ACME BNF and Examples

Microsoft Component-Based Software Development Workshop
June 3-5, 1996

The ACME Development Team
(Bob Monroe, David Garlan, and Dave Wile)

Caveat: These examples are intended solely to explicate and explore some of the proposed features and aspects of the ACME architectural interchange language. The examples may be buggy and the architectures they describe tend to be overly simplified.

ACME BNF version 2.2
Last Revision Date: 4/5/96

```
NOTE The following BNF describes a grammar for specifying architectural
 descriptions.;

 NOTE ARCHITECTURAL-DESCRIPTION is the initial non-terminal for
  describing a valid system. ;

ARCHITECTURAL-DESCRIPTION :=
    { META-DECLARATION ^ '; '; } SYSTEM ^ '; { '; } ;

META-DECLARATION := TEMPLATE | STYLE || ;

NOTE                     S T Y L E S                        ;

 STYLE :=  'Style NAME '( { FORMAL-PARAMETER ^ ', } ')

             '= '{ STYLE-ELEMENT ^ '; { '; } '}
                { 'Harness AGGREGATE-DESCRIPTION } ;

   FORMAL-PARAMETER := NAME ^ ', ': SYNTACTIC-CLASS ;
   STYLE-ELEMENT := TEMPLATE | CONSTRAINT || ;

 NOTE                      T E M P L A T E S                       ;


 TEMPLATE :=  'Template NAME '( { FORMAL-PARAMETER ^ ', } ')
               { 'defining ( '( FORMAL-PARAMETER#def ^ ',  ') }
              DEFINITION ;

   DEFINITION := EXPLICIT-DEFINITION | EXPANSION || ;
     EXPLICIT-DEFINITION := '= D-CONSTITUENT || ;
       D-CONSTITUENT := D-COMPONENT | D-CONNECTOR |
                        D-PORT | D-ROLE | AGGREGATE-DESCRIPTION || ;
     AGGREGATE-DESCRIPTION := '{ DECLARATION ^ '; { ': } '} ;
     EXPANSION := ': SYNTACTIC-CLASS '= SPECIALIZATION ;

       SYNTACTIC-CLASS := ELEMENT-LITERAL-NAME | ELEMENT-LITERAL
                          | ELEMENT-LITERALS || ;
       ELEMENT-LITERAL-NAME := ELEMENT-LITERAL NAME ;
       ELEMENT-LITERAL := 'Component
                     | 'Connector
                     | 'Port
                     | 'Role
                     | 'Aggregation ;
       ELEMENT-LITERALS :=  'Ports
                     | 'Roles
                     | 'Representations
                     | 'Properties
                     | 'Assertions
                          | 'Aggregations;

   SPECIALIZATION := SPECIALIZATIONS | TEMPLATE-INVOCATION | NAME || ;

   SPECIALIZATIONS := SPECIALIZED SPECIFIC-SPECIALIZATION + ;
     SPECIALIZED := TEMPLATE-INVOCATION | NAME || ;
```

```
   TEMPLATE-INVOCATION := NAME '( { ACTUAL-PARAMETER ^ ', } ') ;
   ACTUAL-PARAMETER := CONSTITUENT-OR-NAME | ACTUAL-LIST || ;
   CONSTITUENT-OR-NAME :=  CONSTITUENT | TEMPLATE-EXPANSION | NAME || ;
   CONSTITUENT := COMPONENT | CONNECTOR |
                  PORT | ROLE || ;

   ACTUAL-LIST := CONSTRAINT-LIST |  CONSTITUENT-LIST || ;
   CONSTRAINT-LIST := MIXED-LIST | REPRESENTATION-LIST |
                      PROPERTY-LIST | ASSERTION-LIST || ;
     MIXED-LIST := '{ SINGLE-CONSTRAINT#L ^ '; '} ;
     REPRESENTATION-LIST := 'Representations '{ SINGLE-REPRESENTATION ^
'; '} ;
     PROPERTY-LIST := 'Properties ': '{ SINGLE-PREDICATE#P ^ '; '} |
     ASSERTION-LIST := 'Assertions ': '{ SINGLE-PREDICATE#A ^ '; '} |
   CONSTITUENT-LIST := '{ CONSTITUENT-OR-NAME ^ '; '} ;

   SPECIFIC-SPECIALIZATION  := CONFIG-OP { NAME } 'with DESCRIPTION ;

   CONFIG-OP := 'augment | 'override ;
NOTE Descriptions allow components without ports to be declared, and
connectors without roles, in order that components be specialized
without repeating the roles, I think (Wile--12/95);

   DESCRIPTION := COMPONENT-DESCRIPTION |
              CONNECTOR-DESCRIPTION |
              PORT-DESCRIPTION |
              ROLE-DESCRIPTION || ;

 NOTE                    S Y S T E M                              ;

SYSTEM := 'System NAME { ': NAME#STYLE }
                  '= '{ DECLARATION ^ '; { '; } '} ;

DECLARATION := CONSTRAINTS |
               COMPONENT | CONNECTOR | ROLE | PORT | ATTACHMENTS |
               TEMPLATE-EXPANSION | TEMPLATE-INVOCATION || ;


 NOTE                   C O M P O N E N T S                       ;



 COMPONENT := NAME '= D-COMPONENT ;
 D-COMPONENT :=  'Component ': COMPONENT-DESCRIPTION |> HasPorts ;
   COMPONENT-DESCRIPTION := '{
                  { 'Ports ': '{ EXPLICIT-PORT ^ '; { '; } '} { '; } }
                  { CONSTRAINTS { '; } }
               '} ;

 NOTE                     C O N N E C T O R S                      ;


 CONNECTOR := NAME '= D-CONNECTOR ;
  D-CONNECTOR :=  'Connector ': CONNECTOR-DESCRIPTION |> HasRoles ;
   CONNECTOR-DESCRIPTION :=  '{
                  { 'Roles ': '{ EXPLICIT-ROLE ^ '; { '; } '} { '; } }
                  { CONSTRAINTS { '; } }
               '} ;

 NOTE                        P O R T S                             ;


 EXPLICIT-PORT := TEMPLATE-EXPANSION | IMPLICIT-PORT || ;
 IMPLICIT-PORT := NAME { '= D-PORT } || ;
```

```
 PORT := NAME '= D-PORT ;
   D-PORT :=  'Port ': PORT-DESCRIPTION ;
   PORT-DESCRIPTION := '{ { CONSTRAINTS { '; } } '} ;


 NOTE                    R O L E S                                ;

 EXPLICIT-ROLE := TEMPLATE-EXPANSION | IMPLICIT-ROLE || ;
 IMPLICIT-ROLE := NAME { '= D-ROLE } || ;

 ROLE := NAME '= D-ROLE ;
  D-ROLE :=  'Role ': ROLE-DESCRIPTION ;
   ROLE-DESCRIPTION := '{ { CONSTRAINTS { '; } } '} ;


CONSTRAINTS := { NAME '= } ( SINGLE-CONSTRAINT | CONSTRAINT-LIST ) ;

SINGLE-CONSTRAINT := PROPERTY | ASSERTION | REPRESENTATION || ;
  PROPERTY := 'Property ': SINGLE-PREDICATE ;
  ASSERTION := 'Assertion ': SINGLE-PREDICATE ;
  REPRESENTATION := 'Representation ': SINGLE-REPRESENTATION ;
  SINGLE-PREDICATE := EXTENSION-DESCRIPTION | RAW-PROPERTY || ;

  EXTENSION-DESCRIPTION := LANGUAGE ': EXTERNALLY-PARSED;
    LANGUAGE := NAME |> NamesLanguage;
    EXTERNALLY-PARSED := GRAMMAR Language [ Nonterminals ]
                              <| StartLanguage |> FinishLanguage;

 NOTE             M I S C E L L A N E O U S   P R O D U C T I O N S  ;

SINGLE-REPRESENTATION := { NAME '= }
                 '{ ( EXTERNAL-REP | NAME#SYSTEM | SYSTEM )
                   { 'map ': ABSTRACTION-MAP } '};

 EXTERNAL-REP := 'External { ': TYPE } '= '{ ATT-VALUE '} ';
                 |> VALID-REPRESENTATION ;

 ABSTRACTION-MAP := BINDINGS-MAP-DEF | OTHER-MAP-DEF || ;

 BINDINGS-MAP-DEF :=  'Bindings '= '{ BINDING ^ '; '} ;
 BINDING :=  ROLE-BINDING | PORT-BINDING || ;
    ROLE-BINDING := ROLE-NAME#N 'to ROLE-NAME#V ;
    PORT-BINDING := PORT-NAME#N 'to PORT-NAME#V ;

 ROLE-NAME := NAME#CONNECTOR '. NAME#ROLE ;
 PORT-NAME := NAME#COMPONENT '. NAME#PORT ;

 OTHER-MAP-DEF := LANGUAGE#ABSTRACTION-MAP-TYPE ': EXTERNALLY-PARSED ;

  RAW-PROPERTY :=  ATTRIBUTE { ': TYPE } '= ATT-VALUE ;

NOTE TYPE will perhaps become and ODL/IDL type;

   TYPE := NAME || ;
   ATTRIBUTE := NAME || ;
   ATT-VALUE :=  STRING | NUMBER | NAME || ;


 ATTACHMENTS := { NAME '= } D-ATTACHMENTS ;
   D-ATTACHMENTS := 'Attachments ': ATTACHMENTS-DESCRIPTION ;
   ATTACHMENTS-DESCRIPTION := '{ ATTACHMENT ^ '; { '; } '} ;
 ATTACHMENT := PORT-NAME 'to ROLE-NAME ;

 TEMPLATE-EXPANSION := NAME-LIST '= SPECIALIZATION ;
```

```
   NAME-LIST := NAME | LIST-OF-NAMES || ;
   LIST-OF-NAMES := '( OPTIONAL-NAME ^ ', ') ;

   NOTE Normally, the following is bad form, but in this context it's
OK;
   OPTIONAL-NAME := { NAME } ;

 BINDING :=  ROLE-BINDING | PORT-BINDING || ;
   ROLE-BINDING := ROLE-NAME#N 'to ROLE-NAME#V ;
   PORT-BINDING := PORT-NAME#N 'to PORT-NAME#V ;

 ROLE-NAME := NAME#CONNECTOR '. NAME#ROLE ;
 PORT-NAME := NAME#COMPONENT '. NAME#PORT ;

 NAME := LEXEME <| architectural-entity ;
 STRING := LEXEME <| is-string ;
 NUMBER := LEXEME <| is-number ;
```

Example: pf_style_def.acme
Date: 4/4/96

Purpose: Specify an (overly) simple pf style strictly for showing off acme features in a slideshow.

```
// Describe a simple pf style that prevents cycles.  This style
//  definition demonstrates ACME 2.0's ability to specify a family of
//  architectures as well as individual architectural instances.

Style pipe-filter() = {

  // Declare component templates.
  Template filter() =
     Component: {
          Ports: { stdin, stdout }
          Properties: { type : string = "filter" };  }


  // Declare pipe templates.  Note that the pipe templates
  //  hook the filters up as well as declaring a pipe connector.  The
  //  expressive power of templates has been greatly expanded from
  //  that of earlier versions.
  Template pipe(source_port, sink_port : Port)
           defining(conn:Connector) =
   {  conn = Connector: {
              Roles: {source, sink};
            Properties : {type : string = pipe}; }
      Attachments: { source_port to conn.source;
                  sink_port to conn.sink; } }



  // Declare properties and constraints specified in the style.
  //  Constrain the vocabulary used in this style so that all
  //  components are filters and all connectors are pipes
  Assertion : FOPL = {forall comp:components(System) |
                      (comp.property(type) = "filter") };

  Assertion : FOPL = {forall conn:connectors(System) |
                      (conn.property(type) = "pipe") };

  // Specify a "no-cycles" assertion using a transitive closure
  //  expression
  Assertion : FOPL = {forall comp:components(System) |
                  (comp,comp) !elt of connected+ };

} // end Style pipe-filter definition
```

Example: pf_examples.acme
Date: 4/4/96

Purpose: Use a simple style to show the following features of ACME 2.0:

- **more expressive template invocation**
- **external language integration via visualization language**
- **refinement maps.**

```
// Describe a simple pf style that prevents cycles.  This style
//  definition demonstrates ACME 2.0's ability to specify a family of
//  architectures as well as individual architectural instances.

Style pipe-filter() = {

  // Declare component templates.
  Template filter() =
     Component: {
          Ports: { stdin, stdout }
          Properties: { type : string = "filter" };  }


  // Declare pipe templates.  Note that the pipe templates
  //  hook the filters up as well as declaring a pipe connector.  The
  //  expressive power of templates has been greatly expanded from
  //  that of earlier versions.
  Template pipe(source_port, sink_port : Port)
           defining(conn:Connector) =
   {  conn = Connector: {
              Roles: {source, sink};
            Properties : {type : string = pipe}; }
      Attachments: { source_port to conn.source;
                  sink_port to conn.sink; } }



  // Declare properties and constraints specified in the style.
  //  Constrain the vocabulary used in this style so that all
  //  components are filters and all connectors are pipes
  Assertion : FOPL = {forall comp:components(System) |
                    (comp.property(type) = "filter") };

  Assertion : FOPL = {forall conn:connectors(System) |
                      (conn.property(type) = "pipe") };

  // Specify a "no-cycles" assertion using a transitive closure
  //  expression
  Assertion : FOPL = {forall comp:components(System) |
                  (comp,comp) !elt of connected+ };

} // end Style pipe-filter definition



// Describe an instance of a system constructed using the pipe-filter
//  style.
```

```
System simple_pf : pipe-filter = {

  // Declare the components to be used in this design with the filter
  //  template given in the pipe-filter style definition.
  smooth = filter();
  detect_errors = filter();
  show_tracks = filter();

  // Declare the connectors used in the system.  These more powerful
  //  template invocations now make use of the gensym facility and do
  //  not need to be named.  The templates also make all of the
  //  appropriate attachments.
  () = pipe(smooth.stdout, detect_errors.stdin);
  () = pipe(detect_errors.stdout, show_tracks.stdin);

  // Here I use an external language for visualizations.  In this
  // case the external rep is a first class object
  augment smooth with {
     visualization : {
              vis-x = 50; vis-y = 50;
              vis-height = 100; vis-width = 100;
              vis-shape = rectangle;
              vis-color = black;
            }

  // alternatively, we can embed a visual description as a property:
  augment detect_errors with {
     Property : visualization = {
              vis-x = 50; vis-y = 50;
              vis-height = 100; vis-width = 100;
              vis-shape = rectangle;
              vis-color = black;
            }

  // Here we add an acme representation to the show_tracks filter:
  augment show_tracks with {
    Representation: {
      show_tracks_subsystem;   //identifier, defined elsewhere
      Map : Bindings = {
           stdout to show_tracks_subsystem.stdout;
           stdin  to show_tracks_subsystem.stdin;
            }
    };

  // Here we add an external representation with an abstraction map
  //  that attaches with an external (non-acme) language mapping
  //  description.  This description is a non-acme language that needs
  //  to be interpreted by an ADL specific tool.
  augment detect_errors with {
     Representation: {
       External : source-code = { "/FILTER_LIB/kalman_filter.c" };
       // note: fd_mapping is a non-acme mapping of file-descriptors
       //       to ports
       Map : fd_mapping =
              {  map stdout to code_fd 1
              map stdin to code_fd 0 };
       };
     };

}  // end simple_pf description.
```

Example: Instance description of simple LAS system
Date: 4/4/96

Purpose: Describe a simple proposed architecture for the IWSSD case study of the London Ambulance Service dispatch system. This version uses only basic instance-level (kernel) features of ACME v2.0.

```
// Instance based example - simple LAS architecture:

System LAS_CAD = {

// system components
   call_entry = component {
       ports : { send_call_msg }
     }
   incident_mgr = component {
       ports : { map_request, incident_info_requests,
             send_incident_info, receive_call_msg }
     }
   resource_mgr = component {
       ports : { map_request, incident_info_request,
               receive_incident_info, send_dispatch_request }
     }
   dispatcher = component {
       ports : { receive_dispatch_request }
     }

   map_server = component {
       ports : { request_port1, request_port2 }
     }


// system connectors

   // message passing connectors
   call_info_channel = connector {
       roles : { from, to }
       properties : { conn_type : string = message_pass_channel;
                   msg_flow : flow_direction = from -> to; }
     }
   incident_update_channel = connector {
       roles : { from, to }
       properties : { conn_type : string = message_pass_channel;
                   msg_flow : flow_direction = from -> to; }
     }
   dispatch_request_channel = connector {
       roles : { from, to }
       properties : { conn_type : string = message_pass_channel;
                   msg_flow : flow_direction = from -> to; }
     }

   // RPC connnectors
   incident_info_request_rpc = connector {
       roles : { client_end, server_end }
       property : { conn_type : string = RPC; }
```

```
    }
  map_request_rpc1 = connector {
      roles : { client_end, server_end }
      property : { conn_type : string = RPC; }
    }
  map_request_rpc2 = connector {
      roles : { client_end, server_end }
      property : { conn_type : string = RPC; }
    }


// connect up the attachments
  incident_info_path = attachments : {
     // calls to incident_manager
       call_entry.send_call_msg to call_info_channel.from;
     incident_mgr.receive_call_msg to call_info_channel.to;

     // incident updates to resource manager
     incident_mgr.send_incident_info to
                 incident_update_channel.from;
     resource_mgr.receive_incident_info to
                 incident_update_channel.to;

     // dispatch requests to dispatcher
     respource_mgr.send_dispatch_request to
                 dispatch_request_channel.from;
     dispatcher.receive_dispatch_request to
                 dispatch_request_channel.to;
  }

  rpc_requests = attachments : {
     // calls to map server
     incident_mgr.map_request to map_request_rpc1.client_end;
     map_server.request_port1 to map_request_rpc1.server_end;
     resource_mgr.map_request to map_request_rpc2.client_end;
     map_server.request_port2 to map_request_rpc2.server_end;

     // incident info from incident_mgr
     resource_mgr.incident_info_request to
                 incident_info_request_rpc.client_end;
     incident_mgr.incident_info_requests to
                 incident_info_request_rpc.server_end;
  }


// system assertions and properties:
  assertion : real-time-constraint :
        {latency (send_call_msg to dispatch_request_msg) < 3 min };

}
```

Example: LAS example extended with a style definition and templates
Date: 4/4/96

Purpose: Describe a simple proposed architecture for the IWSSD case study of the London Ambulance Service dispatch system. This version extends the instance based description by using templates.

```
// template based example - simple LAS architecture:

// Describe the style that we are using for the LAS system.

Style envt_control() = {

  // declare component templates.  It's not clear that component
  // templates buy us much here.
  Template server(p:ports) defining comp =
    { comp = Component: {
          Ports: p;
          Properties: { defines_interfaces = server };
    }

  Template manager(p:ports) defining comp =
    { comp = Component: {
          Ports: p;
          Properties: { defines_interfaces = manager };
    }

  Template envt_interface(p:ports) defining comp =
    { comp = Component {
          Ports: p;
          Properties: { defines_interfaces = envt_interface };
    }


  // declare connector templates.  Note that the connector templates
  //  hook the connectors up as well.

  Template rpc(client_port, server_port : Port)
          defining(conn:Connector) =
   {  conn = Connector: {
              Roles: {server_role, client_role};
             Properties : {type : string = rpc_connector};
      }
      Attachments: { client_port to conn.client_role;
                  server_port to conn.server_role; }
   }


  Template msg_channel(sender_port, receiver_port : Port)
          defining(conn:Connector) =
   {  conn = Connector: {
              Roles: {source_role, sink_role};
              Properties: {
               type : string = message_pass_channel;
                 msg_flow : flow_direction = source_role->sink_role;
```

```
            }
        }
        Attachments: { sender_port to conn.source_role;
                       receiver_port to conn.sink_role; }
    }


    // declare properties and constraints specified in the style.

    // constrain the vocabulary used in this style.
    Assertion : FOPL = {forall comp:components(System) |
         (comp.property(defines_interface) contains "server" or
          comp.property(defines_interface) contains "manager" or
          comp.property(defines_interface) contains "envt_interface") };

    Assertion : FOPL = {forall conn:connectors(System) |
          (conn.property(type) = "rpc_connector" or
           conn.property(type) = "message_pass_channel") };


    // specify topological type attachments constraints
    Assertion : FOPL = {forall s,m:server(s) and manager(m) |
           attached(m,s,conn) => (conn.property(type) = "rpc_connector")
};

    Assertion : FOPL = {forall e,m:envt_interface(e) and manager(m) |
           attached(e,m,conn) =>
            (conn.property(type) = "message_pass_channel") };

} // end Style envt_control definition


System LAS_CAD : envt_control = {

// system properties:

// system components
    call_entry = envt_interface( {send_call_msg} );
    dispatcher = envt_interface( {receive_dispatch_request} );

    map_server = server(request_port1, request_port2);

    resource_mgr = manager( {map_request, incident_info_request,
                receive_incident_info, send_dispatch_request} );
    incident_mgr = manager( {map_request, incident_info_requests,
                        send_incident_info, receive_call_msg} );

    // adapt the incident_mgr to support both server and manager
    // interfaces.
    overide incident_mgr.property(defines_interfaces) with "manager,
server"


// define system connectors.  with the new form of template
//  definitions this includes the ability to do attachments in the
//  templates.

    // message passing connectors
    call_info_channel = msg_channel(call_entry.send_call_msg,
                              incident_mgr.receive_call_msg);

    incident_update_channel =
              msg_channel(incident_mgr.send_incident_info,
                    resource_mgr.receive_incident_info);
```

```
    dispatch_request_channel =
              msg_channel(resource_mgr.send_dispatch_request,
                    dispatcher.receive_dispatch_request);

    // RPC connnectors
    incident_info_request_rpc = rpc(resource_mgr.incident_info_request,
                               incident_mgr.incident_info_requests);

    map_request_rpc1 = rpc(incident_mgr.map_request,
                        map_server.request_port1)

    map_request_rpc2 = rpc(resource_mgr.map_request,
                        map_server.request_port2);

}  // end LAS system description.
```

Example: Further extend LAS example with representations
Date: 4/4/96

Purpose: Describe a simple proposed architecture for the IWSSD case study of the London Ambulance Service dispatch system. This version template based description by including two representations and their associated abstraction mappings.

```
// template based example with refinements - simple LAS architecture:

// Describe the style that we are using for the LAS system.

Style envt_control() = {

  // declare component templates.  It's not clear that component
  // templates buy us much here.
  Template server(p:ports) defining comp =
    { comp = Component: {
          Ports: p;
          Properties: { defines_interfaces = server };
    }

  Template manager(p:ports) defining comp =
    { comp = Component: {
          Ports: p;
          Properties: { defines_interfaces = manager };
    }

  Template envt_interface(p:ports) defining comp =
    { comp = Component {
          Ports: p;
          Properties: { defines_interfaces = envt_interface };
    }


  // declare connector templates.  Note that the connector templates
  //  hook the connectors up as well.

  Template rpc(client_port, server_port : Port)
          defining(conn:Connector) =
   {  conn = Connector: {
              Roles: {server_role, client_role};
             Properties : {type : string = rpc_connector};
      }
      Attachments: { client_port to conn.client_role;
                  server_port to conn.server_role; }
   }


  Template msg_channel(sender_port, receiver_port : Port)
          defining(conn:Connector) =
   {  conn = Connector: {
              Roles: {source_role, sink_role};
              Properties: {
               type : string = message_pass_channel;
```

```
                msg_flow : flow_direction = source_role->sink_role;
            }
      }
      Attachments: { sender_port to conn.source_role;
                receiver_port to conn.sink_role; }
   }


   // declare properties and constraints specified in the style.

   // constrain the vocabulary used in this style.
   Assertion : FOPL = {forall comp:components(System) |
        (comp.property(defines_interface) contains "server" or
         comp.property(defines_interface) contains "manager" or
         comp.property(defines_interface) contains "envt_interface") };

   Assertion : FOPL = {forall conn:connectors(System) |
         (conn.property(type) = "rpc_connector" or
          conn.property(type) = "message_pass_channel") };


   // specify topological type attachments constraints
   Assertion : FOPL = {forall s,m:server(s) and manager(m) |
          attached(m,s,conn) => (conn.property(type) = "rpc_connector")
};

   Assertion : FOPL = {forall e,m:envt_interface(e) and manager(m) |
          attached(e,m,conn) =>
           (conn.property(type) = "message_pass_channel") };

} // end Style envt_control definition


System LAS_CAD : envt_control = {

// system properties:

// system components
   call_entry = envt_interface( {send_call_msg} );
   dispatcher = envt_interface( {receive_dispatch_request} );

   incident_mgr = manager( {map_request, incident_info_requests,
                       send_incident_info, receive_call_msg} );
   // adapt the incident_mgr to support both server and manager
   // interfaces.
   overide incident_mgr.property(defines_interfaces) with "manager,
server"


   resource_mgr = manager( {map_request, incident_info_request,
               receive_incident_info, send_dispatch_request} );
   // add the resource_mgr_refinement system as a representation of
   //  the resource manager component.
   augment resource_mgr with {
     Representation: {
        resource_mgr_refinement
      map : Bindings = {
            map_request to planning_engine.map_request;
            incident_info_request to
                  planning_engine.incident_info_requests;
            receive_incident_info to
                  planning_engine.receive_incident_info;
            send_dispatch_request to
                    planning_engine.send_dispatch_request;
```

```
        }
    };


   // the map server has an external representation with a non-ACME,
   //  user-specified abstraction mapping.
   map_server = server(request_port1, request_port2);
   augment map_server with {
     Representation: {
       External : source-code = { "/SERVER_LIB/map_server.c" };
       // note: socket_mapping is a non-acme mapping.
       map : socket_mapping =
              {  route request_port1 to <serverhost>.socket 1001;
             route request_port2 to <serverhost>.socket 1002; };
     };


// define system connectors.  with the new form of template
//  definitions this includes the ability to do attachments in the
//  templates.

   // message passing connectors
   call_info_channel = msg_channel(call_entry.send_call_msg,
                           incident_mgr.receive_call_msg);

   incident_update_channel =
             msg_channel(incident_mgr.send_incident_info,
                   resource_mgr.receive_incident_info);

   dispatch_request_channel =
             msg_channel(resource_mgr.send_dispatch_request,
                   dispatcher.receive_dispatch_request);

   // RPC connnectors
   incident_info_request_rpc = rpc(resource_mgr.incident_info_request,
                             incident_mgr.incident_info_requests);

   map_request_rpc1 = rpc(incident_mgr.map_request,
                       map_server.request_port1)

   map_request_rpc2 = rpc(resource_mgr.map_request,
                       map_server.request_port2);

}  // end LAS system description.



// define the refinement of the resource manager's architecture

System resource_mgr_refinement : envt_control = {

  // define the components
  hospital_resource_db = server( {server_request_port} );
  ambulance_resource_db = server( {server_request_port} );

  planning_engine = manager( {hospital_info, ambulance_info,
                        map_request, incident_info_requests,
                        receive_incident_info,
                        send_dispatch_request} );

  // define the connectors and attach.  note that the planning engine
  //  has four unbound and unattached ports that will be bound when
  //  the system is specified as a representation.
```

```
  hospital_info_request_rpc = rpc(planning_engine.hospital_info,
                      hospital_resource_db.server_request_port);

  ambulance_info_request_rpc = rpc(planning_engine.ambulance_info,
                    ambulance_resource_db.server_request_port);

  // add any properties or assertions about the subsystem below:

}  // end System resource_mgr_refinement definition
```