SMILE and XMILE: A Common Language and Interchange Format for System Dynamics

Karim Chichakly
isee systems, inc.
31 Old Etna Road, Suite 9N
Lebanon, NH 03766
(603) 448-4990/FAX: (603) 448-4992
kchichakly@iseesys.com

Abstract

Proposed four years ago, SMILE has made minimal progress toward a full-fledged specification that System Dynamics software vendors can use to interchange models. This paper attempts to move the development of the standard one step further with a draft specification of both the language that needs to be interchanged (SMILE) and an XML format for that language (XMILE). Central to the success of this standard is the idea of three increasing levels of compliance, only the lowest level (interchange of equations) being required of all vendors.

Keywords: Model Interchange, SMILE, XMILE, XML, System Dynamics Language

Background

In the Spring 2003 System Dynamics Society newsletter, Jim Hines proposed that there be a common interchange format for system dynamics models. Magne Myrtveit originally proposed such an idea at the 1995 ISDC, but Jim hoped to revive interest in the idea and chose the name SMILE (Simulation Model Interchange LanguagE) to keep people lighthearted. The benefits Jim proposed at the time were:

- Sharing of models can lead to greater increases of knowledge and sharing of ideas.
- On-line repositories could be built to facilitate learning.
- Open standards lead to better acceptance in larger corporations as it minimizes their risk with specific vendors.
- It spurs innovation by allowing non-vendors to develop add-ons.

To this formidable list, I would add:

- It allows the creation of a historical record of important works that everyone has access to.
- It allows vendors to expand their market base because suddenly their unique features (and let's be honest – each of the three major players has unique competencies) are available to all system dynamics modelers.

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 Science SIG at the 2006 ISDC[1], I suggested breaking the problem into two pieces: the language we intend to interchange and the interchange format. The first section of this document, on SMILE (Systems ModelIng LanguagE), proposes to begin the process of documenting the base set of functionality that we want from a system dynamics modeling language. The second section on XMILE addresses the XML-based interchange format. As standards, these are two separate documents.

SMILE Specification

1.0 Basic Functionality

It is safe to say that the minimal useful language subset would include most of the capabilities of DYNAMO. After all, it was the first system dynamics modeling language and many of the DYNAMO models that have been written represent some of the seminal work in the field. Given this premise, we can begin with the basic building blocks available in DYNAMO: stocks ("levels" in DYNAMO), flows ("rates" in DYNAMO), auxiliaries, and table functions.

1.1 Stocks

Stocks represent things that accumulate in the system. Their value must be set at the start of the simulation with an initial value. The initial value can be either a constant or an expression. In the case of an expression, the value is evaluated only once, at the beginning of the simulation, to initialize the stock.

During the course of the simulation, the value of a stock can only be changed by its inflows and outflows. In general, a stock is evaluated by adding the sum of its inflows minus the sum of its outflows, all times DT, and adding that to the value of the stock during the previous DT. [2]

A sample DYNAMO stock specification appears below.

```
L POP.K = POP.J + DT*(BIRTHS.JK – DEATHS.JK)
N POP = 100
```

The L line defines the stock equation in terms of its current value (.K), its previous value (.J), and the previous flow values (.JK). The N line defines the stocks initial value.

[1] As an aside, Len Malczynski presented a paper at the 2006 conference that explained why the software vendors may never adopt such a standard. This paper is the start of an effort to prove him wrong.

[2] DT stands for "delta time." Since these are time-based simulations, DT is the increment of time being used to advance through the model. It needs to be small enough to achieve accurate calculation results.

Stocks in SMILE are unconstrained other than by their inflows or outflows. Vendor-specific features such as Non-negativity, which prevents a stock from taking on negative values, are not (directly) supported. Likewise, stocks can only by modified by their inflows and outflows. Therefore, the equation can always be inferred from the list of flows and is never explicitly written (as it is in DYNAMO). The form of all stock equations (using DYNAMO syntax) is:

```
L S.K = S.J + DT*(<sum of inflows> – <sum of outflows>)
```

No other stock formulation is supported in SMILE.

1.2 Flows

Flows represent rates of change of the stocks. They can be defined using any algebraic equation (including a constant value) or by using a table function.

During the course of a simulation, a flow's value is evaluated each DT based on the current state of the system. A sample DYNAMO flow specification appears below.

```
R BIRTHS.KL = BR*POP.K
```

This equation defines the current value of births in terms of the birth rate (BR) and the current population.

Flows in SMILE are unconstrained other than by their equation. Vendor-specific features such as Uniflows, which prevent a flow from taking on negative values, are not supported.

1.3 Auxiliaries

Auxiliaries allow the isolation of any algebraic function that is used. They can both clarify a model and factor out important or repeated calculations. They are defined using any algebraic expression (including a constant value[3]) or a table function.

During the course of a simulation, an auxiliary's value is evaluated each DT based on the current state of the system. A sample DYNAMO auxiliary specification appears below.

```
A BR.K = 0.1*FAM.K
```

This equation defines the birth rate in terms of the base birth rate (0.1) and a food availability multiplier.

[3] DYNAMO explicitly separated the definition of a constant from that of an algebraic expression. The distinction was necessary as auxiliaries required the .K suffix, while constants did not.

1.4 Graphical Functions

Graphical functions are alternately called lookup functions and table functions. They represent a functional mapping between two variables. The domain is consistently referred to as *x* and the range is consistently referred to as *y*.

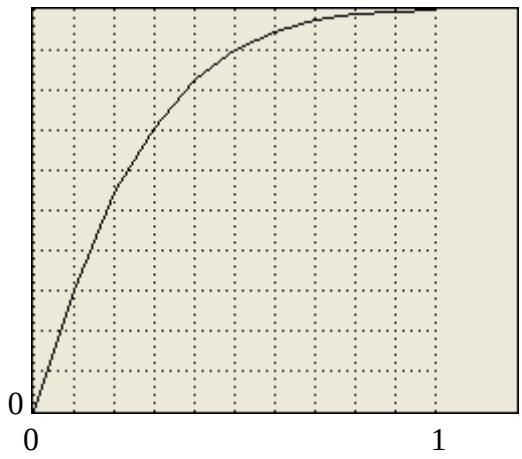
Since every modern program displays table functions as graphs, and most allow the users to edit the functions by drawing the graph, the terminology used here is "graphical functions."

Although some vendors support arbitrary sets of (*x*, *y*)-pairs, DYNAMO only supported a fixed increment for *x*. This is what this standard supports.

A sample DYNAMO table function appears below.

```
A FAM.K = TABLE(FAMT, FOOD.K, 0, 1, .1)
T FAMT = 0/.3/.55/.7/.83/.9/.95/.98/.99/.995/1
```

The table lookup function defines the *y*-values (FAMT), the input (or *x*) variable (FOOD.K), the bounds on *x* (0 to 1) and the *x*-increment (0.1). Note that the supported functionality requires a fixed *x*-increment. XMILE has a representation for arbitrary (*x*, *y*)-pairs, but they are not part of the language. The graph of this function appears below.



The above is a continuous graphical function. Intermediate values are calculated with linear interpolation between the intermediate points. SMILE also supports discrete graphical functions, which use the value associated with the next lower *x*-coordinate for intermediate values (also called a step-wise function). The last two points of a discrete graphical function must have the same *y*-coordinate.

1.5 Groups

Groups (aka sectors) were not supported by DYNAMO. However, they are an important feature for building large models. Groups are useful for collecting related model entities in one place. Some programs allow these separate pieces to be simulated separately.

Every group has a unique name and documentation. It may have other information related to its display, but that is not part of SMILE.

2.0 General Conventions

All statements and constants follow US English conventions. So built-in functions are in English, operators are based on the Roman character set, and constants have US English delimiters (that is, a period is used for a decimal point).

Variable names, comments, and embedded text may be localized.

2.1 Constants

As mentioned, constants follow US English conventions. All constants are floating point numbers in decimal. They can begin with either a digit or a decimal point. There can be any number of digits before or after the decimal point, but must contain at least one digit. A decimal point is not required. The number can be optional followed by an "E" (or "e") and a signed integer constant. The "E" is used as shorthand for scientific notation and represents "times ten to the power of".

Although the number of digits is not explicitly restricted, only so many digits of precision are retained by each vendor's program. However, numbers in all programs should have at least the precision of IEEE single-precision floating point numbers. The same applies for the exponent.

Constants, like variables, can be modified by operators, including unary plus and unary minus. Thus, it is possible to enter negative constants also.

In BNF,

constant ::= { [*digit*]+[**.**[*digit*]*] | [*digit*]***.**[*digit*]+ }[{**E** | **e**} [{**+** | **–**}] [*digit*]*
digit ::= { **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** }

Note the leading sign { **+** | **–** } is not explicitly included as expressions handle this.

Sample constants: 0 –1 .375 14. 6e5 8.123e-10

2.2 Identifiers

Identifiers are used throughout a model to identify various things. The most obvious names are given to variables. However, names are also given to units, subscripts, groups (or sectors), and models (described below). Most of these identifiers will appear in equations (the group name is perhaps the only exception), and as such need to follow certain rules to allow for well-formed expressions.

Due to differences in different vendors' products, it is a challenge to a set of rules that works for everyone. The approach taken here is to define a basic set of rules and then allow escape conventions to handle other cases.

As a basic premise, it is fair to say that identifiers should not be defined such that they could make expressions ambiguous. By this rule, several characters that are used as operator expressions and delimiters can be immediately ruled out. These include:

. , < > ^ * / + – = ( ) [ ] { }

Since we propose to use braces { } for in-line comments, these also need to be ruled out.

We also need to rule out starting characters that may cause confusion. For example, we cannot start an identifier with a digit as that is how constants begin. Some languages restrict what an identifier can start with, rather than ruling things out. Java, for example, states that an identifier can only begin with a letter, an underscore, or a dollar sign. This is, unfortunately, over-restrictive for our use, as we want user to be able to localize their names. In this case, we may want to focus on any additional unary operators that have not yet been covered, specifically !, #, and &.

Finally, white space is used by parsers to break up tokens. Thus, spaces cannot appear in identifiers. The standard approach is to replace spaces with underscores internally, but then change them back to spaces for display outside of equations.

There is an unresolved issue around the use of return ('\r', not '\n') in variable names. Some products allow it and others do not. It is perhaps safest to allow it with the understanding that the standard mapping will map it to an underscore.

For products that allow these characters in identifiers, there needs to be a way of delimiting the additional characters. In VenSim, for example, spaces and other characters are allowed as long as the entire identifier is in quotes (thus identifiers in VenSim cannot have quotes in them). This approach is also taken in scripting shells. In most programming languages, there is an escape character that causes the next character to be taken out literally (rather than interpreted). In all of the C-derived languages, this character is backslash. It is proposed here that both methods be supported as ways of delimiting illegal characters. Double-quotes will be used to delimit entire "identifiers" while the backslash \ will be used to delimit single characters. This necessarily means that double-quotes in variable names always need to be delimited \" (whether in quote or not) and that backslash needs to be delimited \\ when not in quotes.

For products that cannot support other characters in identifiers, there must also be mappings to characters they allow. For example, return would generally be mapped to underscore. The exact definition of these mappings is left for XMILE (but they will likely be defined using an XSLT provided by the specific vendor).

Note that identifiers are *not* case-sensitive and that all built-in function names, operator names, and statement keywords are reserved.

Sample identifiers: Cash_Balance draining wom_multiplier

3.0 Expressions

Equations are defined using expressions. The simplest expression is just a constant.

Expressions are infix (e.g., algebraic), following the general rules of algebraic precedence (parenthesis, exponents, multiplication and division, and addition and subtraction). Unfortunately, our set of operators is much richer than basic algebra, so we have to account for functions, unary operators, and relational operators. In general, the rules or precedence and associativity (the order of computation when operators have the same precedence) follow the established rules of the C-derived languages.

3.1 Operators

The following table lists the supported operators in precedence order. All but the unary operators have left-to-right associativity (right-to-left is the only thing that makes sense for unary operators).

[ ]	Subscripts
( )	Parentheses
^	Exponentiation
+ – NOT	Unary operators positive, negative, and logical not
* / %	Multiplication, division, modulo [should this be included?]
+ –	Addition, subtraction
< <= > >=	Relational operators
= !=	Equality operators (in mathematics, != is relational)
AND	Logical and
OR	Logical or

Note the logical, relational, and equality operators are all defined to return zero (0) if the result is false and one (1) if the result is true.

Sample expressions: a*b (x < 5) AND (y >= 3) (–3)^x

3.2 Structured Statements

One control structure statement is supported:

if *condition* **then** *expression* **else** *expression*

where *condition* is an expression that evaluates to true or false (we follow the convention of C that all non-zero values are true, while zero is false). Generally, this is an expression involving the logical, relational, and equality operators.

Note that many vendors implement this in the DYNAMO (and original FORTRAN) fashion, i.e., as a built-in function:

if_then_else(*condition*, *then-expression*, *else-expression*)

This makes the language a little easier to parse and is a supported alternative (while the former is generally considered easier to comprehend). Note DYNAMO implemented this functionality with the `CLIP` function.

3.3 In-line Comments

Comments are provided to include explanatory text that is ignored by the computer. Comment are delimited by braces { } and can be included anywhere within an expression. This functionality allows the modeler to temporarily turn off parts of an equation or to comment the separate parts of a complex formulation.

Sample comments: a*b { take product of a and b } + c { and add c }

3.4 Documentation

Each variable has its own documentation, which is a block of text unrelated to the equation. Each vendor has their own way to enter documentation, some within expressions. XMILE define how documentation is stored.

3.5 Units

Each variable has its own set of units. Each vendor has their own way to enter units, some within expressions. XMILE defines how units are stored.

Each model has a defined unit of time. The predefined units are:

ns	nanoseconds
us	microseconds
ms	milliseconds
s	seconds
min	minutes
hr	hours
day	days
wk	weeks
mo	months
qtr	quarters
yr	years
time	unspecified time units

In general, units should be abbreviated to their accepted forms, e.g., “mi” for miles, “lb” for pounds, “km” for kilometers, “kg” for kilograms, etc. Unit specifications are expressions in their own right. They are, however, restricted to the operations ^, *, and /.

4.0 Built-in Functions

Certain built-in functions must be relied upon across all systems. This section strives to define the minimum set of built-in functions that must be supported, along with their parameters. The mechanism for defining vendor-specific built-ins is also described.

4.1 Mathematical Functions

ABS: absolute value (magnitude) of a number
Parameters: 1: the number to take the absolute value of
Range: $[0, \infty)$
Example: abs(Balance)

ARCTAN: arctangent of a number
Parameters: 1: the number to take the arctangent of
Range: $(-\pi/2, \pi/2)$
Example: arctan(x)

COS: cosine of an angle in radians
Parameters: 1: the number to take the cosine of
Range: [-1, 1]
Example: cos(angle)

EXP: value of *e* raised to the given power
Parameters: 1: the power on *e*
Range: $(-\infty,\infty)$
Example: exp(x)

INT: next integer less than or equal to the given number
Parameters: 1: the number to find next lowest integer of
Range: $(-\infty,\infty)$; note negative fractional numbers increase in magnitude
Example: int(x)

LN: natural (base-*e*) logarithm of the given number
Parameters: 1: the number to find the natural logarithm of
Range: $[0,\infty)$; note domain is $(0,\infty)$
Example: ln(x)

LOG10: base-10 logarithm of the given number
Parameters: 1: the number to find the base-10 logarithm of
Range: $[0,\infty)$; note domain is $(0,\infty)$
Example: log10(x)

MAX: larger of two numbers
Parameters: 2: the numbers to compare
Example: max(x, y)

MIN: smaller of two numbers
Parameters: 2: the numbers to compare
Example: min(x, y)

MOD: remainder of dividing two numbers
Mathematical definition: a = INT(a/b)*b + MOD(a, b)
Parameters: 2: (*dividend*, *divisor*); *divisor* $\neq 0$
Range: [0, *divisor*) if *divisor* > 0
(-*divisor*, 0] if *divisor* < 0
Example: mod(month, 12)

PI: value of π, the ratio of a circle's circumference to its diameter
Parameters: none
Example: pi

SIN: sine of an angle in radians
Parameters: 1: the number to take the sine of
Range: [-1, 1]
Example: sin(angle)

SQRT: square root of a positive number
Parameters: 1: the number to take the square root of
Range: $[0, \infty)$; note domain is the same
Example: sqrt(x)

TAN: tangent of an angle in radians
Parameters: 1: the number to take the tangent of
undefined for odd multiples of $\pi/2$
Range: $(-\infty,\infty)$
Example: tan(angle)

4.2 Statistical Functions

EXPRND: Sample a value from an Exponential distribution
Parameters: 1 or 2: (*mean*[, *seed*]) $0 \leq$ *seed* < 32,768
If *seed* is provided, the sequence of numbers will always be identical
Example: exprnd(8) samples from an exponential distribution with mean 8

NORMAL: Sample a value from a Normal distribution
Parameters: 2 or 3: (*mean*, *standard deviation*[, *seed*]) $0 \leq$ *seed* < 32,768
If *seed* is provided, the sequence of numbers will always be identical
Example: normal(100, 5) samples from $N(100, 5)$

RANDOM: Sample a value from a uniform distribution
Parameters: 2 or 3: (*minimum*, *maximum*[, *seed*]) $0 \leq$ *seed* < 32,768
If *seed* is provided, the sequence of numbers will always be identical
Example: random(1, 100) picks a random number between 1 and 100

4.3 Delay Functions

DELAY: infinite-order material delay of the input for the requested fixed time
Parameters: 2 or 3: (*input*, *delay time*[, *initial value*])
If *initial value* is not provided, the initial value of *input* will be used
Example: delay(orders, 5)

DELAY1: first-order material delay of the input for the requested fixed time
Parameters: 2 or 3: (*input*, *delay time*[, *initial value*])
If *initial value* is not provided, the initial value of *input* will be used
Example: delay1(orders, 5)

DELAY3: third-order material delay of the input for the requested fixed time
Parameters: 2 or 3: (*input*, *delay time*[, *initial value*])
If *initial value* is not provided, the initial value of *input* will be used
Example: delay3(orders, 5)

DELAYN: Nth-order material delay of the input for the requested fixed time
Parameters: 3 or 4: (*input*, *delay time*, *n*[, *initial value*])
If *initial value* is not provided, the initial value of *input* will be used
Example: delayn(orders, 5, 10) delays orders using a 10^{th} order material delay

SMTH1: 1st-order exponential smooth of the input for the requested time
Parameters: 2 or 3: (*input*, *averaging time*[, *initial value*])
If *initial value* is not provided, the initial value of *input* will be used
Example: smth1(Quality, 5)

SMTH3: 3rd-order exponential smooth of the input for the requested time
Parameters: 2 or 3: (*input*, *averaging time*[, *initial value*])
If *initial value* is not provided, the initial value of *input* will be used
Example: smth3(Quality, 5)

SMTHN: Nth-order exponential smooth of the input for the requested time
Parameters: 3 or 4: (*input*, *averaging time*, *n*[, *initial value*])
If *initial value* is not provided, the initial value of *input* will be used
Example: smthn(Quality, 5, 10) performs a 10^{th} order smooth

4.4 Test Input Functions

PULSE: Generate a one-DT wide pulse at the given time
Parameters: 2 or 3: (*magnitude*, *first time*[, *interval*])
Without *interval*, the PULSE is generated only once
Example: pulse(20, 12, 5) generates a pulse of 20 at time 12, 17, 22, etc.

RAMP: Generate a linearly increasing value over time with the given slope
Parameters: 2: (*slope*, *start time*); begin in-/de-creasing at *start time*
Example: ramp(2, 5) generates a ramp of slope 2 beginning at time 5

STEP: Generate a step increase (or decrease) at the given time
Parameters: 2: (*height*, *start time*); step up/down at *start time*
Example: step(6, 3) steps from 0 to 6 at time 3 (and stays there)

4.5 Time Functions

DT: value of DT, the integration step
Parameters: none
Example: dt

STARTTIME: starting time of the simulation
Parameters: none
Example: starttime

STOPTIME: ending time of the simulation
Parameters: none
Example: stoptime

TIME: current time of the simulation
Parameters: none
Example: time

4.6 Miscellaneous Functions

IF_THEN_ELSE: Select one of two values based on a condition
Parameters: 3: (*condition*, *true value*, *false value*)
If *condition* is non-zero, it is true; otherwise, it is false
Example: if_then_else(x < 3, -4, 11) will be -4 is x < 3 and 11 otherwise

INIT: initial value (value at STARTTIME) of a variable
Parameters: 1: the number to get the initial value of
Example: init(Balance)

4.7 Defining Unsupported Built-ins

There must be a way for vendors to specify the operation of both their own functions and the functions of other vendor that their users wish to use. In the latter case, these can clearly map to either their own functions or to the SMILE functions (preferred when no translation is given for a vendor's function). Ideally, vendor-specific function names would be prefixed by a vendor identifier to avoid conflicting names between both different vendors and SMILE.

As a simple example, let us say that vendor A does not have a LOG10 built-in, but has a general (any base) LOG built-in. That vendor should then be able to map any LOG10(x) function to LOG(x, 10) when the file is read-in. Conversely, if the vendor wishes to use their general LOG function within SMILE, they should be able to provide a translation that maps LOG(x, y) to LN(x)/LN(y) (change of base formula – LOG10 works just as well).

The first kind of translation, from SMILE functions to the vendor's functions, could be handled either by the vendor as the file is read in, or through an XSLT translator. The macro functionality described below would also handle this (they would create a macro for a SMILE function).

The second kind of translation, mapping unsupported functions in the file to SMILE, is the main focus of this section. Every unsupported built-in that a vendor wants to appear within a SMILE file needs to be defined in a SMILE macro. The macros may appear in the same file as the model or in a separate file. It is, however, more likely that each vendor will provide their own file of macros to use with their models.

Macros can also be used to implement complicated options such as stock non-negativity and uniflows. It would, of course, be necessary to add an additional function call to the [out]flow equation to achieve this.

Macros can use all of the syntax of SMILE to achieve their result (it goes without saying that SMILE implementations must support macros). The simplest kind of macro is simply an expression using existing (including through macros) functions and operators. The change of base formula above is a good example.

More complicated macros can define stocks, flows, and auxiliaries to do their work. This would be the approach, for example, to implement a smooth function if one did not already exist.

The name of the macro is the same as the name of the function. In addition, variable numbers of arguments are not supported, but the same macro can be defined multiple times with a different number of arguments. Indeed, a macro can have the same name as a SMILE function, providing it uses a different number of arguments. Finally, the names of any variables (including parameter identifiers) defined within a macro are local to that macro alone and will not conflict with any names within either the model or other macros.

There are also various unsupported options (vendors can also add their own) for building blocks. One way to handle these would be to define built-in macros that are used to envelope an object's equation. Non-negative flows (aka uniflows), for example, could have their equations wrapped in a built-in macro that implements MAX(<flow value>, 0).

Ideally, macros would support these options without having to change the equations. For the simple cases, such as non-negative flows, the format can mimic the built-in macros. However, more complicated options require greater support. For example, non-negative stocks implement the non-negative logic in the stock's *outflows*, not in the stock itself. Furthermore, each outflow needs not only its own value, but the stock's value, and the sum of the values of every higher-priority flow (which, yes, the stock could find for it).

SMILE therefore supports options with macro *filters*. The filters run after the object's value has been computed. If several filters are needed, they would run in the order they appear in the options list. A basic filter would only affect the given object, and so is passed just the object itself. More complicated filters affect inflows or outflows of the

object and need to be invoked for those inflows or outflows and not for the object. They then need to be passed the affected flow, the given stock, and the sum of the inflows or outflows already evaluated (or perhaps both).

The exact format of macros is left to the XMILE document.

5.0 Simulation Specifications

Every SMILE model must specify the start time of the simulation, the stop time of the simulation, DT, and the units of time.

By default, the integration method is Euler's, but other methods are supported as follows:

SMILE Name	Integration Method
Euler	Euler's method
RK2	Runge-Kutta 2
RK4	Runge-Kutta 4
RK_Var	Runge-Kutta Variable Step Size (optional)
Gear	Gear algorithm (optional)

Additional integration methods can be supported optionally. In these cases, a supported fallback method should be also provided, for example, "Gear, RK4". This means that Gear should be used if the product supports it. Otherwise, use RK4.

The language also supports an optional pause interval. By default, a model runs to completion (from STARTTIME to STOPTIME). If, however, the pause interval is specified, the model will pause at all times that match STARTTIME + interval*N; N > 0. Products are free to ignore this specification if they do not support this mode of operation.

6.0 Optional Functionality

It is expected that compliant products will implement the language thus far described in its entirety. There are, however, a number of features that are left to each vendor's discretion as to whether or not to support. These are not intended to be vendor-specific features, but common features that lighter packages may either not support, or support in part. These features include conveyors, queues, arrays, and hierarchical models.

6.1 Conveyors

A conveyor conceptually works like the real thing. Objects get on at one end and some time later (the length of the conveyor), they fall off. Some things can leak out (fall off!) of a conveyor partway, so there is also a leakage flow. In addition, the conveyor has a variable speed control, so you can change the length of time something stays on it.

Note this behavior means a conveyor can have at most two outflows. If there is only one outflow, it must be the stuff coming off the end of the conveyor. If there are two outflows, the first is always the conveyor's output, while the second is the conveyor's leakage.

6.2 Queues

Queues are first-in, first-out objects that track individual batches that enter them (otherwise, they'd just be stocks). The first batch in is the first batch to leave. Queues are important when it is necessary to track batches or when there are input constraints downstream that force the queue outflow to zero (for example a capacity issue on a conveyor).

6.3 Arrays

Arrays add depth to a model in up to N dimensions. Products that support arrays offer different values of N.

Arrays are defined with dimension names. Each named dimension also has a name for each of its indices. For example, a two dimensional array of location vs. product could have a dimension called "location" with three indices "Boston", "Chicago", "LA", and another dimension called "product" with four indices "dresses", "blouses", "skirts", and "pants". If we are looking at sales, we might have a variable sales[location, product] which has elements sales[Boston, dresses], sales[Boston, blouses], etc. Note that both dimension names and subscript names are identifiers. Subscripts also appear within square brackets with each index separated by a comma. Arrays are stored in row-major order, which may only be an issue when looking at data sets.

Some programs do not allow the subscript indices (which are text) to be numbers (just as other identifiers cannot be numbers), while others do allow them to be numbers, but still treat them as textual labels. This will need to be addressed.

6.4 Hierarchy

"Hierarchical models" is really a misnomer. The features explained here could be used to set up a hierarchical system, but by no means need to be used in that way. What this section supports is the idea of independent model pieces (for lack of a better word) interacting with each other in some way (i.e., sharing model inputs and outputs). These pieces may or may not have separate simulation specifications, though this does not make a lot of sense unless the pieces are arranged hierarchically. The semantics of any differences will eventually be described here.

The most relevant issue for SMILE is how these disparate model pieces communicate with each other. It is necessary for each of these pieces to be uniquely named. It is then possible for any object to reference any other object across the entire model, by using the name of the model piece followed by a period and the name of the entity. Note this, in effect, introduces separate namespaces for each of the model pieces. For example, if I am

in a model called “finance” and I need the variable “expenditures” from the model “marketing”, I could reference it as “marketing.expenditures”.

XMILE Specification

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. It is still on the table whether a separate data layer should be included to support the display of graphs and tables when the model is opened (it is likely that we will need to support some 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 sheet and XML style sheets are. Global styles can be defined (such as “all stocks are blue”) and can then be overridden at any level. The presentation information is stored in XSL format.

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 entire XMILE file is enclosed within a <xmile> tag as follows:

```
<xmile xmlns="http://www.systemdynamics.org/XMILE">
   ...
</xmile>
```

This is followed by 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:

- XMILE version: <xmile_version> w/version number
- XMILE compliance level: <xmile_level> w/level number
- SMILE version: <smile_version> w/version number
- SMILE optional features used in model: <smile_options> (defined below)
- Vendor name: <vendor> w/company name
- Product name: <product> w/product name
- Product version: <version> w/version number
- Language code (for variable names and comments – using ISO 639-1; must use 639-2 if language missing from 639-1 – e.g., Hawaiian): <lang> w/code

Optional pieces include:

- Model name: <name> w/name
- Model caption: <caption> w/caption
- Picture of the model in JPG, GIF, TIF, or PNG format: <view> w/filename
  or <view type="jpg"> w/picture data
- 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 – user-defined):
  <contact> w/contact information [perhaps separate out e-mail]
- 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

3.1 SMILE Options

SMILE options is a list of optional functionality used in the model file. The available options are:

```
<uses_conveyor/>
<uses_queue/>
<uses_arrays>N</uses_arrays>, where N is the number of dimensions used
<uses_hierarchy/>
```

3.2 Style Information

Every XMILE file can include some 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, all stocks are non-negative.

The style information for the entire model immediately follows the header. Note that the style information can be repeated in each Model section to override specific global defaults (note it is also legitimate for it to only appear in Model sections that need it).

It is presently thought that the style information will match the CSS format and keywords (augmented). However, it is not envisioned that these will be changed externally and they are within an XML file. Therefore, it is more likely they will appear in XML using the same tags that are used in the objects themselves. This section will assume the latter in its discussion.

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 that has a particular tag can be set at the model level. For example, the following sets the font globally to 12 pt and Tahoma (or failing Tahoma, a sans serif font), the background to white, and the color of objects to blue:

```
<model>
   <font>12 pt Tahoma, sans-serif</font>
   <background>white</background>
   <color>blue</color>
</model>
```

These changes can also be applied directly to objects, e.g.,

```
<connector>
   <color>magenta</color>
</connector>
```

As already stated, these settings can also be from the Model section:

```
<stock>
   <options>
```

```
      <non_negative/>
   </options>
</stock>
<flow>
   <options>
      <non_negative/>
   </options>
</flow>
```

In this particular case, where the same option is being applied to all objects that accept it (stocks and flows are the only objects that can be non-negative), it would be better to apply that option to the model.

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 set 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 `<simspecs>`. The following properties are required:

- Step size: `<dt>` w/value
- Unit of time: `<time_units>` w/SMILE code
- Start time: `<start>` w/time
- Stop time: `<stop>` w/time (after start time)

Optional properties:

- Integration method: `<method>` w/SMILE code (default: Euler's)
- Pause interval: `<pause>` w/interval (default: infinity – can be ignored)

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

4.0 Level 1: Simulation

The simulation layer supports all of the features of the SMILE language, in the required formats. The simulation specifications block has already been described and, along with the header and style blocks, is part of this level. The meat of this layer is within the model block. Note that there can be many model blocks.

Each model block uses the XML tag `<model>`. The model block must begin with a `<name>` field. In most models, this will be empty (i.e., `<name/>`). However, if

`uses_hierarchy` is set, the name is required (in this case, the model contents can be in a separate file referenced through a URL path).

As mentioned above, the model block can include either or both of the style and simulation specs blocks, overriding only those settings that are desired. These blocks should appear just after the name block.

Finally, the model building blocks (stocks, flows, and auxiliaries) defined by SMILE are listed in any order (except those within groups must appear within that group's block).

4.1 Building Block Properties

All building blocks can have the following properties.

- Name: `<name>` w/valid identifier
- Equation: `<eqn>` w/valid expression in a CDATA section (unless constant)
- Units: `<units>` w/valid units
- Documentation: `<doc>` w/block of text
- Options block: `<options>` w/block specific options

Of these, the name and the equation are required in 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.

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

```
<gf>
   <xscale>0,0.5</xscale>     <!-- min and max -->
   <yscale>0,1</yscale>       <!-- min and max -->
   <pts>0.05,0.1,0.2,0.25,0.3,0.33</pts>
</gf>
```

As can be seen, the bounds of the x and y scales are given, followed by a set of *y*-values (tagged `<pts>`). The (fixed) increment along the *x*-axis is determined by dividing the length of the *x*-scale by the number of points.

Alternatively, though not guaranteed to be understood 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>0,1</yscale>       <!-- min and max -->
   <pts>0,0.05;0.1,0.1;0.2,0.2;0.3,0.25;0.4,0.3;0.5,0.33</pts>
</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.

Graphical functions have one option:

```
<discrete/>
```

When this is set, no interpolation is done between points. Instead, the data values represent a step-wise function where the steps occur at each *x*-coordinate. Necessarily, the last point must equal the second to last point (this should be enforced if it is not the case).

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 (the classification is based on the direction of flow when the flow rates are positive –negative inflows do flow outward while negative outflows do flow inward). The basic stock definition is shown below with sample values.

```
<stock>
   <name>Motivation</name>
   <eqn>100</eqn>
   <inflows>increasing</inflows>
   <outflows>decreasing</outflows>
</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 a list of inflows or outflows. If there are multiple inflows, they are listed separated by commas (e.g., `<inflows>in1,in2</inflows>`). If the order of input is important (inflow priority), it is taken to be the order they appear in this comma-separated list. Multiple outflows also appear separated by commas. The order that they appear in this list is their outflow priority, if that is important. I.e., material from the stock is first given to the first flow on this list. If there is still something in the stock, the second flow gets some of it. This example assumes that the stock in question cannot become negative (for example, Inventory).

The options for a stock are:

```
<conveyor>5</conveyor>  <!-- number is length of conveyor -->
<queue/>
<non_negative/>
```

Note that non-negative is not 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 option includes the conveyor's length (transit time). This can be an equation if the conveyor's length varies. If `conveyor` is included, there can be no more than two outflows. The first outflow in the outflows list is *by definition* the conveyor outflow, while the second is the leakage outflow. Other conveyor options are:

```
<capacity>20</capacity>  <!-- maximum allowed on conveyor -->
<in_limit>5</in_limit>   <!-- most that can be taken from q -->
<batch_integrity/>       <!-- maintain q batches -->
```

The last option is only available when there is a queue directly upstream from the conveyor (equivalent to "one at a time and don't split batches", without it we have "as much as possible and split batches" – think about whether split batches ever wants to be off). The `in_limit` is only needed when either there is a queue directly upstream from the conveyor, or there are several inflows to the conveyor (in which case the limit is applied in priority order). The latter case may not be supported as it can easily be reformulated in the same was as non-negativity.

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 (the conveyor and queue drive 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</name>
   <eqn><![CDATA[ rewards*reward_multiplier ]]></eqn>
</flow>
```

The options for a flow are:

```
<non_negative/>
<overflow/>
```

The first is used to document a uniflow, while the second is only used for queue outflows (see stocks above). While it is tempting to also label queue outflows, conveyor outflows, and leakage outflows with special tags, these are by their nature already defined by the stocks that use them. There is no need to repeat this information here.

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</name>
   <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).

4.5 Groups

Groups require a name and may have documentation. The `<group>` tag surrounds the entities in that group. The block must begin with the name and any documentation, as shown below.

```
<group>
   <name>Financial_Sector</name>
   <doc>
      The operation of the Finance department is modeled in
      this sector.
   </doc>
   <stock>
      ...
   </stock>
   <flow>
      ...
   </flow>
   ...
</group>
```

All of the objects within a group belong to that group.

4.6 Built-in Function Translation Macros

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

Macros live outside of all other blocks (and may be the only thing in a file other than its header). The set of macros is tagged with the `<macros>` tag. Within this block is a series of macros, identified by their own `<macro>` tag (note that macros for options, when they exist, will likely use a different tag). Each macro begins with its name and parameters. It must also have an equation that defines its value, as shown below.

```
<macro>
   <name>LOG(X, Y)</name>
```

```
   <eqn><![CDATA[ LN(X)/LN(Y) ]]></eqn>
</macro>
```

This form is 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 (in the same way they are defined in the model). This is shown below.

```
<macro>
   <name>SMOOTH1(input, averaging_time)</name>
   <eqn>Smooth_of_Input</eqn>
   <stock>
      <name>Smooth_of_Input</name>
      <eqn>input</eqn>
      <inflows>change_in_smooth</inflows>
   </stock>
   <flow>
      <name>change_in_smooth</name>
      <eqn><![CDATA[(input – Smooth_of_Input)/averaging_time]]></eqn>
   </flow>
</macro>
```

This is identical to the SMTH1 function without the optional third parameter (which would replace the initial value equation of Smooth_of_Input if it were given), so this is a trivial example (i.e., the equation could just be SMTH1(input, averaging_time) to do the same thing). 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 which 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).

Here is a simple (i.e., self-contained) option filter. This implements non-negativity for flows (i.e., makes them uniflows).

```
<option_filter>
   <flow>
      <name>non_negative(flow, value)</name>
      <eqn><![CDATA[IF value THEN MAX(flow, 0) ELSE flow]]></eqn>
   </flow>
</option_filter>
```

The name of the macro is the same as the option name and it is passed the value of the object calculated without the option, as well as the value of the option (true: 1, false: 0). The result replaces the object's value.

Because options are shared by different objects, we need a way to distinguish between the different object types. This is done by using the correct tag for the object inside the

macro ("<flow>" is used above to show this is a flow option macro). Optional parameters can then be added to the type in order implement more sophisticated macros (shown below).

Here's an attempt at implementing the non-negative option for stocks.

```
<option_filter>
   <stock applyto="outflows">
      <name>non_negative(flow, value, stock, outflow_sum)</name>
      <eqn><![CDATA[
         IF value
         THEN MAX(stock/DT – outflow_sum, flow)
         ELSE flow
      ]]></eqn>
   </stock>
</option_filter>
```

This concept is a little more complicated. Here we are clearly talking about the non-negative option of the stock. But we are now using the `applyto` modifier to tell us this option changes the stock's outflows, not the stock itself.

This also changes the parameters a bit. Because we are modifying a flow, the first parameter is that flow value, not the stock. However, we also include the stock. Finally, because we are looking at the stock's outflows, we also pass in the sum of all the outflows already calculated this time step (i.e., all those with higher priority than the passed flow). The same sort of thing would happen if we operated on the inflows instead (except the sum of inflows already calculated would be passed instead). It is possible that in these cases, we should pass both the sum of the known inflows and the sum of the known outflows.

Further examples are needed to test this theory. In any event, this requires more complex support from the macro interpreter of the host program.

The existence of the above 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. This is necessary for options that affect both the object and those that are upstream or downstream from them.

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 the block any display aspects can be specified (or overridden). There are defined defaults for all of these settings. In addition, the global or model style sheets can change the defaults. Note that the display

tag is not explicitly required in a style sheet, as everything except model options refer to presentation.

There is usually one `<display>` block nested one level inside the `<model>` that describes objects required to properly render the model.

5.1 Display Properties

The following display properties are defined for all presentation objects (including Level 3). Note that not all of them are available for all objects.

- Position: `<pos>` with "`x,y`" [required]
- Name position: `<name_pos>` w/`top`, `bottom`, `left`, `right`, `center` with optional offset [default: `bottom`, except for stocks which default to `top`]
- Background color: `<background>` w/valid CSS color only [default: `white`] specific objects can also specify images
- Color: `<color>` w/valid CSS color [default: `black`]
- Font: `<font>` with CSS conventions, restricted to style, weight, size, family; weight can be `normal` or `bold` only [default: `9 pt sans-serif`]
- Text alignment: `<text-align>` with CSS conventions [default: `left`]
- Text decoration: `<text-decoration>` with `none` or `underline` only [default: `none`]
- Border: `<border>` with CSS conventions [default: `thin solid`]
- Padding for border: `<padding>` with CSS conventions, length only [default: `2 px`]
- Drawing Order: `<z-index>` with CSS conventions [defaults to order in file]

All objects require a position, so there is no default. Unless otherwise specified, the position is the center point of the object.

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.

5.2 Display Properties for SMILE Objects

Some of the SMILE objects require special display properties. At present, only the flow and the group have their own properties.

5.2.1 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 intersection of the flow valve and the pipe.

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, `<cpts>`, that gives the flow's points in order from the start of the flow to its end.

```
<cpts>290,107;335,107</cpts>
```

This is a simple horizontal flow that only has its two endpoints. If clouds are explicitly specified, it is not strictly required that simple flows include their endpoints as they can be inferred from the flow's center and connections. It is, however, recommended if you desire a faithful reproduction of the flow.

Some programs use curved flows instead. These programs also have control points that could be presumably be manipulated to approximate the perpendicular bend behavior supported by this standard.

5.2.2 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 two additional control properties:

```
<show_name/>       <!-- default: true -->
<show_image/>      <!-- default: false (no image) or true -->
```

The first makes the name is visible, presumably on or near the border. The second makes the image (if any) visible, obscuring the contents of the group. Note that even without an image, the contents should be hidden (i.e., `show_image` = hide contents).

The position of a group is its top-left corner, not its center. Resizable objects (including text annotations, graphics frames, and buttons) all require their size to be defined with an (*h*, *v*)-pair.[4]

```
<size>500,400</size>
```

5.3 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 clouds, connectors, and aliases (or ghosts).

5.3.1 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 should be unique within each object type, but are not unique across all objects. They are defined using the `<uid>` tag followed by the identifying number. Every display object must have a unique ID.

[4] *h* represents the horizontal width, while *v* represents the vertical height, of the object

5.3.2 Clouds

Clouds represent the sources and sinks of the system, i.e., the model boundary. Each cloud is associated with a flow and is classified as either a source (at flow's starting point) or a sink (at flow's ending point).

Some might argue that a cloud is not really its own object, but the representation of an unattached flow end. However, clouds can have their own positions (relative to the flow, of course) and their own color. One can even imagine documentation for a cloud, though this is not presently available in any program.

The basic cloud definition is shown below with sample values.

```
<cloud>
   <uid>2</uid>
   <pos>116,50</pos>
   <sink>out</sink>
</cloud>
```

If the cloud is a source (rather than a sink), the `<source>` tag is used. The position is absolute on the model space, not relative to the flow. This means clouds could drift quite far away from their flows; it is recommended that the position get reset if it doesn't make sense when the model is read. Note that display characteristics (e.g., color) can be changed directly within this block.

Finally, not all programs support clouds as their own objects. If no cloud appears in the model for a given flow, it is reasonable (indeed, required) to create one.

5.3.3 Connectors

Algebraic calculations are represented in a stock-flow diagram with connectors between the model elements. Each connector always goes between two entities, which are identified by name or type and unique ID. The basic connector definition is shown below with sample values.

```
<connector>
   <uid>3</uid>
   <from>Inspect</from>
   <to>out</to>
   <pos>283,85</pos>
</connector>
```

The `from` and `to` properties identify the objects the connector goes between (and its direction). If either object is a display object, e.g., an alias, the type of object and its unique ID would appear instead of its name. This would look like this:

```
<from><alias>3</alias></from> <!-- Inspect -->
```

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

The position of the connector is the center of its starting point, i.e., where it attaches to the `from` object. Some programs allow both ends to be controlled. In these cases, a `<to_pos>` tag should be included. Other programs may have multiple points defined along the way. In this case, use a `<pts>` tag, following the same rules as the flow (see above). Be aware, though, that the only information *required* by this standard is the single starting position.

5.3.4 Aliases

An alias (aka ghost) is a second image of an object used to avoid crossed connectors or to communicate across groups or models. The alias typically appears in the same form (or a close approximation) as the original, has the same name, but appears differently in some way. Aliases may also have different display attributes, such as name position and color.

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>3</uid>
   <of>Inspect</of>
   <pos>385,153</pos>
</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 layer 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 also appear within the display block.

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, loops), annotation devices (text blocks, graphics frame), and control devices (buttons). [This leaves the process frame and bundled flows/connectors out of it – maybe the loop should be left out as well.]

There is still some question about where graphs and, perhaps, tables belong. No question they are display objects of some sort and are tied to the interface. But DYNAMO did include plot commands with scaling. And the model is not very useful without at least graphs. On the one hand, it seems inconsistent to put them on the Simulation layer (and into SMILE). On the other hand, everyone shouldn't have to become level-3 compliant just for graphs. A reasonable compromise would be to put them on the display level.

References

Diker VG, Allen RB. 2005. XMILE: Towards an XML Interchange Language for System Dynamics Models. *System Dynamics Review* **21**(4), 351-359.

Hines J. 2003. A SMILE for System Dynamics. *System Dynamics Newsletter* **16**(1): 1, 5-6.

iThink/STELLA 9 Technical Documentation. 2006. isee systems: Lebanon, NH.

Vensim Version 5 User's Guide. 2006. Ventana Systems: Harvard, MA.

W3C. CSS 2.1 Specification. http://www.w3.org/TR/CSS21 [6 November 2006]

W3C. XML 1.1, second ed. http://www.w3.org/TR/2006/REC-xml11-20060816 [29 September 2006]