

Acme

- Architectural interchange language
- CMU and USC-ISI
- Extensible
- Tool support
 - AcmeStudio - Graphical editor
 - AcmeLib - API (Java, C++)
 - AcmeWeb - document generator

Features

- Architecture ontology
 - Semantic elements of the language
- Extension mechanism (*properties*)
 - Supports externally defined sublanguages
- Type mechanism
 - For defining common elements and styles
- Open semantic framework
 - To support automated reasoning

Ontology

- **Components**
 - Computational elements and data stores
- **Connectors**
 - Communication and coordination
- **Ports**
 - Component interfaces possibly including protocols
- **Roles**
 - Connector interfaces
- **Systems**
 - Configurations of components and connectors
 - Specified via *attachments*
- **Representations**
 - For hierarchical decomposition and multiple views
- **Rep-maps**
 - Specifies correspondence between levels of refinement

Example

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

Representations

- Explicit way of indicating structural refinement
- Element may have more than one representation
 - Different views
 - Alternative decompositions
- Parent element acts as a signature
- What properties must a refinement have in order to adequately express its parent?

Rep-Map

- Rep-Map (*abstraction map*) associates abstract component description with the detailed representation
 - Binding list mechanism for representing this abstraction
 - For example, component binding provides a way of associating a port on a component with some port within the representation
- Note that Acme does not define the precise nature of the relationship between an "outer" and an "inner" port/role

Example Representation

```
Component theComponent = {
  Port easyRequests;
  Port hardRequests;
  Representation {
    System details = {
      Component fastButDumbComponent = { Port p; };
      Component slowButSmartComponent = { Port p; };
    };
    Bindings {
      easyRequests to fastButDumbComponent.p;
      hardRequests to slowButSmartComponent.p
    };
  };
};
```

Properties

- Extension mechanism for ADL-specific tools
- Parsed but uninterpreted by Acme itself
- Example uses
 - Data types on ports/roles
 - Interaction protocols
 - Scheduling constraints
 - Resource consumption
- Property sublanguages
 - Visualization properties
 - For tools displaying architectural views
 - Temporal constraints

Properties Example

```
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 { synchronize : boolean = true;
                 max-roles : integer = 2;
                 protocol : Wright = "..."} }

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

Other Acme Features

- Semantic framework
 - Conversion of Acme models into predicates
- Types
 - For checking and abstraction (*Families*)
- Generics

Semantic Framework

- Ability to formally reason about Acme descriptions

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

Family

- A family provides a way of describing a set of similar architectures
 - Architectural style
- Element types that make up the vocabulary of the family
- Set of rules encoded as properties, for using the family

Example Family

```
Family PipesAndFiltersFam = {  
    Component Type FilterT = {};  
    Connector Type PipeT = {};  
};
```

```
System APFSystem : PipesAndFiltersFam = {  
    Component filter1 : FilterT =  
        new FilterT; Component filter2:  
    FilterT = new FilterT;  
    Connector pipe : PipeT = new PipeT; ...  
};
```

Modeling Steps

- Identify concepts that map to Acme
 - System, components, connectors, ports, role, representation
- Define property types and use them to augment the System description
- If appropriate define and use a family aggregating those types

Acme Limitations

- No model for behavior
- No model for functional properties
- No direct way of mapping to code
- In general, no semantics at all beyond typing rules