Architectural Description

Harald Gall, Prof. Dr.

http://seal.ifi.unizh.ch

s. e. a. l.

software evolution & architecture lab

Overview

- Architecture Description Languages (ADLs)
- ACME: an ADL and tool enviroment
- ACMEStudio: the tool for Acme

Architectural Description

- 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.

Architectural Description /2

- 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;
  - there are few tools to help architectural designers with their tasks.

- Response: Architecture Description Languages (ADLs)
  - They 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

ADLs

While all of these languages are concerned with architectural design, each provides certain distinctive capabilities!

- Aesop [GAO94] supports the use of architectural styles
- Adage [CS93] supports the description of architectural frameworks for avionics navigation and guidance
- C2 [MORT96] supports the description of user interface systems using an event-based style
- Darwin [MDEK95] supports the analysis of distributed message-passing systems
- Rapide [LAK + 95] allows architectural designs to be simulated, and has tools for analyzing the results of those simulations
- SADL [MQR95] provides a formal basis for architectural refinement
- UniCon [SDK + 95] has a high-level compiler for architectural designs
- Meta-H [BV93] supports design of real-time avionics control software
- Wright [AG97] supports the formal specification and analysis of interactions between architectural components
- xADL 2.0 [UCI] - supports run-time and design-time elements of a system; architectural types; advanced configuration management concepts such as versions, options, and variants; product family architectures; and architecture "diff"ing (initial support)

ADLs' conceptual basis (ontology)

- ***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.
  - In most ADLs components may have multiple interfaces, each interface defining a point of interaction between a component and its environment.
  - Typical examples of components include
    - clients, servers, filters, objects, blackboards, and databases.

ADLs' conceptual basis /2

- ***Connectors*** represent interactions among components.
  - Computationally speaking, connectors mediate the communication and coordination activities among components.
  - 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:
    - a client-server protocol or an 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.

ADLs' conceptual basis /3

- ***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.

ADLs' conceptual basis /4

- ***Properties*** represent semantic information about a system and its components that goes beyond structure.
  - 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.
  - some ADLs allow one to calculate overall system throughput and latency based on performance estimates of each component and connector [SG98].

ADLs' conceptual basis /5

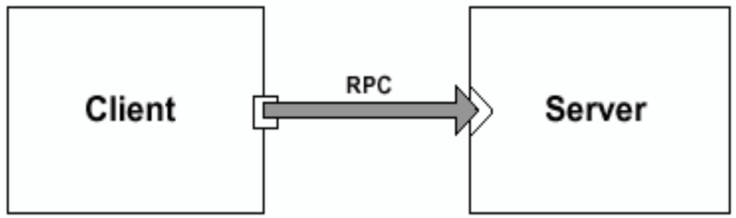
- ***Constraints*** represent claims about an architectural design that should remain true even as it evolves over time.
  - Typical constraints include restrictions on allowable values of properties, topology, and design vocabulary.
  - For example, an architecture might constrain its design so that the number of clients of a particular server is less than some maximum value.

ADLs' conceptual basis /6

- ***Styles*** represent families of related systems.
- An architectural style typically defines a vocabulary of design element types and rules for composing them [SG96].
  - Examples: data flow 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.
  - Examples: traditional multistage compiler framework, 3-tiered client-server systems, the OSI protocol stack, or user interface management systems.

Example: Client-Server

- A client and server component connected by an RPC connector. The server might be represented by a sub-architecture (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.



Acme: An Architecture Description Language

s.e.a.l.
software evolution & architecture lab

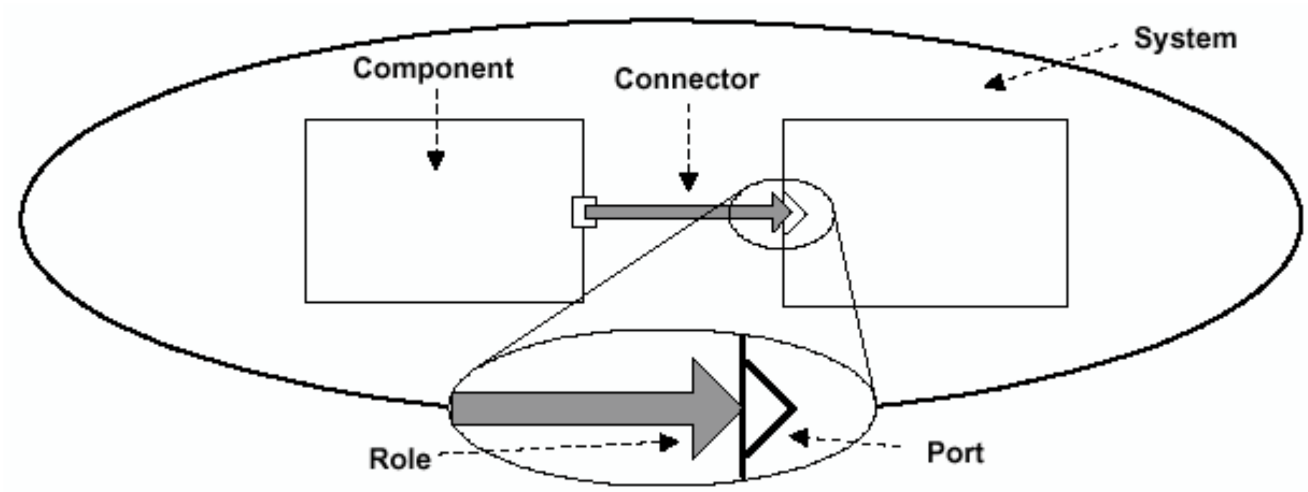
Acme: an ADL [GMW00]

- second-generation ADL; developed by the SEI/CMU
- providing in a simple language the essential elements of architectural design, and supporting natural extensions to support more complex architectural features.
- In particular, Acme embodies the architectural ontology, providing a semantically extensible language and a rich toolset for architectural analysis and integration of independently developed tools.

Acme's 4 aspects of architecture

- Structure the organization of a system into its constituent parts.
- Properties of interest: information about a system or its parts that allow one to reason abstractly about overall behavior (both functional and nonfunctional).
- Constraints: guidelines for how the architecture can change over time.
- Types and styles: defining classes and families of architecture

An Acme C/S Description

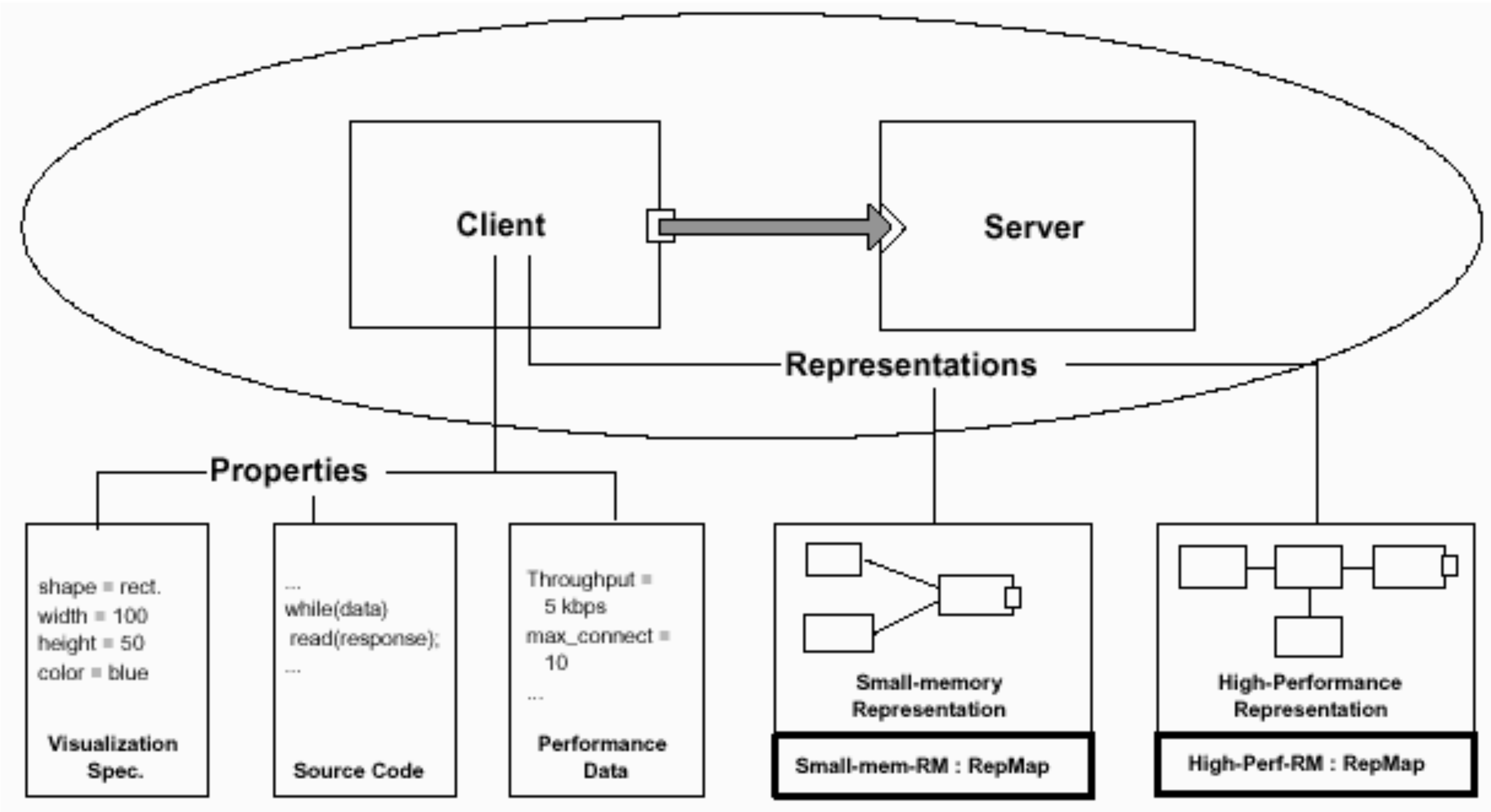



```
System simple_cs = {
  Component client = { Port sendRequest }
  Component server = { Port receiveRequest }
  Connector rpc = { Roles {caller, callee} }
  Attachments : {
      client.sendRequest to rpc.caller ;
      server.receiveRequest to rpc.callee }
}
```

Architectural structure

- 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.

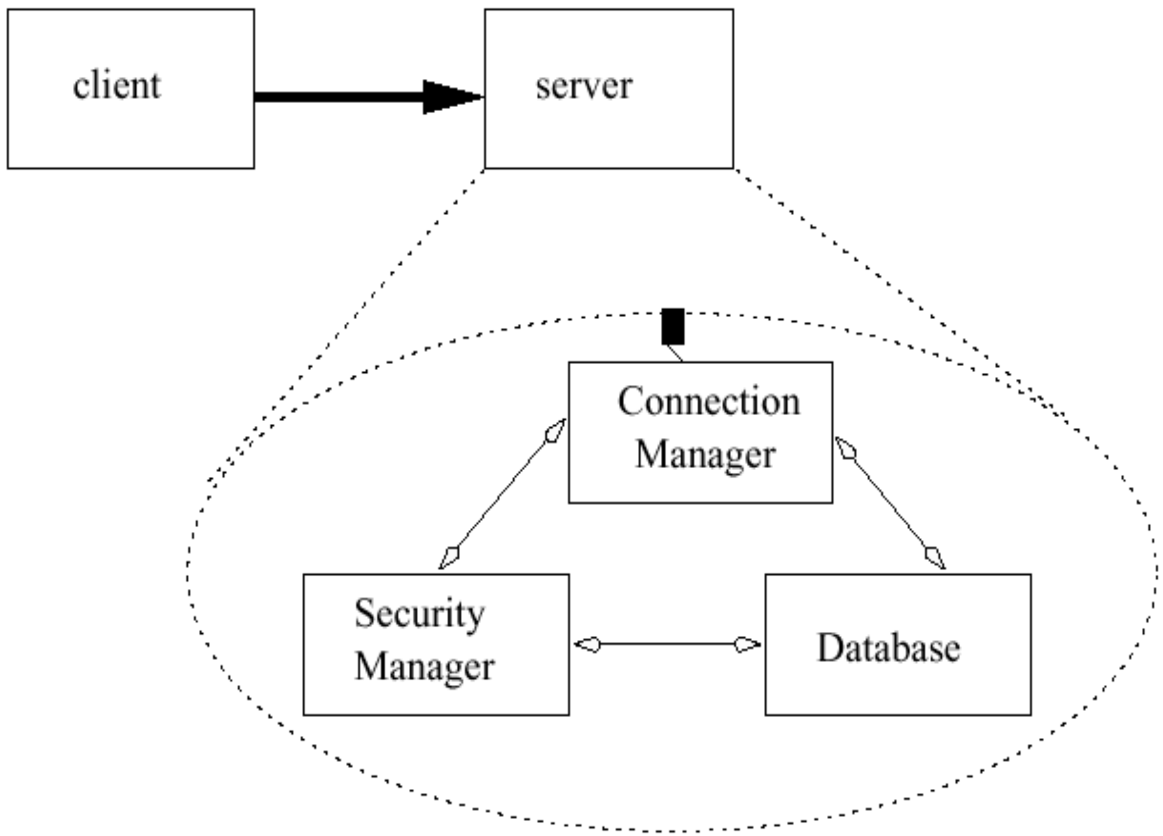
Representations and Properties



Representations and Properties /2

- Representation:
  - to support hierarchical descriptions of architectures, Acme permits any component or connector to be represented by one or more detailed, lower-level descriptions.
- Representation map (rep-map):
  - indicate the correspondence between the internal system representation and the external interface of the component or connector that is being represented.
  - 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.
  - But rep-maps are ***not*** connectors!

Hierarchical C/S system



C/S system with representation

```
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 }
```

Properties

- To accommodate the open-ended requirements for specification of auxiliary information, Acme supports annotation of architectural structure with arbitrary lists 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 (components, connectors, ports, etc.)

C/S system with properties

```
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 }
}
```

Design Constraints

- Design Constraints determine how an architectural design is permitted to evolve over time.
- 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.)
- Constraints can be associated with any design element of an Acme description. The scope of the constraint is determined by that association.
  - 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).
  - 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).

Sample functions for constraints

- 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.

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

Some constraint examples

- 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
- 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.
- self.throughputRate > = 3095
- comp.totalLatency = (comp.readLatency + comp.processingLatency + comp.writeLatency)

Constraints: invariants, heuristics

- Constraints may be attached to design elements in one of two ways:
  - as an invariant: the constraint is taken to be a rule that cannot be violated.
  - as a heuristic: 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.

Constraints example

```
System messagePathSystem = {
  ...
  Connector MessagePath = {
      Roles {source; sink;}
      Property expectedThroughput : float = 512;

      Invariant (queueBufferSize >= 512) and (queueBufferSize <= 4096);

      Heuristic expectedThroughput <= (queueBufferSize / 2);
  }
}
```

Types & 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 vocabulary can be used. This 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.
- 3 kinds of types (interpreted as predicates)
  - property types,
  - structural types,
  - styles (or *families*)

Component type “Client”

```
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;
}
```

Definition of a Pipe-Filter Family

```
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; }
}
```

Roles of Acme

- 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

AcmeStudio

http://acme.able.cs.cmu.edu/acmeweb