# An Overview Of Acme

This document serves as an overview of the capabilities of Acme. It covers the basic language features and includes a few small examples. It is based on excerpts from the paper:

Acme: An Architecture Description Interchange Language, David Garlan, Robert T. Monroe, David Wile, Proceedings of CASCON '97, November 1997.

This paper provides more detailed coverage of background, capabilities and the goals of Acme. In particular, it includes a discussion of the issues that informed the design of the language. For more detailed information on Acme as an ADL, please see this paper.

## Language Features

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 *type mechanism* for abstracting common, reusable architectural idioms and styles; and
4. an *open semantic framework* for reasoning about architectural descriptions.

## Acme 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.

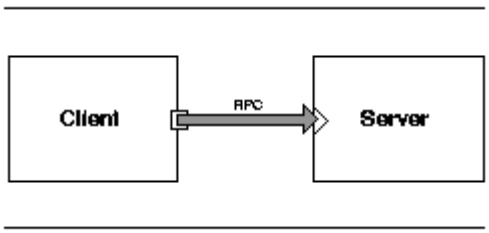


**Figure 1:** Simple Client-Server Diagram

```
System simple_cs = {
    Component client = { Port send-request; };
```

```
    Component server = { Port receive-request; };
    Connector rpc = { Roels { caller, callee}};
    Attachments {
        client.send-request to rpc.caller;
        server.receive-request to rpc.callee;
    }
}
```

**Figure2:** Simple Client-Server System in Acme

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*.

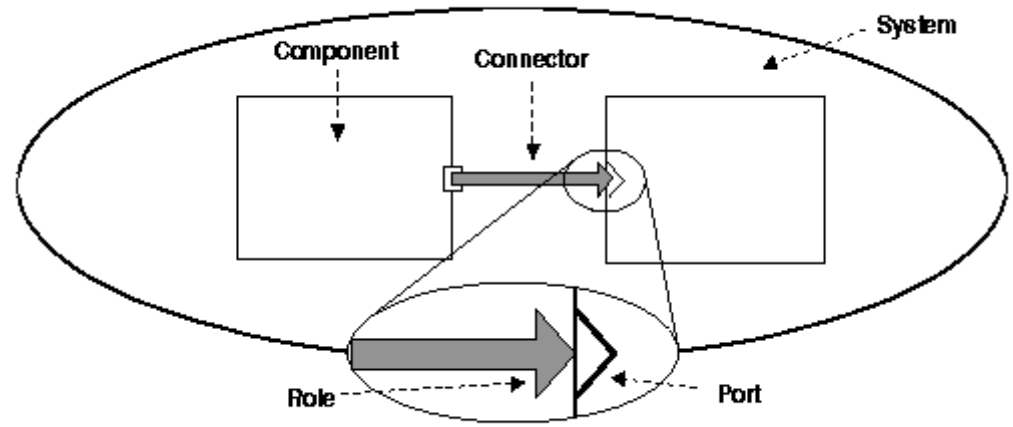


**Figure 3:** Elements of an Acme Description

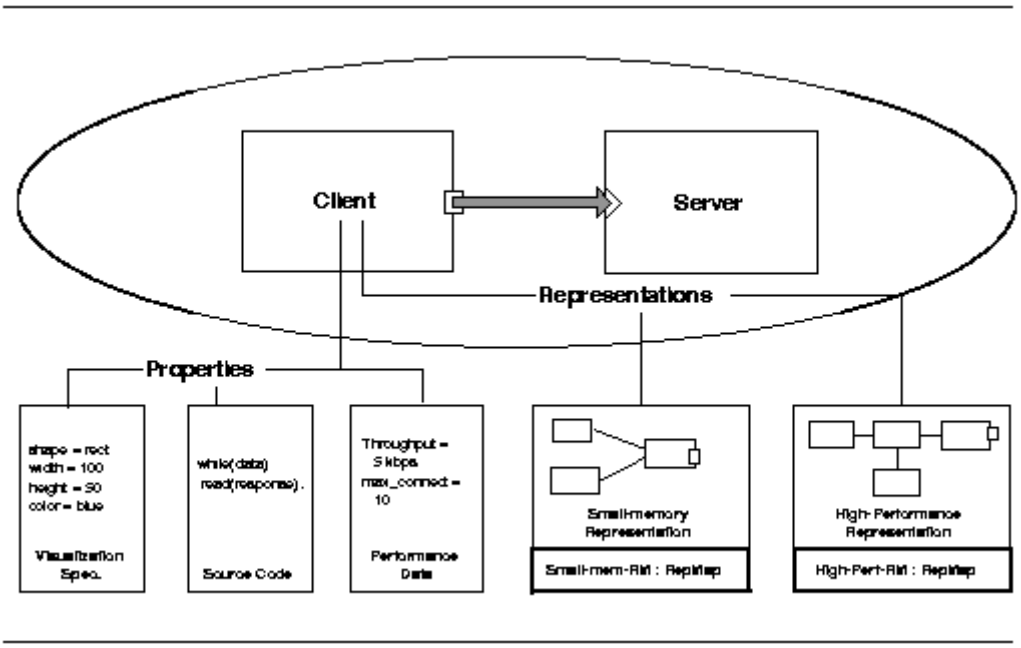


**Figure 4:** Representations and Properties of a Component

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). 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.

## 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.

```
System simple_cs = {
  Component client = {
    Port send-request;
    Property Aesop-style : style-id = client-server;
    Property UniCon-style : style-id = client-server;
    Property source-code : external = "CODE-LIB/client.c";
  }
  Component server = {
    Port receive-request;
    Property idempotence : boolean = true;
    Property max-concurrent-clients : integer = 1;
    source-code : external = "CODE-LIB/server.c";
  }
  Connector rpc = {
    Role caller;
    Role callee;
    Property asynchronous : boolean = true;
    max-roles : integer = 2;
    protocol : Wright = " ... ";
  }
  Attachment client.send-request to rpc.caller;
  Attachment server.receive-request to rpc.callee;
}
```

**Figure 5:** Client-Server System with Properties in Acme

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. 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. 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.

## Acme Families

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 types, a mechanism for the specification of recurring component, connector, port and role structures. These types can then be instantiated within the systems in a Acme description. A Family in Acme includes a set of type defintions which define the common structures or design vocabulary system. NOTE: The Acme Language also includes a template mechanism which is not described here. Please see the original paper for more information on templates. Although Acme is not intended to be a full-fledged ADL, the addition of families greatly enhances the readability and abstraction capabilities of the language.

## 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 andserver 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) ^
  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.