XMILE: An XML Interchange Language for System Dynamics

Information Systems SIG of the Systems Dynamics Society
Language Subcommittee
Karim Chichakly
7 June 2013

Background

In the Spring 2003 System Dynamics Society newsletter, Jim Hines proposed that there be a common interchange format for system dynamics models, called SMILE. More details can be found in the companion SMILE document.

Vedat Diker and Robert Allen later presented a poster at the 2005 ISDC that proposed a working group be formed and that XML be the working language for the standard. At the first meeting of the Information Systems SIG at the 2006 ISDC, I suggested breaking the problem into two pieces: the language we intend to interchange and the interchange format. This second document on XMILE (XML Modeling Interchange LanguagE) documents the interchange format used for SMILE and supersedes documents from 2007 and 2009.

1.0 XMILE Standard Levels

The interchange format is built in three layers (called *compliance levels*). Each level builds upon the prior level in a way that allows programs that only support the lowest level to still read files generated at the highest level. The compliance levels are:

1: Simulation
2: Display
3: Interface

As engineers, perhaps we should give them more cryptic names, such as “Level 307.43b” or “Subset g”, but I think “Level 1 compliance” is sufficiently obscure already.

The first layer, Simulation, is the minimal level needed for compliance. We intend for everyone to support this layer. This represents only the underlying equations of a model in SMILE. If we wished to have an unstructured text format, this could be accomplished by using a variant of DYNAMO or MDL. It represents the basic information necessary to simulate the model (we could also name it the Equations layer).

The second layer, Display, adds the information necessary to both display and edit the stock-flow diagram of the model. We hope everyone will also implement this layer.

The third layer, Interface, adds layout information necessary for management flight simulators, i.e., user interfaces on top of models. This layer also includes all output devices, such as graphs and tables, so it is likely everyone will support part of this.

2.0 Overall Architecture

A XMILE file begins with a header that identifies the vendor, program, and version number that created the file. It also includes information about advanced features used in the model, such as arrays (with the number of dimensions) or conveyors, so programs that do not support the higher-order SMILE functionality can find out right off and tell the user.

The file is conceptually broken into three sections (though, in practice, these pieces are interwoven):

- Model
- Presentation
- Widgets

Any data from simulation runs is kept separate from this information about the structure of the model. A separate data layer is not included as there is no intention to support data interchange.

Although sections include room for vendor-specific extensions, it is recommended to tread lightly in this area.

2.1 Model

The model section corresponds exactly to the Simulation level. It conforms to SMILE and contains all the information necessary to simulate the model.

2.2 Presentation

The presentation section is necessary to support the Display level, but it is not restricted to that level. Presentation involves all aspects of drawing an object, including its position, its color, its font, its relative size, etc.

Presentation information is hierarchical in the same way cascading style sheets and XML style sheets are. Global styles can be defined (such as "all stocks are blue") and can then be overridden at any level.

Note that this is used by *both* the Display and Interface levels.

2.3 Widgets

Widgets are the objects used in the model to support the use of the model, such as graphs, tables, sliders, knobs, etc. As such, widgets live entirely in the Interface level.

This section of the file defines the specific widgets being used with their necessary parameters, but not their presentation. For example, an entry may describe a slider as controlling a specific model variable with a given range and increment.

3.0 XMILE Headers

The XMILE file should be encoded in UTF-8. The entire XMILE file is enclosed within a `<xmile>` tag as follows:

```
<xmile version="1.0" level="2"
   xmlns="http://www.systemdynamics.org/XMILE">
   ...
</xmile>
```

The version number is the XMILE version number (presently 1.0). The level number is the XMILE compliance level as defined in section 1.0. Both of these attributes are required.

The first block within the `<xmile>` tag is the header. The header contains important information about the origin of the model. Some of this information is required, but other pieces are optional. The XML tag for the header is `<header>`. The required pieces are:

- SMILE options: `<smile>` (defined below)
- Vendor name: `<vendor>` w/company name
- Product name: `<product version="..." lang="...">` w/product name – the product version number is required. The language code is optional (default: English) and describes the language used for variable names and comments. Language codes are described by ISO 639-1 unless the language is not there, in which case the ISO 639-2 code should be used (e.g., for Hawaiian).

Optional pieces include:

- Model name: `<name>` w/name
- Model version: `<version>` w/version information
- Model caption: `<caption>` w/caption
- Picture of the model in JPG, GIF, TIF, or PNG format: `<image>` w/URL-format pathname (default protocol is file://) or picture data in Data URI format, using base64 encoding
- Author name: `<author>` w/author name
- Company name: `<affiliation>` w/company name
- Client name: `<client>` w/client name
- Copyright notice: `<copyright>` w/copyright information
- Contact information (e-mail, phone, mailing address, web site):
  `<contact>` block w/contact information broken into `<address>`, `<phone>`, `<fax>`, `<email>`, and `<website>`, all optional
- Date created: `<created>` w/“mm/dd/yyyy hh:mm Xm”, where “X” is “a” or “p” (leading zeroes suppressed)
- Date modified: `<modified>` w/date as above
- Model universally unique ID: `<uuid>` with the ID in IETF RFC4122 format (8-4-4-4-12 hex digits with the dashes)

3.1 SMILE Options

The SMILE options appear under the tag `<smile>`. This is a list of optional functionality used in the model file.

```
<smile version="1.0">
   ...
</smile>
```

The version number is required and is the SMILE version used (presently 1.0). The available options are:

```
<uses_conveyor/>
<uses_queue/>
<uses_arrays>N</uses_arrays>  <!-- N = maximum dims used -->
<uses_submodels/>
```

There is one optional attribute for the `<smile>` tag:

- Namespace: `namespace="…"` with SMILE namespaces, separated by commas. For example, `namespace="std, isee"` means try to resolve unrecognized identifiers against the `std` namespace first, and then against the `isee` namespace. (default: `std`)

3.2 Style Information

Every XMILE file can optionally include style information to set default options. Being style information, this mostly belongs to the Presentation section (which affects both the Display and Interface layers). However, it is also possible to set default styles for Model and Widget sections, for example, setting all stocks to be non-negative.

The style information is cascading across five levels from the entity outwards, with the actual entity style defined by the first occurrence of a style definition for that style property:

1: Styles for the given entity
2: Styles for the entire Display level or Interface level (affects only that level)
3: Styles for the entire model (affects only that Model section)
4: Styles for the entire model file (affects all Model sections)
5: Default XMILE-defined styles when a default appears in this specification

The style information for the entire model file immediately follows the header. This usually includes program defaults when they differ from the standard, though it can also be used for file-specific file-wide settings. Whenever possible, style information uses standard CSS syntax and keywords recast into XML attributes and nodes.

The style block begins with the `<style>` tag. Within this block, any known object can have its attributes set globally (but overridden locally) using its own modifier tags.

Global settings that apply to everything are specified directly on the `<style>` tag or in nodes below it; this is true for `<style>` tags that appear within the `<model>` and `<display>` tags as well, i.e., these settings do not appear within `<model>` or `<display>` tags within the `<style>` tag (the `<style>` tag, by virtue of its position in the file, already says that it affects either the entire file, the `<model>`, the `<display>`, or the `<interface>`). For example, the following sets the color of objects within all models to blue and the background to white:

```
<style color="blue" background="white"/>
```

Unless otherwise indicated or specified, style information appears in XML attributes. For example, `font-family` would be an attribute.

These changes can also be applied directly to objects (again as a child to a `<style>` tag), e.g.,

```
<style color="blue" background="white">
   <connector color="magenta">
</style>
```

Note that when style information applies to a specific object, that style cannot be overridden at a lower level (e.g., the Display) by a change to the overall style (i.e., by the options on the `<style>` tag). Using the example above, to override the color of connectors at a lower level (e.g., the Display), the `<connector>` tag must explicitly appear in that level's style block. If it does not appear there, connectors will be magenta at that level by default, even if the style block at that level sets the default color of all objects to green. In other words, object-specific styles at any level above an object take precedence over an overall style defined at any level.

As already stated, these settings can also be from the Simulation section of XMILE, so all stocks and flows can be set to be non-negative by default:

```
<style color="blue" background="white">
   <non_negative/>
</style>
```

Only stocks or only flows can be also set to be non-negative by default (flows in this example):

```
<style color="blue" background="white">
   <flow>
      <non_negative/>
   </flow>
</style>
```

3.3 Simulation Specifications

Every XMILE file with a Model section must contain at least one set of simulation specifications, as required in the SMILE language. The simulation specifications for the entire model appear immediately after the style information (if present, otherwise after the header). This block must always be present. Note that the simulation specifications can be repeated in each Model section to override specific global defaults. Great care must be taken in these situations, as outlined in the SMILE document.

The simulation specifications block begins with the tag `<sim_specs>`. The following properties are required:

- Start time: `<start>` w/time
- Stop time: `<stop>` w/time (after start time)

Optional properties:

- Step size: `<dt>` w/value (default: 1)
  Optionally specified as the integer reciprocal of DT (for DT <= 1 only): `reciprocal="…"` attribute of `<dt>` with true/false (default: false)
- Integration method: `method="…"` w/SMILE code (default: Euler's)
- Unit of time: `time_units="…"` w/SMILE code (default: "time")
- Pause interval: `pause="…"` w/interval (default: infinity – can be ignored)
- Run selected groups or modules: `<run by="…">` with run type either: `all`, `group`, or `module` (default: all, i.e., run entire model). Which groups or modules to run are identified by `run` attributes on the group or model.

All of the optional properties define default settings that are to be used if the property is not included.

3.4 Dimension Specifications

When the `<uses_arrays>` SMILE option is set, a list of dimension names is required. These dimension names must be consistent across all models.[1] The set of dimension names appear within a `<dimensions>` block as shown in the example below.

```
<dimensions>
   <dim name="N" size="5"/>     <!-- numbered indices -->
   <dim name="Location">        <!-- named indices -->
      <elem name="Boston"/>     <!-- name of 1st index -->
      <elem name="Chicago"/>    <!-- name of 2nd index -->
      <elem name="LA"/>         <!-- name of 3rd index -->
   </dim>
</dimensions>
```

As shown, each dimension name is identified with a `<dim>` tag and a required name. If the elements are not named, only the size of the dimension is given (the lower bound is always one). If the elements have names, they appear in order in `<elem>` nodes. The

[1] A future SMILE version may allow dimension names to also be local to a model.

dimension size should *not* appear when elements have names as the number of element names always determines the size of such dimensions.

3.5 Units Specifications

All user-specified model units are defined in the `<model_units>` tag as shown below:

```
<model_units>
   <units>
      <eqn>models</eqn>             <!-- equation only -->
   </units>
   <units name="models per person per year">
      <eqn>models/(person-yr)</eqn> <!-- name, equation -->
   </units>
   <units name="models per year">
      <eqn>models/yr</eqn>          <!-- name, eqn, alias -->
      <alias>mpy</alias>
   </units>
</model_units>
```

Unit equations (`<eqn>` tag) are defined with SMILE unit expressions. The user-identifiable name appears in the `name="…"` attribute. One `<alias>` tag with the name on the alias appears for each distinct unit alias.

Vendor-provided unit definitions are not required to appear in every model file, but must be made available in this same format in a vendor-specific library.

3.6 Data Import/Export

Persistent data import/export connections are defined with the `<data>` tag, which contains one `<import>` tag for each data import connection and one `<export>` tag for each data export connection. Both tags include the following properties (the first four are optional):

- Type: `type="…"` with “CSV”, “Excel”, or “XML” (default: CSV)
- Enabled state: `enabled="…"` with true/false (default: true)
- How often: `frequency="…"` with either “on_demand” or “automatic” (default: automatic, i.e., whenever the data changes)
- Data orientation: `orientation="…"` with either “horizontal” or “vertical” (default: vertical)
- Source (import) or destination (export) location: `url="…"` w/URL path
- For Excel only, worksheet name: `worksheet="…"` with worksheet name

The `<export>` also specifies both the optional export interval and one of two sources of the data:

- Export interval: `interval="…"` specifying how often, in model time, to export values during the simulation; use `"DT"` to export every DT (default: 0, meaning only once)
- `<all/>` to export all model variables or `<table name=="…"/>` to just export the variables named in the table (note that any array element in the table will export the entire array when interval is set to zero). The `<table>` tag has an optional attribute `use_settings="…"` with a true/false value (default: false), which when true causes the table settings for orientation, interval, and number formatting to be used (thus, when it is set, neither orientation nor interval are meaningful, so should not appear).

Additional data sources and sinks, such as databases and web servers, are desired, but not yet specified.

4.0 Level 1: Simulation

The simulation layer supports all of the features of the SMILE language. With the exception of the presentation aspects of the style block, all of the blocks described above belong to level 1. The meat of this layer is within the model block. Note that there can be many model blocks, one for each submodel defined.

Each model block uses the XML tag `<model>`. Each model tag has optional attributes:

- Name: `name="…"` with the model name. When `uses_submodels` is set, this is required for all but the root model.
- Run enabled: `run="…"` with true/false (default: false). This is only in effect when running the model by module.
- Password if locked: `password="…"` with password. This should be the public key for an RSA-encrypted model block (i.e., all contents with the passworded model block are encrypted).[2]
- Namespace: `namespace="…"` with SMILE namespaces, separated by commas. For example, `namespace="std, isee"` means resolve unrecognized identifiers against the `std` namespace first, and then against the `isee` namespace. If this setting appears, it overrides across the model the file-level namespace setting in the SMILE options block. (default: same as the file-level namespace)
- Optional reference to a separate model file for the model contents: `url="…"` with a URL path. Since the `<module>` tag can also specify the model file URL, this is only needed for models that do not use modules. However, when the same module is shared in multiple parts of the file, it is better to create one URL reference in a model block, rather than specifying it in the module block.

As mentioned above, the model block can include either or both of the style and simulation specifications blocks, overriding only those settings that are desired. These blocks should appear at the very start of the model block.

Finally, the model building blocks (stocks, flows, and auxiliaries) defined by SMILE are listed in any order.

4.1 Building Block Properties

All building blocks can have the following properties.

- Name: `name="…"` attribute w/valid SMILE identifier
- Access: `access="…"` attribute w/valid SMILE access specifier – see Submodels below (default: none)
- Access automatically set to output: `autoexport="…"` attribute with true/false – see Submodels below (default: false)
- Subscript: `subscript="…"` attribute with comma-separated indices for array element (array elements of non-apply-to-all arrays only – see Arrays below) (default: none)
- Equation: `<eqn>` w/valid SMILE expression in a CDATA section if needed
- MathML equation: <mathml> block containing content-based MathML representation of the equation[3]
- Units: `<units>` w/valid SMILE units (default: none)

[2] In iThink 10, this is a proprietarily-encoded private password with no encoding of the model section.
[3] This is optional for a number of reasons, most notably the size of the representation. All vendors, however, should strive to offer MathML output as a user-option.

- Documentation: `<doc>` w/block of text, optionally in HTML format (default: none)
- Dimensions: `<dimensions>` w/`<dim name="…">` for each dim in order (see Arrays below) (default: none)
- Message Poster: `<event_poster>` block described below (default: none)
- Options: as required for each building block (described below)

Of these, the name and the equation are required for all building blocks. In addition, the name must be unique across the model block. For a stock, the equation is for the stock's initial value, rather than for the stock itself.

The documentation can be plain text or can be HTML. If plain text, it should use SMILE identifier escape sequences for non-printable characters (i.e., \n for newline, \r for return, \t for tab, and, necessarily, \\ for backslash), rather than a hexadecimal code such as &#x0A. If HTML, it includes the proper HTML header. Note this is true for all documentation and text fields in a XMILE file.

4.1.1 Graphical Functions

Flows and auxiliaries can be defined as graphical functions. This is done using a `<gf>` block within the building block, as shown below.

```
<gf>
   <xscale min="0" max="0.5"/>
   <yscale min="0" max="1"/>
   <ypts sep=",">0.05,0.1,0.2,0.25,0.3,0.33</ypts>
</gf>
```

As can be seen, the bounds of the x and y scales are given, followed by a set of *y*-values (tagged `<ypts>`). The (fixed) increment along the *x*-axis is determined by dividing the length of the *x*-scale by the number of *y*-values given. The optional separator for the points is a comma by default, but can be overridden with the `sep="…"` attribute.

Alternatively, though not guaranteed to be understood by all packages if the *x*-increment is not constant, the entire graphical function can be defined as a sorted (by *x*) series of (*x*, *y*)-pairs. A representation equivalent to the above graphical function is shown below.

```
<gf>
   <yscale min="0" max="1"/>
   <xpts>0,0.1,0.2,0.3,0.4,0.5</xpts>
   <ypts>0.05,0.1,0.2,0.25,0.3,0.33</ypts>
</gf>
```

Note the *x*-scale is now inferred from the bounds of the given points. In either formulation, the *y*-scale only needs to be included if the desired scale differs from the minimum and maximum *y*-values in the set of points. Thus, the smallest graphical function representation would be:

```
<gf>
   <xscale min="0" max="1"/>
   <ypts>0,0.1,0.5,0.9,1</ypts>
</gf>
```

This definition implies a corresponding set of *x*-values of (0, 0.25, 0.5, 0.75, 1) and a *y*-scale from 0 to 1.

The `<gf>` block has two optional attributes:

- Name: `name="…"` w/valid SMILE identifier (default: anonymous)
- Type: `type="…"` w/valid SMILE graphical function type name (default: `continuous`)

When named, graphical functions can be directly referred to by other entities. For example, we can create a stand-alone (outside all building blocks) named graphical function as shown below:

```
<gf name="rising">
   <xscale min="0" max="1"/>
   <ypts>0,0.1,0.5,0.9,1</ypts>
</gf>
```

Such a graphical function can then be directly referred to in a flow or auxiliary as follows:

```
<gf name="rising"/>
```

As defined in SMILE, named graphical functions can also be used directly in equations, as if they were functions, for example:

```
<eqn>rising(star)</eqn>  <!-- evaluate gf at star -->
```

4.1.2 Message Posters

Events based on entity values can be triggered while the model is being simulated. In XMILE Level 1, these events are limited to pausing the simulation (default) or stopping the simulation.

All events appear in an `event_poster` block with `min` and `max` attributes specifying the lower and upper bounds for all posters (this is a user setting to help them decide where to place events). A series of `threshold` blocks then define the events:

```
<event_poster min="0" max="10">
   <threshold value="5">
      ...
   </threshold>
</event_poster>
```

The threshold has these additional optional attributes:

- Direction: `direction="…"` w/valid SMILE event direction name (default: `increasing`)
- Frequency: `repeat="…"` w/valid SMILE event frequency name (default: `each`)
- Repetition interval: `interval="…"` w/number of unit times (default: disabled; only enabled if present)

Each threshold block must have a unique value and direction (so there can be two threshold blocks at 5 as long as one is increasing and the other is decreasing). Within each threshold block, the actual events are defined, which can be either one that is used every time the threshold is exceeded (*must* be only one for frequency `each`) or a sequence of events that are used one at a time in order each time the threshold is exceeded. Events appear in an `<event>` tag which has one optional attribute:

- Action: `sim_action="…"` w/valid SMILE event action name (default: `pause`)

4.2 Stocks

The minimum requirement for a stock is name, an initial value, and a set of flows. Rather than write a stock equation, SMILE mandates that we classify the flows that affect the stock as either inflows or outflows[4]. The basic stock definition is shown below with sample values.

```
<stock name="Motivation">
   <eqn>100</eqn>
   <inflow>increasing</inflow>
   <outflow>decreasing</outflow>
</stock>
```

Note again that the equation is for the stock's initial value only. If the equation is not constant, the initial values of the included variables will be used to calculate the stock's initial value.

There need not be any inflows or outflows. If there are multiple inflows, they appear with multiple tags in inflow-priority order (if order of inflow to the stock is important). For example, if `in1` must be treated before `in2`, the inflows would appear within the `<stock>` block in this order:

```
<inflow>in1</inflow>
<inflow>in2</inflow>
```

Multiple outflows appear in a similar fashion, i.e., in separate tags. The order that they appear is their outflow priority, if that is important. I.e., material from the stock is first given to the first outflow listed. If there is still something in the stock, the second

[4] The classification is based on the direction of flow when the flow rate is positive. Negative inflows flow outward while negative outflows flow inward.

outflow gets some of it. Inflow priority is only important for queues and capacity-constrained conveyors. Outflow priority is only important for non-negative stocks (for example, Inventory), queues, and conveyors with multiple leakages.

The options for a stock are:

```
<conveyor>
   <len>5</len>          <!-- length of the conveyor -->
   ...                   <!-- conveyor options (below) -->
</conveyor>
<queue/>
<non_negative/>
```

Note that non-negative is not directly supported by SMILE. The flag exists partly for documentation, partly to allow a vendor to invoke a macro to implement the functionality. If this property has been set at the global level, it can be turned off locally with `<non_negative>false</non_negative>`.

The conveyor options include the conveyor's length (transit time). This can be an equation if the conveyor's length varies. Exactly one outflow of the conveyor, *by definition*, must be the conveyor outflow. Leakage flows can be explicitly tagged with the `<leak/>` tag, but absent these tags, by convention the first outflow in the outflows list is the conveyor outflow and all subsequent outflows are leakage outflows. Other conveyor options, which must appear in the conveyor block if provided, are:

```
<capacity>20</capacity> <!-- maximum allowed on conveyor -->
<in_limit>5</in_limit>  <!-- most that can be taken per time -->
<sample>TIME mod 5 = 0</sample>    <!-- sample transit time -->
<arrest>broken</arrest> <!-- arrest conveyor -->
```

The following conveyor options are attributes of the `<conveyor>` tag:

- Discrete: `discrete="…"` with true/false (default: false); true if a discrete conveyor (must be discrete if there is a queue upstream)
- Batch integrity: `batch_integrity="…"` with true/false (default: false); true if only whole batches can be taken from the (required) upstream queue and false if partial batches are allowed
- Number of batches: `one_at_a_time="…"` with true/false (default: true); true if only the front batch from the (required) upstream queue can be taken each DT and false if as many batches as will fit (subject to capacity and inflow limits) can be taken
- Exponential leakage: `exponential_leak="…"` with true/false (default: false); if true, leakage is an exponential decay across the conveyor (rather than linear)

Queues do not have any options. However, their outflows have a priority order and can have the `<overflow/>` option set on all but the first.

See the SMILE document for further details.

4.3 Flows

A flow requires a name and an equation. In the case of conveyor and queue outflows, there is no equation as the conveyor or queue drives them. The conveyor leakage flow, however, does have an equation for its leakage fraction. The basic flow definition is shown below with sample values.

```
<flow name="increasing">
   <eqn>rewards*reward_multiplier</eqn>
</flow>
```

The options for a flow are:

```
<non_negative/>     <!-- mark a uniflow -->
<overflow/>         <!-- mark a queue overflow -->
<leak/>             <!-- mark a conveyor leakage -->
<leak_integers/>    <!-- leak integers from a leakage -->
<multiplier>5/8</multiplier>  <!-- unit conversion -->
```

The first is used to document a uniflow, the second is only used to identify a queue overflow (see section 4.2, “Stocks”), the third is used to explicitly identify leakage flows of a conveyor (optional), and the fourth is a leakage flow option. While it is tempting to also label queue outflows and conveyor outflows, these are by their nature already defined by the stocks that use them; there is no need to repeat that information here.

The last option, `<multiplier>`, specifies the SMILE expression that converts units between stocks when material is transformed in a main chain.

Leakage flows can also have these attributes:

- Leak zone start: `leak_start="…"` with the fractional starting position of the leakage zone within the conveyor (from the inflowing side) (default: 0)
- Leak zone end: `leak_end="…"` with the fractional ending position of the leakage zone within the conveyor (from the inflowing side) (default: 1)

4.4 Auxiliaries

Auxiliaries, like flows, also require a name and an equation. The basic auxiliary definition is shown below with sample values.

```
<aux name="reward_multiplier">
   <eqn>0.15</eqn>
</aux>
```

This particular auxiliary is a constant (i.e., it depends on no other variables and its value does not change).

Auxiliaries have one optional attribute (other than those already listed above):

- Flow concept: `flow_concept="…"` with true/false, which is true if the converter represents a flow concept (default: false)

4.5 Arrays

There are three different kinds of arrays:

- Apply-to-all arrays, where one entity acts as a surrogate for all array elements,
- Non-apply-to-all arrays, where one entity acts as a container for many building block array elements (this container also shares some things across all elements, e.g., units), and
- Apply-to-all arrays with non-apply-to-all graphical functions, where one entity acts as the surrogate for all properties of the array elements except the graphical function used for each element.

4.5.1 Apply-to-all Arrays

A building block that is an apply-to-all array is identified with a `<dimensions>` block, which lists the names of the array dimensions in order using the `<dim>` tag. For example, the following XMILE defines a 2D (X by Y) apply-to-all arrayed converter:

```
<aux name="distance">
   <dimensions>
      <dim name="X"/>
      <dim name="Y"/>
   </dimensions>
   <eqn>SQRT(X^2 + Y^2)</eqn>
   <units>kilometers</units>
</aux>
```

4.5.2 Non-apply-to-all Arrays

Non-apply-to-all arrays are defined by an `<array>` block, which shares the same attributes (name, access, autoexport) as the building blocks. The array block begins with a `<dimensions>` block, which lists the names of the array dimensions in order using the `<dim>` tag. Each array element is then represented using the proper building block tag (which defines the type of array, i.e., stock, flow, or aux) with only the `subscript` attribute allowed. Each element can define its own equation and other simulation attributes (such as event poster and transit time) that is allowed to vary between elements. The documentation, the units, and the non-negative setting are part of the array block, and so are shared for all elements. The event poster can also be shared. The following defines a 2D (X by X) non-apply-to-all arrayed converter:

```
<array name="not_apply_to_all">
   <dimensions>
      <dim name="X"/>
      <dim name="X"/>
   </dimensions>
   <aux subscript="1,1">
      <eqn>TIME</eqn>
   </aux>
   <aux subscript="1,2">
      <eqn>2*TIME</eqn>
   </aux>
   <aux subscript="2,1">
      <eqn>TIME^2</eqn>
   </aux>
   <aux subscript="2,2">
      <eqn>2*TIME^2</eqn>
   </aux>
   <units>Euros</units>
</array>
```

4.5.3 Apply-to-all Arrays with Non-apply-to-all Graphical Functions

Arrayed graphical functions are required to have one equation, so are technically apply-to-all arrays and normally appear in the same way as apply-to-all arrays. However, it is possible to specify a different graphical function for each model element, and in these cases arrayed graphical functions appear in an `<array>` block, just like a non-apply-to-all array. The only information that can appear in each array element is a `<gf>` block describing the graphical function for that element. All other attributes are associated with the array owner and so appear as part of the array block. The following defines a 1D (X) apply-to-all arrayed converter with non-apply-to-all graphical functions.

```
<array name="graphical_not_apply_to_all">
   <dimensions>
      <dim name="X"/>
   </dimensions>
   <aux subscript="1">
      <gf>
         <xpts>1,5,9</xpts>
         <ypts>0,50,25</ypts>
         <xscale min="1" max="9" />
         <yscale min="0" max="50" />
      </gf>
   </aux>
   <aux subscript="2">
      <gf>
         <xpts>1,5,9</xpts>
         <ypts>90,10,50</ypts>
         <xscale min="1" max="9" />
         <yscale min="0" max="100" />
      </gf>
   </aux>
   <eqn>TIME</eqn>
   <units>buildings</units>
</array>
```

4.6 Groups

Groups, aka sectors, collect related model structure together. Groups require a name and may have documentation. The group includes the names of all entities in that group using the `<entity>` tag with a `name` attribute. Display objects (from level 2) can also appear here as described in the "Level 2: Display" section.

```
<group name="Financial Sector">
   <doc>
      The operation of the Finance department is modeled in
      this sector.
   </doc>
   <entity name="Cumulative Revenue"/>
   <entity name="revenue"/>
   ...
</group>
```

Each group tag has optional attributes:

- Run enabled: `run="…"` with true/false (default: false). This is only in effect when running the model by group.
- Locked: `locked="…"` with true/false (default: true). When true, all entities within the group are treated as an indivisible group, e.g., they move or delete with the group object. [Note this is not the same as the model's `<locked>` tag.]

4.7 Submodels

Submodels can optionally have a placeholder entity (known as a module) that references them in every model that uses them. A module appears in a `<module>` block with only two possible attributes:

- Name: `name="…"` attribute w/valid SMILE identifier that refers directly to the named submodel
- Optional location: `url="…"` w/a URL path to the submodel's file. This is required if the submodel, or a reference to the submodel with its own URL, is not in the model file.

When a module does exist, connections from entities to that module automatically denote that those entities are submodel outputs into just that module, so no additional designation is used. Connections from the module to entities automatically turn those entities into submodel inputs (inputs are always marked as such in the XMILE).

An example of an assigned submodel input appears below. Note the `<access>` tag is what identifies it as a submodel input. We know it is assigned because it includes an `<of>` tag (the same tag used for aliases).

```
<aux name="xyz" access="input">
   <eqn>STEP(1, 2)</eqn>
   <of>module.abc</of>
</aux>
```

This is both the simplest way, as well as the preferred way to link the output of a reuseable module to the model above it. In other cases, though, it is not the preferred way to link a module input to its output as it restricts the module containing the input to a single use within the entire model (as the saved model file links the submodel input to exactly one submodel output, or parameter). The preferred way in these other cases is to use the `<to>` tag in the module output:

```
<aux name="abc" access="output">
   <eqn>SINWAVE(5, 4*PI)</eqn>
   <to>module2.xyz</to>
   <to>module3.kjc</to>
</aux>
```

Because of this dual usage, both methods may appear in the same model.

The above submodel output, *abc*, is an explicitly exported submodel output, available to all submodels above or across from the submodel that contains it, as well as all of their submodels. Note that the `<to>` tag can be valid even without the `access` attribute since entities connected to modules make themselves available to submodel inputs in that module (as described above).

Every model entity that is a submodel output can optionally include the following attribute:

- Autoexport: `autoexport="…"` with true/false (default: false). This is true when the software (rather than the user) turned an entity into a submodel output, as, for example, when ghosting across models. When this is true, there are likely connections between modules that were automatically made, complete with connectors. This is indicated with the `autocreate` attribute in a connector.

4.8 Built-in Function Translation Macros

SMILE provides for translation of unrecognized built-ins into SMILE (or anything else, for that matter). There is also a mechanism to define macros to implement non-standard building-block functionality.

4.8.1 Simple Macros

Macros live outside of all other blocks (and may be the only thing in a file other than its header). Each macro is identified by its own `<macro>` tag and begins with its name and parameters. A macro can optionally be assigned to a namespace with the `namespace="…"` attribute, for example, `namespace="isee"`.[5] Macros must also have an equation that defines its value, as shown below.

```
<macro name="LOG">
   <parm>X</parm>
   <parm>Y</parm>
   <eqn>LN(X)/LN(Y)</eqn>
</macro>
```

Parameters to the function appear in the required order. This implicitly gives the expected form of the function: LOG(X, Y). There is only one result, the value of the equation.

Optional blocks provide helpful information to the modeler wishing to use the function:

```
<format>LOG(<value>, <base>)</format>
<desc>Finds the base-<base> logarithm of <value>.</desc>
```

[5] When the `namespace` attribute appears in the `<smile>` tag and specifies a single namespace, all macros default to be in that namespace. This can be used to force all macros in a file to be in the same namespace.

4.8.2 Recursion in Macros

Macros can be recursive:

```
<macro name="FACT">
   <parm>x</parm>
   <eqn>IF x <= 1 THEN 1 ELSE FACT(x – 1)</eqn>
</macro>
```

Nested loops can be implemented by calling recursive functions from within a recursive function, for example, finding the sum of the first *x* factorials:

```
<macro name="FACTSUM">
   <parm>x</parm>
   <eqn>IF x <= 1 THEN 1 ELSE FACT(x) + FACTSUM(x – 1)</eqn>
</macro>
```

With optional parameters that have default values, more complex operations can be implemented, such as finding the index of a value in a 1D-array:

```
<macro name="FIND">
   <parm>A</parm>
   <parm>value</parm>
   <parm default=”1”>i</parm>
   <eqn>IF i > SIZE(A) THEN 0 ELSE
      IF A[i] = value THEN i ELSE FIND(A, value, i + 1)</eqn>
</macro>
```

4.8.3 Complex Macros

The above simple forms are useful when functions can be directly represented by existing built-ins. However, sometimes extra variables (stocks, flows, auxiliaries) are needed. In these cases, the extra variables must also be defined. A macro block is implicitly also a model block, so these variables are defined the same way they are in the model. This is shown below.

```
<macro name="SMOOTH1">
   <parm>input</parm>
   <parm>averaging_time</parm>
   <eqn>Smooth_of_Input</eqn>
   <stock name="Smooth_of_Input">
      <eqn>input</eqn>
      <inflow>change_in_smooth</inflow>
   </stock>
   <flow name="change_in_smooth">
      <eqn>(input – Smooth_of_Input)/averaging_time</eqn>
   </flow>
</macro>
```

This is identical to the SMTH1 function without the optional third parameter (if given, it would replace the initial value equation of Smooth_of_Input), so this is a trivial example[6]. However, it clearly demonstrates the generality and applicability of this mechanism.

Note that any stocks that are defined within a macro must have their own instances for each use of that macro and these instances must persist across the simulation. That is to say, if this SMOOTH1 function is used five times in a model, there must also be five copies of the stock Smooth_of_Input, one for each use. Use of flows and auxiliaries do not require this (auxiliaries, in particular, are useful to simplify the equation into meaningful pieces).

Note also that Level 2 (Display) information can be included in macros to simplify the creation and editing of them.

4.8.4 Loops in Macros

Note that `<macro>` blocks differ from `<model>` blocks only in the addition of parameter, equation, format, and description blocks. Thus, it is possible – and extremely useful – to include simulation specs inside a macro, thus turning it into a loop. For example, a looped version of factorial looks like:

```
<macro name="FACT">
   <parm>x</parm>
   <eqn>Fact</eqn>
   <simspecs>
      <start>1</start>
      <end>x</end>       <!-- note can be parameterized -->
   </simspecs>
   <stock name="Fact">
      <eqn>1</eqn>
      <inflow>change_in_fact</inflow>
   </stock>
   <flow name="change_in_fact">
      <eqn>Fact*TIME</eqn>
   </flow>
</macro>
```

The recursive macro shown to find a value in a 1D-array (in section 4.8.2) can also be expressed this way:

[6] Setting the equation to `SMTH1(input, averaging_time)` does the same thing.

```
<macro name="FIND">
   <parm>A</parm>
   <parm>value</parm>
   <eqn>index</eqn>
   <time_specs>
      <starttime>1</starttime>
      <stoptime>SIZE(A)</stoptime>
   </time_specs>
   <stock name="index">
      <eqn>0</eqn>
      <inflow>find_value</inflow>
   </stock>
   <flow name="find_value">
      <eqn>
            IF (index = 0) and (A[TIME] = value)
            THEN TIME/DT
            ELSE 0
      </eqn>
   </flow>
</macro>
```

4.8.5 Nesting Loops in Macros

It is possible to nest loops, for example, summing all elements in a 2D array with dims X and Y:

```
<macro name="SUM2D">
   <parm>A</parm>
   <eqn>Sum</eqn>
   <simspecs>
      <start>1</start>
      <end>SIZE(A[*, 1])</end>
   </simspecs>
   <stock name="Sum">
      <eqn>0</eqn>
      <inflow>change_in_sum</inflow>
   </stock>
   <flow name="change_in_sum">
      <eqn>SUM1D(A[TIME, *])</eqn>
   </flow>
</macro>
```

```
<macro name="SUM1D">
   <parm>A</parm>
   <eqn>Sum</eqn>
   <simspecs>
      <start>1</start>
      <end>SIZE(A)</end>
   </simspecs>
   <stock name="Sum">
      <eqn>0</eqn>
      <inflow>change_in_sum</inflow>
   </stock>
   <flow name="change_in_sum">
      <eqn>A[TIME]></eqn>
   </flow>
</macro>
```

This method takes advantage of array slicing, building up a two-dimensional result by breaking the 2D array into one-dimensional rows. In the general case, though, it is necessary to use either a separate macro for the inner loop, or separate models within the macro. The first case, using a separate macro for the inner loop (without using slicing), is shown below. In this case, the equation in SUM2D for `change_in_sum` would be changed to `SUMROW(A, TIME)`.

```
<macro name="SUMROW">
   <parm>A</parm>
   <parm>i</parm>
   <eqn>Sum</eqn>
   <simspecs>
      <start>1</start>
      <end>SIZE(A[i, *])</end>
   </simspecs>
   <stock name="Sum">
      <eqn>0</eqn>
      <inflow>change_in_sum</inflow>
   </stock>
   <flow name="change_in_sum">
      <eqn>A[i, TIME]></eqn>
   </flow>
</macro>
```

Explicitly using a second model inside the macro is shown below[7]:

```
<macro name="2DSUM">
   <parm>A</parm>
   <eqn>Sum</eqn>
   <simspecs>
      <start>1</start>
      <end>SIZE(A[*, 1])</end>
   </simspecs>
   <stock name="Sum">
      <eqn>0</eqn>
      <inflow>change_in_sum</inflow>
   </stock>
   <flow name="change_in_sum">
      <eqn>inner.Sum</eqn>
   </flow>

   <model name="inner" mode="independent"> <!-- runs each DT -->
      <simspecs>
         <start>1</start>
         <end>SIZE(A[1, *])</end>
      </simspecs>
      <stock name="Sum" access="output">
         <eqn>0</eqn>
         <inflow>change_in_sum</inflow>
      </stock>
      <flow name="change_in_sum">
         <eqn>A[.TIME, TIME]</eqn>  <!-- warning: need .TIME -->
      </flow>
   </model>
</macro>
```

4.8.6 Option Filters

Macros are also used to implement building block options via filters. Here is a simple (i.e., self-contained) option filter. This implements non-negativity for flows (i.e., makes them uniflows).

```
<macro filter="flow" name="non_negative">
   <parm>flow</parm>
   <parm>option</parm>
   <eqn>IF option THEN MAX(flow, 0) ELSE flow</eqn>
</macro>
```

The macro is identified as an option filter by the use of the `filter="…"` attribute, which must specify the name of a building block ("stock", "flow", or "aux") as option names

[7] It is not decided whether SMILE allows models to be nested in macros, as models cannot be nested in models and macros cannot be nested in macros. However, some way is needed to limit the scope of these internal models or macros. Note if this model cannot be nested, the row index needs to be an input to the submodel (it may anyway – accessing .TIME is dubious, though it could be a default parameter).

can be shared. The name of the macro is the same as the option name. It is passed the value of the object calculated without the option, as well as the value of the option[8]. The result replaces the object's value.

Some building block options actually affect other building blocks connected to that building block. More sophisticated filters that operate on a stock's flow, or stocks connected to a flow, can be written by specifying the `applyto="…"` attribute. In these cases, additional parameters are passed to the filter:

filter type	`applyto` value	Parameters
stock	`inflows`	flow, option, stock, inflow_sum
stock	`outflows`	flow, option, stock, outflow_sum
flow	`upstream`	stock, option, flow
flow	`downstream`	stock, option, flow

In the parameter lists above, *flow* refers to the calculated flow value, *stock* refers to the calculated stock value, *option* refers to the value of the option, *inflow_sum* refers to the sum of the stock's inflows that have higher-priority than this flow, and *outflow_sum* refers to the sum of the stock's outflows that have higher-priority than this flow.

Using these filters, the implementation of non-negative stocks is straightforward.

```
<macro filter="stock" name="non_negative" applyto="outflows">
   <parm>flow</parm>
   <parm>value</parm>
   <parm>stock</parm>
   <parm>outflow_sum</parm>
   <eqn>
      IF value
      THEN MAX(stock/DT – outflow_sum, flow)
      ELSE flow
   </eqn>
</macro>
```

Note this implements the non-negative option of a *stock*, which is done by modifying that stock's outflows rather than the stock itself.

The existence of such a macro does not rule out another macro that operates on the stock instead of its outflows. Such a macro would not have the `applyto` option. In fact, any stock or flow option can implement three filters: one for the stock or flow itself and one for each of the two `applyto` options.

[8] For options that are only true or false, one (1) is passed for true and zero (0) is passed for false.

5.0 Level 2: Display

The display layer supports presentation and editing of SMILE objects. Since this layer affects all objects, the display properties are interspersed throughout the model.

Each display block uses the XML tag `<display>`. Within a display block, any display aspect can be specified (or overridden).

There is usually also one `<display>` block nested one level inside the `<model>` which describes all display objects required to properly render the model. This block can also include a display-level style block which overrides settings in both the file- and model-level style blocks.

5.1 Drawing Plane Settings

The following display attributes on the `<display>` tag define the drawing plane:

- Number of pages: `page_rows="…"` `page_cols="…"` (default: 1, 1)
- Page orientation: `orientation="…"` w/SMILE orientation (default: `portrait`)
- Page size: `page_width="…"` `page_height="…"` (defaults to printer)
- Page order: `page_order="…"` w/SMILE order (default: `row`)
- Show pages: `show_pages ="…"` w/true or false (default: false)
- Scroll position: `scroll_x="…"` `scroll_y="…"` (default: 0, 0)
- Zoom level: `zoom="…"` w/ percentage zoom (default: 100.0)

5.2 Display Properties

The following display properties are defined for all presentation objects (including Level 3). Some of them are attributes on the display tag while others are tags in their own right. Note that not all of them are available for all objects.

- Position: `x="…"` `y="…"` (required)
- Predefined size (stocks, auxiliaries, and modules only): `size="…"` with `large`, `medium`, `small`, or `name_only` (default: `medium`)
- Arbitrary size: `width="…"` `height="…"` (not specified w/predefined size)
- Name position: `label_side="…"` w/`top`, `bottom`, `left`, `right`, `center`
  (default: `bottom`, except for stocks which default to `top`)
- Optional name angle: `label_angle="…"` (default: matches name position)
- Color: `color="…"` w/valid CSS color (default: `black`)
- Background color: `background="…"` w/valid CSS color only (default: `white`)
  specific objects can also specify images
- Font: `font-family="…"` with CSS conventions including fallback fonts
  separated by commas, e.g., "Garamond, serif" (default: `sans-serif`)
  `font-size="…"` with CSS conventions and `pt` units by default
  (default: `9`)

`font-style="…"` with CSS conventions (default: `normal`)
`font-weight="…"` with CSS conventions (default: `normal`);
only `normal` or `bold` allowed
`font-color="…"` with valid CSS color (default: `black`)

- Text alignment: `text-align="…"` with CSS conventions
  (default: consistent with name position; `left` if object has no name pos)
- Text decoration: `text-decoration="…"` with `none` or `underline` only
  (default: `none`)
- Border: `border-style="…"` with CSS conventions (default: `solid`)
  `border-width="…"` with CSS conventions (default: `thin`)
  `border-color="…"` with CSS conventions (default: same as object)
- Padding for border: `padding="…"` with CSS conventions, length only,
  single value allowed, and `px` units by default (default: `2`)
- Margin around text: `margin="…"` with CSS conventions, length only,
  single value allowed, and `px` units by default (default: `2`)
- Image: `image="…"` w/path to image or embedded image in base-64 format
- Drawing Order: `z-index="…"` with CSS conventions (default: order in file,
  i.e., SMILE *auto*)

All objects require a position, so there is no default. When the width and height are given, the position is the top-left of the object. Otherwise, the position is the center point of the object (without the nameplate).

Drawing order may not be supported in all programs as some programs define very specific drawing orders. These programs are not fully Level 2-compliant, but can work towards compliance over time.

Default predefined sizes appear as attributes within a stock, aux, or module block of the style block. The sizes associated with these predefined sizes appear in the file's style block only, again in within the stock, aux, or module block, under the tags `<small>`, `<medium>`, and `<large>` using the `width="…"` and `height="…"` attributes, e.g.,

```
<style>
   <aux size="small">
      <small width="11" height="11"/> <-- should be equal -->
      <medium width="18" height="18"/>
   </aux>
</style>
```

Only the sizes actually used in the model file should be defined.

5.3 Display Properties for SMILE Objects

Formatting and scaling display properties are available for all SMILE objects. Some SMILE objects also require special display properties. At present, only the flow and the group have their own properties.

5.3.1 General SMILE Object Display Properties

All SMILE objects can have explicit ranges and scales that are used by default in input and output devices, respectively. These same properties can appear within the input and output devices to override the entity's setting for that device.

The `<range>` tag is used to specify the default input range for an input device. Without it, any reasonable guess can be used (typically tied to the variable's scale). The `<scale>` tag is used to specify the global scale of a variable. Without it, the scale of variable matches the range of its values. Both tags have two attributes:

- Range/scale minimum: `min="…"` with the minimum value for the range/scale
- Range/scale maximum: `max="…"` with the maximum value for the range/scale

Note that `min` <= `max`. For the `<scale>` tag only, there are two optional attributes that can only appear in output devices (typically graphs):

- Autoscale: `auto="…"` with true/false to override the global scale within that output device; note this is mutually exclusive with `min` and `max` (default: false)
- Autoscale group: `group="…"` with a unique number identifying the group in that output device; note this implies `auto="true"` and is therefore mutually exclusive with `min` and `max` (default: not in a group)

Groups require more than one variable in them and are specifically used to autoscale a group of variables to the same scale starting at the minimum value of all variables in the group and ending at the maximum value of all variables in the group. This is the default scaling for all variables in a comparative plot, so does not need to appear in that case.

The `<format>` tag allows default formatting to be set for values of each variable. Without it, the default settings for each attribute below takes effect:

- Precision: `precision="…"` with value indicating precision of least significant digit, e.g., "0.01" to round to the hundredths place or "0.5" to round to the nearest half (default: best guess based on the scale of the variable)
- Magnitude scale: `scale_by="…"` with the factor to scale all values by before displaying, e.g., "1000" to display thousands (default: no scaling, i.e., 1)
- Special symbols: `display_as="…"` with "number", "currency", or "percent" (default: `number`)
- Include thousands separator: `delimit_000s="…"` with true/false (default: false)

These can also be overridden, using the same attribute names, in variable definitions of individual input or output devices.

5.3.2 Flow Display Properties

The position of a flow is not the center of the complex path taken by the flow, but rather the center of the flow's valve.

A flow is typically drawn through a series of perpendicular bends. Each endpoint and bend can be described by one point. These points are necessary to draw the flow correctly, so the flow requires one additional display property, `<pts>`, that gives the flow's points in order from the start of the flow to its end.

```
<pts>
   <pt x="290" y="107"/>
   <pt x="335" y="107"/>
</pts>
```

This is a simple horizontal flow that only has its two endpoints. Programs that use curved flows instead also have control points that could be presumably be manipulated to approximate the perpendicular bend behavior supported by this standard.

5.3.3 Group Display Properties

Groups can be used to collect related model objects together. As such, they can have both borders and background images. They also have three additional control attributes:

- `locked="…"` with true/false (default: `false`)
- `show_name="…"` with true/false (default: `true`)
- `show_image="…"` with true/false (default: `false`, i.e., no image)

See the SMILE document for more detailed information. The position of a group is always its top-left corner, never its center.

Groups can also include display objects, such as connectors and ghosts. These objects are listed within the group's `<display>` tag using the `<item>` tag with a unique ID (described below), e.g.:

```
<display x="100" y="200" width="300" height="150">
   <item uid="3"/>         <!-- connector w/UID 3 -->
   <item uid="5"/>         <!-- ghost w/UID 5 -->
</display>
```

5.4 Display Objects

Several objects outside SMILE are available at the display level. In particular, objects to support the graphical representation of the model are needed. These include connectors and aliases (aka ghosts). Display properties (position, color, etc.) are applied directly to these objects, i.e., a separate `<display>` tag is not used.

5.4.1 Unique IDs

Display objects do not generally have names or any other way to refer to them. For this reason, every display object includes a unique identifying number. These numbers must be unique across all objects in a `<display>` block. They are defined using the `uid="…"`

attribute with the identifying number inside the quotes. Every display object must have a unique ID.

5.4.2 Connectors

Algebraic calculations are represented in a stock-flow diagram with connectors between the model elements. The basic connector definition is shown below with sample values.

```
<connector uid="3" x="283" y="85">
   <from>Inspect</from>
   <to>out</to>
</connector>
```

The `from` and `to` properties identify the objects the connector goes between (and therefore its direction).

If the object at either end is a display object, e.g., an alias, its unique ID would appear instead of its name. This would look like this:

```
<from><alias uid="5"/></from>       <!-- Inspect -->
```

An XML comment with the entity name is not required, but helps anyone reading the file.

The connector has two options:

- Type: `type="…"` w/SMILE connector type (default: `normal`)
- Points: `<pts>` block in same format as the flow (described below)

The position of the connector is the center of its starting point, i.e., where it attaches to the `from` object. SMILE also allows a set of points to be used to define the connectors path from the `from` object to the `to` object:

```
<pts>
   <pt x="290" y="107"/>
   <pt x="315" y="55"/>
   <pt x="335" y="107"/>
</pts>
```

5.4.3 Aliases

An alias (aka ghost or shadow) is a second image of an object used to avoid crossed connectors or to communicate across groups or models. All aliases require a reference back to the original (sometimes called the parent). The basic alias definition is shown below with sample values.

```
<alias uid="5" x="385" y="153">
   <of>Inspect</of>
</alias>
```

As shown, the `<of>` tag is used to identify which object this is the alias of. Note that if the given object were in another model, the name would have to be fully qualified with the model name, e.g., `QA.Inspect`.

6.0 Level 3: Interface

The interface level supports widgets on the surface of the stock-flow diagram or on a separate page. It includes both the definition of those widgets and their presentation. Because widgets do not require simulation, they appear either within the display block or a separate interface block (using the `<interface>` tag) which is identical to a display block but represents a second drawing plane.

The objects defined by this layer include all of the input devices (sliders, knobs, switches, numeric inputs, graphical inputs), output devices (graphs, tables, numeric displays, status indicators, spatial maps, pie charts, 3D charts, animations), annotation devices (text blocks, graphics frame, buttons), and control devices (message posters). Much of this layer was originally developed by Jeremy Merritt at isee systems, inc.

The `<interface>` tag has two attributes not allowed in the `<display>` tag:

- Home page: `home_page="…"` with page number of the home page (default: 1)
- Simulation speed (seconds per unit time): `<simulation_delay>` with the minimum time in seconds to spend simulating each unit time (default: 0, i.e., no delay)

6.1 Shared Interface Level Properties

A number of properties are re-used by several interface objects. These include references to variables, references to display objects, and references to data.

The `<entity>` tag specifies a specific variable used, for example, to plot or control. The `<entity>` tag has four attributes:

- Name: `name="…"` with the entity's SMILE name
- Subscript: `index="…"` with the comma-separated indices for the array element being plotted (default: none, i.e., the entity is not an array element)
- Value (input devices only): `value="…"` with the setting on the input device (default: constant value of entity being controlled; no default if entity is not constant)
- Equation on (input devices that control non-constant entities only): `eqn_on="…"` with true/false (default: `false`)

The `<data>` tag is used whenever a specific data set must be referred to (rather than defaulting to the data from the latest run). It has one attribute:

- Name: `name="…"` with the data set's name

6.2 Stacked Containers

Tabular and graphical devices, in particular, can be collected into pages within a stacked container. A stacked container has a position, a size, a unique ID (`uid` attribute), and the idea of a current page (the page that is visible to the user – `visible_index` attribute). Devices that are placed within stacked containers (these include graphs, tables, and list inputs) do not have positions, sizes, or unique IDs of their own – they inherit these from the stacked container they are within. For example:

```
<stacked_container x="21" y="527" height="270" width="475"
     visible_index="0" uid="5">
   <graph>...</graph>
   <graph>...</graph>
</stacked_container>
```

Note that it is not possible to mix different types within a stacked container, i.e., all entries must be of the same type, e.g., they must all be graphs.

6.3 Output Objects

6.3.1 Graphs

Graphs are defined within the `<graph>` tag, which has the following attributes and properties, beyond the applicable display attributes, e.g., `background`:

- Graph type: `type="…"` with `time_series`, `scatter`, or `bar` (default: `time_series`)
- Title: `<title>` with graph title (default: none)
- Documentation: `<doc>` with block of text, optionally in HTML format (default: none)
- Data locked: `locked="…"` with true/false (default: `false`)
- Show grid: `show_grid="…"` with true/false (default: `true`)
- Display numbers: `plot_numbers="…"` with true/false (default: `false`)
- Comparative plots: `comparative="…"` with true/false (default: `false`)
- Hide details (e.g., the date/time and other detracting elements when publishing): `hide_detail="…"` with true/false (default: `false`)
- Time precision: `time_precision="…"` with value indicating precision of least significant digit, e.g., 0.01 to round to the hundredths place (default: 0.01)
- Date-time stamp: `date_time="…"` encoded using a restricted ISO-8601 format, either “Complete date plus hours and minutes” or “Complete date plus hours, minutes and seconds” as defined by the w3c (http://www.w3.org/TR/NOTE-datetime) (default: none)

Time-series graphs can optionally include the following additional attributes:

- Starting display time: `from="…"` with the start time (default: STARTTIME)
- Ending display time: `to="…"` with the ending time (default: STOPTIME)

Each graph includes a number of plotted variables. Within the `<graph>` tag, one `<plot>` tag appears for each variable plotted. Besides standard display tags with their defaults (e.g., `color`) and SMILE object display tags (e.g., `precision`, which overrides the variable's specified precision), each `<plot>` tag has the following attributes:

- Plot index: `index="…"` with sequential index number starting at zero for the first plot; this is important for both the legend and plotting numbers on curves (the plotted number is one greater than the index); for scatter plots, index 0 is the *x*-axis variable, index 1 is the *y*-axis variable, and (if a 3D scatter plot) index 2 is the *z*-axis variable
- Line thickness: `pen-width="…"` with the line thickness in pixels (default: 1)
- Line style: `pen-style="…"` with `solid`, `dotted`, `dashed`, or `dot-dashed`, or an SVG dashed line specifier (i.e., stroke-dasharray) (default: `solid`)
- Display y-axis scale: `show_y_axis="…"` with true/false (default: `true`)

Within the `<plot>` tag, the `<entity>` tag defines which entity to plot, the `<scale>` tag defines custom scaling for the entity, and the `<data>` tag defines a specific data set used for the plot.

The default scale for entities with a globally defined scale is that global scale. When no global scale is defined for an entity, the default scale automatically fills the range of values for the entity (for comparative graphs, this is across all runs of that entity in the graph). The `<scale>` tag allows the user to override these settings.

By default, graphs plot an entity's data from the current run. However, this can be overridden by the user, either explicitly (by selecting a specific data set for a given entity) or implicitly (by locking a graph or using a comparative graph). In these cases, the data set must also be named using the `<data>` tag.

Here is a sample time-series graph that plots the variable *altitude*'s current behavior against its base case behavior:

```
<graph title="Altitude Benchmark" plot_numbers="true">
   <plot index="0" color="blue">
      <entity name="altitude">
      <data name="base case">
   </plot>
   <plot index="1" color="red" pen-style="dashed">
      <entity name="altitude">
   </plot>
</graph>
```

6.3.2 Tables

Tables are defined within the `<table>` tag, which has the following attributes and properties, beyond the applicable display attributes, e.g., `background` and `font-size` (and their defaults):

- Title: `<title>` with table title (default: none)
- Documentation: `<doc>` with block of text, optionally in HTML format (default: none)
- Data orientation: `orientation="…"` with either `horizontal` or `vertical` (default: `vertical`)
- Default column width: `column_width="…"` with width in model coordinates
- Report interval: `interval="…"` specifying how often, in model time, to report values during the simulation; use `DT` to export every DT (default: 1)
- Report ending/beginning balances: `report_balances="…"` with either `beginning` or `ending` (default: `beginning`)
- Sum flows across report interval: `report_flows="…"` with either `instantaneous` or `summed` (default: `instantaneous`)
- Data locked: `locked="…"` with true/false (default: `false`)
- Comparative plots: `comparative="…"` with true/false (default: `false`)
- Hide details (e.g., the date/time and other detracting elements when publishing): `hide_detail="…"` with true/false (default: `false`)
- Time precision: `time_precision="…"` with value indicating precision of least significant digit, e.g., 0.01 to round to the hundredths place (default: 0.01)
- Date-time stamp: `date_time="…"` encoded using a restricted ISO-8601 format, either "Complete date plus hours and minutes" or "Complete date plus hours, minutes and seconds" as defined by the w3c (http://www.w3.org/TR/NOTE-datetime) (default: none)
- Wrap text in columns: `wrap_text="…"` with true/false (default: `false`)
- Header style: The same-named display style attributes and defaults prefixed with "`header-`", specifically `header-font-family`, `header-font-size`, `header-font-style`, `header-font-weight`, `header-font-color`, `header-text-align`, `header-text-decoration`, `header-border-style`, `header-border-width`, `header-border-color`, `header-padding`, and `header-margin`.

Each table includes a number of variables. Within the `<table>` tag, one `<item>` tag appears for each variable displayed. Besides the standard display properties with their defaults (e.g., `color` and `font-size`) and SMILE object display tags (e.g., `precision`, which overrides the variable's specified precision), each `<item>` tag has the following attributes:

- Item index: `index="…"` with sequential index number starting at zero for the first variable; this is the column number (vertical orientation) or row number (horizontal orientation) after the time row/column (time is always first)
- Item type: `type="…"` with either `time`, `variable`, or `blank` (blank line) (default: `variable`). Note that the `time` item only needs to be specified to override the default item settings (e.g., the column width) for the time column/row.

Note that, unlike other display objects, the default `text-align` for table variables is `right`, rather than `left`.

Within the `<item>` tag, the `<entity>` tag defines which entity to plot and the `<data>` tag defines a specific data set to be displayed.

Just as with the graph, by default tables plot an entity's data from the current run. However, this can be overridden by the user, either explicitly (by selecting a specific data set for a given entity) or implicitly (by locking a table or using a comparative table). In these cases, the data set must also be named using the `<data>` tag.

Here is a sample table that outputs the variable *altitude*'s current behavior against its base case behavior using beginning balances:

```
<table title="Altitude Benchmark" column_width="63">
   <item type="time" width="100">
   <item index="0">
      <entity name="altitude">
      <data name="base case">
   </item>
   <item type="blank" index="1">
   <item index="2">
      <entity name="altitude">
   </item>
</table>
```

6.3.3 Numeric Displays

The current (instantaneous) value of a single variable is achieved with a numeric display, which is defined with the `<numeric_display>` tag. Other than the standard display properties with their defaults (e.g., `color` and `font-size`) and SMILE object display tags (e.g., `precision`, which overrides the variable's specified precision), each `<numeric_display>` tag has the following attributes:

- Show name: `show_name="…"` with true/false (default: `true`)
- Retain ending value (at end of simulation, rather than show nothing): `retain_ending_value="…"` with true/false (default: `true`)

Each numeric display contains a reference to zero (if unassigned) or one (if assigned) variable(s). The variable is described using the `<entity>` tag.

6.3.4 Lamps and Gauges

Lamps and gauges also indicate the current value of a single variable. A lamp is defined with the `<lamp>` tag and looks similar to an LED or LCD indicator. A gauge is defined with the `<gauge>` tag and looks similar to a speedometer. Other than the standard display properties with their defaults (e.g., `color` and `font-size`) and SMILE object display tags (e.g., `precision`, which overrides the variable's specified precision), each lamp and gauge has the following attributes:

- Show name: `show_name="…"` with true/false (default: `false` for lamps, `true` for gauges)
- Show number (gauges only): `show_number="…"` with true/false (default: `true`)
- Preset size (lamps only): `size="…"` with `small`, `medium`, or `large` (default: `large`)
- Retain ending value (at end of simulation, rather than show nothing): `retain_ending_value="…"` with true/false (default: `true`)
- Flash when in the panic region: `flash_on_panic="…"` with true/false (default: `true`)

Lamps and gauges include a list of zones within a `<zones>` tag, defining non-overlapping value ranges for the variable. Each zone is defined by a `<zone>` with the following attributes:

- Type: `type="…"` with `normal`, `caution`, or `panic` (default: `normal`)
- Color when variable is in range: `color="…"` w/valid CSS color (default: `black`)
- Start of range: `min="…"` with starting value of range
- End of range: `max="…"` with ending value of range
- Sound to play while in range: `sound="…"` with URL-format pathname using the file:// protocol by default (default: no sound)

6.4 Input Objects

6.4.1 Sliders and Knobs

Sliders and knobs are used to change the value of a variable in the model from the interface. Stocks can only be manipulated by knobs; in this case knobs can only change the stock's initial value, i.e., knobs attached to stocks cannot be changed in the middle of a simulation run. Sliders are defined with the `<slider>` tag and knobs are defined with the `<knob>` tag; they are otherwise the same.

Besides the standard display properties with their defaults (e.g., `color` and `label_side`) and SMILE object display properties (e.g., `precision`, which overrides the variable's specified precision), each `<slider>` or `<knob>` tag has the following properties:

- Show name: `show_name="…"` with true/false (default: `true`)
- Show number: `show_number="…"` with true/false; when the number is visible it can be directly edited (default: `true`)
- Show input range: `show_min_max="…"` with true/false (default: `true`)
- Input range: `min="…"` and `max="…"`, overriding entity's input range setting (default: entity's setting)
- Optional reset (slider only): `<reset_to>` with the value to reset the entity to; it has one attribute that define when to reset the entity's value: `after="…"` with either `one_time_unit` or `one_dt`.

The `<entity>` tag is used to define the entity being controlled by the slider or the knob. A sample slider:

```
<slider x="153" y="215" width="155" height="43" min="0"
     max="10" precision="0.1">
   <entity name="interest_rate">
      <value>5.5</value>
   </entity>
   <reset_to after="one_time_unit">3.5</reset_to>
</slider>
```

6.4.2 Switches and Option Groups

A switch forces a variable to be evaluated as zero (when the switch is off) or one (when the switch is on). A switch is defined using the `<switch>` tag, which, like the slider, defines which entity to control with the `<entity>` tag and can optionally be reset (to either 0 or 1, but usually 0) using the `<reset_to>` tag. In addition, it defines these attributes:

- Show name: `show_name="…"` with true/false (default: `true`)
- Switch appearance: `switch_style="…"` with either `toggle` or `push_button` (default: `toggle`)
- Make a sound when clicked: `clicking_sound="…"` with true/false (default: `false`)

Switches can also selectively control the running of groups or modules when the model's simulation specification `run_by` setting is set to `group` or `module`. In these cases, the `<entity>` tag is replaced with either a `<group>` tag or a `<module>` tag, which are otherwise identical to the `<entity>` tag and always have both a `name` and a `value`. In these cases, groups or modules without switches run if their `run` property is set to `true` while those with switches only run if the switch is on (set to one).

A group of mutually-exclusive switches (sometimes known as radio buttons or option buttons) is defined using the `<options>` tag with one `<entity>` tag for each entity in the option group. Option groups cannot be automatically reset after an interval (i.e., the `<reset_to>` tag is not allowed). Option groups have these additional attributes:

- Layout: `layout="…"` with vertical/horizontal/grid (default: `vertical`)
- Spacing: `horizontal_spacing="…"` and `vertical_spacing="…"` with CSS spacing amount in pixels (px) by default (default: 2)

A sample options group that selects between three policy options stored in an array:

```
<options x="827" y="146" width="43" height="25">
   <entity name="policy" index="1">
      <value>1</value>
   </entity>
   <entity name="policy" index="2">
      <value>0</value>
   </entity>
   <entity name="policy" index="3">
      <value>0</value>
   </entity>
</options>
```

6.4.3 Numeric Inputs and List Inputs

Numeric inputs allow values to be entered numerically (versus, for example, using a slider). Numeric inputs are defined using the `<numeric_input>` tag, which shares all of the same attributes and properties as the slider (except `show_number`, which is obviously always true for a numeric input, and `show_min_max`, which has no relevance).

List inputs provide a way to enter tabular data for a related group of variables. They are typically arranged in pages within a stacked container, but do not have to be. A list input is defined using the `<list_input>` tag, which can use any of the display properties with their defaults and adds two new attributes:

- Name of the input page: `name="…"` with page name (default: none)
- Numeric column width: `column_width="…"` with width in model cords; the name column width is the remaining width of the list input (default: half the width of the list input)

The entries within the list input are defined using the `<numeric_input>` tag. The following sample list input accepts values for prices of three different products:

```
<list_input x="446" y="161" width="230" height="97">
   <numeric_input min="5" max="10" precision="0.01">
      <entity name="price" index="Shirt">
         <value>7</value>
      </entity>
   </numeric_input>
   <numeric_input min="10" max="20" precision="0.01">
      <entity name="price" index="Pant">
         <value>15</value>
      </entity>
   </numeric_input>
```

```
      <numeric_input min="20" max="60" precision="0.01">
         <entity name="price" index="Shoe">
            <value>40</value>
         </entity>
      </numeric_input>
   </list_input>
```

6.4.4 Graphical Inputs

Graphical inputs allow graphical functions to be changed from the interface. They are defined using the `<graphical_input>` tag, which shares all the display properties and their defaults. The `<entity>` tag defines which graphical function entity is controlled. If the graphical input contains a different graphical function than the controlled variable, i.e., the user has edited the graphical input but not restored it, the `<entity>` tag will also contain a `<gf>` tag describing the edited graphical function (otherwise, no `<gf>` tag appears). The following graphical input has not been edited by the user so does not affect the controlled entity:

```
   <graphical_input x="311" y="341" width="100" height="103">
      <entity name="effect of overtime on productivity"/>
   </graphical_input>
```

However, this one has been edited by the user and overrides the graphical function in the controlled entity:

```
   <graphical_input x="311" y="341" width="100" height="103">
      <entity name="effect of overtime on productivity">
         <gf>
            <xscale min="0" max="1"/>
            <ypts>1,0.9,0.5,0.1,0</ypts>
         </gf>
      </entity>
   </graphical_input>
```

6.5 Message Poster Events

Within the Interface level (Level 3), poster events, defined with the `<event>` tag, can include visual and aural components. These are defined within the `<event>` tag using the following tags:

- Text message: `<text_box>` with text (described under “Text Boxes” below)
- Image message: `<image>` with URL-format pathname (default protocol is file://) or picture data in Data URI format, using base64 encoding
- Video message: `<video>` with URL-format pathname (default protocol is file://)
- Sound attached to message: `<sound>` with URL-format pathname (default protocol is file://)
- Hyperlink to place to navigate to: `<link>`, as described below

Note that only one visual type message (text, image, video) is allowed per event. Both `<image>` and `<video>` have one optional attribute:

- Resize to completely fill the parent container: `size_to_parent="…"` with true/false (default: `false`)

The `<link>` tag has several attributes:

- Hyperlink target: `target="…"` as described below
- Top left position (when `target` is model or interface): `x="…"` and `y="…"`
- Zoom level (when `target` is model or interface): `zoom="…"` w/ percentage zoom (default: 100.0)
- Optional visual effect (when target is not `URL` or `file`): `effect="…"` with `none` or one of the several predefined visual effects (default: `none`)
  Predefined effect names (all directional names have an implied "from" in front of them, e.g., `wipe_left` means wipe from the left side [to the right]): `dissolve`, `checkerboard`, `bars`, `wipe_left`, `wipe_right`, `wipe_top`, `wipe_bottom`, `wipe_clockwise`, `wipe_counterclockwise`, `iris_in`, `iris_out`, `doors_close`, `doors_open`, `venetian_left`, `venetian_right`, `venetian_top`, `venetian_bottom`, `push_bottom`, `push_top`, `push_left`, `push_right`
- Effect to black and back (only when an effect is given): `to_black="…"` with true/false; it uses the given effect to a black screen and then uses its opposite, if there is one, back out again to show the new location (default: false)

There are several possible hyperlink targets:

- Model diagram: `model` (`x`, `y`, and `zoom` define where)
- Interface diagram: `interface` (`x`, `y`, and `zoom` define where)
- Next page: `next_page`
- Previous page: `previous_page`
- Home page: `home_page` (as defined in `interface` tag)
- Last page navigated from: `back`
- File or web URL: `URL` (`<link>` tag contains target URL; protocol is http:// if not specified)
- File path: `file` (`<link>` tag contains target file's native path) – deprecated, use `URL` with a leading file://.

The following example shows a poster that puts up a text message when 10 is exceeded, but then, when 19 is exceeded, plays a sound and a movie while navigating to the interface:

```
   <event_poster min="1" max="20">
      <threshold value="10">
         <event>
            <text_box>Careful!</text_box>
         </event>
      </threshold>
      <threshold value="19">
         <event>
            <sound>bicycle_bell.wav</sound>
            <link target="interface" x="0" y="0"/>
            <video>c:/Movies/uh-oh.mov</video>
         </event>
      </threshold>
   </event_poster>
```

6.6 Freestanding Windows (Popups)

Freestanding windows, or windows that appear as a result of a user action are defined with a `<popup>` tag. These are used, for example, for graph and table windows, as well as for information windows that may appear when a button is pressed. When used for graphs and tables, the `<popup>` tag appears at the same XML level as the major `<display>` and `<interface>` tags within the model. Otherwise, it appears within the object that contains it (e.g., within a button).

The `<popup>` tag uses the same attributes as the other display objects, but its coordinates and size are in *screen* coordinates rather than model coordinates. When embedded in another object, such as the button, its position is relative to the top-left corner of that object (in screen coordinates). If it is off screen, it should, of course, be forced on screen when it appears. The `<popup>` tag optionally contains one of the following:

- A `<text_box>` tag to display text (described under "Text Boxes" below)
- An `<image>` tag to display an image
- A `<video>` tag to display a movie

When a `<popup>` refers to an existing object, such as a graph or a table, the following attributes are used to describe which object and the visible state of the pop-up:

- Contents: `content="…"` with the unique ID of object to display in window
- Visible: `visible="…"` with true/false (default: `false`)

Freestanding windows, such as floating graphs and tables have absolute screen positions, i.e., they are not relative to any model element.

Here is button that pops up a window containing an image:

```
<button x="972" y="399" width="65" height="34" label="Help">
   <popup x="-151" y="-343" width="612" height="612"
         background="yellow">  <!-- relative screen coords -->
      <image>help.jpg</image>
   </popup>
</button>
```

6.7 Model and Interface Annotations

6.7.1 Text Boxes

The model can be annotated with text boxes. These are represented using the `<text_box>` tag, which contains the block of text, optionally in HTML format, e.g.,

```
<text_box color="red" uid="10">Danger!</text_box>
```

Text boxes, other than the standard display properties with their defaults (e.g., `color` and `font-size`), add two attributes:

- Transparency: `appearance="…"` with either `opaque` or `transparent` (default: `opaque`)
- Include vertical scrollbar: `vertical_scrollbar="…"` with `true`, `false`, or `auto`, where `auto` means it only appears if the text exceeds the vertical bounds of the text box (default: `false`)

6.7.2 Graphics Frames

The model can also be annotated with graphics. These are represented using the `<graphics_frame>` tag. Other than the standard display properties with their defaults (e.g., `border-style`), graphics frames have these attributes:

- Interior fill: `fill="…"` with `none`, `light`, `dark`, or `solid` (default: `none`); `light` is a light shade of the background color, `dark` is a dark shade of the background color, and `solid` is the background color unchanged.

The contents of a graphics frame are defined with either zero or one instance(s) of the `<image>` or `<video>` tag.

6.7.3 Buttons

Buttons are used to control the program, navigate to a hyperlinked location, or to display an information window (containing either text, an image, or a movie). All buttons appear within the `<button>` tag, which includes the standard display properties (e.g., `color`), as well as the following properties:

- Transparency: `appearance="…"` with either `opaque` or `transparent` (default: `opaque`)
- Button corner style: `style="…"` with `square`, `rounded`, or `capsule` (default: `square`)
- Label (optional): `label="…"` with text of label to appear on button; when there is a label, the default text alignment is center (rather than left)
- Image on button (overrides label): `<image>` with URL-format pathname (default protocol is file://) or picture data in Data URI format, using base64 encoding
- Make a sound when clicked: `clicking_sound="…"` with true/false (default: `true`)
- Sound played after click (optional – played after clicking sound, if any): `<sound>` with URL-format pathname (default protocol is file://)

Buttons that navigate include a `<link>` tag, as described above under "Message Poster Events."

Buttons that control the program contain a `<menu_action>` tag, which has the menu name as its data. Supported menu actions are broken into various categories:

- File: `open`, `close`, `save`, `save_as`, `save_as_image`, `revert`
- Printing: `print_setup`, `print`, `print_screen`
- Simulation: `run`, `pause`, `resume`, `stop`, `run_restore` (run *only if* restored)
- Restore objects: `restore_all`, `restore_sliders`, `restore_knobs`, `restore_list_inputs`, `restore_graphical_inputs`, `restore_switches`, `restore_numeric_displays`, `restore_graphs_tables`, `restore_lamps_gauges`
- Data: `data_manager`, `save_data_now`, `import_now`, `export_now`
- Miscellaneous: `exit`, `find`, `run_specs`

In the case of `import_now` and `export_now`, the `<menu_action>` tag has the following attributes:

- Source (import) or destination (export) location: `url="…"` w/URL path
- For Excel only, worksheet name: `worksheet="…"` with worksheet name
- Alternatively, if all on-demand links are to be imported or exported: `all="true"` (as a consequence, when `all` does not appear, it is false)

In the case of `save_data_now`, the `<menu_action>` tag has the following attributes:

- Name of saved run: `run_name="…"` with name of saved run

Buttons that display an information window use a `<popup>` tag, described above under "Freestanding Windows."

Note that buttons that navigate (i.e., use a `<link>` tag) can combine other actions with that navigation. In particular, they can include both a `<menu_action>` and a `<popup>`. They can also behave like a switch if they include a `<switch_action>` tag. The entity being switched on or off is defined within the `<switch_action>` tag in the same way as within a `<switch>` tag, i.e., using an `<entity>` tag, or the identically formatted `<group>` or `<module>` tag. The following button combines several properties to place an image on its face, redefine the clicking sound to a bicycle bell, and, when clicked, navigate to the next page with a visual effect while restoring graphs and tables and turning the bicycle policy group on:

```
<button x="748" y="226" width="294" height="252"
      clicking_sound="false">
   <image size_to_parent="true" width="960" height="720">
      C:/Images/button_ok.jpg
   </image>
   <sound>bicycle_bell.wav</sound>
   <link target="next_page" to_black="true" effect="iris_in"/>
   <menu_action>restore_graphs_tables</menu_action>
   <switch_action>
      <group name="Bicycle Policy">
         <value>1</value>
      </group>
   </switch_action>
</button>
```