Systems Biology Markup Language (SBML) Level 2 Proposal: Array Features

Andrew Finney, Victoria Gor, Eric Mjolsness, Hamid Bolouri

afinney@cds.caltech.edu, gor@aig.jpl.nasa.gov,
Eric.D.Mjolsness@jpl.nasa.gov, hbolouri@cds.caltech.edu

February 26, 2002

Contents

1 Introduction

This document describes proposed features for inclusion in Systems Biology Markup Language (SBML) Level 2. This document describes features enabling the inclusion of arrays of processes, structures or entities in models. These features would allow a model to be assembled from many copies of identical parts. These features enable the representation of patterns of connection amongst array elements.

This document is not a definition of SBML Level 2 or part of it. This document simply presents various features which could be incorporated into SBML Level 2 as the Systems Biology community wishes. This document is intended for detailed review by that community and to provoke alternative proposals. Throughout this document issues that the authors believe will require further discussion have been highlighted.

For brevity the text of this document is with reference to SBML Level 1 (Hucka et al., 2001) i.e. features are described in terms of changes to SBML Level 1. This document uses UML diagrams in the same way except that new features are shown in red.

All types proposed in this document will be derived from the `SBase` type.

The features described in this document are built upon the features defined in Finney et al. (2002).

The appendix A lists all the SBML Level 1 operators with all the operators proposed in this document.

2 Models

The proposed structure of the `Model` type is shown in figure 1. A model would have optional lists of `Domain` and `ArraySpecification` structures.

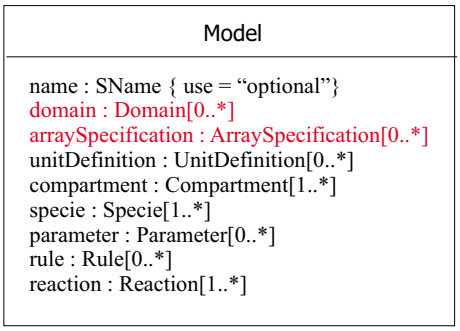

Figure 1: *The definition of the Model type*

`Domain` structures are described in section 3 and `ArraySpecification` structures are described in section 4.

3 Domains

In this proposal the bounds of an array are separated from an actual array definition. Array bounds are defined by these proposed `Domain` structures, which are shown in UML form in figure 2. A `Domain` structure consists of a name (of `SName` type) and inclusive upper and lower bounds of `int` type. These bounds values can be negative. The upper bound must be greater than the lower bound. We suggest that Domains exist in their own namespace i.e. a domain name must unique among all the domains in the model but can have the same name as another non-domain structure in the model.

3.1 Example

The following example shows a `Model` structure incorporating a `Domain` structure called `x` which ranges from -10 through to 10.

```
<model name="tissue">
```

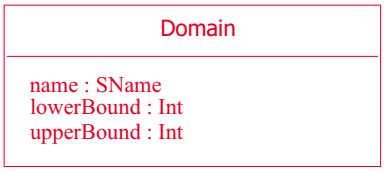

Figure 2: *The definition of the Domain type*

```
    <listOfDomains>
        <domain name="x" upperBound="10" lowerbound="-10"/>
    </listOfDomains>
    ...
</model>
```

4 Array Specifications

Within this proposal the next stage in defining an array is to specify the array 'shape' by specifying the number of dimensions and assigning a domain to each dimension. The proposed `ArraySpecification` structure shown in UML form in figure 3 encapsulates this data.

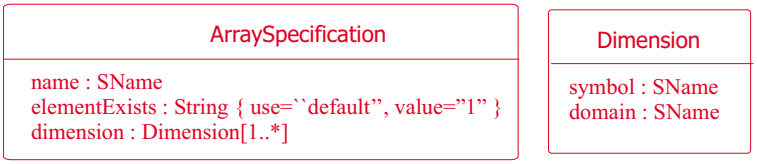

Figure 3: *The definition of the ArraySpecification type*

An `ArraySpecification` structure consists, in its simplest form, of a name field, of type `SName`, and a list of `Dimension` structures. A `Dimension` structure consists of a symbol field and a domain field, both of type `SName`. The domain field refers to a preceding `Domain` structure. The symbol field identifies the `Dimension` structure. The function of the symbol field will be described in section 5.2.

The arrays specified by a given `ArraySpecification` structure will have a dimension for each enclosed `Dimension` structure. The bounds of each dimension are defined by the `Domain` field referenced by the `Dimension` structure.

We suggest that `ArraySpecification` structures exist in their own namespace i.e. a `ArraySpecification` name must unique among all the `ArraySpecification` structures in the model but can have the same name as another non-`ArraySpecification` structure in the model.

4.1 Simple Array Specification Example

The following example shows the specification for a 2 dimensional array of 10 by 5 elements indexed from 0.

```
<model name="tissue">
    <listOfDomains>
        <domain name="x" upperBound="9" lowerbound="0"/>
        <domain name="y" upperBound="4" lowerbound="0"/>
    </listOfDomains>
    <listOfArraySpecfications>
        <arraySpecification name="grid">
            <listOfDimensions>
                <dimension domain="x"/>
                <dimension domain="y"/>
            <listOfDimensions>
        </arraySpecification>
```

```
    </listOfArraySpecifications>
    ...
</model>
```

5 Arrays

The core of this proposal is the idea that almost all the structures in SBML can be defined as arrays as well as single named objects. To this end, we propose that the following SBML types have a new optional field `arraySpecification` of type SName: `Specie`, `Compartment`, `Reaction`, `Parameter` and `Rule`. The `arraySpecificiation` field when present should contain the name of an `ArraySpecification` structure.

The presence of an `arraySpecificiation` field on a structure indicates that the structure represents an array of the objects. All the objects in the array have the properties described by the structures's attributes and substructures. The array shape is defined by the named `ArraySpecification` structure.

Structures that have an `arraySpecificiation` field value share the same namespace as those structures that represent single objects.

5.1 Simple Array Example

The following SBML fragment shows a `Compartment` structure representing a 1 dimensional array.

```
<model name="simple">
    <listOfDomains>
        <domain name="x" upperBound="9" lowerbound="0"/>
    </listOfDomains>
    <listOfArraySpecfications>
        <arraySpecification name="strip">
            <listOfDimensions>
                <dimension symbol="X" domain="x"/>
            </listOfDimensions>
        <arraySpecification>
    </listOfArraySpecifications>
    <listOfCompartments>
        <compartment name="cell" arraySpecification="strip"/>
    </listOfCompartments>
    ...
</model>
```

5.2 Constant Symbols and Expressions

The symbol field of a `Dimension` structure defines a symbol which can be used in numeric expressions. For each element of an array the symbol is assigned the index value of the element. Although these symbol values vary over a domain they are constant for the duration of a simulation. These symbols are called *constant symbols*. It is possible to create expressions using these symbols. It is possible to distinguish between constant and dynamic expressions: in a constant expression all operands and function arguments are either `Dimension` symbols or a constant numeric values.

Constant expressions can then be used in place of most constant attribute values of type `double` throughout SBML. Constant symbols can occur in any numeric expression, for example a rate equation. As symbols given on `Dimension` structures are used in expressions they share the same namespace as other SBML structures such as species.

An examples of this feature is given in section 5.3.

5.3 Using Constant Symbols

The constant symbols defined in an array specification can be used within any structure that references that array specification. For example we can modify the example from section 5.1, to enable each element of the compartment array to have a different volume. The `Compartment` structure is changed from

```
<compartment name="cell" arraySpecification="strip"/>
```

to

```
<compartment name="cell" volume="1 + X/0.1" arraySpecification="strip"/>
```

Notice that although constant symbol `X` varies over the domain `x` this variance only defines the structure of the model so that `X` is not a parameter or variable of any simulation of the model.

5.4 Referencing Array Elements

It is proposed that in place of a symbol for a single object it is possible to refer to a element of an array. This is achieved by using the `[]` array operator, which is similar to the C array operator. In numeric expressions this operator is syntactically equivalent to a function call. The array operator can also be used in any field which refers to a `Specie`, `Compartment`, `Reaction`, `Parameter` or `Rule` structure by name for example the `specie` field on a `SpecieReference` structure can contain an array operator instead of an `SName` type.

The 'array' operand of an array operator must be the name of a structure which has an `arraySpecification` field value. We suggest that the index operand to an array operator be restricted to being a constant expression. An array operator when applied to an array of consisting of n dimensions results in an array of $n - 1$ dimensions and if n is 1 then the result is a single object. The order of application of the array operator to array dimensions follows the order of the `Dimension` structures as given on the corresponding array specification.

We can extend the example shown in section 5.1 to include an array of species placed in an array of compartments:

```
<model name="simple">
    <listOfDomains>
        <domain name="x" upperBound="9" lowerbound="0"/>
    </listOfDomains>
    <listOfArraySpecfications>
        <arraySpecification name="strip">
            <listOfDimensions>
                <dimension symbol="X" domain="x"/>
            </listOfDimensions>
        <arraySpecification>
    </listOfArraySpecifications>
    <listOfCompartments>
        <compartment name="cell" arraySpecification="strip"/>
    </listOfCompartments>
    <listOfSpecies>
        <species
            name="s" arraySpecification="strip" compartment="cell[X]"/>
    </listOfSpecies>
    ...
</model>
```

In this example each species array element is placed in a corresponding compartment.

6 Sparse Arrays and Connections

6.1 Sparse Array Specifications

In practice arrays are not very useful for modeling unless its possible to describe connection schemes between elements of the arrays. For example if one creates a model of a tissue of cells as an array of compartments then the model doesn't become interesting until the interactions between the cells are incorporated. This section begins the process of proposing structures which allow interconnection schemes to be defined.

In this proposal `ArraySpecification` structures include an additional `elementExists` field, as shown in figure 3. This field contains a constant expression which defines whether an element actually occurs at

a given position in the array specification. This expression is conditional, which means that a zero value (false) implies that the element won't occur and a non-zero value (true) implies that an element will occur (conditional expressions are described in more detail in Finney et al. (2002)).

The symbols that can occur in the `elementExists` expression are limited to built-in functions, functions defined by `Function` structures, and constant symbols.

The default value of `elementExists`, 1, ensures that by default the shape of the array specification is determined as described in section 4.1.

Apart from the `elementExists` field, constant symbols only have values for those elements that exist in the array specification in which they are declared.

The index operand to an array operator must refer to a an element which exists in the array. This means that the index must be within the domain to which it is applied and, in the case of sparse arrays be a value to which the `elementExists` expression returns a non-zero value.

6.2 Sparse Array Specification Example

The following example shows a proposed `arraySpecification` structure for triangular arrays where the maximum `y` index is equal to the `x` index.

```
<model name="starfish">
    <listOfDomains>
        <domain name="X" upperBound="9" lowerbound="0"/>
        <domain name="Y" upperBound="9" lowerbound="0"/>
    </listOfDomains>
    <listOfArraySpecfications>
        <arraySpecification name="triangle" elementExists="y <= x">
            <listOfDimensions>
                <dimension symbol="x" domain="X"/>
                <dimension symbol="y" domain="Y"/>
            <listOfDimensions>
        </arraySpecification>
    </listOfArraySpecifications>
    ...
</model>
```

6.3 Using Sparse Array Specifications to Represent Connection schemes

We can use a proposed sparse array specification to present the connections between elements of another array. This is shown in the following example, in which the specification, `grid`, defines a 2 dimensional array and the specification, `connections` defines a sparse 4 dimensional array representing connections between elements of `grid`. `connections` contains adjacent array elements for all pairs of co-ordinates (over the `x` and `y` domains) where the co-ordinates are exactly one array element away from each other.

```
<model name="tissue">
    <listOfDomains>
        <domain name="x" upperBound="9" lowerbound="0"/>
        <domain name="y" upperBound="4" lowerbound="0"/>
    </listOfDomains>
    <listOfArraySpecfications>
        <arraySpecification name="grid">
            <listOfDimensions>
                <dimension domain="x"/>
                <dimension domain="y"/>
            <listOfDimensions>
        </arraySpecification>
        <arraySpecification
            name="connections"
            elementExists="abs(x2 - x1) == 1 || abs(y2 - y1) == 1">
            <listOfDimensions>
                <dimension symbol="x1" domain="x"/>
                <dimension symbol="y1" domain="y"/>
                <dimension symbol="x2" domain="x"/>
```

```
                <dimension symbol="y2" domain="y"/>
            <listOfDimensions>
        </arraySpecification>
    </listOfArraySpecifications>
    ...
</model>
```

The `connections` specification is bidirectional: for every pair of adjacent `grid` co-ordinates there are a pair of elements. `connections` can be simplified and made unidirectional by changing the `elementExists` expression to:

```
x2 - x1 == 1 || y2 - y1 == 1
```

In this case the connections only run from bottom to top and left to right.

This scheme is used in a complete example in section 6.4.

6.4 Example using Connections

Consider the model containing the `ArraySpecification` structures defined in section 6.3. The model can be completed as follows:

```
<model name="tissue">
    <listOfDomains>
        <domain name="X" upperBound="9" lowerbound="0"/>
        <domain name="Y" upperBound="4" lowerbound="0"/>
    </listOfDomains>
    <listOfArraySpecfications>
        <arraySpecification name="grid">
            <listOfDimensions>
                <dimension symbol="x" domain="X"/>
                <dimension symbol="y" domain="Y"/>
            <listOfDimensions>
        </arraySpecification>
        <arraySpecification
            name="connections"
            elementExists="x2 - x1 == 1 || y2 - y1 == 1">
            <listOfDimensions>
                <dimension symbol="x1" domain="X"/>
                <dimension symbol="y1" domain="Y"/>
                <dimension symbol="x2" domain="X"/>
                <dimension symbol="y2" domain="Y"/>
            <listOfDimensions>
        </arraySpecification>
    </listOfArraySpecifications>
    <listOfCompartments>
        <compartment name="cell" arraySpecification="grid"/>
    </listOfCompartments>
    <listOfSpecies>
        <specie name="s" arraySpecification="grid"
            compartment="cell[x][y]" initialAmount="x == 0 ? 1e-10 : 0"/>
    </listOfSpecies>
    </listOfReactions>
        <reaction name="j" arraySpecification="connections"/>
            <listOfReactants>
                <specieReference specie="s[x1][y1]" stoichiometry="1"/>
            </listOfReactants>
            <listOfProducts>
                <specieReference specie="s[x2][y2]" stoichiometry="1"/>
            </listOfProducts>
            <kineticLaw formula="s[x1][y1] * 5"/>
        </reaction>
    </listOfReactions>
</model>
```

As shown before, in section 6.3, the array specification, `grid`, specifies a 2D array and `connections` specifies a sparse array representing the connections between elements of `grid`.

The structure

```
<compartment name="cell" arraySpecification="grid"/>
```

simply creates an 2 dimensional array of compartments. The structure

```
<specie name="s" arraySpecification="grid"
    compartment="cell[x][y]" initialAmount="x == 0 ? 1e-10 : 0"/>
```

creates an 2 dimensional array of species where each species array element is located in the corresponding compartment array element. Only the first column of species has an initial concentration. Notice that the `compartment` field of the `Specie` structure contains an array operator to refer to specific compartments.

The structure

```
<reaction name="j" arraySpecification="connections"/>
    <listOfReactants>
        <specieReference specie="s[x1][y1]" stoichiometry="1"/>
    </listOfReactants>
    <listOfProducts>
        <specieReference specie="s[x2][y2]" stoichiometry="1"/>
    </listOfProducts>
    <kineticLaw formula="s[x1][y1] * 5"/>
</reaction>
```

creates a reaction between adjacent species. Notice that the kinetic law formula contains a reference to a species concentration.

7 Simplified Array Structures for Species

It is possible to incorporate a simplified mechanism for creating species arrays and referencing the elements of those arrays into the scheme proposed in this document.

7.1 Implied Species Arrays

In this proposal the `compartment` field of a `Specie` structure can consist of a name of an array of compartments. This kind of structure represents an array of species with the same specification as the given compartment array. Each element of the specie array is located in a corresponding compartment element. The `arraySpecification` field is then optional.

For example the structure in section 6.4

```
<specie name="s" arraySpecification="grid"
    compartment="cell[x][y]" initialAmount="x == 0 ? 1e-10 : 0"/>
```

can be replaced with the equivalent structure

```
<specie name="s" compartment="cell" initialAmount="x == 0 ? 1e-10 : 0"/>
```

Elements of Specie arrays created this way can be referenced as described in previous sections.

If `arraySpecification` is present in this kind of structure the resulting specie array created is the combination of the compartment and species `arraySpecification` structures. For example the following structures defines a 3 dimension array of species where 2 dimensions are mapped across a grid of compartments:

```
<domain name="i" lowerBound="0" upperBound="5"/>
...
<arraySpecification name="vector">
```

```
    <listOfDimensions>
        <dimension symbol="i" domain="i"/>
    </listOfDimensions>
</arraySpecification>
...
<specie name="ss" arraySpecification="vector"
    compartment="cell" initialAmount="0"/>
```

The compartment array `cell` is defined as before.

7.2 Referencing species array elements

To compliment the above simplification we introduce a new operator '`.`' which has precedence between function operators and the '`*`' and '`/`' operators. The left hand operand of this operator is always a compartment (either a single compartment or a compartment array element). The right hand side is always a specie within that compartment. The operator returns the specie object.

If the compartment on the left hand side is an element of an array then it is assumed that the specie is also an array distributed across the compartments of the array. In which case the '`.`' operator returns the specie element corresponding to the left hand compartment array element.

Consider the reaction structure from section 6.4:

```
<reaction name="j" arraySpecification="connections"/>
    <listOfReactants>
        <specieReference specie="s[x1][y1]" stoichiometry="1"/>
    </listOfReactants>
    <listOfProducts>
        <specieReference specie="s[x2][y2]" stoichiometry="1"/>
    </listOfProducts>
    <kineticLaw formula="s[x1][y1] * 5"/>
</reaction>
```

This can be written as

```
<reaction name="j" arraySpecification="connections"/>
    <listOfReactants>
        <specieReference specie="cell[x1][y1].s" stoichiometry="1"/>
    </listOfReactants>
    <listOfProducts>
        <specieReference specie="cell[x2][y2].s" stoichiometry="1"/>
    </listOfProducts>
    <kineticLaw formula="cell[x1][y1].s * 5"/>
</reaction>
```

The '`.`' operator can be used with arrays of species which have more dimensions than the compartments in which they are located. For example consider the `ss` species array create in the second example in 7.1, this array can be referenced in the following way in a formula:

```
cell[x1][y1].ss[i]
```

This array proposal does not change any of the existing namespace rules of SBML: species located in different compartments cannot have the same name.

7.3 Issue

Given that the simplified species structure and the '`.`' operator is redundant should they be incorporated into SBML Level 2?

8 Array Math

We suggest that, in the context of this proposal, while SBML formulas must return a scalar value, intermediate values within formulas can be arrays. This section describes a set of proposed operators and functions that can be performed on arrays in formulas.

8.1 Operators

We propose that a subset of the matrix operators defined in MATLAB (MathWorks, 1998) are incorporated into SBML. An initial subset for Level 2 might be: `+`, `-`, `*`, `./`, `.*`, `.^` and `'`. Appendix A shows how these operators are integrated into SBML.

8.2 : Array Index Placeholder

In this proposal the character ':' can be used as a place holder for array indices. The result of using such a place holder is an array which is a slice of the original array. The precise definition of this operation is taken from MATLAB (MathWorks, 1998). SBML uses this operation in combination with the 'C' language notation for arrays so that if we consider a two dimensional array, `a`, then `a[:][:]` is equivalent to `a` and `a[i][:]` is equivalent to `a[i]` however there is no equivalent of `a[:][i]`. The form `a[:][i]` is supported by this proposal.

8.3 Built-In Array Functions

We propose the following built-in functions for matrix math:

```
sum(array)
```

This function returns the sum of all the elements of the given array

```
sum(symbol1, lowerBound1, upperBound1,...
    symboln, lowerBoundn, upperBoundn, expression)
```

This function's arguments consist of a sequence of triples followed by an expression. The triples sequence consist of 1 or more triples. Each triple consists of a symbol, which is a new constant integer symbol (i.e. not an expression) sharing the same namespace as species, compartment etc plus an upper and lower bound. The symbols only have scope within following the following arguments. The upper and lower bounds are truncated to integers.

This function simply runs through all the possible symbol values between the computed bounds. For each set of symbol values the final expression is evaluated. The function returns the sum of these final expression values.

```
product(array)
```

This function returns the product of all the elements of the given array

```
product(symbol1, lowerBound1, upperBound1,...
    symboln, lowerBoundn, upperBoundn, expression)
```

Similar to the multiple argument `sum` function: the only difference is that `product` returns the product of the final expression values.

```
map(array, function)
```

returns the array which results from the application of `function` to all the elements of the given array. `function` is a function name only. The corresponding function will take a single argument.

```
reduce(startValue, array, function)
```

This function computes a value x through the application of a function `function` to elements of the array `array`. The initial value of x is `startValue`. `function` takes two arguments: the current value of x and the next element in the array. `function` then returns the new value of x.

For example given the following structure

```
<function name="plus" formula="a + b">
    <listOfArguments>
        <argument name="a"/>
        <argument name="b"/>
    </listOfArguments>
</function>
```

then the expression

```
reduce(0, s, plus)
```

is equivalent to

```
sum(s)
```

In more complex functions the order in which the array elements are applied to the function is significant. `reduce` is computed as a series of nested loops in which the first dimension forms the outermost loop and the last dimension forms the innermost loop.

8.3.1 Issue:

The bound pairs in arguments of the `sum` and `product` functions might be better replaced by domain names.

9 Discussion: Combining arrays with Modularity

The proposed features described in this document could potentially overlap with possible features of the `Modularity` package. Arrays of submodel instances would be useful as would the designation of submodel arguments as constant or dynamic.

A Summary of Proposed Operators

Table 1 lists all the operators that are proposed in this document together with those proposed in Finney et al. (2002).

References

Finney, A., Gor, V., Mjolsness, E., and Bolouri, H. (2002). Systems Biology Markup Language (SBML) Level 2 Proposal: Miscellaneous Features.

Harbison, S. P. and Steele, G. L. (1995). *C: A Reference Manual*. Prentice-Hall.

Hucka, M., Finney, A., Sauro, H. M., and Bolouri, H. (2001). Systems Biology Markup Language (SBML) Level 1: Structures and facilities for basic model definitions. Available via the World Wide Web at `http://www.cds.caltech.edu/erato`.

MathWorks, T. (1998). *Using* MATLAB. MATLAB: The Language of Technical Computing. The MathWorks, Inc., Natik, MA.

Tokens	Operation	Class	Precedence	Associates
name	symbol reference	operand	10	n/a
(*expression*)	expression grouping	operand	10	n/a
$a[k]$	array subscript	postfix	10	left
$a[:]$	array slice	postfix	10	left
.	specie selection	postfix	10	left
$f(\ldots)$	function call	prefix	10	left
!	logical not	unary	9	right
′	matrix transpose	unary	9	right
−	negation	unary	9	right
^	power	binary	8	left
.^	matrix element power	binary	8	left
∗	scalar and matrix multiplication	binary	7	left
.∗	matrix element multiplication	binary	7	left
/	division	binary	7	left
./	matrix element division	binary	7	left
+	scalar and matrix element addition	binary	6	left
−	scalar and matrix element subtraction	binary	6	left
<	less than	binary	5	left
>	greater than	binary	5	left
>=	greater than or equal	binary	5	left
<=	less than or equal	binary	5	left
==	equality	binary	4	left
!=	inequality	binary	4	left
&&	logical and	binary	3	left
||	logical or	binary	2	left
? :	conditional	ternary	1	right

Table 1: *A table of the expression operators available in SBML, operators proposed in this document are shown in red, operators proposed in Finney et al. (2002) are shown in green. In the* **Class** *column, "operand" implies the construct is an operand, "prefix" implies the operation is applied to the following arguments, "unary" implies there is one argument, and "binary" implies there are two arguments. The values in the* **Precedence** *column show how the order of different types of operation are determined. For example, the expression* $a * b + c$ *is evaluated as* $(a * b) + c$ *because the* $*$ *operator has higher precedence. The* **Associates** *column shows how the order of similar precedence operations is determined; for example,* $a - b + c$ *is evaluated as* $(a - b) + c$ *because the* $+$ *and* $-$ *operators are left-associative. The precedence and associativity rules are taken from the C programming language (Harbison and Steele, 1995), except for the symbol ^, which is used in C for a different purpose.*