	ATL Transformation METAH2ACME	Author Baudry Julien Jul.baudry <*at*> gmail.com
	Documentation	May 23rd 2006 Release 0.1

1. ATL Transformation Example

1.1. Example: METAH → ACME

The ACME interchange format was originally conceived as a way to share tool capabilities rovided by a particular ADL with other ADLs, while avoiding the production of many pair wise language translators. The MetaH architectural description language (ADL) and associated toolset support architectural modeling of embedded real-time system applications.

1.1.1. Metamodels

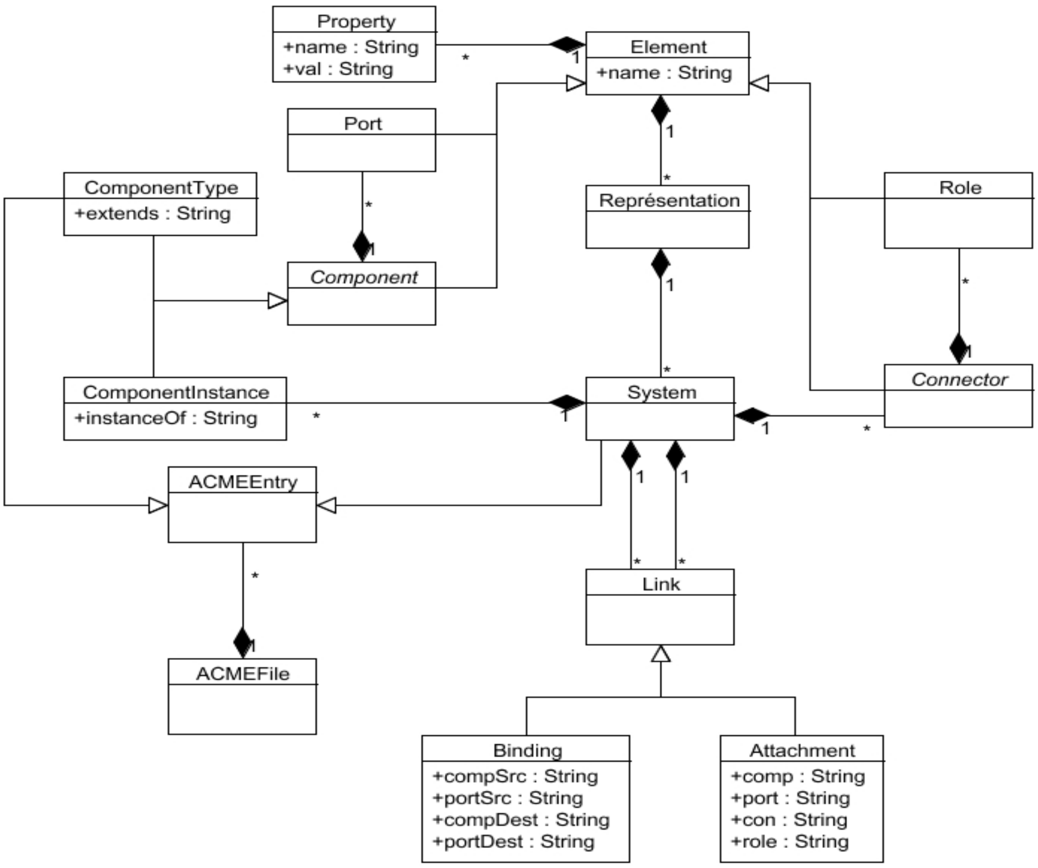

Figure 1 the ACME metamodel

	ATL Transformation **METAH2ACME**	**Author** **Baudry Julien** **Jul.baudry <*at*> gmail.com**
	Documentation	May 23rd 2006 Release 0.1

This transformation is based on a simplified ACME metamodel. An ACME file is modelized by an ACME File element. This element is composed of ACME. All entries inherit, of the abstract ACME Entry element. There are 2 possible entry types: System and Component Type.
The transformation also relies on a limited subset of the METAH language definition. The metamodel considered here is described in Figure 2, and provided in Appendix II in km3 format.

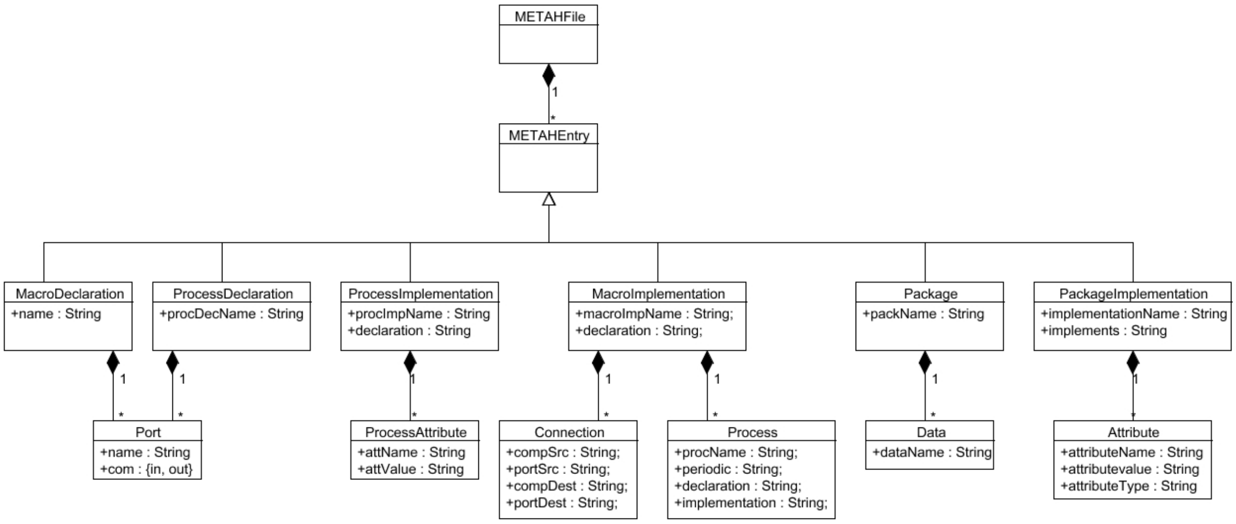

Figure 2 the METAH metamodel

Within this metamodel, a METAH File is associated with a METAHFile element. Such an element is composed of several METAHEntry. There are 2 possible entry types: Package, Package Implementation, Process Declaration, Process Implementation, Macro Declaration, Macro Implementation.

1.1.1 Rule Specification

To facilitate the translation to ACME we use the ACME Family construct to declare a collection of standard MetaH-related types, types used in any MetaH to ACME translation.

```
property type MH_mode_subclass =
 enum {MH_initial, MH_other};

property type MH_port_subclass =
 enum {MH_in, MH_out};

property type MH_process_subclass =
 enum {MH_periodic, MH_aperiodic};
```

```
property type MH_event_subclass =
 enum {MH_interrupt, MH_signal, MH_nudge, MH_node};

property type MH_execution_path = sequence;

property type MH_error_path = sequence

property type MH_Implementation_name = string;

property type MH_Interface_name = string;

port type MH_port = {};

port type MH_event = {};

component type MH_mode =
 {port MH_event_port: MH_port
       = {property MH_port_subclass = MH_in;};};
component type MH_macro = {};

component type MH_monitor = {};

component type MH_package = {};

component type MH_subprogram = {};


component type MH_process =

 {port MH_event_port: MH_port

       = {property MH_port_subclass = MH_in;};};

component type MH_error_model = {};

component type MH_error_state = {};

connector type MH_connector =

 {roles {MH_source; MH_sink};

 property MH_port_identifier: string;};
```

These are the rules to transform a METAH model to a ACME model :

- For a root METAHFile element, an ACMEFile element is created. It contains the same entries.
- For a Process Declaration or a Macro Declaration element, a Component Type element is created :
  - o with the same name,

  - linked to the same port,
  - and extends a predefined ACME type "MH_process".

- For a METAH Port element, an ACME Port element is created :
  - with the same name,
  - and contain 2 properties :
    - the first property "MH_port_type" indicate its type,
    - the second property "MH_port_subclass" indicate port direction, "MH_out" or "MH_In"

- For a Process Implementation element, a Component Type element is created :
  - The name contains the name of the process declaration and the name of the process implementation (a process implementation always refers to a process declaration),
  - The properties contains all available information on the process attributes,
  - The component type extends the component type created with the process declaration referring to him.

- For a Process Attribute element, a Property element is created :
  - With the same name,
  - The value of the attribute is the value of the process and the type of the value

- For a Macro Implementation element, several element are created :
  - A component type, with the same name, extending the component type created by Macro Declaration,
  - A representation, contained by the component type,
  - A system, contained by the representation. Its name is "MH_Little_System". The component declaration contains all available information on the process and the attachments contains all available information on the connections.

- For a process element, an Component Instance element is created :
  - With the same name,
  - The field instanceOf contains the name of the process declaration and the name of the process implementation (a process implementation always refers to a process declaration),
  - The component contains a property to indicate if the process is periodic or not.

- For a connection element, there is two cases :
  - If the connection is between 2 processes, then we create a connector to connect the component created by the processes. The connector contains two roles, and two attachments are created to connect all these elements.
  - Else, we created a binding element.

1.1.3 ATL Code

This ATL code for the ACME to METAH transformation consists of 10 rules. The main rule is the first rule in the following code.

```
module METAH2ACME;
create OUT : ACME from IN : METAH;


--@begin METAHFile2ACMEFile
--@comments METAHFile and ACMEFile are root element of ACME and
METAH metamodel
rule METAHFile2ACMEFile {
	from
		m : METAH!METAHFile
	to
		a : ACME!ACMEFile (
			entries <- m.entries
		)
	}
--@end METAHFile2ACMEFile

--@begin ProcessDeclaration2ComponentType
rule ProcessDeclaration2ComponentType {
	from
		p : METAH!ProcessDeclaration
	to
		c: ACME!ComponentType (
			name <- p.procDecName,
			ports <- p.ports,
			extend <- 'MH_Process'
		)
}
--@end ProcessDeclaration2ComponentType


--@begin Port2Port
--@comments Transform a METAH port into an ACME port with two
properties
rule Port2Port {
	from
		p1 : METAH!Port
	to
		p2 : ACME!Port (
			name <- p1.portName,
			property <- Sequence {port_type,port_subclass}
		),
		port_type : ACME!Property(
			name <- 'MH_port_type',
			val <- p1.portType
		),
```

```
            port_subclass : ACME!Property(
                  name <- 'MH_port_subclass',
                  val <- 'MH_'+p1.portCom
            )
}
--@end Port2Port


--@begin ProcessImplementation2ComponentType
rule ProcessImplementation2ComponentType {
      from
            p1 : METAH!ProcessImplementation
      to
            p2 : ACME!ComponentType (
                  name <- p1.declaration+'_'+p1.procImpName,
                  property <- p1.processAttributes,
                  extend <- p1.declaration
      )
}
--@end ProcessImplementation2ComponentType

--@begin ProcessAttribute2Property
rule ProcessAttribute2Property {
      from
            a : METAH!ProcessAttribute
      to
            p : ACME!Property (
                  name <- 'MH_'+a.attName,
                  val <- a.attValue.toString().concat('
').concat(a.attValueType)
            )
}
--@end ProcessAttribute2Property

--@begin MacroDeclaration2ComponentType
rule MacroDeclaration2ComponentType {
      from
            m : METAH!MacroDeclaration
      to
            c : ACME!ComponentType (
                  name <- m.name,
                  ports <- m.ports,
                  extend <- 'MH_macro'
            )
}
--@end MacroDeclaration2ComponentType
```

```
--@begin MacroImplementation2ComponentType
rule MacroImplementation2ComponentType {
	from
		m : METAH!MacroImplementation
	to
		c : ACME!ComponentType(
			name <- m.macroImpName,
			extend <- m.declaration,
			representations <- Sequence {r}
		),
		r : ACME!Representation (
			systems <- Sequence {s}
		),
		s : ACME!System(
			 name <- 'MH_little_System',
			 componentDeclaration <- m.process,
			 attachments <- m.connections
		)
}
--@end MacroImplementation2ComponentType

--@begin Process2Component
rule Process2Component {
	from
		p : METAH!Process
	to
		c : ACME!ComponentInstance (
			name <- p.procName,
			instanceOf <- p.declaration+'_'+p.implementation,
			property <- period
		),
		period : ACME!Property (
			name <- 'MH_Process_subclass',
			val <-  'MH_'+p.periodic
		)
}
--@end Process2Component

--@begin Connection2Connector
rule Connection2Connector {
	from
		c1 : METAH!Connection (
			not((c1.compSrc.oclIsUndefined())or
			(c1.compDest.oclIsUndefined()))
		)
```

```
	to
		attach1 : ACME!Attachment (
			comp <- c1.compSrc,
			port <- c1.portSrc,
			con <- c1.compSrc+'_to_'+c1.compDest,
			role <- 'MH_sink',
			systemAttachment <-
thisModule.resolveTemp(METAH!MacroImplementation.allInstances()-
>asSequence()->first(),'s')
		),
		attach2 : ACME!Attachment (
			comp <- c1.compDest,
			port <- c1.portDest,
			con <- c1.compSrc+'_to_'+c1.compDest,
			role <- 'MH_source',
			systemAttachment <-
thisModule.resolveTemp(METAH!MacroImplementation.allInstances()-
>asSequence()->first(),'s')
		),
		c2 : ACME!Connector (
			 name <- c1.compSrc+'_to_'+c1.compDest,
			 roles <- Sequence {r1,r2},
			 system <-
thisModule.resolveTemp(METAH!MacroImplementation.allInstances()-
>asSequence()->first(),'s')
		),
		r1 : ACME!Role (
			name <- 'MH_sink'
		),
		r2 : ACME!Role (
			name <- 'MH_source'
		)
}
--@end Connection2Connector

--@begin Connection2Binding
rule Connection2Binding {
	from
		b1 : METAH!Connection (

	((b1.compSrc.oclIsUndefined())or(b1.compDest.oclIsUndefined()))
		)
	to
		b2 : ACME!Binding (
			compSrc <- b1.compSrc,
			compDest <- b1.compDest,
```

```
                portDest <- b1.portDest,
                portSrc <- b1.portSrc,
                systemBinding <-
thisModule.resolveTemp(METAH!MacroImplementation.allInstances()-
>asSequence()->first(),'s')
           )
}
--@end Connection2Binding
```

1.1.4 Transformation example

Here is the METAH model we wanted to transform :

```
process P1 is
 p1_input : in port PORT_TYPE.ANY_TYPE;
 update : out port PORT_TYPE.ANOTHER_TYPE;
 feedback : in port PORT_TYPE.ANOTHER_TYPE;
end P1;

process implementation P1.EXAMPLE is
attributes
 self'Period := 25 ms;
 self'SourceTime := 2 ms;
end P1.EXAMPLE;

process P2 is
 p1_result : out port PORT_TYPE.ANY_TYPE;
 update : out port PORT_TYPE.ANOTHER_TYPE;
 feedback : in port PORT_TYPE.ANOTHER_TYPE;
end P2;

process implementation P2.EXAMPLE is
attributes
 self'Period := 50 ms;
 self'SourceTime := 5 ms;
end P2.EXAMPLE;



macro M is
 m_in : in port PORT_TYPE.ANY_TYPE;
 m_out : out port PORT_TYPE.ANY_TYPE;
end M;

macro implementation M.EXAMPLE is
```

```
 P2 : periodic process p2.example;
 P1 : periodic process p1.example;
connections
 p2.feedback <- p1.update;
 p1.feedback <- p2.update;
 m_out <- p2.p2_result;
 p1.p1_input <- m_in;
end M.EXAMPLE;
```

And here is the result of the transformation :

```
Family MetaH_Family()=
{/ * BEGIN STANDARD METAH DECLARATIONS * /
     ..............
 / * BEGIN EXAMPLE SPECIFIC DECLARATIONS * /

  component type P1 extends PH_Process with{
    port p1_input : MH_port
      = {
      property MH_port_type=ANY_TYPE;
      property MH_port_subclass=MH_in;}
    ;
    port update : MH_port
      = {
      property MH_port_type=ANOTHER_TYPE;
      property MH_port_subclass=MH_out;}
    ;
    port feedback : MH_port
      = {
      property MH_port_type=ANOTHER_TYPE;
      property MH_port_subclass=MH_in;}
    ;
  };

  component type P1_EXAMPLE extends P1 with{
    property MH_Period="25 ms";
    property MH_SourceTime="2 ms";
  };

  component type P2 extends PH_Process with{
    port p1_result : MH_port
      = {
      property MH_port_type=ANY_TYPE;
      property MH_port_subclass=MH_out;}
    ;
```

```
    port update : MH_port
      = {
      property MH_port_type=ANOTHER_TYPE;
      property MH_port_subclass=MH_out;}
    ;
    port feedback : MH_port
      = {
      property MH_port_type=ANOTHER_TYPE;
      property MH_port_subclass=MH_in;}
    ;
  };

  component type P2_EXAMPLE extends P2 with{
    property MH_Period="50 ms";
    property MH_SourceTime="5 ms";
  };

  component type M extends MH_macro with{
    port m_in : MH_port
      = {
      property MH_port_type=ANY_TYPE;
      property MH_port_subclass=MH_in;}
    ;
    port m_out : MH_port
      = {
      property MH_port_type=ANY_TYPE;
      property MH_port_subclass=MH_out;}
    ;
  };

  component type EXAMPLE extends M with{
    Representation{
      system MH_little_system={
        component P2=new p2_example extended with{
          property MH_Process_subclass=MH_periodic;
        };
        component P1=new p1_example extended with{
          property MH_Process_subclass=MH_periodic;
        };
        Connector p1_to_p2=new MH_connector extended with{};
        Connector p2_to_p1=new MH_connector extended with{};
        Attachments{
          p2.feedback to p2_to_p1.MH_sink;
          p1.feedback to p1_to_p2.MH_sink;
          p2.update to p1_to_p2.MH_source;
          p1.update to p2_to_p1.MH_source;
```

```
          };
        }
          Bindings{
            p1.p1_input to m_in;
            m_out to p2.p2_result;
          };
        ;
      };
    };

  };

   system MH_system : MetaH_Family =
    {component MH_component = new M_example;};
```

I. METAH metamodel in km3 format

```
--  @name MetaH
--  @version 1.0
--  @domains developping new architectural design and analysis tools
```

```
--  @authors Julien Baudry (jul.baudry@gmail.com)
--  @date 2006/05/27
--  @description High-level software and hardware architecture
specification language
--  @see This metamodel has been extracted from information
available on this site :
http://www.sei.cmu.edu/pub/documents/98.reports/pdf/98sr006.pdf

package MetaH {

	--@begin METAHFile
	class METAHFile {
		reference entries[*] container : METAHEntry;
	}
	--@end METAHFile

	--@begin METAHEntry
	abstract class METAHEntry {
	}
	--@end METAHEntry


	--@begin Package
	--@comments MetaH package objects are used to describe
collections of subprograms and statically allocated
	--@comments and persistent data. Package objects may be
shareable across processes and may contain ports.
	class Package extends METAHEntry {
		attribute packName : String;
		reference data [*] ordered container : Data;
	}
	--@end Package

	--@begin PackageImplementation
	class PackageImplementation extends METAHEntry {
		attribute implementationName : String;
		attribute implements : String;
		reference attributes [*] ordered container : Attribute;
	}
	--@end PackageImplementation

	--@begin Data
	--@comments Type define in a Package
	class Data {
		attribute dataName : String;
	}
```

```
	--@end Data

	--@begin Attribute
	class Attribute {
		attribute attName : String;
		attribute attValue : Integer;
		attribute attValueType : String;
		attribute attType : String;
	}
	--@end Attribute

	--@begin ProcessDeclaration
	--@comments A MetaH process describes a single, schedulable
thread of execution. There are two subclasses of
	--@comments MetaH processes, periodic processes and aperiodic
processes
	class ProcessDeclaration extends METAHEntry {
		attribute procDecName : String;
		reference ports [*] ordered container : Port;
	}
	--@end ProcessDeclaration

	--@beginProcessImplementation
	class ProcessImplementation extends METAHEntry {
		attribute procImpName : String;
		attribute declaration : String;
		reference processAttributes [*] ordered container :
ProcessAttribute;
	}
	--@end ProcessImplementation

	--@begin ProcessAttribute
	class ProcessAttribute {
		attribute attName : String;
		attribute attValue : Integer;
		attribute attValueType : String;
	}
	--@end ProcessAttribute

	--@begin Process
	class Process  {
		attribute procName : String;
		attribute periodic : String;
		attribute declaration : String;
		attribute implementation : String;
	}
```

```
	--@end Process

	--@begin Port
	--@comments A port represents a point of contact between a process/macro and its environment.
	class Port {
		attribute portName : String;
		attribute portCom : String;
		attribute portPackage : String;
		attribute portType : String;
	}
	--@end Port

	--@begin Connection
	--@comments The connections part of an implementation specification declares connections between the interface
	--@comments elements of the various components in an implementation.
	class Connection extends METAHEntry {
		attribute compSrc : String;
		attribute portSrc : String;
		attribute compDest : String;
		attribute portDest : String;
	}
	--@end Connection

	--@begin MacroDeclaration
	--@comments A macro object is a hierarchical structuring mechanism, largely a syntactic feature to help structure
	--@comments large specifications that has little individual semantic impact. A macro object may contain
	--@comments process, macro, and connections between objects in the interfaces of these components.
	class MacroDeclaration extends METAHEntry {
		attribute name : String;
		reference ports [*] ordered container : Port;
	}
	--@end MacroDeclaration

	--@begin MacroImplementation
	class MacroImplementation extends METAHEntry {
		attribute macroImpName : String;
		attribute declaration : String;
		reference process [*] ordered container : Process;
		reference connections [*] ordered container : Connection;
	}
```

```
	--@end MacroImplementation
}

package PrimitiveTypes {
	datatype Boolean;
	datatype Integer;
	datatype String;
}
```

II. ACME metamodel in km3 format

```
--  @name ACME
--  @version 1.2
--  @domains developping new architectural design and analysis tools
--  @authors Julien Baudry (jul.baudry@gmail.com)
```

```
--  @date 2006/05/21
--  @description ACME is a simple, generic software architecture
description language (ADL)
--  @see This metamodel has been extracted from information
available on the ACME site : http://www.cs.cmu.edu/~ACME/.

package ACME {

	--@begin ACME File
	class ACMEFile {
		reference entries[*] container : ACMEEntry;
	}
	--@end ACME FILE

	--@begin ACME Entry
	abstract class ACMEEntry {
	}
	--@end ACME Entry

	--@begin Element
	--@comments Generic element of ACME. Any element (port, role,
component, connector and system) has properties and representations.
	abstract class Element {
		-- identifier
		attribute name : String;
		reference representations [*] ordered container :
Representation;
		reference property [*] ordered container : Property;
	}
	--@end Element

	class Type extends Element {}

	--@begin System
	-- @comments A system in ACME is a set of components and
connectors. Systems are first order entities in ACME.
	class System extends Element, ACMEEntry {
		-- set of components
		reference componentDeclaration [*] ordered container :
ComponentInstance;
		-- set of connector
		reference connectorDeclaration [*] ordered container :
Connector oppositeOf system;
		-- set of attachment between component and connector
		reference attachments [*] ordered container : Link
oppositeOf systemAttachment;
```

```
		reference bindings [*] ordered container : Link
oppositeOf systemBinding;
	}
	--@end System

	--@begin representation
	-- @comments A Representation is used to further describe an
element in terms of the ACME system construct.
	-- @comments Elements in ACME may have more than one
representation.
	class Representation  {
		reference systems [*] ordered container : System;
	}
	--@end representation

	--@begin Component
	--@comments Components are the basic building blocks in an ACME
description of a system.
	--@comments Components expose their functionality through their
ports.
	--@comments A component may have several ports corresponding to
different interfaces to the component.
	abstract class Component extends Element {
		-- set of port
		reference ports [*] ordered container : Port;
	}
	--@end Component

	--@begin Component Instance
	class ComponentInstance extends Component {
		attribute instanceOf : String;
	}
	--@end Component Instance

	--@begin Component Type
	class ComponentType extends Component, ACMEEntry {
		attribute extend : String;
	}
	--@end Component Type

	--@begin Port
	--@comments A port represents a point of contact between the
component and its environment.
	class Port extends Element {
	}
	--@end Port
```

```
	--@begin Connector
	--@comments Connectors define the nature of an interaction 
between components.
	--@comments A connector includes a set of interfaces in the 
form of roles
	class Connector extends Element {
		-- set of role
		reference roles [*] ordered container : Role;
		reference system : System oppositeOf 
connectorDeclaration;
	}
	--@end Connecor

	--@begin Role
	--@comments A role represents a point of contact between the 
connector and its environment.
	class Role extends Element {
	}
	--@end Role

	--@begin Property
	--@comments Elements in ACME include properties which can be 
used to describe aspects of its computational behavior or structure
	class Property {
		attribute name : String;
		attribute val : String;
	}
	--@end Property

	--@begin Link
	abstract class Link {
		reference systemBinding : System oppositeOf bindings;
		reference systemAttachment : System oppositeOf 
attachments;
	}
	--@end Link

	--@begin Attachment
	--@comments Each attachment represents an interaction between a 
port of a component and a role of a connector
	class Attachment extends Link {
		attribute comp : String;
		attribute port : String;
		attribute con : String;
		attribute role : String;
```

```
	}
	--@end Attachment

	--@begin Binding
	--@comments For a component, a binding provides a way of
associating a port on a component with some port within the
representation.
	class Binding extends Link {
		attribute compSrc : String;
		attribute portSrc : String;
		attribute compDest : String;
		attribute portDest : String;
	}
	--@end Binding
}

package PrimitiveTypes {
	datatype Boolean;
	datatype Integer;
	datatype String;
}
```

References

[1] ACME official website : http://www.cs.cmu.edu/~acme/
[2] METAH official website : http://www.htc.honeywell.com/metah/
[3] Mapping METAH Into ACME. Mario R. Barbacci and Charles B. Weinstock. Software Engineering Institute, July 1998 : http://www.sei.cmu.edu/pub/documents/98.reports/pdf/98sr006.pdf