client or server component with the set of ports passed as an actual parameter. The *rpc* template is slightly more sophisticated than the client and server templates in that it not only declares an rpc connector but also attaches a client to a server. The "defining(conn:Connector)" clause indicates that a unique identifier *conn* needs to be generated each time this template is expanded. As a result of the defining clause it is possible to refer to the newly created connector within the template's body, as is required for the template to attach the new connector to the passed in components. Use of this style leads to concise descriptions of architectures and permits the explicit delineation of reusable architectural structures.

Although Acme is not intended to be a full-fledged ADL, the addition of templates and styles greatly enhance the readability and abstraction capabilities of the language. Both templates and styles can, however, be eliminated by direct expansion. This permits Acme tools to translate any Acme description into the more primitive core language for straightforward interchange. As a result, Acme is able to satisfy the secondary goals of readability and support for abstraction without compromising Acme's primary goal of supporting the interchange of software architecture descriptions between heterogeneous ADL's.

3.4 Acme's Open Semantic Framework

Acme is primarily concerned with the architectural structure of systems, and hence does not embody specific computational semantics for architectures. Rather, Acme relies on an open semantic framework that provides a basic structural semantics while allowing specific ADLs to associate computational or run-time behavior with architectures using the property construct.

The open semantic framework provides a straightforward mapping of the structural aspects of the language into a logical formalism based on relations and constraints. In this framework, an Acme specification represents a derived predicate, called its prescription. This predicate can be reasoned about using logic or it can be compared for fidelity with real world artifacts that the specification is intended to describe.

To illustrate, consider the simple client-server example architecture specification of Figures 1 and 2, where a client is linked to a server through a single connector. This system has the following prescription:

```
exists  client, server, rpc |
  component(client) ^
  component(server) ^
  connector(rpc) ^
  attached(client.send-request,
           rpc.caller) ^
  attached(server.receive-request,
           rpc.callee)
```

These predicates can be reasoned about using standard first-order logical machinery. They can also be used as the formal specification of an implementation. (In this case, it requires that one be able to find the artifacts *client* and *server* that purport to be components, a connector artifact *rpc*, and attachments that are specified by the predicate.)

This simple translation scheme is, however, not quite sufficient. Two implicit aspects of the specification also need to be included in the prescription: the first is the closed world assumption which states that all components, connectors, ports and roles have been identified by the existential variables in the specification, all attachments have been specified, and that no more exist; Second, the existential variables must refer to distinct entities. With these additions, the example's prescription reads:

```
exists  client, server, rpc |
  component(client) ^
```

```
Style client-server = {

  Component Template client(rpc-call-ports : Ports) = {
      Ports rpc-call-ports;
      Properties { Aesop-style : style-id = client-server;
                   Unicon-style : style-id = cs;
                   source-code : external = "CODE-LIB/client.c" }}

  Component Template server(rpc-receive-ports : Ports) = {
      Ports rpc-receive-ports;
      Properties { Aesop-style : style-id = client-server;
                   Unicon-style : style-id = cs; ... }}

  Template rpc(caller_port, callee_port : Port) defining (conn : Connector) =
   { conn = Connector {
        Roles {caller, callee}
        Properties { synchronous : boolean = true;
                     max-roles : integer = 2; }
                     protocol : Wright = "..." }}
     Attachments { conn.caller to caller_port;
                   conn.callee to callee_port; }}
}

System complex_cs : client-server = {
  c1 = client(send-request);   c2 = client(send-request);
  c3 = client(send-request);   s1 = server(receive-request);
  s2 = server(receive-request);

  rpc(c1.send-request, s1.receive-request);
  rpc(c2.send-request, s1.receive-request);
  rpc(c3.send-request, s2.receive-request);
}
```

Figure 6: Client-Server System Using Templates and Style

```
  component(server) ^
  connector(rpc) ^
  attached(client.send-request,
           rpc.caller) ^
  attached(server.receive-request,
           rpc.callee) ^
  client != server ^
  server != rpc ^
  client != rpc ^
  (for all y:component (y) =>
            y = client | y = server) ^
  (for all y:connector(y) => y = rpc) ^
  (for all p,q: attached(p,q) =>
     (p=client.send-request ^
      q=rpc.caller)
   | (p=server.receive-request ^
      q=rpc.callee))
```

In addition to basic structural information, properties also need to be handled. Property names correspond to predicates that take the entity to which the property applies as an argument and return the value of that property name for the given entity. The values of properties are treated as primitive atoms, without their own semantics. So, for example,

```
Component client = {
  Port send-request;
  Properties {
    Aesop-style : style-id = client-server;
    UniCon-style : style-id = cs} }
```

adds to the prescription the clauses:

```
Aesop-style(client) = client-server ^
Unicon-style(client) = cs
```

Although the value of a property is considered an atomic entity in terms of Acme's structural semantics, tools that manipulate and analyze Acme descriptions can interpret the property values as needed. An example of this approach is the protocol property of the RPC connector specified in the "Wright" sublanguage in example 5.

```
Connector rpc  = {
  Roles {caller, callee}
  Property protocol : Wright = "..."; }
```

Tools that don't understand the meaning of the Wright sublanguage can ignore the value of this property, processing it as an uninterpreted string. Tools that do understand the Wright sublanguage can interpret the value of the protocol specification to discover more detailed ADL-specific semantics of the connector.

4 Discussion

Returning to the language design goals enumerated earlier, we can now see how Acme attempts to reconcile the competing goals for the language.

Acme addresses its primary goal—the need for an ADL interchange format—by providing an extremely simple basis for architectural representation. Essentially, any tool that can handle the seven basic architectural element types (components, connectors, etc.) can interact with other architectural tools. The simplicity of the structural core of Acme (i.e. Acme without templates and styles) is reflected in the fact that its BNF occupies only a single page. For most ADLs it is trivial to write a parser and an unparser for that core language. Moreover, any architectural description using the more expressive capabilities of templates can be automatically translated into the simpler core language. Of course, the more a given tool can take advantage of property annotations, the more it can do with the descriptions (in the form of analysis, code generation, transformation, etc.).

Despite its simplicity, however, Acme provides a non-trivial basis for architectural representation and analysis—addressing the second goal. Three features contribute to this. First is the use of explicit connectors. This permits new architectural glue to be defined, and elevates Acme above typical module interconnection languages in which only a small set of connector types (usually procedure call and shared variables) are supported. Second is the use of multiple representations. As noted earlier, this permits the encoding of multiple views, refinement relations, and simple encapsulation schemes. Third is the use of templates and styles for encapsulating reusable patterns and idioms.

With respect to analysis, it is worth commenting here on Acme's type system. Acme provides a fixed set of types, including the seven basic architectural types (component, connector, etc.) and simple property types (integer, boolean, string). Within this set Acme supports a strong typing discipline. (For example, the actual and formal parameters of a template must agree.) However, Acme does not treat templates as type constructors themselves. So, for example, a template that creates a pipe connector does not actually introduce a new type of connector—rather it provides shorthand for creating a standard connector, endowed with pipe features (e.g., input and output roles).

The decision to use this relatively weak type system was based primarily on methodological considerations. In general, for architectural description it is more important that the parts have the right structure (and properties) than that they are declared using a particular set of forms. So for example, if I create an architectural description in which all of the connectors "look like" pipes, I should be able to use it in all of the contexts that I could have used the same description declared with a pipe constructor template. This increases the flexibility of the language, but at the cost of requiring analysis tools to do the checking that a type system would otherwise have provided for free.

The third goal for Acme—providing a foundation for new ADL development—is supported in three ways. First, the core constructs of Acme provide a baseline for architectural description that are a good starting point for almost any ADL. Second, the template mechanism permits the packaging of common syntactic forms; specific ADLs can be defined simply by fixing a set of template libraries (or styles) and then restricting developers to those forms. Third, Acme's open semantic framework does not preempt the development of more detailed ADL-specific semantics. By binding very few decisions about the computational semantics of an Acme description, language designers who build new ADLs on top of Acme can supply those ADLs with whatever semantic model is appropriate for the extended language.

Acme's fourth goal is to serve as a vehicle for conventionalization about standards for architectural information. This is supported by the property mechanism, which permits the use of new sublanguages for property types and values. Although the success of this approach will depend on the willingness of the architectural community to build consensus around common properties that many ADLs are capable of handling, early indications are that this is already happening.

It is worth emphasizing that this goal sets Acme apart from most other language design efforts. Typically a language is designed as a fully-formed, complete artifact. It is then presented to a community of users, who will either adopt it or not. Acme is different. It recognizes that architectural representation is an evolving, multi-faced target. Rather than attempting to completely pin the target down, Acme instead provides the context in which interested parties can participate in developing standards and conventions for representing and analyzing architectural information centered around a shared core of basic concepts.

The final goal for Acme is to provide an expressive notation for architects. While expressiveness is invariably a subjective quality, Acme has attempted to address the issue by adopting a rich set of base architectural types, by providing a flexible template encapsulation mechanism, and by allowing the definition of new property sublanguages.

5 Example

To illustrate how Acme can be used to support architectural interchange, we briefly describe our experience integrating Wright and Rapide using Acme.[4]

Wright is an ADL that allows one to specify and analyze the abstract behavior of architectures [1]. Components and connectors behaviors are specified using a event/process notation based on CSP [9]. In particular, connector semantics are defined by a protocol that specifies the behavior of participating components. The Wright toolset allows one to use a commercial model checking technology to *statically* check properties such as: (a) whether a component interface (also specified by a process) is consistent with the connector to which it is attached; (b) whether a connector is internally consistent; and (c) whether a configuration of components and connectors is complete (in the sense that there are sufficient connections to satisfy the interface requirements of the components).

Rapide is an ADL that allows one to specify systems in terms of partially ordered sets of events [10]. Component computations are triggered by received events, and in turn trigger other computations by sending events to other components. The Rapide toolset permits simulation of such descriptions, animation of those simulations, and analysis of the resulting trace graphs to check for anomalous behavior.

Rapide and Wright are similar in several respects. Both are targeted at modelling abstract behavior of architectural designs, both use the notion of events to define behaviors, and both share a basic structural model of interconnected, communicating components.

But they also differ in significant ways. First, from a user's point of view the two languages offer different, but complementary capabilities for analysis: Wright supports static analysis using model checking technology, while Rapide supports dynamic analysis using simulation technology. Second, notationally the formalisms used to characterize event behavior are different: Wright adopts the functional

[4] A complete description of this work is beyond the scope and space limitations of this paper. Here we summarize the salient features to illuminate the benefits and limitations of Acme in practice.

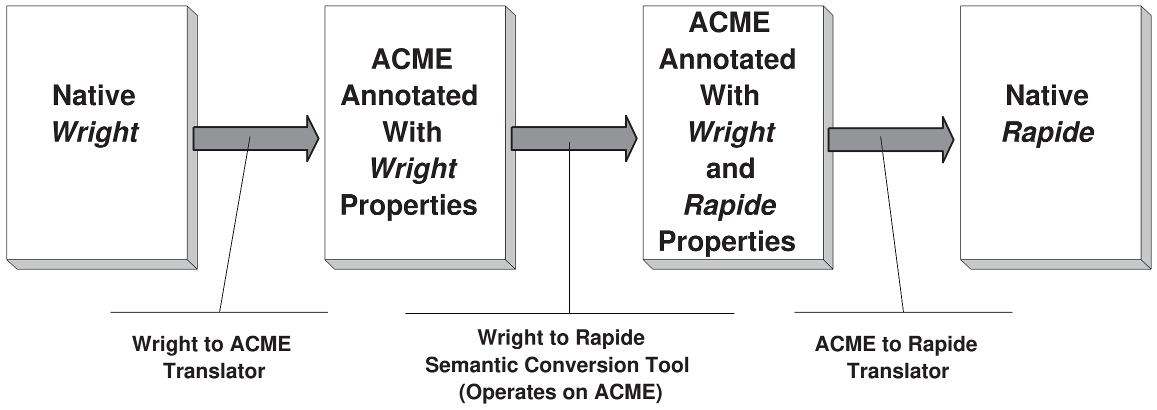

Figure 7: Wright-to-Rapide Translation via Acme

style of CSP, while Rapide uses an imperative model. Third, the two languages differ in their treatment of connectors. Rapide provides a small, fixed set of primitive connectors, while Wright permits the description of new types of connectors. Fourth, Rapide supports dynamic reconfiguration of architectures, while Wright focuses on static configurations.

As developers of the Wright toolset we wanted to take advantage of Rapide tools to add simulation and animation capabilities for our architectural descriptions. To do this we used Acme as an interchange format between the two toolsets. As illustrated in Figure 7, Wright descriptions (developed using our Wright tools) are shipped to Rapide tools via Acme interchange in three steps. First, Wright descriptions are translated into Acme. Next, a Wright-Rapide translator traverses the Acme representation to produce a new one that includes Rapide specifications. The resulting Acme description is then translated into Rapide text, which can be processed by Rapide tools.

The first and third steps are straightforward. Wright architectural structure maps easily into the Acme ontology described earlier. Specifications of component and connector behavior are also easily mapped into annotations of the structural graph. Similarly, Acme descriptions annotated with Rapide specifications are easily unparsed to native Rapide text. In both of these steps Acme libraries provide basic parsing and unparsing routines that greatly simplify the process.

The hard work occurs in the middle step. The key challenge was to bridge the semantic gap between Wright and Rapide. There were two main aspects of this. The first was to map from the functional style of Wright to the imperative style of Rapide. This turned out to be straightforward, using standard program transformation techniques.

The second, and more substantive aspect was to deal with the problem that connectors are not first class in Rapide, but are in Wright. We considered two approaches. The first was to limit the translation to those Wright specifications that use the connector types understood by Rapide. The second was to map non-trivial connectors in Wright to components in Rapide. We decided to adopt the second approach, since it would permit a larger set of Wright specifications to be mapped in to Rapide.[5] Thus the Wright-Rapide translator first converts each non-trivial connector into a

[5]The downside is that the original Wright description and the resulting Rapide description no longer have isomorphic structure, complicating the mapping of results of Rapide tools to the original descriptions. Note also that the richer capabilities of Rapide to describe dynamic architectures was not an issue since we were only interested in one-way translation.

component, and uses simple event-binding connectors to connect the parts. It then transforms each Wright semantic annotation into a Rapide semantic annotation.

Acme provided two key benefits compared to a direct, non-Acme based, translation between Wright and Rapide. First, it substantially simplified the handling of the structural aspects of the architectural description and translation. This made it straightforward to map Wright structures into Acme, and also to transform the original descriptions into (non-isomorphic) Rapide structures. Second, it had the important side effect of augmenting the Wright toolset with a set of Acme-based tools. Once Wright is translated into Acme we can use Acme tools for graphical browsing, conversion to web documents, and persistent storage.

6 Current Status, On-going Work

Work to date on Acme has focused on developing a coherent language that satisfies the requirements of a diverse set of stakeholders and goals. We have completed the design of the initial release of the Acme language. Over the past two years the preliminary language design has been discussed at several meetings of researchers and practitioners, who have provided critical feedback and guidance. This is the first written account of the language to appear at a conference.

Actual use of Acme has taken two forms. The first has been the exploration of language capabilities through case studies of system architectures. The most complex of these was an architecture for a missile command system, involving about a dozen pages of Acme. The Second has been a set of case studies in which we use Acme to support the interchange of architectural designs between various ADLs. Currently, we are able to transfer designs via Acme between UniCon and Aesop, as well as from Wright to Rapide. Tools to support more sophisticated interchange between Rapide, Wright, and SADL are in development. Although our experience with inter-ADL exchange is not yet broad enough to declare Acme a success as an interchange language, based on our initial experience with the five ADLs listed above, the prognosis looks good.

There are currently a number of Acme-based tools available. These tools include (a) an Acme-Web visualization tool that converts a textual Acme description into a "World-Wide-Weblet" that can be viewed using standard web browsers[6]; (b) a system that animates pipe-and-filter architectures described with Acme; (c) a web-based Acme repository for templates, styles, and architectural descriptions: and (d) an "expander" tool that converts architectural descriptions using templates and styles into a simple "core" description (without templates) that can be more readily interchanged between tools.

Current work on Acme is centered around three activities. First, we are extending our tools to provide better capabilities for analysis of Acme descriptions and working with other ADL developers to create tools that expand the set of ADLs that can translate to and from Acme. Second, we are continuing our efforts to develop community-based consensus around common attributes. In particular, we hope to develop a standard for characterizing trace behavior that will allow broad-based reuse of architectural "animation" tools.

Third, we are exploring richer semantic models along two major dimensions: constraint logic families and property families. Along the first dimension we are exploring temporal logic to express dynamic aspects of architectural evolution. In order to specify properties of entire architecture families—styles, refinements, or dynamic architectures—the closed world and uniqueness assumptions mentioned earlier will need to be relaxed.

The second dimension of exploration is to enable better semantic discrimination of types of properties. In general, an Acme specification's prescription is never an end in itself, but rather forms the hypothesis (the justification, really) that some other, more important predicates hold. For example, a derivative property from the combination of "Aesop-style(client) = client-server" and cycle-freeness might be that certain kinds of deadlock are impossible. We

[6]Examples and information about the tool can be found through the URL: http://www.cs.cmu.edu/~able/acme-web/

need to be able to distinguish such derived predicates from those that are stated axiomatically about an architectural specification.

Acknowledgements

The Wright-Rapide translation was largely carried out by Zhenyu Wang, whom we gratefully acknowledge. The research reported here was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grants F33615-93-1-1330 and N66001-95-C-8623; and by National Science Foundation under Grant CCR-9357792 and a Graduate Research Fellowship. 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.

References

[1] 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.

[2] 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.

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

[4] David Garlan, 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.

[5] 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.

[6] David Garlan, Bob Monroe, and David Wile. ACME: An interchange language for software architecture. Technical Report CMU-CS-95-219, Carnegie Mellon University, 1997. Available in draft form at http://www.cs.cmu.edu/~acme.

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

[8] 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.

[9] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[10] 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.

[11] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *SIGSOFT'96: Proceedings of the Fourth ACM Symposium on the Foundations of Software Engineering*. ACM Press, October 1996.

[12] M. Moriconi, X. Qian, and R. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):356–372, April 1995.

[13] 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.

[14] 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.