3

Acme: Architectural Description of Component-Based Systems

David Garlan
Computer Science Department, Carnegie Mellon University
Pittsburgh, PA 15213-3890 USA

Robert T. Monroe
Computer Science Department, Carnegie Mellon University
Pittsburgh, PA 15213-3890 USA

David Wile
USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292 USA

Abstract

Over the past decade there has been considerable experimentation with the design of architecture description languages that can provide a formal basis for description and analysis of the architectures of component-based systems. As the field has matured there has emerged among the software architecture research community general consensus about many aspects of the foundations for architectural representation and analysis. One result has been the development of a generic architecture description language, called *Acme*, that can serve as a common representation for software architectures and that permits the integration of diverse collections of independently developed architectural analysis tools. In this paper we describe the Acme language and tools, and our experience in using it to integrate architecture analysis tools and to describe component-based systems.

3.1 Introduction

An important problem for component-based systems engineering is finding appropriate notations to describe those systems. Good notations make it possible to document component-based designs clearly, reason about their properties, and automate their analysis and system generation.

One approach to describing component-based systems is to use object modeling notations (for example, as in [Szy98]). Each component can be represented by a class, component interfaces can be represented by class interfaces, and interactions between components can be defined in terms of associations.

Object modeling of component-based systems has a number of nice features. Ob-

ject notations are familiar to an increasingly large number of software engineers. They provide a direct mapping to implementations. They are supported by commercial tools. They have well-defined methods for developing systems from a set of requirements.

But object modeling notations have a number of drawbacks with respect to the description of component-based systems. First, they provide only a single form of primitive interconnection – method invocation. This makes it difficult to represent richer types of component interaction as first class design elements. Second, they have weak support for hierarchical description, making it difficult to describe systems at increasing levels of detail. Third, they do not support the definition of families of systems. While they can be used to describe patterns and to define a vocabulary of object types, they don't have explicit syntactic support for characterizing a class of system in terms of the design constraints that each member of the family must observe. Fourth, they do not provide direct support for characterizing and analyzing non-functional properties. This makes it difficult to reason about critical system design properties, such as system performance and reliability.

An alternative approach, and one that overcomes these problems, is to use an architecture description language (ADL). Over the past decade a number of languages have been developed to handle the high-level description of complex software systems, exposing their gross structure as a collection of interacting components, and allowing engineers 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, conformance to architectural standards, and anticipated dimensions of evolution [GP95, GS93, MT97, PW92].

While different ADLs focus on different aspects of architecture, as the field has matured there has emerged among the software architecture research community general consensus about many aspects of the foundations for architectural representation and analysis. One result has been the development of a generic, second-generation architecture description language, called *Acme*, that can serve as a common representation for software architectures, and that permits the integration of diverse collections of independently developed architectural analysis tools. In the remainder of this paper we describe the Acme language and tools, and our experience using it. We begin by surveying the conceptual basis for architecture description. Then we describe Acme as a concrete example of a language for architecture description. Finally, we summarize current experience with Acme and outline directions for future research.

3.2 Architectural Description

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 [GP95, GS93, PW92]. Typical properties of concern include compatibility between components [AG97], conformance to standards [AGI98], performance [SG98], schedulability [BV93], and reliability.

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 few tools to help 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 [GAO94], Adage [CS93], C2 [MORT96], Darwin [MDEK95], Rapide [LAK+95], SADL [MQR95], UniCon [SDK+95], Meta-H [BV93], and Wright [AG97]. While all of these languages are concerned with architectural design, each provides certain distinctive capabilities: Adage supports the description of architectural frameworks for avionics navigation and guidance; Aesop supports the use of architectural styles; C2 supports the description of user interface systems using an event-based style; Darwin supports the analysis of distributed message-passing systems; Meta-H provides 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; 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 formal specification and analysis of interactions between architectural components.

Although there is considerable diversity in the capabilities of different ADLs, all share a similar conceptual basis, or ontology [Gar95, MT97], that determines a common foundation of concepts and concerns for architectural description. The main elements of this ontology are:

- *Components* represent the primary computational elements and data stores of a

† In this paper we use the term "ADL" to refer to both the language and its supporting toolset.

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. In most ADLs components may have multiple interfaces, each interface defining a point of interaction between a component and its environment.

- *Connectors* represent interactions among components. Computationally speaking, connectors mediate the communication and coordination activities among components. That is, 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. Connectors also have interfaces that define the roles played by the various participants in the interaction represented by the connector.
- *Systems* represent configurations (graphs) of components and connectors. In modern ADLs a key property of system descriptions is that the overall topology of a system is defined independently from the components and connectors that make up the system. (This is in contrast to most programming language module systems where dependencies are wired into components via import clauses.) Systems may also be hierarchical: components and connectors may represent subsystems that have "internal" architectures.
- *Properties* represent semantic information about a system and its components that goes beyond structure. As noted earlier, different ADLs focus on different properties, but virtually all provide *some* way to define one or more extra-functional properties together with tools for analyzing those properties. For example, some ADLs allow one to calculate overall system throughput and latency based on performance estimates of each component and connector [SG98].
- *Constraints* represent claims about an architectural design that should remain true even as it evolves over time. Typical constraints include restrictions on allowable values of properties, topology, and design vocabulary. For example, an architecture might constrain its design so that the number of clients of a particular server is less than some maximum value.
- *Styles* represent families of related systems. An architectural *style* typically defines a vocabulary of design element types and rules for composing them [SG96]. Examples include dataflow architectures based on graphs of pipes and filters, blackboard architectures based on shared data space and a set of knowledge sources, and layered systems. Some architectural styles additionally prescribe a framework† as a set of structural forms that specific applications can specialize.

† Terminology distinguishing different kinds of families of architectures is far from standard. Among the terms used are "product-line frameworks," "component integration standards," "kits," "architectural patterns," "styles," "idioms," and others. For the purposes of this paper, the distinctions between these kinds of architectural families is less important than the fact that they all represent a set of architectural instances.

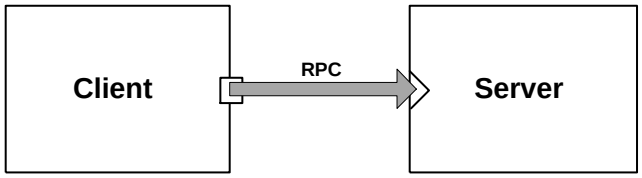

Fig. 3.1. Simple Client-Server Diagram.

Examples include the traditional multistage compiler framework, 3-tiered client-server systems, the OSI protocol stack, and user interface management systems.

As a very simple illustrative example, Figure 3.1 depicts a system containing a client and server component connected by a RPC connector. The server itself might be represented by a subarchitecture (not shown). Properties of the connector might include the protocol of interaction that it requires. Properties of the server might include the average response time for requests. Constraints on the system might stipulate that no more than five clients can ever be connected to this server and that servers may not initiate communication with a client. The style of the system might be a "client-server" style in which the vocabulary of design includes clients, servers, and RPC connectors.

This ontological basis of ADLs has a natural map to the descriptive needs of component-based systems. First, ADLs allow one to describe compositions of components precisely, making explicit the ways in which those components communicate. Second, they support the use of multiple component interfaces, a key feature of many component-based systems approaches. Third, they support hierarchical descriptions and encapsulation of subsystems as components in a larger system. Fourth, they support the specification and analysis of non-functional properties. Fifth, many ADLs provide an explicit home for describing the detailed semantics of communication infrastructure (through specification of connector types). Sixth, ADLs allow one to define constraints on system composition that make clear what kinds of compositions are allowed. Finally, architectural styles allow one to make precise the differences between kinds of component integration standards.

3.3 Acme

To elaborate on how architectural description allows one to characterize component based systems, we now describe a particular ADL, called Acme.† As a second-generation ADL, Acme has the distinctive property that it builds on the experience of other ADLs, providing in a simple language the essential elements of architectural

† In this paper we provide a high-level view of the language, emphasizing aspects that are particularly relevant to the description of component-based systems. For a more detailed treatment see [Mon98].

design, and supporting natural extensions to support more complex architectural features. In particular, Acme embodies the architectural ontology described above, providing a semantically extensible language and a rich toolset for architectural analysis and integration of independently developed tools.

Acme supports the definition of four distinct aspects of architecture. First is structure—the organization of a system into its constituent parts. Second is properties of interest—information about a system or its parts that allow one to reason abstractly about overall behavior (both functional and nonfunctional). Third is constraints—guidelines for how the architecture can change over time. Fourth is types and styles—defining classes and families of architecture. We now consider each in turn.

3.3.1 Structure

Architectural structure is defined in Acme using seven core types of entities: *components, connectors, systems, ports, roles, representations, and rep-maps.*† The first five are illustrated in Figure 3.2.

Consistent with the ontology outlined earlier, Acme *components* represent computational elements and data stores of a system. A component may have multiple interfaces, each of which is termed a *port*. A port identifies a point of interaction between the component and its environment, and can represent an interface as simple as a single procedure signature. Alternatively, a port can define a more complex interface, such as a collection of procedure calls that must be invoked in certain specified orders, or an event multicast interface.

Acme *connectors* represent interactions among components. 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.

Acme *systems* are defined as graphs in which the nodes represent components and the arcs represent connectors. This is done by identifying which component ports are *attached* to which connector roles.

Figure 3.3 contains an Acme description of the simple architecture of Figure 3.1. 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 defined by listing a set of *attachments* that bind component ports to connector roles. In this case, the client's requesting port

† In earlier presentations, this part of the language was referred to as "Kernel Acme."

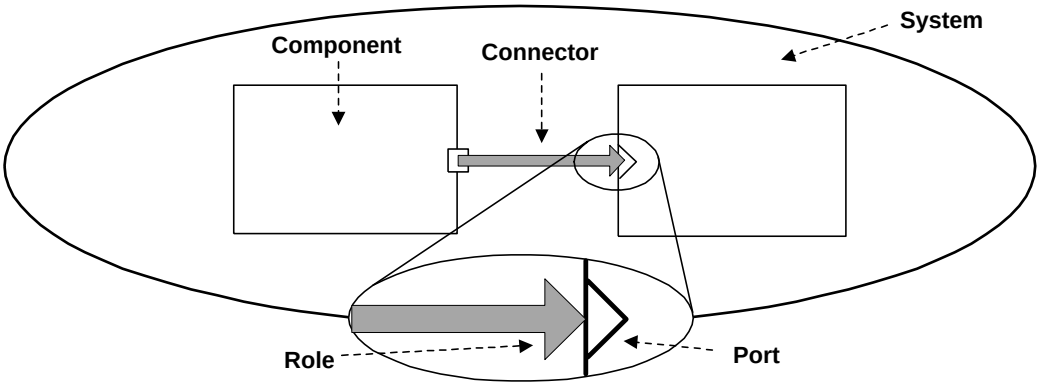

Fig. 3.2. Elements of an Acme Description.

```
System simple_cs = {
  Component client = { Port sendRequest }
  Component server = { Port receiveRequest }
  Connector rpc  = { Roles {caller, callee} }
  Attachments : {
      client.sendRequest to rpc.caller ;
      server.receiveRequest to rpc.callee }
}
```

Fig. 3.3. Simple Client-Server System in Acme.

is bound to the rpc's caller role, and the servers's request-handling port is bound to the rpc's callee role.

To support hierarchical descriptions of architectures, Acme permits any component or connector to be represented by one or more detailed, lower-level descriptions. Each such description is termed a *representation*. Representations of a component are illustrated abstractly in Figure 3.4.

The ability to associate multiple representations with a design element† allows

† We use the term "design element" to refer to any of the Acme building blocks: components, connectors, and so on.

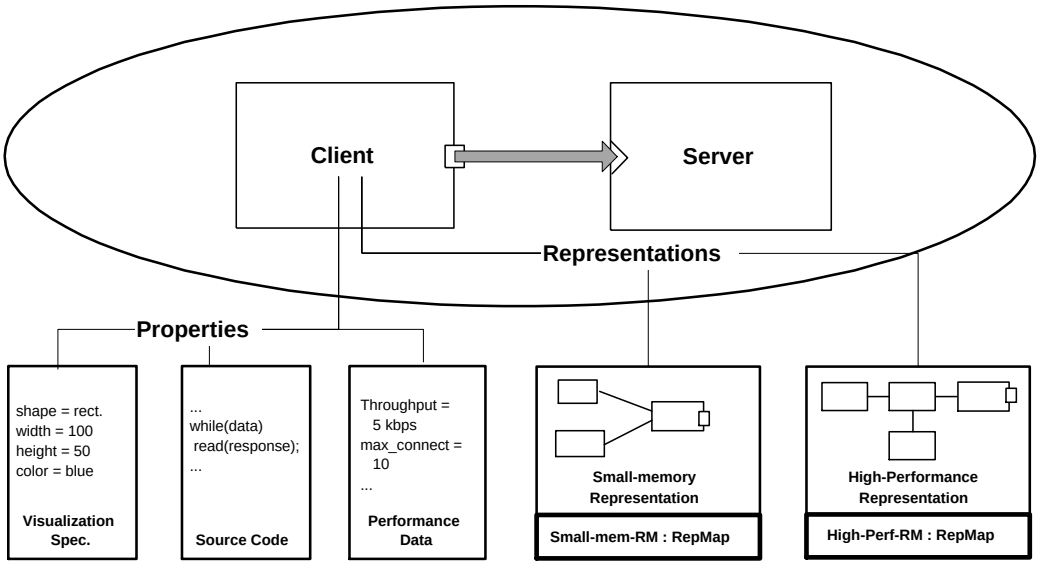

Fig. 3.4. Representations and Properties of a Component.

Acme to encode multiple views of architectural entities (although there is nothing currently built into Acme that supports resolution of inter-view correspondences).

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 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.

Figures 3.5 and 3.6 illustrate the use of representations in elaborating the simple client-server example. In this case, the *server* component is elaborated by a more detailed architectural representation.

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

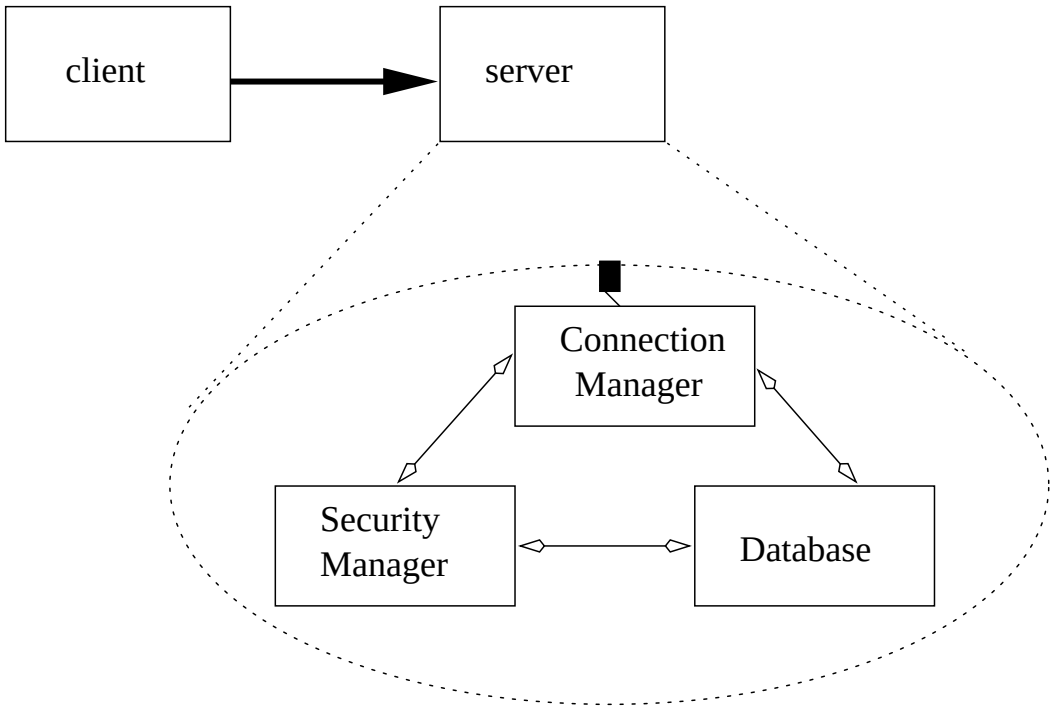

Fig. 3.5. Hierarchical Client-Server System.

3.3.2 Properties

The seven classes of design element outlined above are sufficient for defining the *structure* of an architecture as a graph of components and connectors. Explicit description of structure is useful for documenting systems as compositions of components.

However, there is much more to architectural description than structure. But what exactly? Looking at the range of first-generation ADLs, each typically has its own forms of auxiliary information that determines such things as the run-time semantics of the system, protocols of interaction, scheduling constraints, and resource consumption.

Clearly, the needs for documenting extra-structural properties of a system's architecture depend on the nature of the system, the kinds of analyses required, the tools at hand, and the level of detail included in the description.

To accommodate the open-ended requirements for specification of auxiliary information, Acme supports annotation of architectural structure with arbitrary lists

```
System simpleCS = {
  Component client = { ... }
  Component server = {
        Port receiveRequest;
        Representation serverDetails = {
          System serverDetailsSys = {

            Component connectionManager = {
               Ports { externalSocket; securityCheckIntf; dbQueryIntf } }

            Component securityManager = {
               Ports { securityAuthorization; credentialQuery; } }

            Component database = {
               Ports { securityManagementIntf; queryIntf; } }

            Connector SQLQuery = { Roles { caller; callee } }
            Connector clearanceRequest = { Roles { requestor; grantor } }
            Connector securityQuery = {
               Roles { securityManager; requestor } }
            Attachments {
               connectionManager.securityCheckIntf to clearanceRequest.requestor;
               securityManager.securityAuthorization to clearanceRequest.grantor;
               connectionManager.dbQueryIntf to SQLQuery.caller;
               database.queryIntf to SQLQuery.callee;
               securityManager.credentialQuery to securityQuery.securityManager;
               database.securityManagementIntf to securityQuery.requestor; }

        }
        Bindings { connectionManager.externalSocket to server.receiveRequest }
    }
  }
  Connector rpc  = { ... }
  Attachments { client.send-request to rpc.caller ;
                server.receive-request to rpc.callee }
```

Fig. 3.6. Client-Server System with Representation.

of properties. Each property has a name, an optional type, and a value. Any of the seven classes of Acme architectural design entities can be annotated with a property list.

Figure 3.4 pictures several properties that might be associated with the example architecture, and Figure 3.7 shows the simple client-server system elaborated with several properties in Acme. In the latter figure, properties document such things as the client's expected request rate and the location of its source code. For the *rpc*

```
System simple_cs = {
  Component client = {
        Port sendRequest;
        Properties { requestRate : float = 17.0;
                     sourceCode : externalFile = "CODE-LIB/client.c" }}

  Component server = {
        Port receiveRequest;
        Properties { idempotent : boolean = true;
                     maxConcurrentClients : integer = 1;
                     multithreaded : boolean = false;
                     sourceCode : externalFile = "CODE-LIB/server.c" }}

  Connector rpc  = {
        Role caller;
        Role callee;
        Properties { synchronous : boolean = true;
                     maxRoles : integer = 2;
                     protocol : WrightSpec = "..." }}

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

Fig. 3.7. Client-Server System with Properties.

connector, properties document the protocol of interaction described as a Wright specification [AG97] (elided in the figure).

Properties serve to document details of an architecture relevant to its design and analysis. However, from Acme's point of view properties are uninterpreted values—that is, they have no intrinsic semantics. Properties become useful, however, when tools use them for analysis, translation, display, and manipulation.

The types of properties are defined using a set of built-in primitive property types (including integer, string, and boolean), and type constructors for records, sets, and lists.

3.3.3 Design Constraints

One of the key ingredients of an architecture description is a set of design constraints that determine how an architectural design is permitted to evolve over time. Such constraints can be considered a special kind of property, but since they play such a central role in architectural design, Acme provides special syntax for describing them. (Of course, this also permits the creation of tools for checking constraint satisfaction of an architectural description.)

Connected(comp1, comp2)	True if component comp1 is connected to component comp2 by at least one connector
Reachable(comp1, comp2)	True if component comp2 is in the transitive closure of Connected(comp1, *)
HasProperty(elt, propName)	True if element elt has a property called propName
HasType(elt, typeName)	True if element elt has type typeName
SystemName.Connectors	The set of connectors in system SystemName
ConnectorName.Roles	The set of the roles in connector ConnectorName

Fig. 3.8. Sample Functions for Constraint Expressions.

Acme uses a constraint language based on first order predicate logic (FOPL).† That is, design constraints are expressed as predicates over architectural specifications. The constraint language includes the standard set of FOPL constructs (conjunction, disjunction, implication, quantification, and others). It also includes a number of special functions that refer to architecture-specific aspects of a system. For example, there are predicates to determine if two components are connected (both directly and indirectly), and if a component has a particular property. Other functions return the set of components in a given system, the set of ports of a given component, the set of representations of a connector, and so forth. Figure 3.8 lists a representative set of example functions. (For a detailed description see [Mon99].)

Constraints can be associated with any design element of an Acme description. The scope of the constraint is determined by that association. For example, if a constraint is attached to a system then it can refer to any of the design elements contained within it (components, connectors, and their parts). On the other hand, a constraint attached to a component can only refer to that component (using the special keyword *self*, and its parts (that is, its ports, properties, and representations).

To explore a few examples, consider the following constraints that might be associated with a system:

connected(client, server)

will be true if the components named *client* and *server* are connected directly by a connector.

Forall conn : connector in systemInstance.Connectors @ size(conn.roles) = 2

will be true of a system in which all of the connectors are binary connectors.

† Acme constraints were initially developed in the Armani system [Mon99, Mon98], and recently incorporated into the language.

```
System messagePathSystem = {
  ...
  Connector MessagePath = {
      Roles {source; sink;}
      Property expectedThroughput : float =  512;
      Invariant (queueBufferSize >= 512) and (queueBufferSize <= 4096);
      Heuristic expectedThroughput <= (queueBufferSize / 2);
   }
}
```

Fig. 3.9. *MessagePath* Connector with Invariants and Heuristics.

Forall conn : connector in systemInstance.Connectors @
Forall r : role in conn.Roles @
Exists comp : component in systemInstance.Components @
Exists p : port in comp.Ports @ attached(p,r) and (p.protocol = r.protocol)

will be true when all connectors in the system are attached to a port, and the attached (port, role) pair share the same protocol. Here the port and role protocol values are represented as properties of the port and role design elements.

Constraints can also define the range of legal property values, as in

self.throughputRate >= 3095

and indicate relationships between properties, as in

comp.totalLatency =
(comp.readLatency + comp.processingLatency + comp.writeLatency)

Constraints may be attached to design elements in one of two ways: as an *invariant* or a *heuristic*. In the first case, the constraint is taken to be a rule that cannot be violated. In the second case, the constraint is taken to be a rule that should be observed, but may be selectively violated. Tools that check for consistency of an Acme specification will naturally treat these differently. A violation of an invariant makes the architectural specification invalid, while a violation of a heuristic is treated as a warning.

Figure 3.9 illustrates how constraints might be used for a hypothetical *MessagePath* connector. In this example an invariant prescribes the range of legal buffer sizes, while a heuristic prescribes a maximum value for the expected throughput.

3.3.4 Types and Styles

An important general capability for the description of architectures is the ability to define styles—or families—of systems. Styles allow one to define a domain-specific or application-specific design vocabulary, together with constraints on how that

```
Component Type Client = {
    Port Request = {Property protocol: CSPprotocolT};
    Property request-rate: Float;
    Invariant Forall p in self.Ports @ p.protocol = rpc-client;
    Invariant size(self.Ports) <= 5;
    Invariant request-rate >= 0;
    Heuristic request-rate < 100;
}
```

Fig. 3.10. Component Type "Client."

vocabulary can be used. This in turn supports packaging of domain-specific design expertise, use of special-purpose analysis and code-generation tools, simplification of the design process, and the ability to check for conformance to architectural standards.

The basic building block for defining styles in Acme is a type system that can be used to encapsulate recurring structures and relationships. In Acme an architect can define three kinds of types: property types, structural types, and styles. Property types were discussed earlier.

Structural types make it possible to define types of components, connectors, ports, and roles. Each such type provides a type name and a list of required substructure, properties, and constraints. Figure 3.10 illustrates the definition of a *Client* component type. The type definition specifies that any component that is an instance of type *Client* must have at least one port called *Request* and a property called *request-rate* of type float. Further, the invariants associated with the type require that all ports of a *Client* component have a *protocol* property whose value is *rpc-client*, that no client more than 5 ports, that a component's request rate is larger greater than 0. Finally, there is a heuristic indicating that the request-rate should be less than 100.

The third kind of type is a style, which (for historical reasons) is called a *family* in Acme. Just as structural types represent sets of structural elements, a family represents a set of *systems*.

An Acme family is defined by specifying three things: a set of property and structural types, a set of constraints, and default structure. The property and structural types provide the design vocabulary for the family. The constraints determine how instances of those types can be used. The default structure prescribes the minimal set of instances that must appear in any system in the family.

Figure 3.11 illustrates the definition of a "Pipe and Filter" family, together with a sample system declaration using the family. The family defines two component types, one connector type, and one property type. The single invariant of this family prescribes that all connectors must be pipes. No default structure is declared in

```
Family PipeFilterFam = {

  Component Type FilterT = {
        Ports { stdin; stdout; };
        Property throughput : int;
  };
  Component Type UnixFilterT extends FilterT with {
        Port stderr;
        Property implementationFile : String;
  };
  Connector Type PipeT = {
        Roles { source; sink; };
        Property bufferSize : int;
  };
  Property Type StringMsgFormatT = Record [ size:int; msg:String; ];
  Invariant Forall c in self.Connectors @ HasType(c, PipeT);

}

System simplePF : PipeFilterFam = {

    Component smooth : FilterT = new FilterT
    Component detectErrors : FilterT;
    Component showTracks : UnixFilterT = new UnixFilterT extended with {
        Property implementationFile : String = "IMPL_HOME/showTracks.c";
    };

    // Declare the system's connectors
    Connector firstPipe : PipeT;
    Connector secondPipe : PipeT;

    // Define the system's topology
    Attachments { smooth.stdout to firstPipe.source;
                  detectErrors.stdin to firstPipe.sink;
                  detectErrors.stdout to secondPipe.source;
                  showTracks.stdin to secondPipe.sink; }
}
```

Fig. 3.11. Definition of a Pipe-Filter Family.

the family. The system *simplePF* is then defined as an instance of the family. This declaration allows the system to make use of any of the types in the family, and it must satisfy all of the family's invariants.

Figure 3.11 also illustrates several other points about the use of types and families. First, types can be used to create default instances with the *new* keyword (as in *new FilterT*). This causes the value of the instance to have the minimal structure defined by type. But an instance can also add to this default structure, as illustrated in

the definition of the *showTracks* component. That component adds a new property (*implementationFile*).

This raises an important question: what does it mean for an instance to satisfy a type? In Acme, types are interpreted as predicates, and asserting that an instance satisfies a type is the same as asserting that it satisfies the predicate denoted by the type. The predicate associated with a type is constructed by viewing declared structure as asserting the *existence* of that structure in each instance. In other words, a type defines the *minimal* structure of its instances.† (Hence, in the example of Figure 3.11 it is essential to include the invariant asserting that all connectors have type *pipe*.)

The use of a predicate-based type system has several important consequences. First, design elements (and systems) can have an arbitrary number of types. For example, the fact that a structural element is declared to be of a particular type, does not preclude it from satisfying other type specifications. This is an important property since it permits, for example, a system to be considered a valid instance of a style, even though it was not explicitly declared as such.

Second, the use of invariants fits smoothly within the type system. Adding a invariant to a structural type or family simply conjoins that predicate with the others in the type. This means that the type system becomes quite expressive – essentially harnessing first order predicate logic (FOPL) to create useful type distinctions.

Third, the process of type checking becomes one of checking satisfaction of a set of predicates over declared structures. Hence, types play two useful roles: (a) they encapsulate common, reusable structures and properties, and (b) they support a powerful form of checkable redundancy.

The use of predicates does, however, raise the issue that, in general, checking for satisfaction of FOPL predicates is not decidable. Therefore, systems that rely on predicate-based type systems usually do so with the aid of a theorem prover (for example, PVS [ORS92]). In Acme, however, we constrain the expressiveness of types so that type checking remains decidable. This is done by ensuring that quantification is only over finite sets of elements. Finiteness comes from the fact that Acme structures can only declare a finite number of subparts (components, ports, representations, and others).

3.4 Using Acme

The success of any system description language is ultimately defined by its impact on the practice of developing complex software systems. While Acme is still relatively

† The semantics of the Acme type system is similar to – but considerably simpler than – that of other predicate-based type systems, such as the one used by PVS [ORS92]. For a formal treatment of the semantics, see [Mon98].

young, as such languages go it has already begun to play several key roles in the software architecture community.

The first role is as an architecture description language. As a second-generation ADL, Acme has attempted capture the essential elements of architectural modeling. Hence the language presents a relatively simple core set of concepts for defining system structure, coupled with the ability to extend those concepts using properties, constraints, types, and styles that are appropriate to the context of use. A number of case studies have been carried out using Acme, including a substantial missile guidance system, and a global tracking system for the U.S. Department of Defense (carried out by Lockheed-Martin Corporation).

The second role is as a basis for new architecture design and analysis tools. Currently over a dozen tools and three design environments have been built to operate on Acme descriptions. The tools perform a variety of tasks, including type checking Acme (including satisfaction of invariants and constraints) [Mon99], generation of Web-based documentation, automated graph layout, animation of runtime behavior in architectural terms [GB99, LAK+95], dependence analysis for predicting the impacts of changes [SRW98], and performance and reliability analyses (for certain styles) [SG98].

The environments provide graphical front ends for creating Acme descriptions and support various analysis capabilities. The primary environment, called AcmeStudio, provides a Windows-based front end, together with capabilities for customizing the environment to support style-specific visualizations and tool invocation. A second prototype environment, called Armani, uses a commercial graphical front end (Visio) and a Java back end. It developed the initial implementation of the constraint checker for Acme descriptions. A third, more experimental environment designed at ISI emphasizes the creation of domain-specific architectural styles and analyzers. It uses the Microsoft's PowerPoint editor for graph manipulation coupled with analyzers reacting to changes to a DCOM representation of the architectural elements and their attached properties [GB99].

The third role of Acme is as a basis for integrating existing tools. As noted earlier, a large number of ADLs have been developed, each with their own stand-alone toolset. Acme can be used to integrate many of these tools by providing a common representation for interchanging architectural descriptions. (Indeed, historically, this was the initial motivation for creating Acme.) The basis for tool integration using Acme is the ability of other tools to read and write Acme descriptions. This is accomplished by encoding ADL-specific information as properties, attached to a generic structural skeleton. To the extent that tools share properties, they can exchange semantically relevant information for analysis.

To take one example, described in detail in [GW99], both Rapide and Wright are ADLs that model abstract behavior of architectural designs in terms of event traces. Rapide tools are primarily used to simulate event-based behavior, while Wright tools use a model checker to perform static behavioral analysis. The complementary

capabilities of the two systems were combined by using Acme as an interchange language.

Naturally, the ability to perform such integration depends both on the ability of existing ADLs to read and write Acme, and on the ability to relate semantic information from one ADL to another. With respect to the first issue, a number of ADLs can handle Acme, including UniCon, Wright, Aesop, C2, and SADL. With respect to the second, work is underway to define common property sublanguages that will be accessible to several tools. Examples include property languages for visualization, for representation of events and event patterns, and for timing information.

The fourth role of Acme is as a starting point for the development of new domain-specific ADLs. Many companies are recognizing the importance of product-line frameworks, often based on external component integration standards. Ideally developers of such systems would use tools and environments tailored to those frameworks and standards. Unfortunately, there is little available commercially. Practitioners must therefore rely on either general purpose modeling notations (such as UML) or domain-specific tools that may not match their domain. Acme provides an alternative by supporting a path along which one can produce more specialized ADLs by tailoring the general descriptive framework provided by Acme to the needs of the specific domain (for example, using the ISI Acme environment or AcmeStudio).

3.5 Conclusion and Future Work

We have argued that architecture description languages provide an appropriate foundational basis for describing component-based systems designs. We then illustrated the nature of such description by presenting Acme. Finally, we described the roles that languages such as Acme are starting to play. While we would not claim that architectural descriptions should supplant all other means for specifying component-based systems, experience to date demonstrates that they are a useful adjunct to existing mechanisms, especially for specifying and reasoning about nonfunctional properties of a system.

However, much remains to be done both in practice and in research. On the practical side of things, it will clearly be important to experiment further with architectural descriptions in real settings. It will also be important to develop new tools that are useful to practicing software architects, and to continue to integrate existing tools.

On the research side a number of avenues are worth exploring. First is extensions of the type system to support parameterized and higher-order types.† An initial proposal for this has already been sketched out.

† An earlier version of Acme had a rudimentary form of this, called "templates" that appeared in earlier documentation, but was never fully implemented or integrated with the current tools.

Second is extensions to support the definition of structural patterns. These extensions would permit class diagram-like patterns to be added to family definitions.

Third is better integration with external tools and notations. In particular, it is important to be able to relate architectural descriptions, such as Acme's, to notations such as UML. Several proposals have been published on how to do this (for example, [MR99, HNS99]), but it is not yet clear what aspects of object modeling are best exploited in this mapping.

Fourth is the development of an "ADL Toolkit" that would provide meta-tools for producing new architecture description languages. While this can be done now, for example, using Acme's extension mechanisms, the capabilities would be more accessible to practicing system designers if there were better tool support for the process.

3.6 Acknowledgments

The development of Acme has been a community effort, with contributions from many researchers and practitioners. We thank the many people who have informed the language design and worked to incorporate Acme into their tools and development environments.

The research reported here was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the U.S. DoD Advanced Research Projects Agency (DARPA) under grants F30602-97-2-0031, and F33615-93-1-1330; and by National Science Foundation under 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 U.S. Department of Defense, the United States Government, or the National Science Foundation. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon.

Bibliography

[AG97] Allen, R. and Garlan, D. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[AGI98] Allen, R., Garlan, D., and Ivers, J. Formal modeling and analysis of the HLA component integration standard. In *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, Lake Buena Vista, Florida, November 1998. ACM.

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

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

[GAO94] Garlan, D., Allen, R., and Ockerbloom, J. 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] Garlan, D., 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.

[GB99] Goldman, N. M. and Balzer, R. M. The isi visual design editor. In *Proceedings of the 1999 IEEE International Conference on Visual Languages*, September 1999. To appear.

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

[GS93] Garlan, D. and Shaw, M. An introduction to software architecture. In Ambriola, V. and Tortora, G., editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

[GW99] Garlan, D. and Wang, Z. A case study in software architecture interchange. In *Proceedings of Coordination '99*. Springer Verlag, April 1999.

[HNS99] Hofmeister, C., Nord, R. L., and Soni, D. Describing software architecture with UML. In *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, San Antonio, TX, February 1999.

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

[MDEK95] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference, ESEC'95*, September 1995.

[Mon98] Monroe, R. T. Capturing software architecture design expertise with armani. Technical Report CMU-CS-163, Carnegie Mellon University, October 1998.

[Mon99] Monroe, R. T. *Rapid Development of Custom Software Design Environments*. PhD thesis, Carnegie Mellon University, July 1999.

[MORT96] Medvidovic, N., Oreizy, P., Robbins, J. E., and Taylor, R. N. 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.

[MQR95] Moriconi, M., Qian, X., and Riemenschneider, R. Correct architecture refinement. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):356–372, April 1995.

[MR99] Medvidovic, N. and Rosenblum, D. S. Assessing the suitability of a standard design method for modeling software architectures. In *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, San Antonio, TX, February 1999.

[MT97] Medvidovic, N. and Taylor, R. N. Architecture description languages. In *Software Engineering – ESEC/FSE'97*, volume 1301 of *Lecture Notes in Computer Science*, Zurich, Switzerland, September 1997. Springer.

[ORS92] Owre, S., Rushby, J. M., and Shankar, N. PVS: A prototype verification system. In Kapur, D., editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

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

[SDK+95] Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G. 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.

[SG96] Shaw, M. and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[SG98] Spitznagel, B. and Garlan, D. Architecture-based performance analysis. In *Tenth International Conference on Software Engineering and Knowledge Engineering (SEKE'98)*, San Francisco, CA, June 1998.

[SRW98] Stafford, J., Richardson, D., and Wolf, A. Aladdin: A Tool for Architecture-Level Dependence Analysis of Software. Technical Report CU-CS-858-98, Department of Computer Science, University of Colorado, Boulder, CO, April 1998.

[Szy98] Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 19998.