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

Andrew Finney, Victoria Gor, Ben Bornstein, Eric Mjolsness, Hamid Bolouri

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

April 19, 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 and in Finney et al. (2002).

2 Models

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

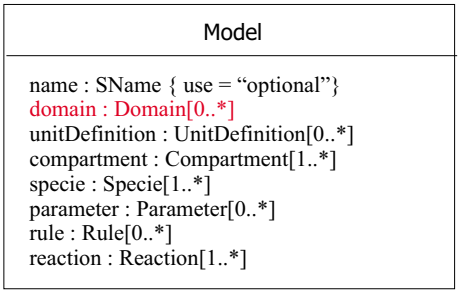

Figure 1: *The definition of the Model type*

`Domain` structures are described in section 3.

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 inclusive upper and lower bounds of `int` type and a set of symbols which have values over the domain. The bound values can be negative. The upper bound must be greater than the lower bound.

The following example shows a `Model` structure incorporating a `Domain` structure which ranges from -10 through to 10. The symbols `x` and `y` are defined to have values over the given domain.

```
<model name="tissue">
    <listOfDomains>
        <domain upperBound="10" lowerbound="-10">
            <symbol name="x"/>
            <symbol name="y"/>
        </domain>
```

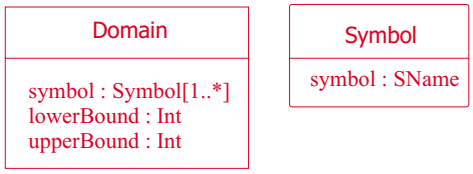

Figure 2: *The definition of the Domain type*

```
    </listOfDomains>
    ...
</model>
```

3.1 Constant Symbols and Expressions

The symbol field of a `Symbol` structure defines a symbol which can be used in numeric expressions. The scope of these symbols is defined in section 4. 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 constant symbols or a constant numeric values.

Constant expressions can then be used in place of most constant attribute values of type `double` throughout SBML although there are limitations on which specific expression they can be used in, see section 4. Constant symbols can occur in any numeric expression, for example a rate equation. As constant symbols are used in expressions they share the same namespace as other SBML structures such as species.

4 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. We propose that following SBML types, `Specie`, `Compartment`, `Reaction`, `Parameter` and `Rule` can be defined as arrays of objects.

To declare and operate on arrays we introduce the `[]` array operator, which is similar to the C array operator. As in C the array operator is prefixed by a symbol name and should contain a constant expression. Several array operators can be combined to indicate multiple dimensions. As in C there are 2 uses of the array operator: declaring arrays and accessing array elements.

4.1 Array Declaration

The array operator can be used in the `name` fields of `Specie`, `Compartment`, `Reaction` and `Parameter` structures and the object reference field of `Rule` structures to indicate that the given structure is an array rather than a single object. All the objects in the array have the properties described by the structures's attributes and substructures.

Structures that are declared as arrays share the same namespace as those structures that represent single objects.

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

```
<model name="simple">
    <listOfDomains>
        <domain upperBound="9" lowerbound="0">
            <listOfSymbols>
                <symbol name="x"/>
            </listOfSymbols>
        </domain>
    </listOfDomains>
```

```
    <listOfCompartments>
        <compartment name="cell[x]"/>
    </listOfCompartments>
    ...
</model>
```

4.1.1 Symbol Scope

Domain symbols used in the expressions contained in array declarations can be used in attributes of the same structure and sub structures of that structure. This is the only place in which these symbols can be used other than in array declarations.

The following example shows how a symbol can be used in numeric fields within the symbols scope:

```
<model name="ref">
    <listOfDomains>
        <domain upperBound="9" lowerbound="0">
            <listOfSymbols>
                <symbol name="x"/>
            </listOfSymbols>
        </domain>
    </listOfDomains>
    <listOfCompartments>
        <compartment name="cell[x]" volume="x * 0.2" />
    </listOfCompartments>
    ...
</model>
```

Notice that although constant symbol `x` varies between 9 and 0 this variance only defines the structure of the model so that `x` is not a parameter or variable of any simulation of the model.

4.1.2 Simple Sparse Arrays

Given the scheme described above it is possible to define sparse arrays. Array elements are only created once for each value of a domain symbol. The index of a created array element is the result of the constant expression contained in the array operator. In an array declaration involving more than one array operator, i.e. an declaration for a multi-dimensional array, each symbol only runs over its range once. For example consider the following model:

```
<model name="notsimple">
    <listOfDomains>
        <domain upperBound="9" lowerbound="0">
            <listOfSymbols>
                <symbol name="x"/>
                <symbol name="y"/>
            </listOfSymbols>
        </domain>
    </listOfDomains>
    <listOfCompartments>
        <compartment name="cell2D[x][y]"/>
        <compartment name="cellDiagonal[x][x]"/>
    </listOfCompartments>
    ...
</model>
```

In the model `notsimple` the array `cell2D` is square array containing 100 elements. `cellDiagonal` is a 2D array containing elements only on the diagonal.

The following example creates an array where elements occur at even locations:

```
<model name="even">
    <listOfDomains>
        <domain upperBound="9" lowerbound="0">
            <listOfSymbols>
                <symbol name="x"/>
                <symbol name="y"/>
```

```
            </listOfSymbols>
        </domain>
    </listOfDomains>
    <listOfCompartments>
        <compartment name="cell2D[2*x][2*y]"/>
    </listOfCompartments>
    ...
</model>
```

4.1.3 Declaring arrays of rules

Declaring arrays of rules is slightly different from the cases described above. Rule structures don't contain a `name` field that declares a new symbol instead they reference another structure. When declaring an array of rules the array operator is used in the object reference field i.e. the `specie`, `compartment` and `name` field for `SpecieConcentrationRule`, `CompartmentVolumeRule` and `ParameterRule` structures respectively. The symbol prefixing the array operator should be an array of the appropriate type e.g. a specie array for the `specie` field. The expressions enclosed in the array operator in the object reference field operate in the same way as described above except that the declaration uses those expressions to link back to the referenced array. For example the following model has a rule applied to an array of species.

```
<model name="rules">
    <listOfDomains>
        <domain upperBound="9" lowerbound="0">
            <listOfSymbols>
                <symbol name="x"/>
            </listOfSymbols>
        </domain>
    </listOfDomains>
    ...
    <listOfSpecies>
        <species name="s[x]" initialAmount="0.1"/>
    </listOfSpecies>
    ...
    <listOfRules>
        <specieConcentrationRule specie="s[x]" type="rate" formula="0.1"/>
    </listOfRules>
</model>
```

As rules do not declare symbols it is possible for more than one rule to be applied to the same array. For example consider the following example:

```
<model name="rules">
    <listOfDomains>
        <domain upperBound="9" lowerbound="0">
            <listOfSymbols>
                <symbol name="x"/>
            </listOfSymbols>
        </domain>
        <domain upperBound="5" lowerbound="0">
            <listOfSymbols>
                <symbol name="w"/>
            </listOfSymbols>
        </domain>
        <domain upperBound="9" lowerbound="6">
            <listOfSymbols>
                <symbol name="v"/>
            </listOfSymbols>
        </domain>
    </listOfDomains>
    ...
    <listOfSpecies>
        <species name="s[x]" initialAmount="0.1" compartment="cell"/>
    </listOfSpecies>
    ...
    <listOfRules>
        <specieConcentrationRule specie="s[w]" type="rate" formula="0.1"/>
        <specieConcentrationRule specie="s[v]" type="rate" formula="0.2"/>
```

```
    </listOfRules>
</model>
```

4.1.4 Rules applied to a whole arrays or slices of arrays

Rules can be defined which apply to the whole of an array, or slices of arrays, in which the formula expression returns a whole array value rather than a single value to be inserted into each array element. This kind of rule is declared simply by not applying the array operator to an array symbol, or using the array slice operator (see section 7.2), in the rule object reference field. Whole matrix operators are described in more detail in section 7. For now assume that we can, for instance, multiply arrays then consider the example model:

```
<model name="rules">
    <listOfDomains>
        <domain upperBound="9" lowerbound="0">
            <listOfSymbols>
                <symbol name="x"/>
            </listOfSymbols>
        </domain>
    </listOfDomains>
    ...
    <listOfSpecies>
        <species name="s1[x]" initialAmount="0.1"/>
        <species name="s2[x]" initialAmount="0.2"/>
    </listOfSpecies>
    ...
    <listOfRules>
        <specieConcentrationRule specie="s1" type="rate" formula="s1 * s2"/>
    </listOfRules>
</model>
```

The rule for `s1` defines that the rate of change of the values of the array of `s1` is the product of the matrices `s1` and `s2`.

4.1.5 Issues

- It is possible avoid the use of the array operator in the name field for array declarations and instead use XML structures instead. This would mean that a SBML parser would avoid having to parse the content of the `name` field. For example the following model

```
<model name="simple">
    <listOfDomains>
        <domain upperBound="9" lowerbound="0">
            <listOfSymbols>
                <symbol name="x"/>
            </listOfSymbols>
        </domain>
    </listOfDomains>
    <listOfCompartments>
        <compartment name="cell[x]"/>
    </listOfCompartments>
    ...
</model>
```

  can be recast as:

```
<model name="simple">
    <listOfDomains>
        <domain upperBound="9" lowerbound="0">
            <listOfSymbols>
                <symbol name="x"/>
            </listOfSymbols>
        </domain>
    </listOfDomains>
    <listOfCompartments>
        <compartment name="cell">
```

```
            <listOfDimensions>
                <dimension formula="x"/>
            </listOfDimensions>
        </compartment>
    </listOfCompartments>
    ...
</model>
```

- We could eliminate simple sparse arrays by restricting the expressions inside array operators used in declarations to be just symbols rather than constant expressions and by forcing the symbol in each dimension to be unique within the declaration.

4.2 Array Element Reference

In this section mechanisms for accessing specific elements of an array are described. In numeric expressions the array operator is syntactically equivalent to a function call and returns the value of the indexed array element. The array operator can also be used in any field which refers to a `Specie`, `Compartment`, `Reaction` or `Parameter` by name, for example the `specie` field on a `SpecieReference` structure can contain an array operator instead of an `SName` type. In these fields the array operator is used to refer to specific element.

The 'array' operand of an array operator must be the name of a structure which has been declared as an array. We suggest that for SBML Level 2, 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.

We can use the array operator to create an array of species distributed across an array of compartments:

```
<model name="ref">
    <listOfDomains>
        <domain upperBound="9" lowerbound="0">
            <listOfSymbols>
                <symbol name="x"/>
            </listOfSymbols>
        </domain>
    </listOfDomains>
    <listOfCompartments>
        <compartment name="cell[x]"/>
    </listOfCompartments>
    <listOfSpecies>
        <species name="s[x]" compartment="cell[x]"/>
    </listOfSpecies>
    ...
</model>
```

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

4.2.1 Issue

We could use a new element to reference elements from an object reference field instead of putting the array operator inside the object reference fields. For example the above example can be re-formulated as:

```
<model name="ref">
    <listOfDomains>
        <domain upperBound="9" lowerbound="0">
            <listOfSymbols>
                <symbol name="x"/>
            </listOfSymbols>
        </domain>
    </listOfDomains>
    <listOfCompartments>
        <compartment name="cell">
            <listOfDimensions>
                <dimension formula="x"/>
            </listOfDimensions>
```

```
        </compartment>
    </listOfCompartments>
    <listOfSpecies>
        <species name="s" compartment="cell">
            <listOfDimensions>
                <dimension formula="x"/>
            </listOfDimensions>
            <listOfCompartmentElementReferences>
                <dimension formula="x"/>
            </listOfCompartmentElementReferences>
        </species>
    </listOfSpecies>
    ...
</model>
```

We will still need to be able to parse the array operator in numeric expressions.

4.2.2 Example of using element references in both numeric and object reference fields

The following example shows how the array operator can be used in a numeric expression.

```
<model name="ref">
    <listOfDomains>
        <domain upperBound="9" lowerbound="0">
            <listOfSymbols>
                <symbol name="x"/>
            </listOfSymbols>
        </domain>
    </listOfDomains>
    <listOfCompartments>
        <compartment name="cell"/>
    </listOfCompartments>
    <listOfSpecies>
        <species name="s1[x]" compartment="cell"/>
        <species name="s2[x]" compartment="cell"/>
    </listOfSpecies>
    <listOfReactions>
        <reaction name="r[x]">
            <listOfReactants>
                <specieReference specie="s1[x]"/>
            </listOfReactants>
            <listOfProducts>
                <specieReference specie="s2[x]"/>
            </listofProducts>
            <kineticLaw formula="s1[x] * 0.1"/>
        </reaction>
    </listOfReactions>
</model>
```

4.2.3 Referring to elements of a sparse array

The index operand to an array operator must refer to an element which exists in an array. For example the following model is inconsistent:

```
<model name="ref">
    <listOfDomains>
        <domain upperBound="9" lowerbound="0">
            <listOfSymbols>
                <symbol name="x"/>
            </listOfSymbols>
        </domain>
    </listOfDomains>
    <listOfCompartments>
        <compartment name="cell[x * 2]"/>
    </listOfCompartments>
    <listOfSpecies>
        <species name="s" compartment="cell[1]"/>
```

```
    </listOfSpecies>
    ...
</model>
```

The element `cell[1]` has not been declared.

5 Conditional Sparse Arrays and Connections

5.1 Conditional Sparse Arrays

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 the structures `Specie`, `Compartment`, `Reaction` and `Parameter` and `Rule` include an additional `elementExists` field. This field should only occur when the structure is declared as an array. `elementExists` contains a constant expression which defines whether an array element actually occurs at a given position in the array. 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)).

We propose that for SBML Level 2 that `elementExists` values are constant expressions.

The default value of `elementExists`, 1, ensures that by default the shape of the array specification is determined just by the array operator alone.

Apart from the `elementExists` field, constant symbols only have values for those elements that exist in the array 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 a value to which the `elementExists` expression returns a non-zero value.

5.1.1 Issue

To enable dynamic structures in SBML we could allow the `elementExists` expression to be dynamic. This would allow the model to change dynamically during the simulation. Perhaps other more explicit dynamic restructuring of a model should be considered in SBML.

5.2 Conditional Sparse Array Example

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

```
<model name="starfish">
    <listOfDomains>
        <domain upperBound="9" lowerbound="0"/>
            <symbol name="x"/>
            <symbol name="y"/>
        </domain>
    </listOfDomains>
    <compartment name="cell[x][y]" elementExists="y <= x">
    ...
</model>
```

5.3 Using Conditional Sparse Arrays to Represent Connection schemes

We can use a proposed sparse array to represent the connections between elements of another array. This is shown in the following example, in which `grid` is a 2 dimensional array of compartments and, `connections` is a sparse 4 dimensional array of reactions 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 upperBound="9" lowerbound="0">
            <symbol name="x"/>
            <symbol name="y"/>
            <symbol name="x1"/>
            <symbol name="x2"/>
            <symbol name="y1"/>
            <symbol name="y2"/>
        </domain>
    </listOfDomains>
    <listOfCompartments>
        <compartment name="grid[x][y]"/>
    </listOfCompartments>
    <listOfSpecies>
        <species name="s[x][y]" initialAmount="0.1" compartment="grid[x][y]"/>
    </listOfSpecies>
    <listOfReactions>
        <reaction
                name="connections[x1][y1][x2][y2]"
                elementExists="abs(x2 - x1) == 1 || abs(y2 - y1) == 1">

            <listOfReactants>
                <specieReference specie="s[x1][y1]"/>
            </listOfReactants>
            <listOfProducts>
                <specieReference specie="s[x2][y2]"/>
            </listOfProducts>
            <kineticLaw formula="s[x1][y1] * 0.1"/>

        </reaction>
    </listOfReactions>
</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.

Often its convenient to use `Function` structures in the `elementExists` field. The following example, uses a `Function` structure to define an more explicit connection scheme.

```
<model name="spotty">
    <listOfFunctions>
        <function name="explicit"
            formula="(x==1 && y==1) || (x==3 && y==3) || (x==0 && y==9)">
            <listOfArguments>
                <argument name="x"/>
                <argument name="y"/>
            </listOfArguments>
        </function>
    </listOfFunctions>
    <listOfDomains>
        <domain upperBound="9" lowerbound="0">
            <symbol name="a"/>
            <symbol name="b"/>
        </domain>
    </listOfDomains>
    ...
    <listOfReactions>
        <reaction name="spots[a][b]" elementExists="explicit(a, b)">
```

```
            ...
        </reaction>
    </listOfReactions>
</model>
```

6 Simplified Array Structures for Related Objects

It is possible to incorporate a simplified mechanism for creating species, compartment and parameter arrays and for referencing the elements of those arrays.

6.1 Implied Species Arrays

In this proposal the `compartment` field of a `Specie` structure can consist of a name of an array of compartments, without an array operator. 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 specie array doesn't have to be explicitly declared.

For example given

```
<domain lowerBound="0" upperBound="9"/>
    <listOfSymbols>
        <symbol name="x"/>
        <symbol name="y"/>
    </listOfSymbols>
</domain>
...
<compartment name="cell[x][y]"/>
```

the structure

```
<specie name="s[x][y]" 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 the specie `name` field includes an array operator in this kind of structure the resulting specie array created is the combination of the compartment and species arrays. For example the following structures defines a 3 dimension array of species where 2 dimensions are mapped across a grid of compartments:

```
<listOfDomains>
    <domain lowerBound="0" upperBound="9"/>
        <listOfSymbols>
            <symbol name="x"/>
            <symbol name="y"/>
        </listOfSymbols>
    </domain>
    <domain lowerBound="0" upperBound="5"/>
        <listOfSymbols>
            <symbol name="i"/>
        </listOfSymbols>
    </domain>
<listOfDomains>
...
<compartment name="cell[x][y]"/>
...
<specie name="ss[i]" compartment="cell" initialAmount="0"/>
```

6.2 Implied Compartment Arrays

In SBML Level 1 it is possible to specify nested compartments. For example the fragment:

```
<listOfCompartments>
    <compartment name="a">
    <compartment name="b" outside="a">
</listOfCompartments>
```

Defines a compartment `b` which is enclosed by `a`.

We can extend the structure defined in section 6.1 to cover nested compartments.

For example given

```
<domain lowerBound="0" upperBound="5"/>
    <listOfSymbols>
        <symbol name="i"/>
    </listOfSymbols>
</domain>
```

then

```
<listOfCompartments>
    <compartment name="a[i]"/>
    <compartment name="b" outside="a"/>
</listOfCompartments>
```

is equivalent to

```
<listOfCompartments>
    <compartment name="a[i]"/>
    <compartment name="b[i]" outside="a[i]"/>
</listOfCompartments>
```

6.3 Implied Parameter Arrays

We can apply the above concept to parameters if we introduce a new `SName` field, `foreach`, to the parameter structure. This field allows us to reference any other symbol from this structure and thus attach a parameter to each element of the referenced array.

For example the model:

```
<listOfDomains>
    <domain lowerBound="0" upperBound="5"/>
        <listOfSymbols>
            <symbol name="i"/>
        </listOfSymbols>
    </domain>
<listOfDomains>
...
<compartment name="cell[i]"/>
...
<parameter name="p" foreach="cell" value="0"/>
...
```

is equivalent to

```
<listOfDomains>
    <domain lowerBound="0" upperBound="5"/>
        <listOfSymbols>
            <symbol name="i"/>
        </listOfSymbols>
    </domain>
<listOfDomains>
...
```

```
<compartment name="cell[i]"/>
...
<parameter name="p[i]" value="0"/>
...
```

6.4 Referencing nested 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 an expression representing a containing object. The right hand side is a symbol representing an object within the left hand object.

If the object on the left hand side is an element of an array then it is assumed that the right hand object is also an array distributed across the elements of the array. In which case the '.' operator returns the element corresponding to the left hand array element.

Consider the following model:

```
<model>
    <listOfDomains>
        <domain lowerBound="0" upperBound="5"/>
            <listOfSymbols>
                <symbol name="i"/>
            </listOfSymbols>
        </domain>
    <listOfDomains>
    <listOfCompartments>
        <compartment name="cell[i]"/>
    </listOfCompartments>
    <listOfSpecies>
        <species name="s1[i]" compartment="cell[i]"/>
        <species name="s2[i]" compartment="cell[i]"/>
    </listOfSpecies>
    <reaction name="j[i]">
        <listOfReactants>
            <specieReference specie="s1[i]" stoichiometry="1"/>
        </listOfReactants>
        <listOfProducts>
            <specieReference specie="s2[i]" stoichiometry="1"/>
        </listOfProducts>
        <kineticLaw formula="s1[i] * 5"/>
    </reaction>
</model>
```

This can be written as

```
<model>
    <listOfDomains>
        <domain lowerBound="0" upperBound="5"/>
            <listOfSymbols>
                <symbol name="i"/>
            </listOfSymbols>
        </domain>
    <listOfDomains>
    <listOfCompartments>
        <compartment name="cell[i]"/>
    </listOfCompartments>
    <listOfSpecies>
        <species name="s1" compartment="cell"/>
        <species name="s2" compartment="cell"/>
    </listOfSpecies>
    <reaction name="r[i]">
        <listOfReactants>
            <specieReference specie="cell[i].s1" stoichiometry="1"/>
        </listOfReactants>
        <listOfProducts>
            <specieReference specie="cell[i].s2" stoichiometry="1"/>
```

```
        </listOfProducts>
        <kineticLaw formula="cell[i].s1 * 5"/>
    </reaction>
</model>
```

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 following model:

```
<model>
    <listOfDomains>
        <domain lowerBound="0" upperBound="5"/>
            <listOfSymbols>
                <symbol name="i"/>
            </listOfSymbols>
        </domain>
        <domain lowerBound="0" upperBound="10"/>
            <listOfSymbols>
                <symbol name="j"/>
            </listOfSymbols>
        </domain>
    <listOfDomains>
    <listOfCompartments>
        <compartment name="cell[i]"/>
    </listOfCompartments>
    <listOfSpecies>
        <species name="s1[j]" compartment="cell"/>
        <species name="s2[j]" compartment="cell"/>
    </listOfSpecies>
    <reaction name="r[i][j]">
        <listOfReactants>
            <specieReference specie="cell[i].s1[j]" stoichiometry="1"/>
        </listOfReactants>
        <listOfProducts>
            <specieReference specie="cell[i].s2[j]" stoichiometry="1"/>
        </listOfProducts>
        <kineticLaw formula="cell[i].s1[j] * 5"/>
    </reaction>
</model>
```

It is possible to create a model with more than one level of nesting. For example:

```
<model>
    <listOfDomains>
        <domain lowerBound="0" upperBound="5">
            <listOfSymbols>
                <symbol name="i"/>
            </listOfSymbols>
        </domain>
    </listOfDomains>
    <listOfCompartments>
        <compartment name="a[i]"/>
        <compartment name="b" outside="a"/>
    </listOfCompartments>
    </listOfSpecies>
        <specie name="s1" compartment="b" initialAmount="0.1"/>
        <specie name="s2" compartment="b" initialAmount="0.1"/>
    </listOfSpecies>
    <listOfReactions>
        <reaction name="r[i]">
            <listOfReactions>
                <specieReference specie="a[i].b.s1"/>
            </listOfReactions>
            <listOfProducts>
                <specieReference specie="a[i].b.s2"/>
            </listOfProducts>
        </reaction>
    </listOfReactions>
</model>
```

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

6.5 Issue

- The simplified structures and the '.' operator described in this section are redundant. Should they be incorporated into SBML Level 2?
- The above definition implies that elements of an array defined using the implied forms described in this section can be referenced by either the '.' operator or indexing the array directly. For example given:

```
<listOfDomains>
    <domain upperBound="0" lowerBound="9">
        <symbol name="x"/>
    </domain>
    <domain upperBound="0" lowerBound="3">
        <symbol name="y"/>
    </domain>
</listOfDomains>
<listOfCompartments>
    <compartment name="cell[x]"/>
</listOfCompartments>
<listOfSpecies>
    <specie name="s[y]" compartment="cell"/>
</listOfSpecies>
```

  elements of `s` can be referenced in two equivalent ways:

```
<specieReference specie="s[x][y]"/>
```

  or

```
<specieReference specie="cell[x].s[y]"/>
```

  Is this a good idea?

- By using the same domain symbol twice in the same nested structure can have confusing results. For example given:

```
<listOfDomains>
    <domain upperBound="0" lowerBound="9">
        <symbol name="x"/>
    </domain>
</listOfDomains>
<listOfCompartments>
    <compartment name="cell[x]"/>
</listOfCompartments>
```

  then

```
<listOfSpecies>
    <specie name="s[x]" compartment="cell"/>
</listOfSpecies>
```

  is equivalent to

```
<listOfSpecies>
    <specie name="s[x][x]" compartment="cell[x]"/>
</listOfSpecies>
```

  This means that a sparse array of species has been created. Should a parse detect this as an error or simply warn the user?

7 Array Math

This section describes a set of proposed operators and functions that can be performed on arrays in formulas.

7.1 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.

7.2 : 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). For example given a 2 dimensional array `x` the expression `x[:][i]` returns the 1 dimensional array of the ith row of the array.

SBML uses the `:` 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]`.

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

This function's arguments consist of an expression followed by a sequence of triples. 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.

```
sumOverDomain(expression, domainSymbol1, ... domainSymboln)
```

This function's arguments consist of an expression followed by a sequence of domain symbols. The domain symbols only have scope in the expression argument. The domain symbols cannot be any of the domain symbols already used in the scope outside the function call.

This function simply runs through all the possible symbol values with the domains. 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(expression, symbol1, lowerBound1, upperBound1,...
    symboln, lowerBoundn, upperBoundn)
```

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

```
productOverDomain(expression, domainSymbol1, domainSymboln)
```

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

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

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

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. The default value of `startValue` is zero.

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(plus, s)
```

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.

```
argmin(expression, symbol)
```

returns the potential value of previously declared `symbol` which clauses `expression` to return the minimum value. `argmin` returns a value but doesn't change `symbol`. `symbol` can be a scalar or array value. When `symbol` is an array then the set of values that minimize `expression` is returned.

```
argmax(expression, symbol)
```

Similar to `argmin` except that the function returns the value that a maximizes `expression`.

7.4 Issues

- Some of the above matrix functions are redundant should they be included in SBML?
- Given that `argmin` and `argmax` can't be mapped to SBML Level 1 should they be included in Level 2? If yes should they be part of a seperate package (called `optimization`)?

8 Complex Example using Array Math

The following example employs array math.

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level2" version="1" level="2">
<model name="GRN_Equations">

    <listOfFunctions>
        <function name="CM"
            formula=
                "abs(x2-x1) <= 0.2 && abs(y2-y1) <= 0.2 && ( (x2 != x1) || (y2!=y1) )">
            <listOfArguments>
                <argument name="x1"/>
                <argument name="y1"/>
                <argument name="x2"/>
                <argument name="y2"/>
            </listOfArguments>
        </function>
        <function name="sigmoid" formula = "0.5* (1.0+ (x/sqrt(1.0+x*x)))">
            <listOfArguments>
                <argument name="x"/>
            </listOfArguments>
        </function>
    </listOfFunctions>


    <listOfDomains>
      <domain upperBound="8" lowerBound="0">
        <listOfSymbols>
          <symbol name="ic"/>
          <symbol name="I"/>
          <symbol name="K"/>
          <symbol name="i_cnt"/>
          <symbol name="j_cnt"/>
        </listOfSymbols>
      </domain>

      <domain upperBound="2" lowerBound="0">
        <listOfSymbols>
          <symbol name="is"/>
          <symbol name="js"/>
          <symbol name="J"/>
          <symbol name="L"/>
        </listOfSymbols>
      </domain>

      <domain upperBound="2" lowerBound="1">
        <listOfSymbols>
          <symbol name="i_crd"/>
        </listOfSymbols>
      </domain>
    </listOfDomains>

    <listOfCompartments>
      <compartment name="cell[ic]"/>
    </listOfCompartments>

    <listOfSpecies>
      <specie name="ss[is]" compartment="cell" initialAmount="0"/>
    </listOfSpecies>

    <listOfParameters>
      <parameter name="h[is]"        value="1"/>
      <parameter name="lambda[is]"   value="1"/>
      <parameter name="tau[is]"      value="1"/>
      <parameter name="Tin[is][js]"  value="1"/>
      <parameter name="TOne[is][js]" value="1"/>
      <parameter name="Tout[is][js]" value="1"/>
      <parameter name="TTwo[is][js]" value="1"/>
      <parameter name="CS[i_cnt][j_cnt]" value="0"/>
      <parameter name="crd[i_crd]" foreach="cell"
                 value="i_crd == 1 ?
```

```
                          (ic div 3 + 1) * 0.1 :
                          (ic mod 3 + 1) * 0.1"/>

      <parameter name="mass" foreach="cell" value="1"/>
      <parameter name="init_eq_dist[i_cnt][j_cnt]"
                 value="( (cell[i_cnt].crd[1]-cell[j_cnt].crd[1])^2 +
                          (cell[i_cnt].crd[2]-cell[j_cnt].crd[2])^2 ) / 2"/>
    </listOfParameters>

    <listOfRules>
        <specieConcentrationRule specie="cell[ic].ss[is]" type="rate"
            formula = "(
                      (-lambda[is]*cell[ic].ss[is]) +
                      sigmoid(
                        h[is] +
sumOverDomain(cell[ic].ss[L]*Tin[is][L], L) +
sumOverDomain(CM(cell[ic].crd[1], cell[ic].crd[2], cell[K].crd[1], cell[K].crd[2]) *
                sumOverDomain(cell[K].ss[L] * Tout[is][L], L), K) +

sumOverDomain(CM(cell[ic].crd[1], cell[ic].crd[2], cell[K].crd[1], cell[K].crd[2]) *
                 sumOverDomain(
                              cell[K].ss[I]  *
                              cell[ic].ss[I] *
                              Tone[J][is]    *
                              TTwo[is][J]),
                              I,
                              J)
                       )
                    ,K) / tau[is]" />

        <parameterRule parameter="mass[ic]"
                       formula="sumOverDomain(cell[ic].ss[L], L)+1"/>

        <parameterRule name="CS[i_cnt][j_cnt]"
                       formula="init_eq_dist[i_cnt][j_cnt] *( mass[i_cnt]^-3+
                                                              mass[j_cnt]^-3)"/>
        <parameterRule parameter="crd"
                       formula="
                       argmin(1/2 *
                         sumOverDomain(
                           CM(cell[I].crd[1],cell[I].crd[2],cell[J].crd[1],cell
[J].crd[2]) * ( (((cell[I].crd[1]-cell[J].crd[1])^2 +
(cell[I].crd[2]-cell[J].crd[2])^2)^-2)-CS[I][J])^2, I, J), crd )"/>

    </listOfRules>
</model>
</sbml>
```

This model doesn't contain any reactions. We're assuming that SBML Level 2 will allow models to contain no reactions.

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).

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.*

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.