MoML – A MODELING MARKUP LANGUAGE IN XML – VERSION 0.4

by

Edward A. Lee and Steve Neuendorffer

Memorandum No. UCB/ERL M00/12

14 March 2000

MoML – A MODELING MARKUP LANGUAGE IN XML – VERSION 0.4

by

Edward A. Lee and Steve Neuendorffer

Memorandum No. UCB/ERL M00/12

14 March 2000

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

MoML — A Modeling Markup Language in XML — Version 0.4

Edward A. Lee and Steve Neuendorffer

University of California at Berkeley
{eal, neuendor}@eecs.berkeley.edu
March 14, 2000

1.0 MOML PRINCIPLES

MoML is an XML modeling markup language. It is intended for specifying interconnections of parameterized, hierarchical components. It makes no assumptions about the meaning of the components or their interconnections. It provides a concrete syntax for the GSRC abstract syntax (see http://www.gigascale.org/semantics). It also includes some features that are not in the abstract syntax, such as annotations for visual rendition.

MoML is extensible in that components and their interconnections can be decorated with data specified in some other language (such as another XML language). The intent is to keep MoML very small by representing only the features of the abstract syntax. A MoML parser is expected to ignore decorations that it does understand. Thus, tools that can read MoML are able to exchange meaningful data even if they do not share semantic models, type systems, or other features of design languages. An example of a tool that requires only MoML is a visualizer or browser for hierarchical designs.

XML defines a rudimentary syntax for specifying hierarchical data. An XML language (sometimes called a dialect) follows this syntax and imposes additional rules defined in a document type definition (DTD). MoML is such a dialect, and as such, is fully defined by its DTD.

The key features of MoML include:

- *Web integration.* XML, the popular *extensible markup language*, provides a standard syntax and a standard way of defining the content within that syntax. The syntax is a subset of SGML, and is similar to HTML. It is intended for use on the Internet, and is intended for precisely this sort of specialization into dialects. File references are via URIs (in practice, URLs), both relative and absolute, so MoML is equally comfortable working on a localized computer or on a network.
- *Implementation independence.* The MoML language is designed to work with a variety of tools. A modeling tool that reads MoML files is expected to provide a class loader in some form. Given the name of a class, the class loader must be able to instantiate it. In Java, the class loader could be that built in to the JVM. In C++ or other languages, the class loader would have to implemented by the modeling tool.
- *Extensibility.* Components can be parameterized in two ways. First, they can have named properties with string values. Second, they can be associated with an external configuration file that can be in any format understood by the component. Typically, the configuration will be in some other XML dialect, such as PlotML or GraphicML.
- *Support for visual rendering.* Models in MoML can provide annotations that serve as hints or specifications for a visual rendering tool, such as a block diagram editor. This recognizes the reality that hierarchical component-based designs are a good match for visual renditions. For example, components can specify a location and can reference an external configuration file that defines a visual rendition, such as an icon.
- *Classes and inheritance.* Components can be defined in MoML as classes which can then be instantiated in a model. Components can extend other components through an object-oriented inheritance mechanism. It is important to recognize that this mechanism operates at the level of the abstract syntax, and therefore can be used in a variety of contexts.
- *Semantics independence.* MoML defines no semantics for an interconnection of components. It instead provides a mechanism for attaching a "director" to a model. The director defines the semantics of the interconnection. MoML knows nothing about directors except that they are instances of classes that can be loaded by the class loader.

The key observation in the design of MoML is that the most important decision for such a language is the abstract syntax supported by the language, not the concrete syntax. It is far less important what punctuation is used, and how the textual data is structured, than what the data represents. MoML is intended to provide a concrete syntax for the GSRC abstract syntax, although the GSRC abstract syntax is evolving, so some of the features of the GSRC abstract syntax are missing from MoML. The concrete syntax follows from the abstract syntax by designing an XML dialect to most concisely represent this abstract syntax.

A MoML tool has been constructed using Ptolemy II [1], which provides a sanity check and a reference implementation. Some of the examples below illustrate how MoML is used with Ptolemy II, but keep in mind that MoML is designed carefully to be tool independent. Its key depen-

dence is on the abstract syntax, and in principle, it can be used with any tool that is compatible with the abstract syntax.

1.1 Clustered Graphs

A model is given as a clustered graph, an abstract syntax for netlists, state transition diagrams, block diagrams, etc. An *abstract syntax* is a conceptual data organization. It can be contrasted with a *concrete syntax*, which is a syntax for a persistent, readable representation of the data, such as EDIF for netlists. MoML is a concrete syntax for the clustered graph abstract syntax. A particular graph configuration is called a *topology*.

Certain features of the GSRC abstract syntax extend beyond clustered graphs and are not yet supported by MoML. In particular, MoML does not yet support heterarchy. Also, the reference implementation currently only supports tree-structured containment relationships, rather than the directed acyclic graphs (DAGs) modeled in the GSRC abstract syntax.

A topology is a collection of *entities*, *ports*, and *relations*. We use the graphical notation shown in figure 1, where entities are depicted as rounded boxes and relations as diamonds. Entities contain ports, shown as filled circles, and relations connect the ports. We consistently use the term *connection* to denote the association between connected ports (or their entities), and the term *link* to denote the association between ports and relations. Thus, a connection consists of a relation and two or more links.

Relations are intended to serve a mediators, in the sense of the Mediator design pattern of Gamma, *et al.* [2]. "Mediator promotes loose coupling by keeping objects from referring to each other explicitly..." For example, a relation could be used to direct messages passed between entities. Or it could denote a transition between states in a finite state machine, where the states are represented as entities. Or it could mediate rendezvous between processes represented as entities. Or it could mediate method calls between loosely associated objects, as for example in remote method invocation over a network.

1.2 Abstraction

Composite entities (clusters) are entities that can contain a topology (entities and relations). Clustering is illustrated by the example in figure 2. A port contained by a composite entity has inside as well as outside links. Such a port serves to expose ports in the contained entities as ports of the composite. This is the converse of the "hiding" operator often found in process algebras.

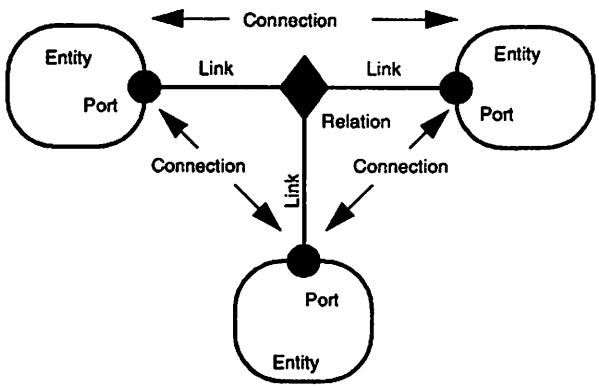

Figure 1. Visual notation and terminology.

Ports within an entity are hidden by default, and must be explicitly exposed to be visible (linkable) from outside the entity[1]. The composite entity with ports thus provides an abstraction of the contents of the composite.

2.0 SPECIFICATION OF A MODEL

In this section, we describe the XML elements that are used to define MoML models.

2.1 Data Organization

As with all XML files, MoML files have two parts, one defining the MoML language and one containing the model data. The first part is called the *document type definition*, or DTD. This dual specification of content and structure is a key XML innovation. The DTD for MoML is given in figure 3. If you are adept at reading these, it is a complete specification of the language. However, since it is not particularly easy to read, we explain its key features here.

Every MoML file must either contain or refer to a DTD. The simplest way to do this is with the file structure shown below:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE model PUBLIC
"-//UC Berkeley//DTD MoML 1//EN"
"http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<model name="modelname" class="classname">
    model definition ...
</model>
```

Here, "model definition" is a set of XML elements that specify a clustered graph. The syntax for these elements is described in subsequent sections. The first line above is required in any XML file. It asserts the version of XML that this file is based on (1.0) and states that the file includes external references (in this case, to the DTD). The second through fourth lines declare the document type (model) and provide references to the DTD.

The references to the DTD above refer to a "public" DTD. The name of the DTD is `-//UC Berkeley//DTD MoML 1//EN`,

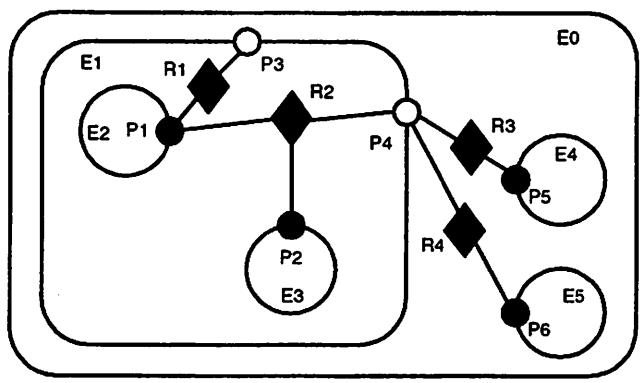

Figure 2. Abstraction and aggregation.

1. The GSRC abstract syntax rejects level-crossing links. MoML can express these, but they are discouraged, and a particular MoML tool is likely to reject them.

```
<!ELEMENT model (class | configure | director | doc | entity | import |
                 link | property | relation | rendition)*>
<!ATTLIST model name CDATA #REQUIRED
                class CDATA #IMPLIED>

<!ELEMENT class (class | configure | director | doc | entity | import | link |
                 property | relation | rendition)*>
<!ATTLIST class name CDATA #REQUIRED
                extends CDATA #IMPLIED>

<!ELEMENT configure (#PCDATA)>
<!ATTLIST configure source CDATA #IMPLIED>

<!ELEMENT director (configure | property)*>
<!ATTLIST director name CDATA "director"
                   class CDATA #REQUIRED>

<!ELEMENT doc (#PCDATA)>

<!ELEMENT entity (class | configure | director | doc | entity | import | link | port |
                  link | port | property | relation | rendition)*>
<!ATTLIST entity name CDATA #REQUIRED
                 class CDATA #IMPLIED>

<!ELEMENT import EMPTY>
<!ATTLIST import source CDATA #REQUIRED
                 base CDATA #IMPLIED>

<!ELEMENT link EMPTY>
<!ATTLIST link port CDATA #REQUIRED
               relation CDATA #REQUIRED
               vertex CDATA #IMPLIED>

<!ELEMENT location EMPTY>
<!ATTLIST location value CDATA #REQUIRED>

<!ELEMENT port (configure | doc | property)*>
<!ATTLIST port class CDATA #IMPLIED
               name CDATA #REQUIRED>

<!ELEMENT property (configure | doc | property)*>
<!ATTLIST property class CDATA #IMPLIED
                   name CDATA #REQUIRED
                   value CDATA #IMPLIED>

<!ELEMENT relation (property | vertex)*>
<!ATTLIST relation name CDATA #REQUIRED
                   class CDATA #IMPLIED>

<!ELEMENT rendition (configure | location | property)*>
<!ATTLIST rendition class CDATA #REQUIRED>

<!ELEMENT vertex (location | property)*>
<!ATTLIST vertex name CDATA #REQUIRED
                 pathTo CDATA #IMPLIED>
```

Figure 3. MoML version 1 DTD.

which follows the standard naming convention of public DTDs. The leading dash "-" indicates that this is not a DTD approved by any standards body. The first field, surrounded by double slashes, in the name of the "owner" of the DTD, "UC Berkeley." The next field is the name of the DTD, "DTD MoML 1" where the "1" indicates version 1 of the MoML DTD. The final field, "EN" indicates that the language assumed by the DTD is English.

In addition to the name of the DTD, the DOCTYPE element includes a URL pointing to a copy of the DTD on the web. If a particular MoML tool does not have access to a local copy of the DTD, then it finds it at this web site.

The "model" element may be replaced by a "class" element, as in:[1]

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE class PUBLIC "..." "http://..."
<class name="modelname" class="classname">
    class definition ...
</class>
```

We will say more about class definitions below.

The DTD may be given directly as a relative or absolute URL instead of a public DTD, using the following syntax:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE model SYSTEM "DTD location">
<model name="modelname" class="classname">
   model definition ...
</model>
```

Here, "DTD location" is a relative or absolute URL.

A third option is to create a standalone MoML file that includes the DTD. The result is rather verbose, but has the general structure shown below:

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE model [
   DTD information
]>
<model name="modelname" class="classname">
   model definition ...
</model>
```

2.2 Overview of XML

An XML document consists of the header tags "<?xml ... ?>" and "<!DOCTYPE ... >" followed by exactly one *element*. The element has the structure:

```
start tag
body
end tag
```

where the start tag has the form

```
<elementName attributes>
```

and the end tag has the form

```
</elementName>
```

The body, if present, can contain additional elements as well as arbitrary text. If the body is not present, then the element is said to be *empty*; it can optionally be written using the shorthand:

```
<elementName attributes/>
```

where the body and end tag are omitted.

The attributes are given as follows:

```
<elementName attributeName="attributeValue" .../>
```

Which attributes are legal in an element is defined by the DTD. The quotation marks delimit the attributes, so if the attribute value needs to contain quotation marks, then they must be given using the special XML entity """ as in the following example:

```
<elementName attributeName="&quot;foo&quot;"/>
```

The value of the attribute will be

```
"foo"
```

(with the quotation marks).

In XML """ is called an *entity*, creating possible confusion with our use of entity in MoML. In XML, an entity is a named storage unit of data. Thus, """ references an entity called "quot" that stores a double quote character.

The keyword "SYSTEM" (which was seen above) indicates that an external URL or URI gives an entity (above it is the location of the DTD). This choice of keyword is positively peculiar, but we must live with it. The keyword "CDATA" (which we will encounter below) refers to "character data."

2.3 Names and Classes

Most MoML elements have *name* and *class* attributes. The name is a handle for the object being defined or referenced by the element. In MoML, the same syntax is used to reference a pre-existing object as to create a new object. If a new object is being created, then the class attribute (usually) must be given. If a pre-existing object is being referenced, or if the MoML reader has a built-in default class for the element, then the class attribute is optional. If the class attribute is given, then the pre-existing object must be an instance of the specified class.

A name is either absolute or relative. Absolute names begin with a period "." and consist of a series of name fields separated by periods, as in ".x.y.z". Each name field can have alphanumeric characters or the underscore "_" character. The first field is the name of the top-level model or class object. The second field is the name of an object immediately contained by that top-level.

Any name that does not begin with a period is relative to the current context, the object defined or referenced by an enclosing element. The first field of such a name refers to or defines an object immediately contained by that object. For example, inside

1. We omit the DTD name and URL henceforth for conciseness.

of an object with absolute name ".x" the name "y.z" refers to an object with absolute name ".x.y.z".

A name is required to be unique within its container. That is, in any given model, the absolute names of all the objects must be unique. There can be two objects named "z", but they must not be both contained by ".x.y".

2.4 Model Element

A very simple MoML file looks like this:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE model PUBLIC "..." "http://...">
<model name="modelname" class="classname">
</model>
```

A *model* element has name and class attributes. This value of the class attribute must be a class that instantiable by the MoML tool. For example, in the Ptolemy II reference implementation, we can define a model with:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE model PUBLIC "..." "http://...">
<model name="ptIImodel"
       class="ptolemy.actor.TypedCompositeActor">
</model>
```

Here, `ptolemy.actor.TypedCompositeActor` is a class that a Java class loader can find and that the MoML parser can instantiate. In Ptolemy II, it is a container class for clustered graphs representing executable models or libraries of instantiable model classes.

2.5 Entity Element

A model typically contains entities, as in the following Ptolemy II example:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE model PUBLIC "..." "http://...">
<model name="ptIImodel"
       class="ptolemy.actor.TypedCompositeActor">
   <entity name="source"
       class="ptolemy.actor.lib.Ramp"/>
   <entity name="sink"
       class="ptolemy.actor.lib.SequencePlotter"/>
</model>
```

Notice the common XML shorthand here of writing "`<entity ... />`" rather than "`<entity ...></entity>`." Of course, the shorthand only works if there is nothing in the body of the entity element.

An entity can contain other entities, as shown in this example:

```
<model name="ptIImodel"
       class="ptolemy.actor.TypedCompositeActor">
   <entity name="container"
       class="ptolemy.actor.TypedCompositeActor">
      <entity name="source"
       class="ptolemy.actor.lib.Ramp"/>
   </entity>
</model>
```

An entity must specify a class unless the entity already exists in the containing entity or model. The name of the entity reflects the container hierarchy. Thus, in the above example, the *source* entity has the full name "`.ptIImodel.container.source`".

The definition of an entity can be distributed in the MoML file. Once created, it can be referred to again by name as follows:

```
<model name="top" class="classname">
   <entity name="x" class="classname"/>
   ...
   <entity name="x">
      <property name="y">
   </entity>
</model>
```

The property element is added to the pre-existing entity with name "x" when the second entity element is encountered.

In principle, MoML supports multiple containment, as in the following:

```
<model name="top" class="classname">
   <entity name="x" class="classname"/>
   ...
   <entity name="y" class="classname">
      <entity name=".top.x"/>
   </entity>
</model>
```

Here, the element named "x" appears both in "top" and in ".top.y". Thus, it would have two full names, ".top.x" and ".top.y.x". However, the Ptolemy II reference implementation does not (yet) support this, as it implements a strict container relationship, where an object can have only one container. Thus, attempting to parse the above MoML will result in an exception being thrown.

2.6 Properties

Entities (and some other elements) can be parameterized. There are two mechanisms. The simplest one is to use the *property* element:

```
<entity name="source"
        class="ptolemy.actor.lib.Ramp">
    <property name="init" value="5"
        class="ptolemy.data.expr.Parameter"/>
</entity>
```

The property element has a name, at minimum (the value and class are optional). It is common for the enclosing class to already contain properties, in which case the property element is used only to set the value. For example:

```
<entity name="source"
        class="ptolemy.actor.lib.Ramp">
    <property name="init" value="5"/>
</entity>
```

In the above, the enclosing object (*source*, an instance of `ptolemy.actor.lib.Ramp`) must already contain a property with the name *init*. This is typically how library components

are parameterized. It is up to a MoML tool, such as Ptolemy II, to interpret the value string.

A property can be declared without a class and without a pre-existing property if it is a *pure property*, one with only a name and no value. For example:

```
<entity name="source"
        class="ptolemy.actor.lib.Ramp">
    <property name="abc"/>
</entity>
```

A property can also contain a property, as in

```
<property name="x" value="5">
    <property name="y" value="10"/>
</property>
```

A second, much more flexible mechanism is provided for parameterizing entities. The *configure* element can be used to specify a relative or absolute URL pointing to a file that configures the entity, or it can be used to include the configuration information in line. That information need not be MoML information. It need not even be XML, and can even be binary encoded data (although binary data cannot be in line; it must be in an external file). For example,

```
<entity name="sink"
        class="ptolemy.actor.lib.SequencePlotter">
    <configure source="filename"/>
</entity>
```

Here, *filename* can give the name of a file containing data. (For the SequencePlotter actor, that external data will have PlotML syntax; PlotML is another XML dialect for configuring plotters.) Configure information can also be given in the body of the MoML file as follows:

```
<entity name="sink"
        class="ptolemy.actor.lib.SequencePlotter">
    <configure>
        configure information
    </configure>
</entity>
```

With the above syntax, the configure information must be textual data without any markup (no "<" or ">"). If you wish to include markup, use the standard XML syntax for preventing the parsing of the markup:

```
<entity name="sink"
        class="ptolemy.actor.lib.SequencePlotter">
    <configure> <![CDATA[
        configure information with markup
    ]]></configure>
</entity>
```

Everything between "<![CDATA[" and "]]>" will be passed to the class as configuration information. The data must be textual, but it can now contain markup. The only constraint is that it cannot contain the termination string "]]>", so it cannot itself contain a similarly escaped body of CDATA information. This mechanism is particularly useful if the configuration is XML data conforming to some other DTD (i.e., non-MoML XML or HTML).

You can give both a source attribute and in-line configuration information, as in the following:

```
<entity name="sink"
        class="ptolemy.actor.lib.SequencePlotter">
    <configure source="filename">
        configure information
    </configure>
</entity>
```

In this case, the file data will be passed to the application first, followed by the in-line configuration data.

In Ptolemy II, the configure element is supported by any class that implements the Configurable interface. That interface defines a configure() method that accepts an input stream. Both external file data and in-line data are provided to the class as a character stream by calling this method.

2.7 Doc Element

Some elements can be documented using the *doc* element. For example,

```
<entity name="source"
        class="ptolemy.actor.lib.Ramp">
    <property name="init" value="5">
        <doc> Text here ... </doc>
    </property>
    <doc> Text here ... </doc>
</entity>
```

With the above syntax, the documentation information must be textual data without any markup (no "<" or ">"). If you wish to include markup, use the standard XML syntax for preventing the parsing of the markup. For example, to use HTML in the documentation, do something like this:

```
<entity name="source"
        class="ptolemy.actor.lib.Ramp">
    <doc><![CDATA[
<H1>Title</H1>
<P>Text</P>
    ]]></doc>
</entity>
```

Everything between "<![CDATA[" and "]]>" will be recorded as documentation. The only constraint is that it cannot contain the termination string "]]>", so it cannot itself contain a similarly escaped body of CDATA information.

More than one doc element can be included in an element. Utilities such as graphical editors are responsible for consolidating the documentation given by each in the order in which they are given.

2.8 Ports

An entity can declare a port:

```
<entity name="A" class="classname">
```

```
    <port name="out"/>
</entity>
```

In the above example, no class is given for the port, so the port is required to already exist in the class for entity A. Alternatively, we can specify a class name, as in

```
<entity name="A" class="classname">
   <port name="out" class="classname"/>
</entity>
```

In this case, a port will be created if one does not already exist. If it does already exist, then its class is checked for consistency with the declared class (the pre-existing port must be an instance of the declared class). In Ptolemy II, the typical classname for a port would be `ptolemy.actor.TypedIOPort`.

In Ptolemy II, it is often useful to declare a port to be an input, an output, or both. To do this, enclose in the port a property named "input" or "output" or both, as in the following example:

```
<port name="out" class="ptolemy.actor.IOPort">
    <property name="output"/>
</port>
```

This is an example of a pure property. Note that this convention is not part of the MoML definition. This illustrates one of the ways in which MoML extensible.

2.9 Relations and Links

To connect entities, you create relations and links. The following example describes the topology shown in figure 4:

```
<model name="top" class="classname">
   <entity name="A" class="classname">
      <port name="out"/>
   </entity>
   <entity name="B" class="classname">
      <port name="out"/>
   </entity>
   <entity name="C" class="classname">
      <port name="in">
         <property name="multiport"/>
      </port>
   </entity>
   <relation name="r1" class="classname"/>
   <relation name="r2" class="classname"/>
   <link port="A.out" relation="r1"/>
   <link port="B.out" relation="r2"/>
   <link port="C.in" relation="r1"/>
   <link port="C.in" relation="r2"/>
</model>
```

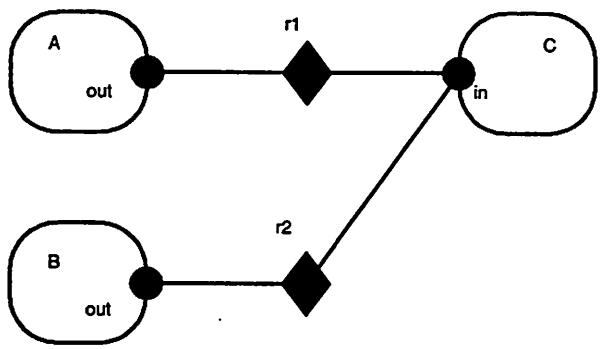

Figure 4. Example topology.

Notice that this example has two distinct links to `C.in` from two different relations. The order of these links may be important to a MoML tool, so any MoML tool must preserve the order in which they are specified.

In the Ptolemy II reference implementation, the typical classname for a relation would be `ptolemy.actor.TypedIORelation`. As usual, the class attribute may be omitted if the relation already exists in the containing entity.

2.10 Classes

So far, entities have been instances of externally defined classes accessed via a class loader. They can also be instances of classes defined in MoML. To define a class in MoML, use the *class* element, as in the following example from Ptolemy II:

```
<class name="Gen"
       extends="ptolemy.actor.TypedCompositeActor">
    <entity name="ramp"
       class="ptolemy.actor.lib.Ramp">
       <port name="output"/>
       <property name="step" value="2*PI/50"/>
    </entity>
    <entity name="sine"
        class="ptolemy.actor.lib.Sine">
        <port name="input"/>
        <port name="output"/>
    </entity>
    <port name="output"
        class="ptolemy.actor.TypedIOPort"/>
    <relation name="r1"
        class="ptolemy.actor.TypedIORelation"/>
    <relation name="r2"
        class="ptolemy.actor.TypedIORelation"/>
    <link port="ramp.output" relation="r1"/>
    <link port="sine.input" relation="r1"/>
    <link port="sine.output" relation="r2"/>
    <link port="output" relation="r2"/>
</class>
```

The class element may be the top-level element in a file, in which case the `DOCTYPE` should be declared as "class". It can also be nested within a model. The above example specifies the topology

shown in figure 5. Once defined, can be instantiated as if it were a class loaded by the class loader:

```
<entity name="instancename" class="classname"/>
```

The class name follows the same convention as entity names. In fact, a class *is* an entity with the additional feature that one can create new instances of it with the entity element.

In the above example, the relative name of the class is "Gen". The class name might be ".Gen" if the class is defined at the top level, as follows:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE class PUBLIC "..."  "http://...">
<class name="Gen"
    extends="ptolemy.actor.TypedCompositeActor">
    class definition ...
</class>
```

Alternatively, it may have full name ".top.Gen" if it is defined as follows:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE class PUBLIC "..."  "http://...">
<model name="top"
    extends="ptolemy.kernel.CompositeEntity">
    <class name="Gen"
        extends="ptolemy.actor.TypedCompositeActor">
      class definition ...
   </class>
</model>
```

This allows a library of class definitions to be conveniently collected within a single MoML file.

The Gen class given at the beginning of this subsection generates a sine wave with a period of 50 samples. It is not all that useful without being parameterized. Let us extend it and add properties:

```
<class name="Sinegen" extends="Gen">
    <property name="samplingFrequency"
        value="8000.0"
        class="ptolemy.data.expr.Parameter">
        <doc>The sampling frequency in Hertz.</doc>
    </property>
    <property name="frequency"
        value="440.0"
        class="ptolemy.data.expr.Parameter">
        <doc>The frequency in Hertz.</doc>
    </property>
    <property name="phase"
        value="0.0"
        class="ptolemy.data.expr.Parameter">
        <doc>The phase, in radians.</doc>
    </property>
    <property name="ramp.step"
        value="frequency*2*PI/samplingFrequency">
        <doc>Formula for the step size.</doc>
    </property>
    <property name="ramp.init" value="phase"/>
</class>
```

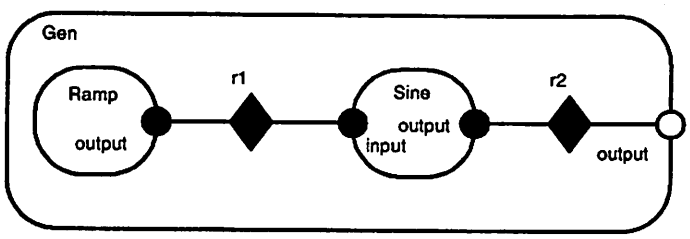

Figure 5. Sine wave generator topology.

This class extends Gen by adding three properties, and then sets the properties of the component entities to have values that are expressions.

2.11 Directors

Recall that a clustered graph in MoML has no semantics. However, a particular model has semantics. It may be a dataflow graph, a state machine, a process network, or something else. To give it semantics, MoML allows the specification of a director associated with a model, an entity, or a class. The following example gives discrete-event semantics to a Ptolemy II model:

```
<model name="top"
        class="ptolemy.actor.TypedCompositeActor">
    <director
        class="ptolemy.domains.de.kernel.DEDirector">
        <property name="stopTime" value="100.0"/>
    </director>
    ...
</model>
```

This example also sets a property of the director.

2.12 Import Element

Given the ability to have class definitions and clusters, it is unlikely that interesting designs will reside in a single file. You can import definitions in another file by giving a relative or absolute URL in an element like this:

```
<import source="URL"/>
```

or

```
<import base="URL" source="URL"/>
```

The (optional) base specifies a URL with respect to which the relative source URL should be interpreted. If no base is specified, then the base of the current document (the one containing the import statement) is used, or if the current document has no base, then the current working directory is used.

For example, you might import a library of classes. The imported file must be a MoML file. If it defines classes with names that match classes previously defined, then the new definitions replace the old. Imported models are always defined at the top level of the hierarchy, regardless of where the import element is found. Thus, if the imported file contains

```
<model name="top"
```

```
    class="ptolemy.actor.CompositeEntity">
    <class name="Gen"
       extends="ptolemy.actor.TypedCompositeActor">
       ...
    </class>
</model>
```

then the class should be referenced by the absolute name ".top.Gen" always, even if the `import` element occurs within an entity definition.

Notice that since an import element may result in a class definition that replaces a previous class definition, it matters where in a MoML file you place the import element. Any elements before it use definitions in place before the imported file is read. Any elements after it will use the new definitions.

2.13 Annotations for Visual Rendering

The abstract syntax of MoML, clustered graphs, is amenable to visual renditions as bubble and arc diagrams or as block diagrams. To support tools that display and/or edit MoML files visually, there are two simple annotations that can be attached to entities and relations. A tool that does not support visual renditions just ignores these annotations. A visual rendition might be an icon, the layout of a circuit, an image of the structure of a component, or anything else that can be rendered visually.

First, an entity can specify a rendition as in the following example:

```
<entity name="ramp" class="ptolemy.actor.lib.Ramp">
   <port name="output"/>
   <rendition class="iconClass">
      <location value="100, 100"/>
   </rendition>
</entity>
```

The *iconClass* depends on the visual rendering tool being used. The location element specifies the location of the icon in the visual field. MoML makes no assumptions about how this location is specified; its value is just a string. The location element is not required, so a MoML tool should be prepared to place the icon without a specified location.

The second type of annotation supports paths that connect ports. Consider the following example:

```
<relation name="r"
   class="ptolemy.actor.TypedIORelation">
   <vertex name="v1">
      <location value="100, 100"/>
   </vertex>
   <vertex name="v2" pathTo="v1">
      <location value="100, 200"/>
   </vertex>
</relation>
<link port="A.out" relation="r" vertex="v1"/>
<link port="B.in" relation="r" vertex="v1"/>
<link port="C.in" relation="r" vertex="v2"/>
```

This assumes that there are three entities named *A*, *B*, and *C*. The relation is annotated with a set of vertices, which will normally be rendered as graphical objects with a location. The vertices are linked together with paths, which in a simple visual tool might be straight lines, or in a more sophisticated tool might be autorouted paths.

Figure 6 illustrates how the above fragment might be rendered. The square boxes are icons for the three entities. They have ports with arrowheads suggesting direction. There is a single relation, which shows up visually only as a set of lines and two vertices. The vertices are shown as small squares.

The link elements specify not just a relation, but also a vertex within that relation. This tells the visual rendering tool to draw a path from the specified port to the specified vertex.

3.0 PTOLEMY II IMPLEMENTATION

MoML is intended to be a generic modeling markup language, not one that is specialized to Ptolemy II. As such, Ptolemy II may be viewed as a reference implementation of a MoML tool. In Ptolemy II, MoML is supported by two packages, the moml package and the actor.gui package.

The moml package contains the classes shown in figure 7, which is a UML static structure diagram. The basis for the MoML parser is the parser distributed by Microstar. This parser is used in Ptolemy II in both applications and applets, as shown in figure 8. The moml package (figure 7) also includes a set of attribute classes that decorate the objects in a model with MoML-specific information.

The parse() methods of the MoMLParser class read MoML data and construct a Ptolemy II model. The exportMoML() methods of Ptolemy II objects can be used to produce a MoML file given a model. Thus, MoML can be used as the persistent file format for Ptolemy II models.

3.1 Command-line Invocation

A model defined as a MoML file may be executed on the command-line by typing

```
ptolemy filename.xml
```

This assumes that the `ptolemy` executable is in your path. That executable creates an instance of the class PtolemyApplication, shown in figure 8. That class contains an instance of ModelFrame, which defines a top-level window that serves as an interface for executing a model. An example of such a top-level window is shown in figure 9. The ModelFrame is actually just a top-level window and a menubar containing an instance of ModelPane. The ModelPane has two parts. On the left, it displays all the top-level parameters of a model and its director, permitting the user to interactively edit them. On the right, it stacks the displays of any components in the model that implement the Placeable interface, such as signal plotters.

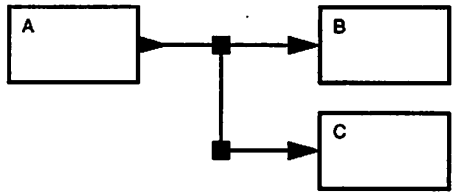

Figure 6. Example showing how MoML might be visually rendered.

The `ptolemy` executable has the following usage:

```
Usage: ptolemy [ options ] [file ...]

Options that take values:
 -class <classname>
 -<parameter name> <parameter value>

Boolean flags:
 -help -test -version
```

Notice that more than one MoML file can be given. The result is that multiple files will be executed in the same Java virtual machine, in separate threads. By default, models are opened using the PtolemyApplication class. However, any other class with a main() method can be specified instead using the -class option.

If a model has top-level parameters, the default value of those parameters can be given on the command line. Also, the director parameters can be set by the same mechanism. For example,

```
bash-2.02$ cd $PTII/ptolemy/moml/demo
```

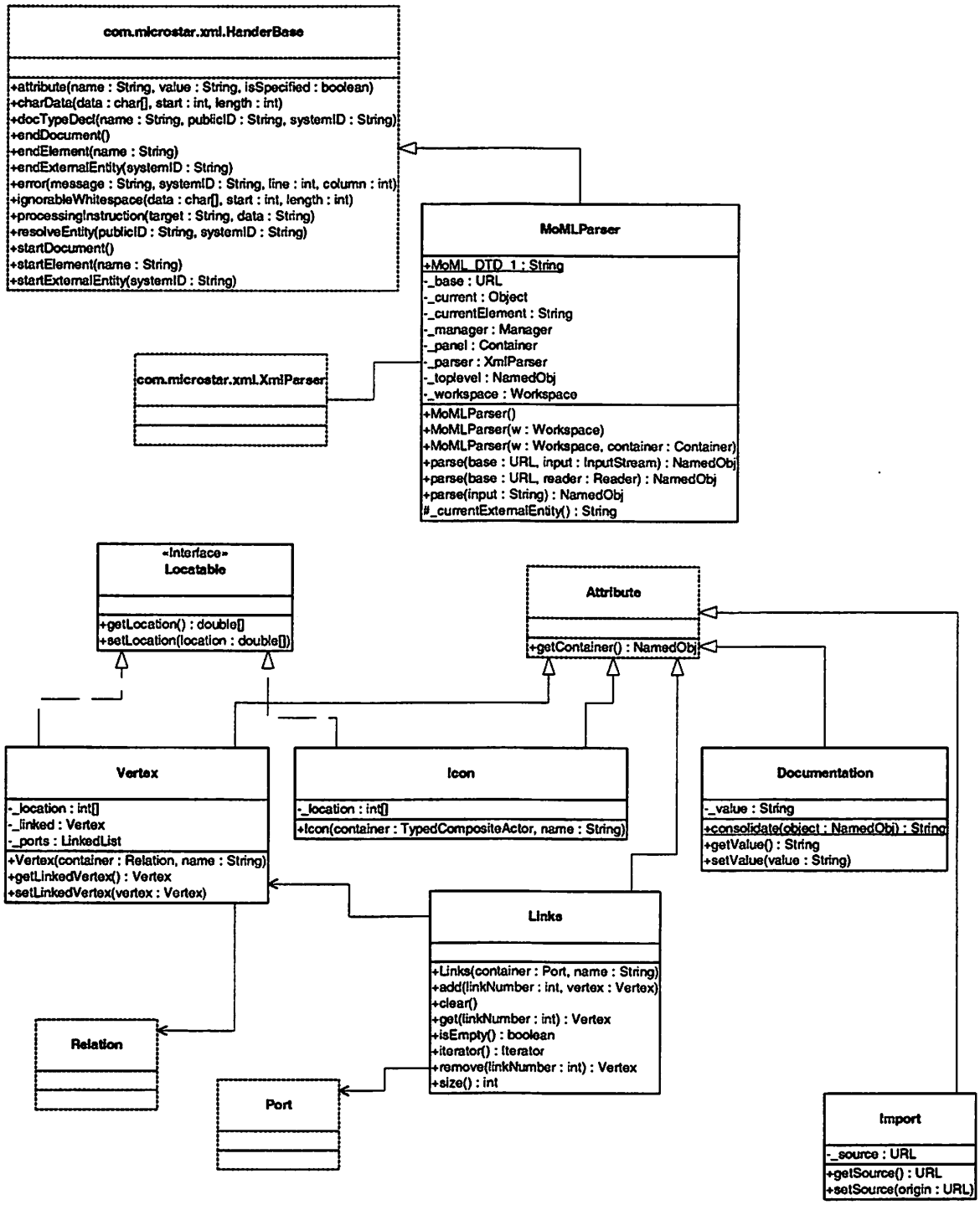

Figure 7. Classes supporting MoML in the moml package.

```
bash-2.02$ ptolemy -iterations 1000 modulation.xml
```

results in the display shown in figure 9.

3.2 Applets

The same XML file can be used in an application or an applet. The applet mechanism is explained in detail in this subsection. An applet is a Java class that can be referenced by an HTML file and accessed over the web. For details, see the Ptolemy II documentation.

3.3 Support Methods

A number of methods have been added to the Ptolemy II kernel classes to support MoML. The following methods of NamedObj (and derived class) are particularly useful:

```
deferMoMLDefinitionTo(deferTo: NamedObj)
```

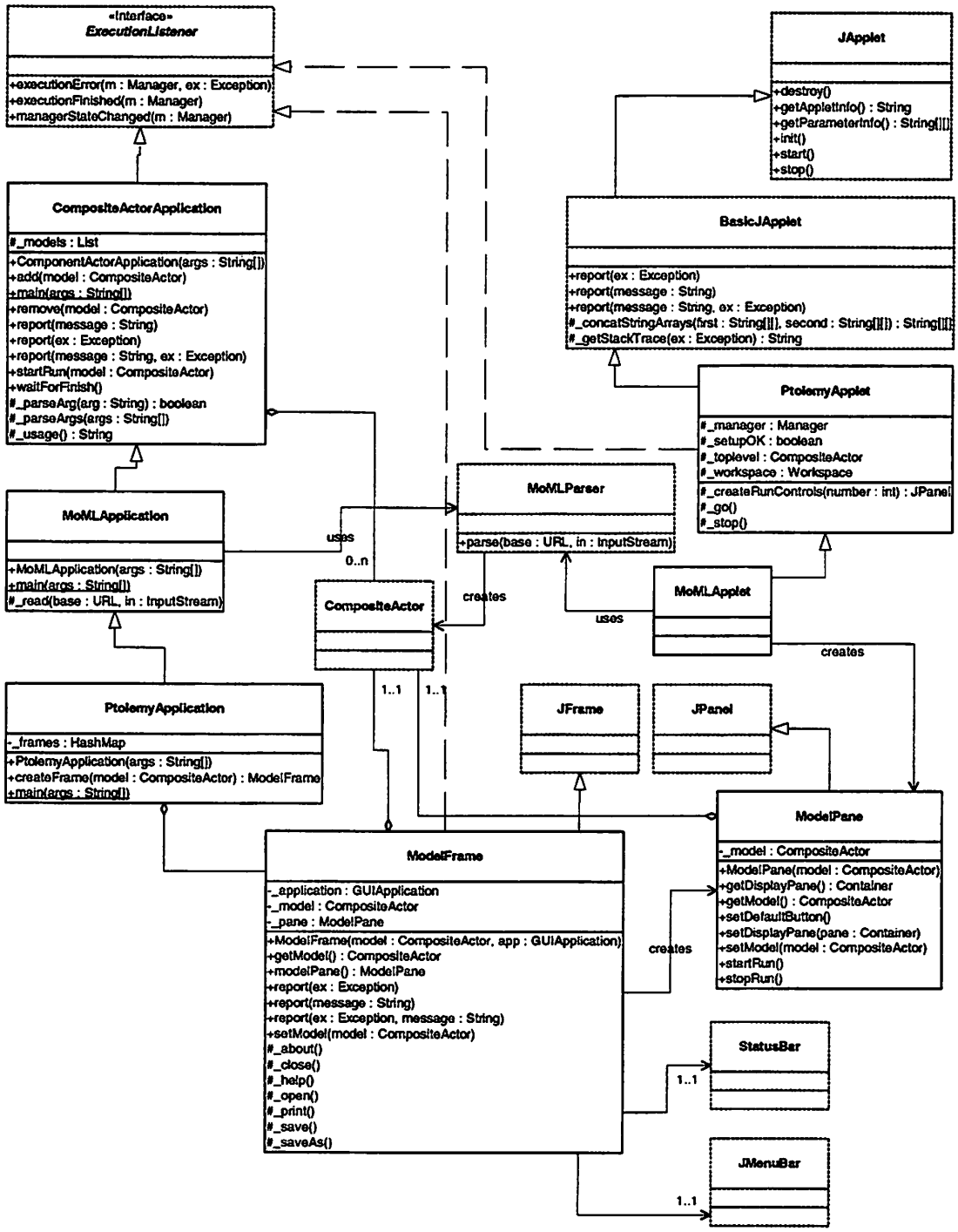

Figure 8. Applet and application classes in the actor.gui package.

```
deferredMoMLDefinitionFrom(): List
exportMoML(): String
exportMoML(output: Writer)
exportMoML(output: Writer, depth: int)
_exportMoMLContents(output: Writer, depth: int)
```

The first two of these permit an object to instantiate another "by reference." This means in particular that when the object is asked to describe itself using MoML, it defers to the reference.

The rest of these methods support exporting MoML. Since any object derived from NamedObj can export MoML, MoML becomes an effective persistent format for Ptolemy II models. It is much more compact than serializing the objects, and it reflects a stable initial state of the objects rather than whatever happens to be the current state (say, in the middle of a model execution).

3.4 Special Attributes

A number of classes derived from Attribute are shown in figure 7. These classes are used to decorate a Ptolemy II object with additional information that is relevant to a GUI or other user interface.They generate their own MoML representations. Some of these are described here.

Doc element. When a MoML file is parsed by Ptolemy II, a doc element is converted to an instance of the special property of class Documentation. This property is contained by the entity, port, or relation that encloses the doc element. There may be more than one instance of Documentation contained by a single object. To extract all documentation that has been so associated with an object, use code like the following:

```
import ptolemy.kernel.util.NamedObj;
import ptolemy.moml.Documentation;

NamedObj obj = object with documentation;
Iterator docs = obj.attributeList(Documenta-
tion.class);
while (docs.hasNext()) {
   Documentation doc = (Documentation)docs.next();
   System.out.println(doc.getValue());
}
```

This code would be used, for example, by a GUI wishing to present documentation.

3.5 Inheritance

MoML supports inheritance by permitting you to extend existing classes. Ptolemy II reads MoML that uses this inheritance mechanism. For example, consider the following MoML file:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE model PUBLIC "..." "http://...">
<model name="top"
       class="ptolemy.kernel.CompositeEntity">
    <class name="base"
       extends="ptolemy.kernel.CompositeEntity">
       <entity name="e1"
          class="ptolemy.kernel.ComponentEntity">
       </entity>
    </class>
    <class name="derived" extends="base">
       <entity name="e2"
          class="ptolemy.kernel.ComponentEntity"/>
    </class>
</model>
```

Here, the "derived" class extends the "base" class by adding another entity to it. However, there is a key limitation in Ptolemy II. Invoking the exportMoML() methods discards the inheritance link. For the above example, `top.exportMoML()` will produce:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE model PUBLIC "..." "http://...">
<model name="top"
       class="ptolemy.kernel.CompositeEntity">
    <class name="base"
       extends="ptolemy.kernel.CompositeEntity">
       <entity name="e1"
          class="ptolemy.kernel.ComponentEntity"/>
    </class>
    <class name="derived"
          extends="ptolemy.kernel.CompositeEntity">
       <entity name="e1"
```

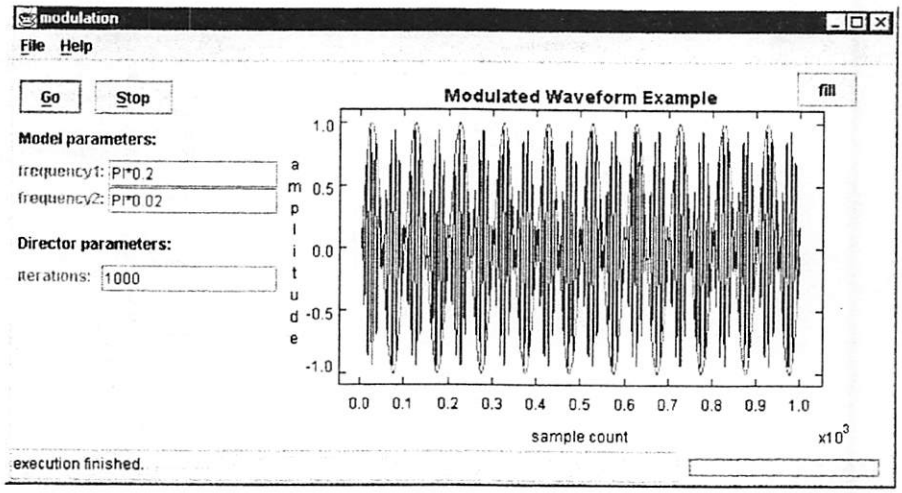

Figure 9. Sinusoidal modulation example specified in MoML.

```
        class="ptolemy.kernel.ComponentEntity"/>
    <entity name="e2"
        class="ptolemy.kernel.ComponentEntity"/>
  </class>
</model>
```

The derived class no longer extends the base class. This is caused by the addition of an entity to the derived class. The same problem arises if the base class is instantiated and extended using MoML code like:

```
<entity name="derived" class="base">
<entity name="e2"
    class="ptolemy.kernel.ComponentEntity"/>
</entity>
```

The reasons for this flaw are subtle and fairly deep, and a good mechanism for correcting it remains an open research question. Fortunately, this flaw is not visible at the level of a block diagram editor, because no known block diagram editor provides visual inheritance mechanisms like those in MoML. It is again an open research question to identify such a mechanism.

We explain further. In object-oriented languages such as C++ and Java, all instances of a class share the same code for their methods. Only the instance variables differ among instances of a class. In Ptolemy II, however, the "code" for a composite entity is its topology. Ptolemy II has no mechanism for sharing the same topology among instances of the composite entity. Instead, it clones the base topology when creating an instance or a derived topology. The difficulty arises when either the base or the derived topology changes.

If the base topology changes after the derived topology has been created, then ideally, the derived topology should reflect those changes. Unfortunately, this is difficult to do. All changes to the base would have to be mirrored in the derived, which would require all code that modifies the base, including user-written code in actors, to mirror the changes in all derived classes. Such an approach would place an undue burden on code developers, and the resulting code would have very little chance of being correct.

One possibility would be to prohibit changes in the base topology after a derived topology is created. However, this is excessively restrictive. In Ptolemy II, the deferredMoMLDefinitionsFrom() method of NamedObj returns a list of objects that were derived by cloning a particular object. This list can be used by a user interface to re-clone the base, replacing the derived instances with new ones.

If the derived topology changes after cloning, as in the above example, and these changes involve only *addition* to the topology (new entities, ports, relations, or links) or changes to parameter values, then in principle it would not be difficult to generate those additions in the MoML body. However, even *detecting* the changes is non-trivial, since they can occur arbitrarily deeply in the topology. A block diagram editor can largely avoid the problem by providing no mechanism for editing the derived topology. Only the base topology can be edited.

The mechanism in Ptolemy II is conservative, in that if a MoML file is exported, it correctly reflects the topology. However, it loses information about the heritage of composite objects that are derived from others.

There is an additional subtlety. If a topology is modified by directly making kernel calls, then exportMoML() will normally export the modified topology. However, if a derived component is modified, then exportMoML() may fail to catch the changes. In particular, consider the following example:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE model PUBLIC "..." "http://...">
<model name="top"
        class="ptolemy.kernel.CompositeEntity">
    <class name="base"
        extends="ptolemy.kernel.CompositeEntity">
        <entity name="e1"
            class="ptolemy.kernel.ComponentEntity"/>
    </class>
    <entity name="derived" class="master"/>
</model>
```

Here, the derived class does not modify the topology, so exportMoML() produces:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE model PUBLIC "..." "http://...">
<model name="top"
        class="ptolemy.kernel.CompositeEntity">
    <class name="master"
        extends="ptolemy.kernel.CompositeEntity">
        <entity name="e1"
            class="ptolemy.kernel.ComponentEntity"/>
    </class>
    <entity name="derived" class=".top.master"/>
</model>
```

This MoML code will be exported *even if the derived class has been modified by making direct kernel calls.* This actually can prove to be convenient. It means that if a model mutates during execution, and is later saved, that a user interface can ensure that only the original model, before mutations, is saved. It does this by creating a class for the model, instantiating the class without modifying it, and executing the instance.

4.0 ACKNOWLEDGEMENTS

Many thanks to Ed Willink of Racal Research Ltd. for many helpful suggestions, only some of which have made it into this version of MoML. Also, thanks to Tom Henzinger, Alberto Sangiovanni-Vincentelli, and Kees Vissers for helping clarify issues of abstract syntax.

5.0 REFERENCES

[1] J. Davis, R. Galicia, M. Goel, C. Hylands, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay and Y. Xiong, "Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java," Technical Report UCB/ERL No. M99/40, University of California, Berkeley, CA 94720, July 19, 1999. (http://ptolemy.eecs.berkeley.edu/papers/99/HMAD).

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.