SMILE: A Common 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. 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 Systems SIG at the 2006 ISDC[1], I suggested breaking the problem into two pieces: the language we intend to interchange and the interchange format. This first 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. A separate document on XMILE addresses the XML-based interchange format.

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

[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.

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.

Stocks in SMILE are only constrained by their inflows and 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
```

[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.

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

Flows in SMILE are only constrained by their equation[3]. Vendor-specific features such as uniflows, which prevent a flow from taking on negative values, are not (directly) 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[4]) 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.

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*.

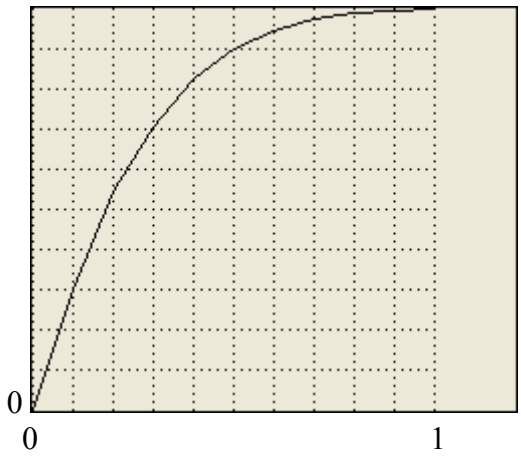
Since every modern program displays table functions as graphs, and most allow the users to edit the functions by drawing the graph, the terminology used here is "graphical functions."

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. The graph of this function appears below.

[3] Conveyors, an optional feature, can place additional constraints on a flow.
[4] 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.



The above is a continuous graphical function. There are three types of graphical functions supported by SMILE, with these names:

Name	Description
`continuous`	Intermediate values are calculated with linear interpolation between the intermediate points. Out-of-range values are the same as the closest endpoint (i.e, no extrapolation is performed).
`extrapolate`	Intermediate values are calculated with linear interpolation between the intermediate points. Out-of-range values are calculated with linear extrapolation from the last two values at either end.
`discrete`	Intermediate values take on the value associated with the next lower *x*-coordinate (also called a step-wise function). The last two points of a discrete graphical function must have the same *y*-coordinate. Out-of-range values are the same as the closest endpoint (i.e, no extrapolation is performed).

Graphical functions in SMILE can optionally be named (the name can then be used in an equation as described in section 3.2).

Although some vendors support arbitrary sets of (*x*, *y*)-pairs, DYNAMO only supported a fixed increment for *x*. This is what this base SMILE standard supports. However, the SMILE language also includes table functions composed of arbitrary (*x*, *y*)-pairs as an optional feature (see section 6.1).

The SMILE language also includes analytic table function definitions as an optional feature (parameters to be defined at a later date). These table functions definitions must also include a representative set of points.

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 a constant must contain at least one digit. A decimal point is not required. The number can be optionally 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, a fixed number of 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 (section 3.0) handle this case.

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

2.2 Identifiers

Identifiers are used throughout a model to give variables, namespaces, units, subscripts, groups (or sectors), macros, and model names. Most of these identifiers will appear in equations[5], and as such need to follow certain rules to allow for well-formed expressions.

[5] The group name is perhaps the only exception.

Due to differences in different vendors' products, it is challenging to find 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.

2.2.1 Identifier Form

Identifiers are formed by a sequence of one or more characters that include roman letters (A-Z or a-z), underscore (_), dollar sign ($), digits (0-9), and Unicode characters above 127. Identifiers cannot begin with a digit or a dollar sign, and cannot begin or end with an underscore.

An identifier may be enclosed in quotation marks, which are not part of the identifier itself. However, an identifier must be enclosed within quotation marks if it violates any of the above rules. Within quotation marks, a few characters must be specified with an escape sequence that starts with a backslash. All other characters are taken literally. The valid escape sequences appear below. If any character other than those specified below appears after a backslash, the identifier is invalid.

Escape sequence	Character
`\"`	quotation mark (")
`\n`	newline
`\t`	tab
`\\`	backslash
`\_`	unbreakable underscore (whitespace)

Sample identifiers:

```
Cash_Balance   draining2  "wom multiplier"   "revenue\ngap"
```

2.2.2 Identifier Equivalence

Case-insensitive: Identifiers may use any mixture of uppercase and lowercase letters, but identifiers that differ only by case will be considered the same. Thus, `Cash_Balance`, `cash_balance`, and `CASH_BALANCE` are all the same identifier. For Unicode characters, case-insensitivity is defined by the Unicode Collation Algorithm (UCA – http://www.unicode.org/unicode/reports/tr10/), which is compliant with ISO 14651. For C, C++, and Java, this algorithm is implemented in the International Components for Unicode (ICU – http://site.icu-project.org/).

Whitespace: Whitespace characters include the space ( ), newline (`\n`), tab (`\t`), and underscore (_). Within an identifier, whitespace characters are considered equivalent. Thus, `wom_multiplier` is the same identifier as `"wom multiplier"` and `"wom\nmultiplier"`.

Additionally, groups of whitespace characters are always treated as *one* whitespace character for the purposes of distinguishing between identifiers. Thus, `wom_multiplier` is the same identifier as `wom_____multiplier`.

Finally, leading and trailing whitespace characters are ignored for the purposes of distinguishing between identifiers. Thus, `wom_multiplier` is the same identifier as `"___wom_multiplier___"` (note the quotes are necessary for leading and trailing whitespace).

These rules can be overridden if necessary by the explicit use of the `\_` escape sequence. This is treated as a true underscore character rather than as a whitespace character, so it only compares to itself.

2.2.3 Namespaces

To avoid conflicts between identifiers in different libraries of functions, each library, whether vendor-specific or user-defined, should exist within its own namespace. Note that identifiers within submodels, by definition, appear in their own named local namespace (see section 6.5).

A namespace is an identifier like any other. An identifier from a namespace other than the local one is only accessible by qualifying it with its namespace, a period (.), and the identifier itself, with no intervening spaces. For example, the identifier `find` within the namespace `funcs` would be accessed as `funcs.find` (and not as `funcs . find`). Such a compound identifier is known as a *qualified name* (and those without the namespace are known as *unqualified*).

Namespace identifiers must be unique across a model file and cannot conflict with any other identifier. SMILE predefines its own namespace and a number of namespaces for vendors:

Name	Purpose
`std`	All SMILE statement and function identifiers
`isee`	All isee systems identifiers
`vensim`	All Ventana Systems identifiers
`powersim`	All Powersim Software identifiers
`anylogic`	All XJ Technology identifiers
`forio`	All Forio Simulations identifiers
`simile`	All Simulistics identifiers

Namespaces can be nested within other namespaces. For example, `isee.utils.find` would refer to a function named `find` in the `utils` namespace of the `isee` namespace.

Unqualified names are normally only resolved within the containing model (or macro), i.e., they are assumed local to their model. However, an entire SMILE file, or individual models within a SMILE file, can avoid the use of qualified names from other namespaces, especially for function names in equations, by specifying that the given file or model uses one or more namespaces. In this case, unqualified names will first be resolved locally, within the containing model. If no match is found, the specified namespaces are searched in order until a match is found. By obeying the specified

namespace order, SMILE allows user to control how conflicting identifiers are resolved. However, warnings should be generated for such conflicts.

Note this namespace resolution capability is only available for explicitly defined namespaces and not for the implicit namespaces of submodels (see section 6.5). Any identifiers that are accessed across models must be qualified.

By default, all SMILE files are in the `std` namespace, but this can be overridden by explicitly setting one or more namespaces. It is intended that most SMILE files will always specify that they use the `std` namespace, thus obviating the need to include `std.` in front of all SMILE identifiers.

2.2.4 Identifier Conventions

Identifiers defined by SMILE, including registered vendor namespaces (see section 2.2.3), should be chosen so that they do not require quotation marks. Note that the registered vendor names are all in lowercase; this is intentional. It is also preferred that vendors also choose the identifiers within their namespaces such that they do not require quotation marks.

2.2.5 Reserved Identifiers

The operator names `AND`, `OR`, and `NOT`, the statement keywords `IF`, `THEN`, and `ELSE`, the names of all built-in functions, and the SMILE namespace `std`, are reserved identifiers and cannot be used for any other purpose.

2.3 Data Types

There is only one data type in SMILE: real numbers. Note that some parts of the language, for example array indices, require integer values. These are, however, represented as real numbers.

2.4 Containers

All containers in SMILE are lists of numbers. As much as possible, the syntax and operation of these containers are consistent. Only one container is inherent to SMILE: graphical functions. Three containers are optional in smile: arrays, conveyors, and queues.

Neither a graphical function nor an array can change its size during a simulation. However, the size of a conveyor (its length) can change and the size of a queue changes as a matter of course during a simulation.

Since all four containers are lists of numbers we may wish to operate on (for example, find their mean or examine an element), they are uniformly accessed with square bracket notation as defined in section 6.4, Arrays. There are also a number of built-in functions that apply to all of them. These features are optional and are only guaranteed to be present if arrays are supported.

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 – in that order). 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 for 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).

Operators	Precedence Group (in decreasing order)
[ ]	Subscripts
( )	Parentheses
^	Exponentiation
+ – NOT	Unary operators positive, negative, and logical not
* / MOD	Multiplication, division, modulo
+ –	Addition, subtraction
< <= > >=	Relational operators
= <>	Equality operators [in mathematics, <> is considered 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.

Modulo is defined to return the floored modulus proposed by Knuth. In this form, the sign of the result always follows the sign of the divisor, as one would expect.[6]

Sample expressions: `a*b` `(x < 5) and (y >= 3)` `(-3)^x`

3.2 Function calls

Parentheses are also used to provide parameters to function calls, e.g., `ABS(x)`, and take precedence over all operators (as do the commas separating parameters). Note that functions that do not take parameters do not include parentheses when used in an equation, e.g., `TIME`. There are several cases where variable names can be (syntactically) used like a function in equations:

[6] Note the standard C modulo operator uses the truncated modulus which gives a sign consistent with the dividend. The choice of modulus method affects integer division, which does not exist in SMILE, as well as the `INT` function, which must return the floor of the number. By definition, a = INT(a/b)*b + a MOD b.

- Named graphical function: The graphical function is evaluated at the passed valued, e.g., given the graphical function named `cost`, `cost(2003)` evaluates the graphical function at $x = 2003$.
- Named model: A model that has a name, defined submodel inputs, and one submodel output can be treated as a function in an equation, e.g., given the model named `maximum` with one submodel input and one submodel output that gives the maximum value of the input over this run, `maximum(Balance)` evaluates to the maximum value of `Balance` during this run. When there is more than one submodel input, the order of the parameters must be defined as they are for a macro definition. For more information, see sections 5.7 (macros) and 6.5 (submodels).
- Array name: An array name can be passed the flat index (i.e., the linear row-major index) of an element to access that element. Since functions can only return one value, this can be useful when a function must identify an element across a multidimensional array (e.g., the `RANK` built-in). For example, given the three-dimensional array `A` with bounds `[2, 3, 4]`, `A(10)` refers to the tenth element in row-major order, i.e., `A[1, 3, 2]`. See section 6.4 for more information about arrays.

3.3 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 some 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.

3.4 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 }`

3.5 Documentation

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

3.6 Units

Each variable has its own set of units. XMILE defines how units are stored.

Each model has a defined unit of time. The predefined time 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. Units are specified with SMILE expressions (called the unit equation), restricted to the operators ^ (exponentiation), - or * (multiplication), and / (division) with parentheses as needed to group units in the numerator or denominator.[7] Exponents must be integers. When there are no named units in the numerator (e.g., units of "per second"), the integer one must be used as a placeholder for the numerator (e.g., 1/s). The integer one can be used at any time to represent dimensionless units.

Units can optionally be given user-identifiable names (e.g., "per second") to show the user in place of their equations. This name cannot contain any operators, but is not recognized within equations. If no name is given, the equation is always presented to the user.

Alias names can also be assigned to units so that several different names can be treated equivalently both for unit checking and for defining unit equations. For example, the unit with the user name "cubic centimenters" and equation "cm^3" can be given the alias "cc" which can then be used to define any derived unit, e.g., "g/cc" (density).

[7] Note some packages do not allow multiple division operators nor parentheses except to group the multiplication in the denominator.

4.0 Simulation Specifications

Every SMILE model must specify the start time and the stop time of the simulation. If DT is not specified, it defaults to one. For DT <= 1, DT can be specified as an integer reciprocal (e.g., 7 for DT = 1/7). If the units of time are not specified, they default to `time`. Units of time are specified with the SMILE abbreviations (section 3.6).

The language also supports an optional pause interval. By default, a model runs to completion (from STARTTIME to STOPTIME). However, if 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.

The simulation specifications can specify the option to only run selected groups (each selected group is marked), only run selected submodels (each selected submodel is marked), or run the entire model (default).

4.1 Integration Methods

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

SMILE Name	Integration Method
Euler	Euler's method (default)
RK2	Runge-Kutta 2
RK4	Runge-Kutta 4
RK45	Runge-Kutta mixed 4^{th}- 5^{th}-order (optional)
RK_Var	Runge-Kutta Variable Step Size (optional)
Gear	Gear algorithm (optional)

The last three integration methods are optional. 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.

Some vendors do not offer RK2, as it is less useful than it once was when computing power was expensive. SMILE defines RK4 to always be the fallback for RK2, i.e., RK2 implies "RK2, RK4."

4.2 Simulation Events

Events based on entity values can be triggered while the model is being simulated. Any implementation that does not support simulation events is free to ignore them.

Within the simulation, these events are limited to pausing or stopping the simulation. Events are specified as a series of threshold values that, when exceeded, trigger the specified action:

SMILE Name	Action
`pause`	Pause the simulation (default)
`stop`	Stop the simulation

For each value, thresholds can be exceeded in either of two directions:

SMILE Name	Action occurs when entity value becomes:
`increasing`	Larger than the threshold (default)
`decreasing`	Smaller than the threshold

The number of times the event occurs during the simulation (its frequency) can also be controlled:

SMILE Name	Event occurs:
`each`	Each time the threshold is exceeded (default)
`once`	Only the first time the threshold is exceeded each run
`once_ever`	Only the first time the threshold is exceeded this session

When the frequency is set to `each`, an optional repetition interval can also be specified which causes the event to be triggered again every so many unit times (at the specified interval) that the variable remains above the threshold.

Each unique threshold value and direction can be given more than one event. In this case, the events are triggered in order based on which instance the threshold has been exceeded since the start of the run. For example, if there are three events assigned to a threshold of 5 (increasing), the first event will be triggered the first time the variable goes above 5, the second event will be triggered the second time it goes above 5, and the third will be triggered the third time it goes above 5. If the variable goes above 5 after that, no further events will be triggered. When multiple events are assigned in this way, the frequency can only be `once` or `once_ever` (i.e., it cannot be `each`).

A range (minimum and maximum) that contains all events can also be specified to more readily allow the user to edit the thresholds in context. By default, this range should be initialized to the variable's known range at the time the events are first created (the user then has to modify them if they are no longer appropriate; they do not readjust if the variable's range changes).

5.0 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.

5.1 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)

ARCCOS: arccosine of a number
Parameters: 1: the number to take the arccosine of
Range: $(0, \pi)$
Example: arccos(x)

ARCSIN: arcsine of a number
Parameters: 1: the number to take the arcsine of
Range: $(-\pi/2, \pi/2)$
Example: arcsin(x)

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)

INF: value of infinity
Parameters: none
Example: inf

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)

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

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

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)

5.2 Statistical Functions

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

LOGNORMAL: Sample a value from a log-normal distribution
Parameters: 2 or 3: (*mean*, *standard deviation*[, *seed*]) $0 \leq seed < 2^{32}$
If *seed* is provided, the sequence of numbers will always be identical
Example: lognormal(10, 1) samples from a lognormal distribution with mean 10 and standard deviation 1.

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

POISSON: Sample a value from a Poisson distribution
Parameters: 2 or 3: (*mean*[, *seed*]) $0 \leq seed < 2^{32}$
If *seed* is provided, the sequence of numbers will always be identical
Example: poisson(3) samples from a Poisson distribution with a mean arrival rate of 3 arrivals per unit time

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

5.3 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

FORCST: Perform a trend extrapolation over a time horizon
Parameters: 3 or 4: (*input*, *averaging time*, *horizon*, [, *initial trend*])
If *initial trend* is not provided, zero will be used
Example: forcst(Quality, 5, 10) calculates value of quality 10 time units in the future

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

TREND: Find trend in input over a given time frame
Parameters: 2 or 3: (*input*, *averaging time*, [, *initial value*])
If *initial value* is not provided, zero will be used
Example: trend(Quality, 5) calculates the fractional change in Quality per unit time

5.4 Test Input Functions

PULSE: Generate a one-DT wide pulse at the given time
Parameters: 2 or 3: (*magnitude*, *first time*[, *interval*])
Without *interval* or when *interval* = 0, the PULSE is generated only once
Example: pulse(20, 12, 5) generates a pulse value of 20/DT 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)

5.5 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

5.6 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)

5.7 Defining Unsupported Built-ins

SMILE provides a way for vendors to specify the operation of both their own functions and the functions of other vendors that their users wish to use. In the latter case, these can map to either their own functions (if available) or to the SMILE functions. As described in section 2.2.3, vendor-specific function names should appear in their own namespace, and thus be prefixed by a vendor identifier to avoid conflicting names between both different vendors and SMILE, e.g., `isee.HISTORY`.

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)`.[8]

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 could also handle this (by creating a macro for the 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.

5.7.1 Macros Implementing Functions

SMILE implementations must support macros. Macros can use all of the syntax of SMILE to achieve their result. The simplest kind of macro is simply an expression using existing functions (including macros) and operators. In this regard, its value is specified in the same way as an auxiliary. The change of base formula above is a good example:

```
LOG(x, y):
   LN(x)/LN(y)
```

[8] Change of base formula – `LOG10` works just as well as `LN`.

Macros can be recursive, so a slightly more complicate macro would call itself:

```
FACT(x):
   IF x <= 1 THEN 1 ELSE x*FACT(x – 1)
```

More complicated macros can use stocks, flows, and auxiliaries to define their behavior. This would be the approach, for example, to implement a smooth function if one did not already exist. The factorial function, defined recursively above, can also be defined using stocks and flows (using DYNAMO syntax where FACT is the result):

```
L FACT.K = FACT.J + DT*CHANGE.JK
R CHANGE.KL = FACT.K*TIME
```

This, of course, requires that the simulation specs for this macro be properly set to run from 1 to *x*, with the default DT of one. A more complex macro is, in fact, just a model. However, for a model to be used as a function, it also needs to specify the expected parameters, their order, and the macro equation (i.e., the function return value). The simulation specs also need to be specifiable in terms of the macro parameters.

The name of the macro is the same as the name of the function. 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 parameters. Default values for trailing function parameters are supported, giving limited support for variable numbers 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.

Macros can also include text showing their usage format and text to describe their purpose, both of which are helpful documentation for the user of the function.

5.7.2 Macros Extending Building Block Behavior

There are also some defined, but unsupported, options for building blocks (vendors can also add their own). 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 the stock could find for it).

SMILE therefore supports building block options with macro *filters*, also called *option filters*. The filters run after the object's value has been computed and allow the object's

value to be altered (filtered) based on the option setting. If several filters are needed, they would run in the order they appear in the object's list of options. A basic filter would only affect the given object, and so is passed just the object itself and the value of the option setting. 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. This is summarized in the table below.

filter type	applied to	Parameters
stock, flow, aux	itself	object value, option value
stock	its inflows	flow value, option value, stock value, inflow sum
stock	its outflows	flow value, option value, stock value, outflow sum
flow	upstream stock	stock value, option value, flow value
flow	downstream stock	stock value, option value, flow value

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

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 enhanced graphical functions, conveyors, queues, arrays, and submodels.

6.1 Enhanced Graphical Functions

For many applications based on real-world data streams, a fixed *x*-increment in the definition of a graphical function is too restrictive. Sampling can be extremely irregular, making it difficult to both find a common *x*-increment to capture all points and also interpolate missing points. This feature allows the use of arbitrary (*x*, *y*)-points to define a graphical function.

There are also some canonical shapes for graphical functions that can be described with a few parameters (to be defined). Support for editing these shapes is optional. [Since such graphical functions are required to include a set of points conforming to the standard, every package should be able to use these graphical functions.]

6.2 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 one or more leakage flows can also be defined. In addition, the

conveyor has a variable speed control, so you can change the length of time something stays on it.

Since the outflows have different purposes, it is necessary to be specific about which outflow does what. While leakage flows can be explicitly marked, it is not required. In this case, by convention, if there is only one outflow, it must be the stuff coming off the end of the conveyor. If there are two or more outflows, the first is always the conveyor's output, while the remaining outflows are the conveyor's leakage.

Besides the length of a conveyor (in time units), a conveyor has the following optional parameters (the first four are defined with SMILE expressions):

- Capacity: Maximum contents of a conveyor (default: `INF`)
- Inflow limit: Maximum amount of material that can flow into the conveyor in each unit time (default: `INF`)
- Sample: When provided, the transit time changes only when this expression is true. (default: 1, i.e., the transit time is updated every DT)
- Arrest: When provided, the conveyor shuts down (stops moving, leaking, etc.) whenever this expression is true. (default: 0, i.e., the conveyor never arrests)
- Discrete: Whether the conveyor is discrete (moving batches of material) or continuous (moving a constant stream, e.g., sand and gravel). When discrete, the initial value is applied to the start of each unit time within the conveyor (rather than evenly distributed) and the inflow limit, while still per unit time, can be fulfilled within one DT (in the continuous case, the per DT limit is DT*inflow limit). This is only meaningful when there is *not* a queue immediately upstream of the conveyor (when there is a queue, the conveyor must be discrete). (default: false)
- Batch integrity: Whether batches from an upstream queue can be split into smaller units to meet the limits on the conveyor (they cannot when this is true). This is only meaningful when there is a queue immediately upstream of the conveyor. (default: false)
- Number of batches: How many batches can be taken from the front of the upstream queue to meet the conveyor limits: restricted to one or as many as possible. This is only meaningful when there is a queue immediately upstream of the conveyor (otherwise, it always takes as much as possible). (default: one)
- Exponential leakage: True if the leakage should be an exponential decay across the conveyor (false if the leakage is linear, expressed as fraction of the inflowing amount to leak by the time it exits the conveyor). This is only meaningful if the conveyor has one or more leakage outflows. (default: false)

Each leakage outflow of the conveyor should be marked as such (but does not have to be) and has the following options:

- Leakage zone start: Fraction, in [0, 1], of conveyor length that marks the start of the leakage zone (from the inflowing side), i.e., the fractional length that does not leak on the inflowing side (default: 0)
- Leakage zone end: Fraction, in [0, 1], of conveyor length that marks the end of the leakage zone (from the inflowing side); note (1 – leakage zone end) is the fractional length that does not leak on the outflowing side (default: 1)
- Leak integers: True to leak only integers (default: false)

6.3 Queues

Queues are first-in, first-out objects that track individual batches that enter them (otherwise, they'd just be stocks). The first batch to enter 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 (e.g., a capacity limit on a conveyor).

The value of a queue outflow is determined by the queue's inflow, the queue's contents and what is downstream of the queue. A conveyor, in particular, can limit the outflow of a queue based on its inflow and capacity limits. [Queues flowing to regular stocks and clouds are not limited in any way, so their outflows will always empty the queue.] Queues can have multiple outflows and some of these (other than the first one) can be designated overflows, i.e., flows that take excess capacity from the front of a queue when a higher priority queue outflow has been blocked due to capacity or inflow constraints.

6.4 Arrays

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

Arrays are defined using dimension names. Each named dimension can either specify a name (a SMILE identifier) for each of its subscript indices, or consecutive numbers can be used, starting at one (the latter restriction may eventually be removed, allowing them to start at any integer). For example, a two-dimensional array of location vs. product could have a dimension called `Location` with three indices `Boston`, `Chicago`, and `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.

Dimension names are identifiers from the same namespace as variables. As such, they need to be unique across an entire model (except macros). Subscript index names, on the other hand, only need to be unique within a given dimension name as they are only valid when used as a subscript into an array that is defined with that dimension name.

Within equations, subscripts are SMILE expressions that appear within square brackets with each index separated by a comma. If a subscript expression results in an invalid subscript index (i.e., it is out of range), a zero (0) should be returned and, optionally, a warning should be given to the user. The user should be allowed to treat all subscript indices – even named ones – as numbers (specifically integers, but SMILE only has real numbers). Arrays are assumed to be stored in row-major order, which is important for initialization, in data sets, and when using flat indices (as described in section 3.2).

Within the equations of arrays, dimension names can be used in subscripts. A dimension name is a placeholder for the subscript index used by the element in which the equation appears. For example, an array `profit[Location, Product]` could have the single equation:

```
revenue – sales
```

which, since when all indices are dimension names none need appear, is shorthand for:

```
revenue[Location, Product] – sales[Location, Product]
```

Each element of *profit* would have the dimension names bound to their indices. For example, when *profit*[*Boston*, *blouses*] is evaluated, it gets bound to the above equation as follows:

```
revenue[Boston, blouses] – sales[Boston, blouses]
```

Clearly for dimension names to be used in this way, the array containing the equation must itself be sized using those dimension names, though not necessarily in the same order (e.g., `profit[Product, Location]` can use the same equation).

This raises the issue of how an array's equation is defined in SMILE. There are two ways to do this:

- One equation for the entire array (the "Apply to All" option in STELLA): One equation is given for the array and can include dimension names. No element equations appear.

- One equation for each element of the array: Each element of the array appears with its own equation. No equation appears for the entire array.

If the array is a graphical function, it must use the first option, i.e., one equation. However, there can be separate graphical functions defined for each element of the array.

6.4.2 Array Operations

As much as possible, arithmetic operators should behave in the expected linear algebra ways. Operations with a scalars fall out using the dimension name syntax. Addition and subtraction of same-sized arrays fall out in the same way. However, for historical

reasons, the remaining SMILE operators also perform element by element operations instead of linear algebra.

Multiplication of arrays should be properly supported using some as yet undefined operator (maybe ".*" since it is the dot product, though this is the opposite of what MATLAB uses). If array *A* has size $M \times N$ and array *B* has size $N \times P$, then array *C* of size $M \times P$ can have the equation `A[M, N].*B[N, P]`. The result should follow linear algebra multiplication. Likewise, a vector of size $1 \times M$ can be multiplied by another vector of size $M \times 1$ (using transpose – below) to get a scalar, the dot product.

Note that division is only supported on an element-by-element basis, i.e., matrix inversions are not supported.

Transposition of arrays is handled in three ways:

- Transposition operator: If reversing the dimensions is sufficient (i.e., turning a $1 \times M$ array into an $M \times 1$ array, transposing a square matrix, or turning an $M \times N \times P$ array into a $P \times N \times M$ array), the transposition operator, ' (apostrophe), can be used after the array, e.g., `A'` or `A[M, 1, N]'`. This unary operator has the highest arithmetic precedence (just above exponentiation) and right-to-left associativity.
- Dimension names: If the dimension names being transposed are unique, the dimension names can be used directly. For example, if array `A` has dimensions $M \times N \times P$ and array `B` has dimensions $P \times M$, a slice of `A` can be assigned to `B` with the equation (in `B`): `A[M, 2, P]`. If all the dimensions match, but are in a different order, just the variable name is needed, e.g., to assign an $M \times N \times P$ array `C` to a $N \times P \times M$ array `D`, just set `D`'s equation to: `C`.
- Dimension positions: When dimension names are not unique and the desired order is not an exact reversal, neither of the two methods above will work. In this case, the dimension position operator, @ (at-sign), must be used with an absolute dimension position of the dimension in the entity that contains the equation. Given an array `A` with dimensions $M \times N \times N$ that we wish to assign to an $N \times N \times M$ array `B` with first *N* in `A` corresponding to the first *N* in `B`, the equation for `B` should be: `A[M, @1, @2]`. Note that `@1` resolves to the first dimension in `B` (i.e., the first *N*) while `@2` resolves to the second dimension in `B` (i.e., the second *N*). Note that the equation `A[M, @2, @1]` (or the more obscure but equivalent `A[@3, @2, @1]`) is the same as `A'`, i.e., it just reverses the dimensions.

6.4.3 Array Slicing

An array can be sliced by using a "*" for a dimension name. For example, `A[1, *]` extracts the first row of matrix *A* while `A[*, 3]` extracts the third column.

More complex slicing can be done using ranges with the start and ending indices separated by `:` (colon). For example, `A[1:3]` extracts the first three elements of *A* (even if that dimension has named indices) and `SUM(A[1:3])` finds the sum of those elements.

6.4.4 Array Builtins

A few built-ins are necessary to support arrays fully. Some are expansions of existing built-ins. Others are unique to arrays.

MIN:	smallest value in an array – *extends MIN(x, y)*
Parameters:	1: the array to examine, e.g, min(A)
	2: any mix of arrays and scalars, e.g., min(A, 0)
MAX:	largest value in an array – *extends MAX(x, y)*
Parameters:	1: the array to examine, e.g., max(A)
	2: any mix of arrays and scalars, e.g., max(A, 0)
SUM:	sum of values in an array
Parameters:	1: the array to sum, e.g., sum(A[M, *]) sums all rows of A
MEAN:	mean of values in an array (just SUM/SIZE)
Parameters:	1: the array to find the mean of, e.g., mean(A)
STDDEV:	standard deviation of values in an array
Parameters:	1: the array to find the standard deviation of, e.g., stddev(A)
RANK:	index of element of given rank in 1D-array sorted in ascending order
	or flat index of element of given rank in *N*-D-array in ascending order
Parameters:	2: the array and the rank, e.g., rank(A, 1) gives index of MIN value
	3: array, rank, secondary sort array; breaks ties using second array
SIZE:	size of array
Parameters:	1: the array, e.g., size(A[1, *]) gives the size of one row of A
SELF:	refers to entity containing the equation (valid with SIZE and PREVIOUS)
Parameters:	none, e.g., size(self[1, *]) gives size of one row of ourselves,
	or previous(SELF, 0) retains our previous value in the next DT
PREVIOUS:	previous value of entity
Parameters:	2: variable and initial value expression, e.g., previous(A, 0)

All of these functions except `SELF` and `PREVIOUS` should work on all container objects. For example, `MIN` should be able to be applied to a graphical function to find the minimum *y*-value, to a queue to find the minimum value in the queue, or to a conveyor to find the minimum value in the conveyor. Additionally, the [] notation used to access array elements should also work on all containers. Thus, `var[3]` returns the *y*-value of the third data point in a graphical function or the third element (from the front) of a queue or a conveyor. If any of these objects is arrayed, two sets of subscripts are used, the first for the array dimensions and the second for the element within that container. For example, if the array *A* of size *M* x *N* is a queue, `A[2, 3][1]` accesses the front element of queue `A[2, 3]`.

6.5 Submodels

The features explained here are general enough for hierarchical models or models made up of separate unrelated pieces. This section supports the idea of independent model pieces 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 having them usually only makes sense when the pieces are arranged hierarchically.

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 to use the model name as the local namespace and refer to objects across model boundaries using qualified names. For example, the variable named `expenditures` in the model `marketing` can be referenced from any other model as `marketing.expenditures`. Note this single qualifying level forces submodel names to be unique across a model (there are exceptions in Level 3). The top level of a model has no name, so the variable `cost` at the top level is simply `.cost`. [Note that display IDs, described in section 7, must also be qualified when referenced across models (particularly in interface objects), e.g., `.12` or `marketing.4`.]

Submodels typically appear in a container created by and visible to the user (called a *module*). The SMILE format includes modules for this purpose; these can be ignored by packages that do not use them and must be auto-created if not there for packages that do. The module must have the same name as the model it contains. However, if the module has been created and named, but not yet assigned a model, it can (and will) have a name that does not correspond to any model.

A module can also have an icon or picture assigned to it. Depending on the package, this will be shown inside the existing icon, or will replace the module icon.

The scope of variables within a submodel is local to that model; variables in a model are not normally visible from outside the model.[9] It is therefore necessary to explicitly specify the variables that are accessible outside the model, i.e., the *outputs* of the model.

Some implementations also require specific variables be set aside for *inputs* to the model – i.e., to pick up the outputs from other models. These will necessarily need to be specified for these implementations. They can, however, be ignored on systems that do not care. In addition, systems that do care should automatically create them if they do not appear within a SMILE document.

The variables that are used for this cross-model communication, the inputs and outputs, are given an additional *access* attribute. The access is restricted to the following values:

SMILE Name	Access Level
`input`	placeholder for input value (acts like an alias)
`output`	public access output (restricted by upward connections)

[9] If the scope of variables in a model should be global, the correct mechanism is a *group*, not a submodel.

Note there is a third access level that is not explicitly represented: an output can have restricted access, available only within the next lower module. These act like function parameters and are typically represented by connections from that entity to the module that represents the model they are being passed to. An entity with input access can also have this restricted local (implicit) output access, to act as a pass-through from one module up into the containing model and then down again into the other module (which is only necessary if the variable is also used in the containing model).

Note there is no option for an input that is also a global output. This requires an intermediate auxiliary. As placeholders, inputs do not have to be assigned to an output from another module. When assigned, an input is also referred to as a cross-level ghost (or alias). Within the model file, it is generally preferred that the output they are assigned to refers to the qualified name of the input instead of the reverse. This allows the same submodel to be used in many places with different parameters either within the same model or across different models. For symmetry, simplicity, and for upward connected outputs that may be used across many models (and so have different upward connections), it is also allowed to specify the output in the input instead using the output's qualified name, in a way that mirrors how aliases are defined (section 7.4.4).

By default, a submodel inherits the simulation specifications of the overall model. However, each submodel can specify its own simulation specs. In such cases, there are two modes of operation:

`normal`	use same time base as rest of model, but different DT
`independent`	do a complete run for every DT in the containing model

In the first mode, the overall model's DT must be an integer multiple of the submodel's DT. Note the submodel's start and end are forced to be the same as the overall model.

In the second mode, the length of the simulation specifies how many times to iterate before returning a value. This performs a simple loop.

6.6 Macro-based Auxiliaries and Submodels

Because the SMILE language has its own macro language, it is possible to allow the user to implement their own macros within the SMILE language (and thus within simulation packages). This allows users to extend the basic functionality of any package, and, since the macros are in SMILE, it allows these extensions to be shared. All SMILE implementations must support SMILE macros, so the optional behavior is restricted to the ability to create and edit these macros.

The logical consequence, from a user's point-of-view, is to allow the user to implement simple macros in auxiliaries and complex macros in submodels. In the latter case, the module can refer to a named macro rather than a named model, the only difference being a macro allows the parameters and return value to be specified. The auxiliary case requires, as an equation option, an indication that the equation is implemented in a macro with the same name (no equation then appears within the auxiliary's definition).

7.0 Display

The display characteristics describe how the model is drawn.

7.1 Style Information

Style information describes drawing characteristics, such as color and font. Style information is hierarchical and cascading (like web page style sheets). The model file can have default settings for the entire file. Each model can override or add to those for the model. Each display definition within the model can then override or add to the file and the model styles for that display. Each individual object can then override or add to these to set its specific style. There are also default styles to be used if none of these styles are specified.

7.2 Drawing Plane

A drawing plane is assumed that is large enough to hold the entire model. This plane has integer coordinates that more-or-less correspond to pixel locations at normal zoom on a 72 dpi display. The coordinate (0, 0) is at the top left of the drawing plane. The *x*-coordinates increase to the right and the *y*-coordinates increase to the bottom. When referring to a number of coordinates in the drawing plane (also known as model coordinates), the term *display units* will be used.

Several display options are used define the drawing plane:

Property	Values (defaults in [])
Number of pages	*rows* and *cols* [1, 1]
Page orientation	*portrait* and *landscape* [*portrait*]
Page size	*width* and *height* of a page (display units)
Page order	Page number ordering: *row* or *col* [*row*]
Show pages	True to display page boundaries [*false*]
Scroll position	*x*- and *y*-coordinate of top-left of visible area (display units) [0, 0]
Zoom level	Zoom level as a percentage [100.0]

7.2 Display Properties

The following display properties are defined for all SMILE objects. The succeeding sections go into more depth.

Property	Values (defaults in [])
Position	*x*- and *y*- coordinate on drawing plane
Predefined size	*stock_size*, *aux_size*, or *module_size* with predefined sizes (in the file style block) at values *name_only*, *small*, *medium*, and *large*; flow valves cannot be resized, but always use *aux_size* = *medium* [*medium*]
Arbitrary size	*width* and *height* (in display units); invalid with predefined size
Name position	*top*, *bottom*, *left*, *right*, or *center* [*bottom*, except stocks: *top*]

Name angle	Angle (degrees) from center of entity to center of name, where 0° is the positive *x*-axis and proceeds counterclockwise [matches name position]
Color	RGB color or named constant, e.g., CSS [*black*]
Background color	RGB color or named constant, e.g., CSS [*white*]
Font	Family, size, and style; style is limited to *normal*, *bold*, *italic*, *underline*, and *oblique* [9 *pt*, *sans-serif*]
Text alignment	*left*, *center*, *right*, or *justify* [*left*]
Border	Any combination of thickness (*thin*, *medium*, *thick*) and style (*solid*, *double*, *dotted*, *dashed*, *ridge* (3D above surface), *groove* (3D below surface)) [*thin solid*]
Padding	Thickness of padding inside border (in display units) [2]
Image	File path or reference to embedded image
Drawing order	*auto* or an integer indicating required order (objects on top of other objects have a higher integer value) [*auto*]

7.2.1 Position

When an arbitrary size (width and height) is specified, the position attribute specifies the top-left corner of the object (without any nameplate). This is used for anything that can be resized, e.g., stocks, auxiliaries, groups, buttons, and text annotations.

When a predefined size is specified (the default for stocks, auxiliaries, and modules), the position attribute specifies the center of the object (without any nameplate). For a flow, it is the center of the flow valve. Since building blocks are not rendered at the same size across software packages and some software packages may ignore the specified predefined size (it may not even make sense for their representation), specifying the center allows each package to render such objects in the most natural way for that package.

7.2.2 Name Position

The name is most often in one of five fixed positions relative to the object it identifies. These positions are above the object with bottom vertical alignment (text alignment: center), below the object with top vertical alignment (text alignment: center), to the left of the object and centered vertically (text alignment: right), to the right of the object and centered vertically (text alignment: left), and sharing the same center as the object and centered vertically (text alignment: center).

Names can also be moved away from these positions. SMILE supports this with an optional name angle, which is the angle of the radius (of a circle around the object) connecting the center of the object to the center of its name. This angle is in degrees in polar coordinates, i.e., 0° lies on the positive *x*-axis and the numbers increase counterclockwise. The use of an angle emphasizes that names move around the shape of the object, for example, in a circle around an auxiliary and in a rectangle around a stock.

The name positions correspond to 0° (right), 90° (top), 180° (left), and 270° (bottom). When the name is exactly in one of these positions, the angle is not used (i.e., it defaults to whichever of these values corresponds to the name position). The name angle is undefined when the name position is center, as the radius becomes zero in this case.

Some packages allow the names to be placed at an arbitrary location on the drawing plane, completely unrelated to the object position. SMILE does not support this general case, assuming that names should be close to the objects being named.

7.2.3 Background Color, Border, Padding, and *Image*

These are restricted to specific objects (defined later) that have a concept of a foreground and a background, or may need a border. Like the background color, the image appears on the background, but may need a specific background color to fill transparent areas.

The background color can also be applied to a model as a whole, in which case it refers to the color of the drawing plane.

7.2.4 Text Alignment

This mostly applies to simple text annotations. Building blocks (stocks, flows, auxiliaries, etc.) have predefined alignments based on the name position.

7.2.5 Drawing Order

The default setting, *auto*, draws objects in the order they appear in the SMILE file. Drawing order may not be supported in all software packages as some define very specific drawing orders. However, these packages should work toward compliance with this part of the standard.

7.3 Display Properties Unique to Specific Objects

A number of objects have unique display properties. These are described in the following sections.

7.3.1 Flows

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 that gives the flow's points in order from the start of the flow to its end:

Property	Values (defaults in [])
Points	List of *x*- and *y*-coordinates

Some programs use curved flows instead. These are not supported by the standard.

7.3.2 Groups

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 properties:

Property	Values (defaults in [])
Locked	*true* or *false* [*false*]
Show name	*true* or *false* [*true*]
Show image	*true* or *false* [*false* if no image defined, *true* otherwise]

When the locked property is true, the objects within the group remain fixed relative to the group, i.e., when the group is moved, they maintain their relative positions within the group. When the show name property is true, the group name is visible, presumably on or near the border. The show image property hides the contents of the group (show image also means hide contents) and, if there is a group image, also makes the image visible.

7.4 Display-only Objects

A number of objects are available only at the display level. In particular, objects to support the graphical representation of the model are needed. These include connectors and aliases (aka ghosts).

7.4.1 Unique IDs

Display objects do not generally have names or any other way to refer to them. For this reason, every display object must have a unique identifying number (display ID). These integers must be unique across all objects in a model (but not within the entire model file – they are local to a model).

7.4.2 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 either by name or unique display ID. There are two properties (besides position) that are required for all connectors, from and to:

Property	Values
From	Name or ID of object connector is coming from
To	Name or ID of object connector is going to

Note there is no ambiguity between names and IDs; names must be valid identifiers and IDs must be valid integers, and these definitions do not overlap. However, it is encouraged that the file format discern between these in some other way to remove the parsing requirement this implies.

Connectors also have two optional properties:

Property	Values (defaults in [])
Type	Type of connector (see below) [*normal*]
Points	List of *x*- and *y*-coordinates (same as flow)

Note that the position of the connector is the center of its starting point, i.e., where it attaches to the "from" object. Usually the connector then follows a linear or semicircular path to the "to" object. Some programs may have multiple points defined along the way. In this case, the "points" property should be used instead. Be aware, though, that the only information *required* by this standard is the single starting position.

The connector type affects how it is displayed:

Type	Effect
Normal	Normal connector; sometimes called *action* connector (solid line)
Info	Information connector (dashed line)
Delay	Information connector with delay (dashed line with hash marks)

7.4.3 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. The alias typically appears in the same form as (or a close approximation to) the original and 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 position and a reference back to the original (sometimes called the parent):

Property	Values
Of	Name of object this is an 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`.

8.0 Interface

Interface objects can appear on the surface of the stock-flow diagram or on a separate diagram plane. These 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), and control devices (buttons, text- and multimedia-based message posters).

The SMILE language only includes time-series graphs and tables.