
A Unified Approach for Representing Structurally-Complex Models in SBML Level 3

Exploring the possibility of amalgamating several SBML Level 3 package proposals,
based on the data model of the Simile modelling environment

Robert Muetzelfeldt
r.muetzelfeldt@ed.ac.uk

April 2010

Extended abstract

The aim of this note is to explore a unified approach to handling several of the proposed extensions to the SBML Level 3 Core specification. The approach is illustrated with reference to Simile, a modelling environment which appears to have most of the capabilities of the various SBML Level 3 package proposals which deal with model structure. Simile (<http://www.simulistics.com>) is a visual modelling environment for continuous systems modelling which includes the ability to handle complex disaggregation of model structure, by allowing the modeller to specify classes of object and the relationships between them.

The note is organised around the 6 packages listed on the SBML Level 3 Proposals web page (http://sbml.org/Community/Wiki/SBML_Level_3_Proposals) which deal with model structure, namely **comp**, **arrays**, **spatial**, **geom**, **dyn** and **multi**. For each one, I consider how the requirements which motivated the package can be handled using Simile's unified approach. Although Simile has a declarative model-representation language (in both Prolog and XML syntax), I use Simile diagrams and equation syntax throughout, since this is more compact and readable than large chunks of XML.

For the **comp** (Hierarchical Model Composition) package, I show that Simile can represent the hierarchical structure of models through the nesting of submodels. Simile's model-representation language does not support inclusion-by-reference. However, the Simile environment does support plug-and-play modularity, aided by an interface specification file which stores the connections between variables in a submodel and the rest of the model.

The requirements of the **arrays** (Arrays and Sets) package are handled by two Simile features. The first is that Simile supports n-dimensional array variables, and has its own array language for processing these. The second is that Simile allows submodels to be declared as being multiple-instance, which handles the various requirements for "arrays of things" as presented in the arrays package. This section includes an implementation of the sample use-case model (the 'activator' model) included in the documentation for this package.

The **spatial** (Spatial Diffusion) and **geom** (Geometry) packages are treated together, because of the lack of documentation on the former and the considerable overlap between the two concepts. Somewhat paradoxically, Simile has no spatial/geometric constructs built in, even though it was originally developed for spatial modelling (in fact, for modelling agro-forestry systems in spatial terms). Instead, the modeller describes a system in terms of:

- objects (which can be interpreted as corresponding to spatial units, such as 'grid square');
- attributes for these objects (such as x and y coordinates), and
- relationships between the objects (which can be interpreted as spatial relationships, such as 'is_next_to').

This enables a wide variety of 1D, 2D and 3D spatial configurations to be modelled, as well as geometry-based behaviours, such as the movement of particles on a 3D surface. But the onus is on the modeller to capture spatial semantics, rather than using pre-defined language constructs.

In the section on the **dyn** (Dynamic Structures) package, I consider two aspects of this topic. The first deals with modelling a dynamically-changing number of entities. Simile has a built-in mechanism for handling this, since a submodel can have a dynamically-changing number of instances, and symbols are provided for specifying the rate of creation and destruction of instances. The second deals with dynamically-changing model structure: for example, switching between two alternative calculation pathways during the course of a simulation. I show that Simile supports this by allowing a submodel to be 'conditional', with its existence at any point in time dependent on the value of a Boolean switch.

The requirements of the **multi** (Multi-state Multi-component Species) package are met by using two Simile submodels. The first is a multiple-instance submodel with one instance for each state (i.e. separate species) of the species type. The second defines the particular states, both as substrate and product, involved in a particular reaction, using a Boolean expression which corresponds directly to the Selector described in this package. This case is revealing in that Simile was not designed to handle this class of problem, yet its generic nature allowed a solution to be developed, and in a fairly intuitive way.

The conclusion is that Simile can indeed meet most of the requirements of these various packages, using a generic set of constructs - basically, the multiple-instance submodel, the concept of a relationship (association) between submodels, and array variables. This suggests the possibility of having a single SBML Level 3 extension package similar to the Simile data model, rather than a series of separate packages. Such an approach has a number of potential advantages and disadvantages compared with having the current set of discrete packages. These are discussed in some depth, but no doubt members of the SBML community will have additional thoughts.

Contents

1 Introduction.....	3
1.1 SBML Level 3.....	3
1.2 Aim and approach.....	3
1.3 Simile.....	4
1.4 Challenges in preparing this document.....	8
2 The SBML Level 3 extension packages.....	9
2.1 Package 'comp': Hierarchical Model Composition	9
2.2 Package 'arrays': Arrays & sets	11
2.3 Packages 'spatial' and 'geom': Spatial diffusion and Geometry	17
2.4 Package 'dyn': Dynamic structures	21
2.5 Package 'multi': Multi-state, multi-component species	24
3 Summary of SBML Level 3 packages and how they would be handled in Simile.....	29
4 Discussion and conclusions.....	30
4.1 Assessment.....	31
4.2 What are the implications for SBML Level 3?.....	32
4.3 What next?.....	32

1 Introduction

1.1 SBML Level 3

SBML (the Systems Biology Markup Language) is an XML-based language for representing models of biological processes (<http://www.sbml.org>). It is widely-used standard, with some 130 software tools claiming some level of support for it. The Biomodels database (<http://www.ebi.ac.uk/biomodels-main>) contains some 450 models represented in SBML, over half of which are curated (checked against the results presented in a paper describing the models).

Up to SBML Level 2 Version 4, the specification consisted of a single document covering the whole language. In response to various proposals for extending the language, SBML Level 3 consists of a Core specification (first released in Dec 2009), and a number of extensions, called 'packages'. The SBML community maintains a list of proposed packages (http://sbml.org/Community/Wiki/SBML_Level_3_Proposals), which vary from a brief description of an idea, a proposal document, to a fully-fledged draft specification.

Some of these packages relate to what might be called structural aspects of a model: for example, hierarchical composition, multiple entities, spatial disaggregation, and array variables (**Table 1**). This document focuses on these packages: the Section heading gives the Section in this document where that package is analysed.

Section	Short label	Full name	Description	Status (from package web pages)
2.1	comp	Hierarchical model composition	A means for defining how a model is composed from other models.	One active proposal: Hoops <i>et al</i> , Sept 2007. Preceded by several presentations, documents and a workshop, dating from 2002.
2.2	arrays	Arrays and sets	Support for expressing arrays or sets of things. Example: an array of identical compartments.	Two active proposals: Finney <i>et al</i> , Sept 2003; Shapiro <i>et al</i> , Dec 2004. Preceded by various documents, workshops <i>etc</i> dating back to 2000.
2.3	spatial	Spatial diffusion	Support for describing processes that involve a spatial component.	Brief notes only.
2.3	geom	Geometry	Object structures for describing one-, two- and three-dimensional characteristics of SBML entities. Example: the shape of a three-dimensional compartment.	No proposal, but some quite detailed notes, from 2008/09. Note that each page links to another with more information.
2.4	dyn	Dynamic structures	Support for creating and destroying entities during a simulation.	No proposal, no documents etc, only brief two-paragraph outline from 2008.
2.5	multi	Multi-state, multi-component species	Object structures for representing entity pools with multiple states and composed of multiple components, and reaction rules involving them.	One active proposal: Le Novère <i>et al</i> , March 2010. Various documents, earlier proposals, notes <i>etc</i> from 2001.

Table 1. The SBML Level 3 packages analysed in this document. Note that the **spatial** and **geom** packages are considered together, because of the large amount of overlap between them.

1.2 Aim and approach

The **aim of this document** is to explore the possibility of handling the requirements of these 6 packages with a single, unified approach, rather than with 6 separate approaches. If this turns out to be both possible and

desirable, then that suggests that these 6 packages could be replaced by a single one which covers the full range of requirements. This seems desirable for a number of reasons:

First, and most obviously, less effort would be needed to define SBML Level 3 extensions, since only one package would be required, rather than several.

Second, overlap between packages would be eliminated, avoiding the possibility of the same concept being modelled in different ways in different models.

Third, the demands on developers of SBML Level 3-compliant software would be greatly reduced. If they wished to claim that they fully supported the SBML Level 3 functionality, they would only need to invest in handling one generic specification, rather than a variety of separate specifications.

Fourth, the problem of fragmentation of the the SBML-compliant software base would be reduced. With a number of separate specifications, there is the danger that any one vendor would choose to support only one or two of the packages. Hence, any one model would only be processable by some (possibly quite small) subset of "SBML-supporting" tools. I appreciate that the whole point of the SBML Level 3 development process, based on the concept of discrete packages, is to allow just such selective support by tool developers. I nevertheless maintain that, if possible, it is better to avoid restricting the number of tools that can handle any one model.

Finally, it should be easier for people to understand each other's models. If there are a number of packages around, then a scientist would have to fully understand each package in order to be able to understand the full range of SBML models. If there is only one package, then the amount of learning is reduced, even if this package itself takes more time to master.

The thing that brings me to undertake this investigation is my experience in working with Simile (<http://www.simulistics.com>), a modelling environment which has a high degree of expressiveness: that is, it is capable of representing a wide range of concepts used in modelling complex systems. This includes hierarchical, spatial and individual-based modelling. It achieves this through a unified and generic data model, allowing a wide range of situations to be modelled using a small number of model element types.

So, the hypothesis that I will be exploring is that Simile is capable of addressing the requirements of each of the 6 packages listed in **Table 1**.

To do this, I will first give a brief introduction to Simile itself (Section 1.3), and then (Section 2) consider each of the 6 packages in turn. For each package, I will outline the rationale for it (as given by the package proposers), and give examples of how the specification itself and/or sample use cases can be represented in Simile. Section 3 summarises the results of these analyses in a single table, in order to display the unified way in which Simile handles the various requirements. Finally, Section 4 assesses the results of this exercise, and considers possible implications for SBML Level 3.

1.3 Simile

Simile (<http://www.simulistics.com>) is a visual modelling environment developed to meet the needs of ecological and environmental research.

***Disclosure.** Simile is a commercial product. I am a Director, occasional employee and share-holder in Simulistics, the company which develops and markets Simile. The views expressed here are my own and not those of the company, and are offered in good faith to reflect my understanding of Simile and its capabilities at the time of writing.*

Simile is characterised by two main features:

- continuous-systems modelling in terms of System Dynamics (stock-and-flow) modelling;
- the representation of complex model structure in terms of classes of object, associations between classes and array variables.

1.3.1 Continuous-systems modelling in Simile

Simile supports **System Dynamics** (stock-and-flow) modelling (http://en.wikipedia.org/wiki/System_dynamics). System Dynamics is, like the notation for biological reactions, a way of describing a model based on differential equations in a form that is more accessible for practitioners in a particular domain. In biology, researchers like to talk about reaction pathways. In ecological and environmental research, it is natural to describe a system in terms

of stocks of material (carbon, water, nutrients, populations), and flows in to, out of and between these stocks (what are generally referred to as 'transport processes' within the SBML community), so this is a natural notation to use. However, we should not forget the very close correspondence between the two notations: SBML models can be re-expressed as differential equation models or (equivalently) as models for the amount (stock) of the various chemical species involved. Conversely, System Dynamics models can be converted into SBML, using the `<sbml:rateRule>` element for state variables (stocks), and the `<sbml:assignmentRule>` element for flows and intermediate variables.

1.3.2 Representation of complex model structure in Simile

Unlike most other System Dynamics software, Simile provides a very expressive notation for representing **complex substructure** (disaggregation) in models. For example, a Simile model can contain:

- various component models, organised hierarchically, nested to any depth;
- a population of individuals, where the number of individuals can change dynamically over time;
- a spatial submodel, consisting of a large number of spatial units (e.g. grid squares, hexagons, polygons...);
- age/size/sex classes, or indeed classes based on any criterion the modeller wants;
- array variables, of any number of dimensions.

Simile achieves this by a simple but powerful mechanism: the submodel. The submodel can be simply a mechanism for dividing up visually a large model. In this case, it has no mathematical significance: it is simply being used to display the structure of the model in a meaningful way, and for supporting modular modelling (allowing one submodel to be easily swapped with an alternative submodel of the same part of the system).

But Simile also allows a submodel to be a **multiple-instance submodel**, by the simple act of setting a property for this submodel. Thus, a (flat) submodel of a single cell can become a model for 10 000 cells simply by changing the submodel dimension property from 1 to 10 000. Alternatively, and almost as easily, the submodel can be characterised as defining a population of cells by selecting an alternative property: in this case, the number of instances can change dynamically. In either case, the modeller can use another submodel to define an association (relationship) between submodel instances: for example, to capture the notion that one cell *is_next_to* another cell.

For those readers familiar with UML (the Unified Modelling Language, typically used for object-oriented design in databases and software engineering, <http://www.uml.org>), it may be helpful to think in terms of a close analogy between a Simile diagram and a UML class diagram, with a multiple-instance submodel corresponding to a UML class, and an association submodel corresponding to a UML association class. I will explore this view below (Section 1.2.4) and towards the end of this document, since it suggests a notation for introducing the features that Simile has into SBML.

1.3.3 An example Simile model: farmers and fields

The expressiveness of Simile's model-representation language is best understood by reference to an example. This example includes some simple dynamic processes (what would be represented by reactions or rate rules in SBML), but the emphasis here is on Simile's ability to represent object structure.

The model aims to simulate the grain store of a number of farmers, where each farmer owns a number of fields, and receives the harvest from only those fields.

The model diagram (**Fig. 1**) consists of 3 submodels:

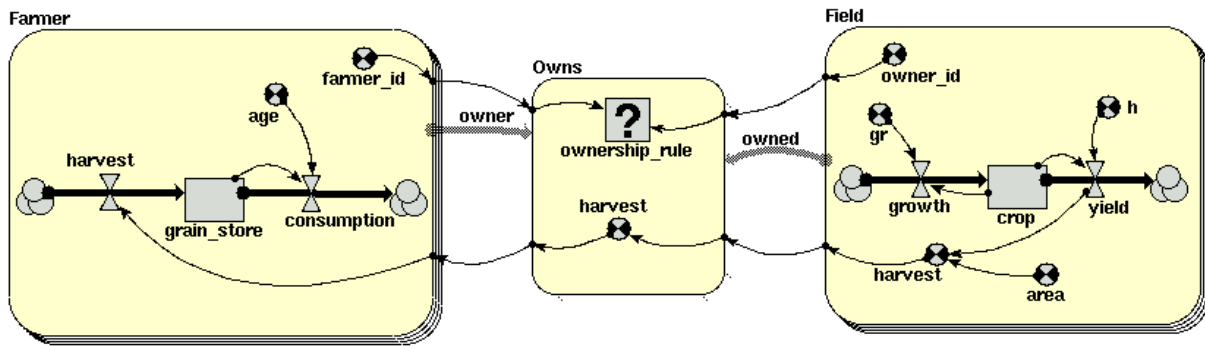


Fig. 1. Simile model diagram for a simple farmers-and-fields model.

The submodel **Farmer** represents the attributes and processes associated with each of several farmers. In Simile terms, this is a multiple-instance submodel: the visual clue for this is the multiple boundary, like a stack of cards. Each farmer has an ID, an age (which changes during the simulation), and a grain store. The grain store is a state variable, represented by a rectangle, and its rate of change is calculated as the net difference between two flow processes: harvest and consumption.

The submodel **Field** represents the attributes and processes associated with each of several fields: this is also a multiple-instance submodel, since there are several fields. Each field is owned by a farmer, has an area, and a state variable representing the biomass of the crop growing in it, on a per-metre-square basis. The biomass of the crop increases through growth, and decreases through yield. The harvest from the field is obtained by multiplying the yield per square metre by the field's area.

The submodel **Owns** has the job of associating a field with the farmer who owns it, so that the grain harvested from each field can go to the farmer who owns it. This is also a multiple-instance submodel, which in Simile is termed an 'association submodel'. In this case, the number of instances is worked out by evaluating the ownership_rule: in this case, a simple test which checks that the farmer_id of the farmer is the same as the owner_id of the field. There is one instance for each farmer-field pair.

This example illustrates clearly the two modelling notations which Simile supports. First, the grain_store and crop state variables, plus associated flows, influences and variables, is closely based on the standard System Dynamics notation, and is readily understood by anyone with experience of that notation. In SBML terms, this corresponds to the description of a set of reactions and transport processes. The other notation - the multiple-instance submodel and the association submodel - is the thing which sets Simile apart from other modelling environments, and provides the expressiveness required for handling the varied needs of the SBML Level 3 packages discussed in this document.

1.3.4 Relationship of Simile submodels to UML classes

Simile's multiple-instance and association submodel notation is analogous to UML notation for defining classes of object and the association between them. It is a powerful and expressive notation for representing complex, disaggregated systems in a concise but mathematically-meaningful manner. **Fig. 2** shows the close correspondence between the Simile model diagram (**Fig. 1**) and a UML class diagram. It shows - in standard UML class diagram notation - two main classes (Farmer and Field) with an Owns association between them; this association has its own association class.

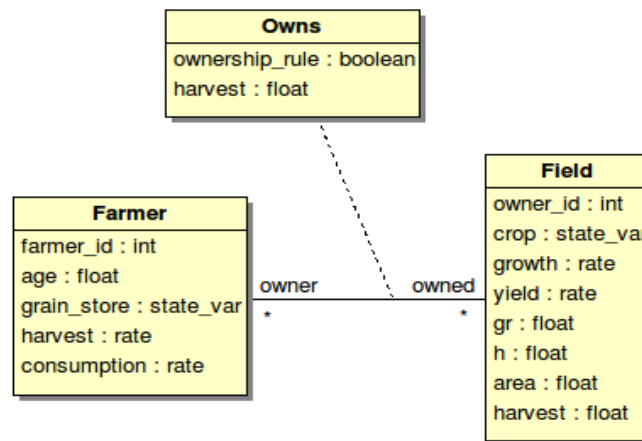


Fig. 2. The object class structure of the farmers-and-fields example as a UML class diagram.

1.3.5 Simile array variables

This section is included because it is directly relevant to at least one of the SBML Level 3 packages under consideration - the **arrays** package. It is included here to provide a basic understanding of Simile array handling before showing how they can be applied to the specific requirements of the **arrays** package as discussed in that section (Section 2.2).

Simile has two types of multi-value data structure: the array and the list. The array has a fixed number of values, while the list has a number which can vary during the course of the simulation. An array can be created explicitly by the modeller, or can be created implicitly by exporting values from a multiple-instance submodel with a fixed number of instances. A list can only be created by exporting values from a multiple-instance submodel with a (potentially) varying number of instances. In this section, I will discuss only Simile arrays.

Consider the following Simile model (**Fig. 3**):

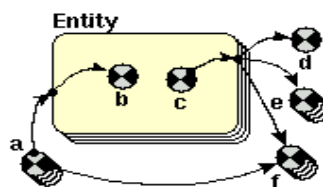


Fig. 3. A model diagram showing various aspects of Simile array variables.

It consists of a multiple-instance submodel (**Entity**), with a fixed number (say, 4) of instances. The following 6 equations give a flavour for Simile's notation for working with array variables. The left-hand side is the symbol as it appears on the model diagram. The right-hand side of each equation is exactly what is entered as the expression for that variable.

a = [10,20,30,40]

a is made into an array variable, because its values are an array (indicated by the [...] of 4 values).

b = element([a],index(1))

b is a scalar in each instance of the submodel, whose value is the index(1)'th value of the array **[a]**, where index(1) is a Simile function which returns the index number of each instance of the submodel (i.e. 1, 2, 3 or 4).

c = element([5,10,15,20],index(1))

c is a scalar in each instance of the submodel, whose value is the index(1)'th value of the 4-value array,

d = sum([c])

d is a scalar, the sum of the values of the array **[c]**

e = [c]

e is an array equal to the (implicitly-created) array **[c]** ('implicitly-created' by having been exported from the submodel)

f = 2*[c]*[a]

f is a 4-element array, each of whose values is 2 times the corresponding elements of the implicitly-created array **[c]** times the explicitly-created array **[a]**.

Note that Simile uses the [...] notation to have two meanings. One is to designate that a variable in an expression is actually an array, e.g. [a]. The other is to create an array whose values are the comma-separated list of values inside the square brackets, e.g. [10,20,30,40].

Note also that Simile's array notation does not correspond to matrix algebra notation, but is a notation developed to enable common operations to be performed with array variables without having to use programming constructs such as a for...next loop.

In addition to the features exemplified above:

- Simile arrays can be nested to any depth, e.g. [[1,2,3],[4,5,6]] is a 2D (2x3) array; [[a]] is a 2D array variable.
- You can construct arrays using the `makearray(Array,N)` function: for example:
`makearray(3,5)` creates the array [3,3,3,3,3]
- You can combine array operations with a conditional expression, e.g. the following expression for the variable f above
`if [a]<30 then [c] else 99`
`makes [f] equal to [10,20,99,99]`

There is much more to Simile's array notation than can be covered here: for more information, see <http://www.simulistics.com/help/equations/arrays.htm>.

1.4 Challenges in preparing this document

I faced a number of challenges and difficulties in exploring the possibility of a unified approach to the various SBML Level 3 packages.

First, the SBML community has been discussing some of these proposed extensions for many years - in some cases, 10 or more. These discussions have been through formal or semi-formal proposals, notes of various types, workshops, contributions to the sbml-discuss forum, and no doubt quite a few informal exchanges of ideas. Thus, in proposing a unified approach to handling proposed SBML extensions, it is sometimes difficult to know just what has been discussed, and to pin down the various ideas which have been floating around.

Second, there are various overlaps behind the proposed extensions packages, as indeed has been recognised by several contributors to discussions on the extension packages. For example, the main use-case example model presented for the **arrays** package actually involves spatial diffusion, so could well be used as an example for the spatial diffusion package as well. This is actually quite a strong argument in favour of a unified approach, since it is undesirable to provide alternative but equally-valid mechanisms for implementing the same modelling concept.

Third, some of the packages seem to involve quite distinct sets of concepts. For example, the various documents on the **geom** (geometry) package cover both the case of a set of spatial units, within each of which various continuous processes (e.g. biochemical reactions) occur, and the case of a geometric shape (e.g. a sphere, on the surface of which particles can occur, interact and move. Both aspects are, of course, perfectly valid elements of a proposal, but are clearly quite different in nature, and thus complicate the task of describing how they can be handled within a unified approach.

Fourth, Simile is based on System Dynamics (stock-and-flow) rather than biological reactions for continuous processes. This is not particularly relevant to the present discussion (which is concerned with the structural aspects of models rather than the representation of processes), but it may make it harder for the reader from the Systems Biology community to relate to some of the examples.

Finally, the author comes from an ecological/environmental modelling background rather than a Systems Biology one, and most applications of Simile have been at this level. The reader is asked to be tolerant of any misuse of terminology or apparent ignorance of well-understood concepts within the Systems Biology domain.

2 The SBML Level 3 extension packages

In this Section, I will consider each of the SBML Level 3 packages listed in **Table 1**, summarising the rationale and status of the package, and exploring how its requirements can be met in Simile.

2.1 Package 'comp': Hierarchical Model Composition

2.1.1 Rationale and current status of the proposal

Home page for the proposed package

http://sbml.org/Community/Wiki/SBML_Level_3_Proposals/Hierarchical_Model_Composition

Rationale

The following is the stated aim of the proposed extension package, as taken directly from its home page:

"Model composition refers to the ability to include models as submodels inside other models. This requires defining the interfaces between the models and rules for connecting parts of models together. The motivation is to enable the creation of standard, vetted models and use them as library components when creating larger models, much as is done in software development, electronics design, and other engineering fields."

The following paragraph is taken from the proposal itself:

"SBML Level 2 has no direct support for allowing a model to include other models as submodels. Software tools either have to implement their own schemes outside of SBML, or (in principle) one could use annotations to augment a plain SBML Level 2 model with the necessary information to allow a software tool to compose a model out of submodels. There is a clear need for an official SBML language facility supporting hierarchical model composition."

Status

There is one active proposal: Hoops, S. *et al* (2007) Hierarchical Model Composition.

This proposal mentions and links to two previous proposals (Ginkel, June 2002; Finney, Oct 2003). However, neither are listed on the package's home page as being 'active'.

Analysis

There are two separate (though linked) concept involved here. The first is the idea that an externally-defined model can be included in some other model. This is often referred to as 'component-based' or 'modular' modelling, and is clearly an important rationale for the proposal, as evidenced by the two paragraphs quoted above. The external model may be included by reference (as seems to be the focus for this proposal), but it could also be included by copying (i.e. the text of the external model is incorporated into the parent model). In the latter case, it could be that this process is supported by some modelling tool, though (with the right data model) it could also be done by copy-and-pasting text in a text editor.

The second concept is that a model can be organised hierarchically, in terms of submodels nested to any depth. This could be achieved by including another, externally-defined model 'by reference', but not necessarily: the language could simply allow one submodel to (textually) contain other submodel(s) within it.

The proposal by Hoops *et al* includes covers both aspects. It includes sample code which (in summary) has the form shown in **Box 1**. **Submodel_1** is included by reference to an externally-define model, and is thus an example of the first concept mentioned above. **Submodel_2** is included in the file itself, and is thus an example of the second concept.

```

<sbml>
  <model>
    . . . . . normal SBML elements
    <listOfSubmodels>
      <submodel id="Submodel_1">
        <model xref="..." xpointer="..." />
      </submodel>
      <submodel id="Submodel_2">
        <model id="HierarchicalModel_2">
          . . . . . normal SBML elements
        </model>
      </submodel>
    </listOfSubmodels>
  </model>
</sbml>

```

Box 1. Skeleton XML code for nested submodels.

2.1.2 Simile handling of the 'comp' requirements

Simile's model-representation language has strong support for hierarchical model organisation: a model can contain submodels, nested to any depth. The Simile language provides no notation to allow an externally-defined model to be included in a model by reference (unless the model has been compiled into a DLL - I will ignore this mechanism, since the model is no longer represented in a markup language). In contrast, the Simile software does provide support for modular model construction, and indeed supports plug-and-play modularity. An externally-defined model can be included multiple times in a particular model, but this is by copying rather than by reference.

Fig. 4 illustrates both aspects. The main part of the diagram shows a model with several nested submodels. These were constructed by the user drawing the 4 submodel envelopes, and are represented hierarchically in Simile's model-representation language. The complete model specification is held in a single file, and there is no notation for this file referencing an external (sub) model.

However, the user can work with the model in a modular fashion within Simile itself. In the present example, **Submodel 4b** is an alternative to **Submodel 4a**: both have an input from the variable **b**, and both calculate a value for **d**. Any submodel can be saved as a stand-alone Simile model, so both **Submodel 4a** and **Submodel 4b** could have been constructed either as a submodel in Simile then saved, or as a stand-alone Simile model. Simile allows the two versions to be swapped around, and automatically makes the links between the variables outside the submodel and those inside it. This process requires Simile to know what these links are: this information is held in a separate file, the Interface Specification File.

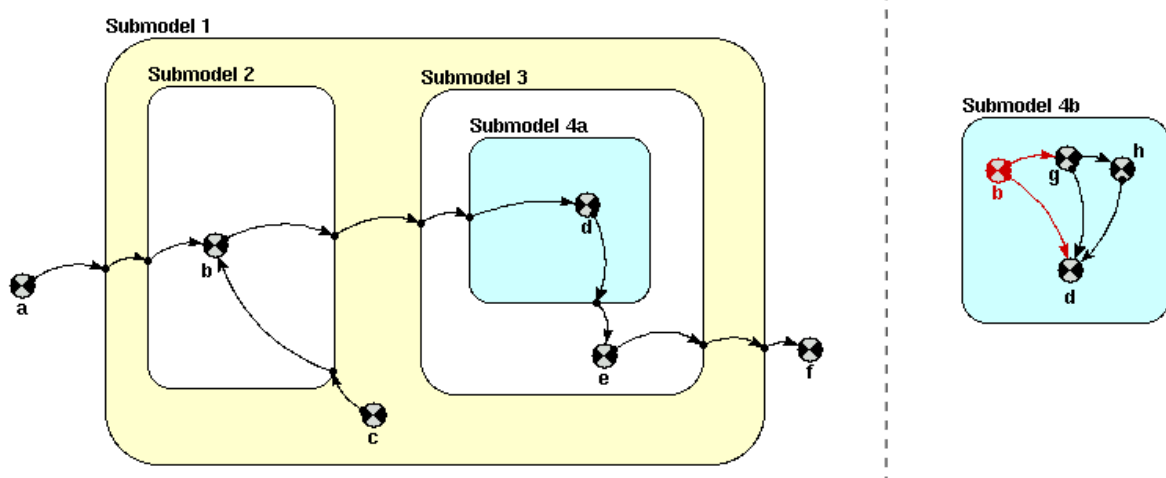


Fig. 4. Simile model diagram showing nested submodels and links between variables. Submodel 4b is an alternative to Submodel 4a, and can be swapped in using Simile's mechanism for plug-and-play modularity.

Simile does not attach any particular semantic interpretation to the notion that one submodel is inside another. In UML, for instance, there is a distinction between 'composition' and 'aggregation': composition meaning that the existence of some component is dependent on its containing component (like a University department in a University); aggregation meaning that some component happens to be part of another, but could have a separate existence (example: a car wheel). This aspect of Simile partly reflects the fact that the concept of submodel is pretty neutral: sometimes it can correspond to a physical object, sometimes to a class of objects (as detailed in the following Sections), and sometimes it is simply a convenient way of dividing up a complex model into meaningful blocks.

This could have implications in relation to the present exercise. SBML already has the concept of a **compartment**, and the **comp** proposal introduces the new concept of a **component/submodel**. In other words, a particular model that made use of the **comp** package could have two concepts that deal with containership. There may be perfectly good reasons for maintaining this distinction: I merely point out that Simile does not have it, so this would need to be factored in to any discussion about building on the Simile data model in SBML Level 3.

The most significant feature of the **comp** package proposal which Simile cannot currently handle is the idea of a **re-useable object**, where re-use is achieved by **reference** rather than by **copying**. At first sight, this seems like quite a major limitation of Simile. After all, other object-oriented modelling environments, such as Modelica (<http://www.modelica.org>) and Simulink (<http://www.mathworks.com/products/simulink>), support the idea that there can be a library of components, and a model can be built up by selecting instances of these components and inserting them into the model design. A classic example is a modelling environment for electronic circuits, where the library contains 'Transistor', 'Resistor', etc, and the model contains 'transistor1', 'transistor2', etc - instances of Transistor.

However, things are not always what they seem. The fundamental difference here is that a model diagram in Modelica or Simulink is, in UML terms, an **object** (or instance) diagram, whereas a Simile diagram corresponds more closely to a UML **class** diagram (Section 1.2.4). This is not to dismiss the value of having re-useable classes. It is just that Simile already supports (in quite a strong sense) the idea of class and instance. Moreover, after some 14 years of watching Simile being used in modelling in ecology and environmental science, we have rarely encountered situations where some additional notion of re-useable class would have been beneficial.

2.2 Package 'arrays': Arrays & sets

2.2.1 Rationale and current status of the proposal

Home page for the proposed package

http://sbml.org/Community/Wiki/SBML_Level_3_Proposals/Arrays_and_Sets

Rationale

The following is the stated aim of the proposed extension package, as taken directly from its home page:

"The primary motivation for proposing this capability is that many types of models use large numbers of more-or-less identical components, and it is convenient (if not practically necessary in very large models) to be able to use an indexing scheme to reference these components. (Or as Bruce Shapiro put it succinctly in 2006, "Arrays allow us to describe a bunch of stuff without listing every item explicitly every time".) Another motivation is the desire to support models having elements and structures whose spatial geometries are not important; these elements might more conveniently be referenced as indexed entities rather than individually-named entities. A final motivation is to support abstract mathematical models."

This section draws on two types of information from the available documentation for the SBML Level 3 **arrays** package:

- Proposed specifications from the latest draft proposal, with a description of how the relevant part of the specification can be handled in Simile (Section 2.2.2);
- An example model which is presented as a typical use-case for the **arrays** package (Section 2.2.3), followed by the implementation of the model in Simile (Section 2.2.4). (Note that this model has a strong spatial element, so could also be taken as an example for the spatial/geometries packages (Section 3). It is considered here because this is where it is presented in the SBML Level 3 documentation.)

Status

Two active proposals:

Finney *et al*, Sept 2003

Shapiro *et al*, Dec 2004

Preceded by various documents and workshops going back to 2000.

2.2.2 Proposed specification

What I have done here is to take each significant item from the latest **arrays** proposal (Shapiro, B.E., Gor, V. and Mjolsness, E. December 2004: Systems Biology Markup Language (SBML) Level 3 Proposal: Dynamic Arrays), and assess whether there is an existing counterpart in Simile. Every reference to a Section here is to the relevant section in the Shapiro *et al* proposal. For brevity, I will not detail the original proposal in each case: the interested reader is referred to the original document. However, I do provide relevant quotes (in italics) from the original proposal where it helps to set the scene.

Please note that the proposal talks about arrays of several different SBML things: variables, species, compartments, reactions etc. Simile only has variables and submodels: variables can be array variables, and submodels can be multiple-instance submodels. So when I show how the SBML requirement can be handled in Simile, it will always be in terms of a Simile array variable or a Simile multiple-instance submodel.

ARRAY DEFINITIONS (Shapiro et al, Section 3)

"Objects can be defined as arrays in two ways, either explicitly or implicitly. Any object can be defined as an array explicitly via a dimension object. Section describes explicit array definitions, and section 3.2 discusses implicit array definitions."

Explicit array definitions (Shapiro et al, Section 3.1)

"An object can be explicitly declared to be an array by attaching a list of dimensions to the object's definition. The number of dimension statements is equal to the number of array indices; a vector would have one dimension statement, a matrix two, and so forth."

Explicit definition of array variables

Simile has several methods for the explicit creation of array variables (see Section 1.2.5 above). In the following examples, the array variable reference (left-hand side of the equals sign) shows how the array would be referred to in some other expression. The Simile user would simply type the expression on the right-hand side in the equation dialogue box for the variable with the alphanumeric label (i.e. with the enclosing square brackets).

- Creation by provision of an array of values:

$[a] = [10,20,30]$ Creates a 1D 3-element array, with values 10, 20 and 30 for the 3 elements.

$[[a]] = [[10,20],[30,40],[50,60]]$ Creates a 3x2 array, with the values shown.

- Creation by inclusion of array variables in expressions:

$[a] = [b]$ Creates the array $[a]$ from the array $[b]$.

$[a] = 2*[b]*[c]$ if $[a]=[1,2,3]$ and $[b]=[4,5,6]$, then $[a]=[8,20,36]$

- Creation using the makearray(Pattern,N) function:

The 'makearray(Pattern,N) creates an array containing N elements according to the pattern given in Pattern.

$[a] = \text{makearray}(3,5) \rightarrow [3,3,3,3,3]$

The 'makearray' function is very powerful, and can be used to construct a wide variety of arrays. See the Simile documentation for more information (under built-in functions), and the sparse array example below.

Explicit definition of multiple-instance submodels

The equivalent in Simile of what the SBML Level 3 arrays proposal calls an 'array' of objects is the Simile 'fixed-membership multiple-instance submodel'. As the name indicates, this has a fixed number of instances. It is created by setting the submodel property 'Use fix number of dimensions' to a number greater than 1.

Simile also allows a submodel to be declared as having a variable number of instances, that is, the number of instances can change during the course of the simulation run. In Simile this is called a 'population submodel', and is an example of a 'variable-membership multiple-instance submodel'. This can be considered as another way in which Simile can meet the requirement of the SBML Level 3 **arrays** proposal for an array of objects, but there is a significant technical difference between the two types of submodel in Simile. The fixed-membership multiple-instance submodel exports a variable as an actual array in the normal sense (one can access a value using the appropriate index), whereas a population submodel exports a variable as a list, and it is not possible to access the *i*th value of this list.

Implicit array definitions (Shapiro et al, Section 3.2)

"Arrays can be defined implicitly because variables "inherit" a dimension from certain "enclosing" or referenced objects, as described in the following sections."

- Implied compartment arrays (Shapiro et al, Section 3.2.1)

"If compartment B is inside of compartment A and A is an array, then B has the same implied dimensions as A. If B is an array inside A, then B has implied dimensions $\dim(B) \times \dim(A)$."

As noted previously, Simile has no concept of a compartment, but instead the more generic concept of a submodel, which can be used to have the role of a compartment container. The following discussion is therefore in terms of Simile submodels.

If a simple submodel is nested inside a multiple-instance submodel, then the inner submodel has, implicitly, the dimensions of the outer, multiple-instance submodel.

If a multiple-instance submodel (Sub1), with dimension 5, is nested inside another multiple-instance submodel (Sub2), with dimension 3, then the inner submodel has implicitly the dimensions of a 2D array, i.e. dimensions (3,5).

- Implied parameter arrays (Shapiro et al, Section 3.2.2)

Any variable inside a multiple-instance submodel has implicitly the dimensions of the submodel it is in.

- Implied rule arrays (Shapiro et al, Section 3.2.3)

Shapiro *et al* give the example of "an array of 100 rate rules $x_i' = x_i y_i$ ".

In Simile, we can handle this easily using a multiple-instance submodel, as follows. Any state variable and rate variable inside a multiple-instance submodel has implicitly the dimensions of the submodel it is in. The following example keeps closely to the Shapiro et al notation: the submodel is called **i**, the state variable is **x**, the rate of change is **dxdt**, and the other variable is called **y**. Use the following equations:

$dxdt = x * y$ (note that there is no need for the index *i*)

$y = \text{rand_const}(0.05, 0.15)$ (*y* is a random value between 0.05 and 0.15 for each instance)

and run the simulation. The graph shows the trajectory of each of the 100 instances.

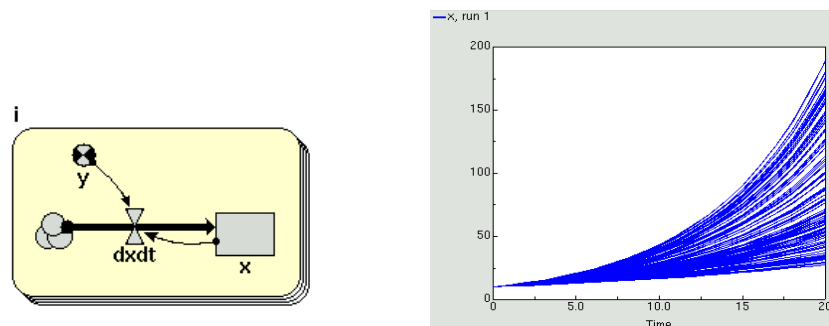


Fig. 5. Simile diagram and sample results for a simple multiple-instance 'rate-equation' model.

- Implied reaction rates (Shapiro et al, Section 3.2.4)

Simile has no concept of a reaction, so this is irrelevant.

- Implied event arrays (Shapiro et al, Section 3.2.5)

Simile has no concept of an event, so this is irrelevant. Current development plans include an event mechanism. When this happens, then events will automatically take on the dimensions of the submodel they occur within.

INITIAL ASSIGNMENT RULES (Shapiro et al, Section 4)

These are already handled by Simile's existing mechanisms. Any state variable in a multiple-instance submodel can be set using the normal mechanisms. Any parameter can have a value assigned:

- by setting to a constant, in which case all instances take the same value;
- by assigning a random value to each instance;
- by setting a value which is different for each submodel instance);
- by getting values from external data files (e.g. a comma-separated value file exported from a spreadsheet).

SPARSE ARRAYS (Shapiro et al, Section 5)

"Sparse arrays can be defined by first defining a default value of zero to every element of an array, and then assigning specific values to the non-zero elements."

As noted above, Simile provides several mechanisms for constructing arrays, including conditional assignment of array element values, and the `makearray()` function. These can be used to construct sparse arrays.

Shapiro *et al* give the example of a 3x3 array consisting of zeroes with the exception of the diagonal in which each element has a value 1. In Simile, this can be achieved by the expression:

```
[[a]] = makearray(makearray(if place_in(1)==place_in(2) then 1 else 0,3),3)
```

with `[[a]]` then equal to `[[1,0,0],[0,1,0],[0,0,1]]`

The nested `makearrays` create a 2D, 3x3 array. 'place_in' is a built-in function which references each array index (i.e. 1, 2 and 3 in this case). Its argument (1 or 2 in this case) references the inner and outer arrays (rows and columns) respectively.

Note that the proposed SBML Level 3 approach first sets up an array in which every element is zero, then re-assigns a non-zero value to specific elements. In Simile, the array is specified in a single step, with a conditional expression to decide on the value to assign.

CONDITIONAL OBJECTS (Shapiro et al, Section 6)

"We allow any array object to have an optional exists field to indicate that some values of an array are undefined and never need to be considered."

The appropriate construct to use in Simile to handle this situation is the conditional multiple-instance submodel - i.e. a multiple-instance submodel with a 'condition' symbol in it, with the Boolean expression in the condition element defining whether a particular instance exists or not. (See Section 2.4.4 for an explanation of 'conditional submodels' in Simile.) This is not the same as a data structure containing only some elements out of a possibly much larger set, one which can be passed around within the model. However, my guess is that any problem requiring the proposed SBML Level 3 'conditional objects' can be handled using this mechanism.

CONNECTION RULES (Shapiro et al, Section 7)

"Connection rules allow users to define different types of geometric interactions."

*"Suppose that `cell` is a compartment array. Then the following defines a **connectionRule** `xneighbor` from `cell[from]` to `cell[to]` whenever $|x[from] - x[to]| < \text{radius}[from]$, i.e. whenever two cells are close along the x-axis."*

This is precisely the type of situation for which Simile's 'association submodel' was designed, and indeed which contributes to Simile's expressiveness. Examples of its use are given in the following subsection (2.2.3), and in section 2.3 (on the **spatial** and **geom** packages.)

2.2.3 'Arrays' use-case model

Bruce Shapiro has presented a "use-case motivating the need for this package", at http://sbml.org/Community/Wiki/SBML_Level_3_Arrays_and_Sets#Use-case_motivating_the_need_for_this_package.

"The model is based on the "Activator Model" in Fig. 4 of the 2005 [paper by Jönsson et al.](#). The model consists of 253 cells, each of which behaves according to a small set of equations. The cellular template is based on real data, a laser scanning confocal horizontal cross-sectional image of an arabidopsis meristem with GFP labeled nuclei. Cell-cell connectivity was derived using a [Delaunay triangulation](#) procedure, which analysis has shown to be approximately 97% correct in the meristem. The template looks like this (colors do not have significance here)."

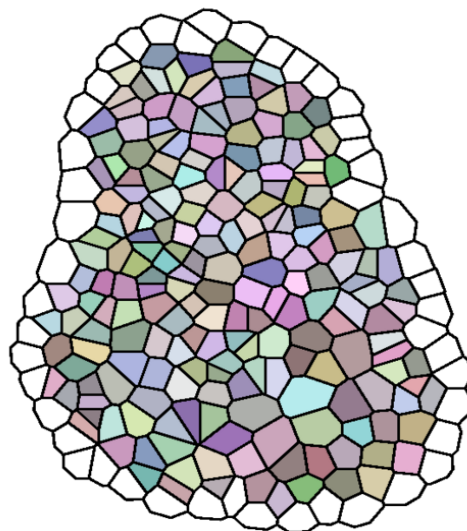


Fig. 6. Cellular template for the 'activator' model (taken from the above-referenced web page, which itself references Fig. 4 of Jonsson et al (2005)).

The dynamics of the system can be expressed on paper very concisely: a mere 4 ODEs (see the referenced paper, equations 5-8).

Bruce Shapiro then states:

"The overall model that results from running the code to generate 253 cells in the configuration above

contains 2025 internal reactions and 2119 intercellular reactions. Using an array notation, the cells and reactions do not have to be named individually—they are simply referred to using integer indexes. This vastly simplifies code and allows using loops and other ways of manipulating the cells and reactions in the model."

An SBML model - without the array notation - has indeed been constructed with the number of reactions specified above, each one coded separately - it runs to some 3.1 MB! It can be viewed here:

<http://computableplant.caltech.edu/models/Activator/index.html>.

This is clearly an impractical way to handle this degree of disaggregation in a model.

2.2.4 Simile implementation of the use-case example model

Fig. 7 is the Simile diagram for a quick re-implementation of the model. The only change is that the cells have been modelled as being arranged on a regular grid, rather than as tessellated polygons. In fact, this is not such a significant simplification, and the Simile model diagram will be little different for the latter approach. The reason for using a regular grid is that I do not have the data defining the polygons, rather than a limitation in Simile's ability to express the model in terms of tessellated polygons.

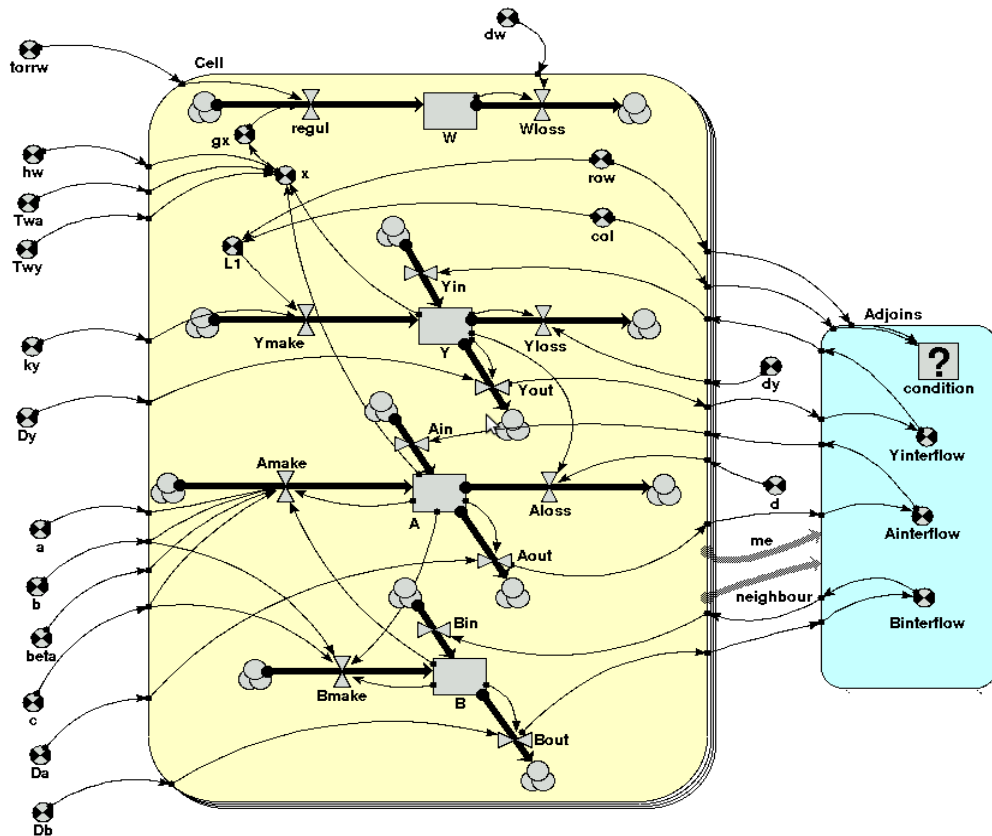


Fig. 7. Simile model diagram for the 'activator' model.

The main feature of the model is the **Cell** submodel. This is a multiple-instance submodel, with as many instances as cells: 400 in this particular implementation, but it is trivial to change the number. Cells have a **row** and a **col** attribute, with the value for each cell being worked out from its instance number, such that each cell has a unique pair of row and col values. Each cell also has 4 state variables (**W**, **Y**, **A** and **B**). Some processes contributing to the rate of change of each state variable are internal to the cell (shown as horizontal here), while others represent the flow in from neighbouring cells or flows out to neighbouring cells (the diagonal flow arrows).

The **Adjoins** submodel is a Simile association model, and defines which cells are a neighbour of each cell. It has

as many instances as there are pairs of neighbouring cells. In this case, it is worked out simply from the **row** and **col** values (they must be within 1 of each other, but not the same). This submodel contains 3 "...interflow" variables, whose value for each instance of the **Adjoins** submodel is the flow of the metabolite from one cell to one of its neighbours.

Finally, there are a number of parameters. These are outside the **Cell** and **Adjoins** submodels, because (it is assumed) their value is the same for all cells and connections between cells. (If you want to make the value of a parameter different for each cell, it is a simple matter of re-locating it inside the **Cell** submodel, and perhaps inputting its value from an external .csv file, or calculating it from some cell attribute.)

Running the model

The following diagram shows the result of running the model, which was set up with a high value for L1 for the middle square, and zero values for the rest. This implementation has not been validated against the original model, but it is likely that any corrections or additions required will be minor.

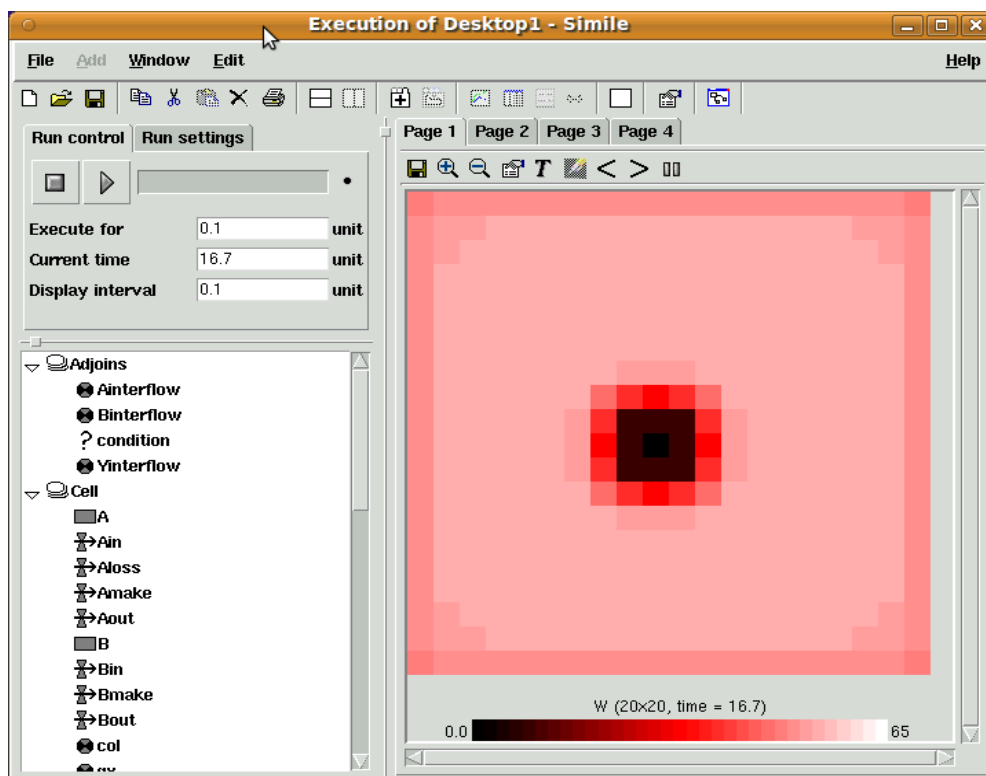


Fig. 8. Sample output from running the 'activator' model.

It is clear that Simile is able to handle this example in a way envisaged when it was put forward as a use-case for the **arrays** package. (It is also, of course, another example of spatial modelling.)

2.2.5 Assessment of Simile's abilities to handle the requirements of the 'arrays' package

As stated above, there are two rather different types of requirement here: the ability to describe multiple things; and the ability to have array variables. Simile handles the first using the multiple-instance submodel, and the second by having array variables and the ability to incorporate such variables in mathematical expressions.

2.3 Packages 'spatial' and 'geom': Spatial diffusion and Geometry

2.3.1 Rationale and current status of the proposal

I am treating these two packages together. The **spatial** package description consists of only a few sketch notes

which include a mention to geometry, while the **geom** package clearly refers to spatial-modelling aspects, sometimes explicitly.

Home page for the proposed package

http://sbml.org/Community/Wiki/SBML_Level_3_Proposals/Geometry

However, note that there is a link at the bottom of this page to an "activity page for the SBML Level 3 geometry effort" (http://sbml.org/Community/Wiki/SBML_Level_3_Geometry), and there is a link at the bottom of that page to "Geometry in SBML", which actually contains the formal specification: http://sbml.org/Geometry_in_SBML

Rationale

"The goal of supporting spatial characteristics in SBML Level 3 is to allow the representation of the geometric features of compartments and the spatial distribution of model quantities and processes. SBML models today are nonspatial: compartments are topological structures only, with dimensionality, size and containment being their only physical attributes. This was partly a conscious design decision, because there are far more nonspatial modeling tools available today and so our priorities went towards that. However, it is clearly insufficient for many potential modeling uses, including the problems described above. Today there are a number of computational modeling systems supporting spatial characteristics in biochemical network models."

The focus of the spatial features effort is on extending SBML to support representing at least the following: (1) the size and shape of physical entities, whether compartments or reacting species; (2) the absolute or relative spatial location of reacting species in compartments, for instance in a volume, on membrane surfaces, or along microtubules; (3) the rates of diffusion of species through compartments; and (4) the definition of rate equations and algebraic constraints describing phenomena either at specific locations, or distributed across compartments."

The following notes from Nicolas Le Novère express succinctly a number of important points and distinctions in this area:

There are actually two topics hidden behind "spatial" SBML extensions, that should belong to two different extensions: the support of spatial models and the encoding of geometry.

Spatial:

We need to extend SBML so it can support PDE, but also finite elements, single-particle models etc.

Examples: The differential location of a given species in subcellular compartments affects its function, e.g. the CaMKII differential phosphorylation in post-synaptic density and in cytosol or the hysteresis generated by the differential location of kinases and phosphatases. Another example is the MAPK cascade from the membrane to the nucleus.

That requires the encoding of initial conditions (specific coordinates - link to the array proposal? - or distributions) and movement laws.

Geometry:

We need to extend SBML to describe the geometry of physical objects, whether compartments or species.

Examples: the morphology of neuronal compartments is central to the treatment of signals, whether electrical or calcium diffusion. The topology of supra-macromolecular structures in the post-synaptic junction affects the signal transduction.

That requires the encoding of physical entity topology - link with the complex proposals? - and the encoding of deformation laws.

Status

There is no active proposal for these packages.

The two main content pages (links given above) relating to these packages are undated, but the page history shows that the main content was added in Sept 2008 and Nov 2009.

2.3.2 Implementing spatial and geometrical concepts in Simile

This is a complex topic, since it covers a wide range of concepts.

The first thing to make clear is that Simile does not allow models to be expressed as PDEs, whether involving spatial dimensions or not. Any spatial model expressed in terms of PDEs must first be re-cast in terms of discretised space.

The second point is that Simile was designed to handle spatial concepts from an early stage. It actually started life as AME - the Agroforestry Modelling Environment - for modelling the biological and spatial interactions between trees and crops, with spatially-referenced trees interacting with a spatial grid containing the soil and crop.

The third point is that - paradoxically - Simile has no explicit notions of space or geometry built into its modelling language. Any spatial or geometrical aspects must be expressed in terms of objects, attributes of objects, and associations between objects. This might seem odd, given the previous paragraph, but was a deliberate decision in order to achieve the genericness for handling space which we considered to be an important goal.

A simple spatial model in Simile

An example will make this clearer. (This actually covers some of the same ground as the 'activator' model presented above, in Section 2.2.4, but is included here for completeness, and to expand on some aspects.) Imagine that you want to model spatial diffusion across a regular grid. One option is to use a modelling environment which has grid space built in as the only spatial representation, or as one of several built-in spatial representations the user can select. You specify the grid attributes (e.g. number of rows and columns), and the dynamics which occur in each grid square (perhaps in the form of a set of reactions). The software also allows you to choose a method for handling transport between squares (perhaps you can choose between a donor-controlled flow or a flow proportional to the gradient between two grid squares), and perhaps you can choose which neighbourhood model to use (e.g. von Neumann (4-cell) or Moore (8-cell) neighbourhood).

In Simile, you do things very differently. You add a submodel to your model diagram (see **Fig. 9**) and specify that it has multiple instances: say 100. You call the submodel 'Grid square', perhaps, but this is just a label: it has no significance to Simile. You provide two variables, say **row** and **column** (again, no built-in significance), and engineer their values so that each pair have a unique combination of values between 1 and 10: this is easy to do. You put whatever dynamics you want in the **Grid square** submodel. You then add an association submodel (perhaps called something relevant, like **Adjoins**), and express the condition for the association to hold between any two grid squares, in terms of the row and column values for each square. Finally, you take influences from the **Grid square** submodel through the **Adjoins** submodel and back again, to allow each square to have access to the values in the squares that adjoin it, and provide the appropriate equations for handling the interaction. (A more detailed example is shown in Section 2.2.4 above, with sample display of model results.)

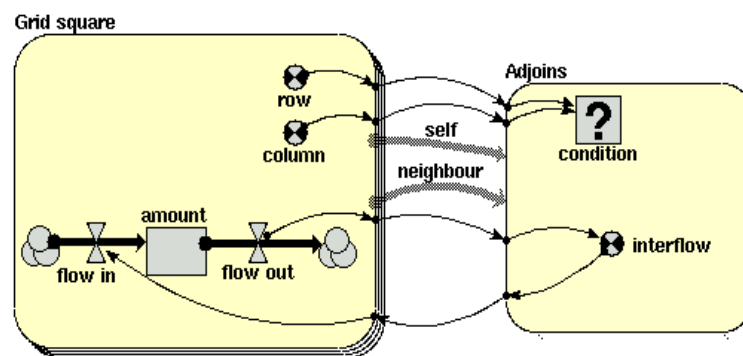


Fig. 9. A very simple spatial-grid model in Simile

So, the key difference is that, in the most spatial modelling software, you select a spatial configuration, while in Simile you design a spatial configuration. The assumption is that all aspects of space which are of relevance in a particular model can be represented in terms of structures (objects and their relation to each other) and values for variables. Since Simile has a rich language for handling both concepts, there is no need to provide built-in primitives for space or geometry. The downside of the Simile approach is that the modeller has to do more work. The upside is that the modeller has much more freedom - to specify a spatial configuration, and to specify how spatial units interact given a certain spatial configuration. Moreover, the modeller is using the same constructs as are used for the other aspects of modelling discussed in this document, rather than having to learn new ones

specifically provided for handling space and geometry.

This approach gives the modeller a great deal of freedom in designing spatial configurations. You can have 1D, 2D or 3D space. In 1D, you can have layers, perhaps for a finite-element solution to modelling diffusion along a tubule. In 2D, you can choose to model space as (for example) squares, rectangles, hexagons, overlapping circles or polygons. In the same model, you could have some aspects represented in one of these ways, combined with moving particles characterised with (x,y) coordinates in the same spatial region, with each particle being aware of the spatial unit it is within as it moves around. In 3D, you could have cubes or irregular 3D volumes. In all cases, the modeller can specify relationships such as proximity, and various forms of interaction between the spatial units. **Fig. 10** shows some examples of 2D displays used to display Simile spatial models while the model is running.

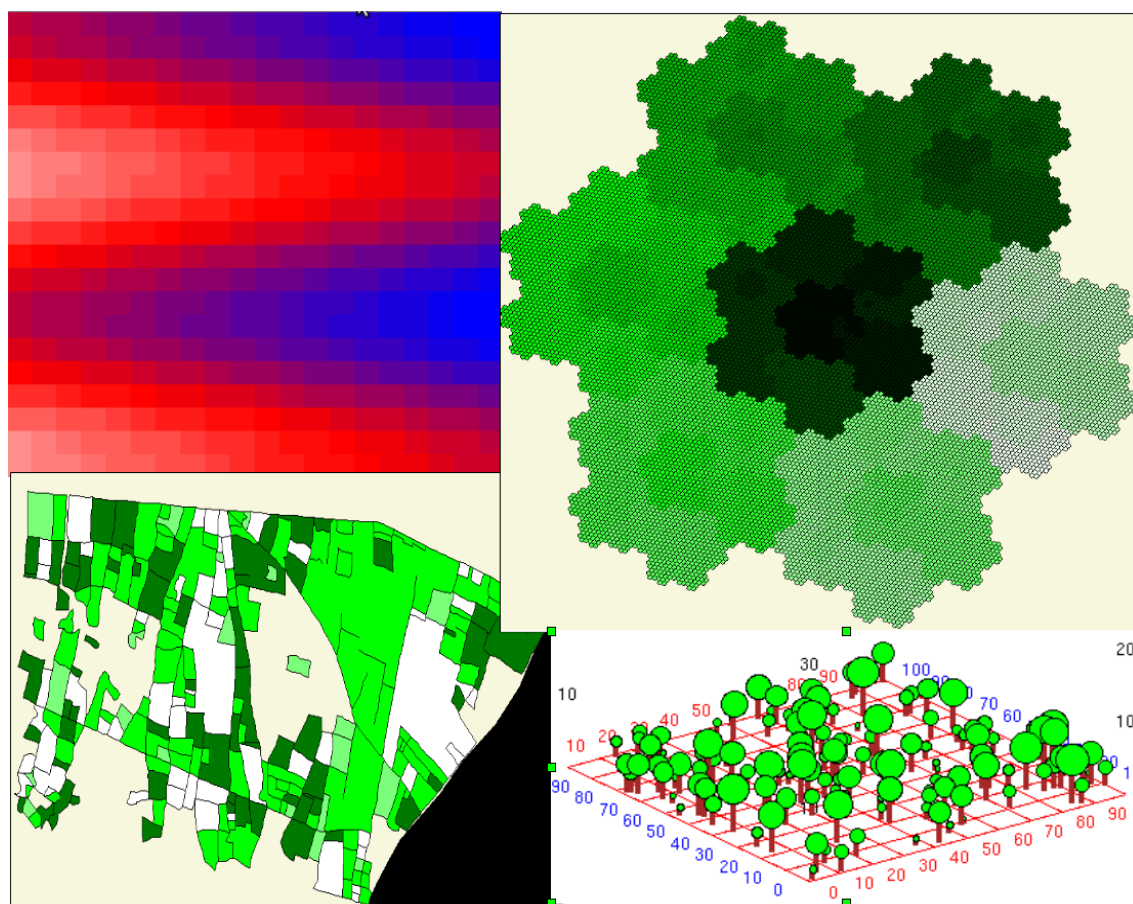


Fig. 10. Sample of Simile spatial displays, showing its ability to handle a variety of spatial configurations. From top-left: regular grid, hierarchically-recursive hexagonal grid, polygons, and individual location in 2D space.

There are issues of computational efficiency associated with the Simile approach, compared with software tools which have dedicated facilities for handling particular spatial or geometric configurations. These will be discussed in the final Discussion section (Section 4) of this document.

2.3.3 Individual-based ('particle-based') representation

One of the submissions for handling spatial and geometrical aspects in a future SBML Level 3 package (http://www.sbml.org/Community/Wiki/SBML_Level_3_Geometry) suggest that it should be capable of supporting a particle-based approach, allowing individual particles to move about in 2D, 2.5D (surface) or 3D space.

The Le Novère Compneur group describe their simulator, Meredys (<http://www.ebi.ac.uk/compneur-srv/meredys.html>), which simulates movement of individual molecules, interactions between them on the basis of

their geometrical properties (e.g. distance of a reactive site from the molecule's centre of gravity), and their ability to form transient complexes which behave as one.

As mentioned above, Simile supports modelling in terms of individual objects. In fact, the model diagram would look very much like the above figure (**Fig. 9**), with the difference being mainly one of interpretation. Rather than the instances of the multiple-instance submodel being interpreted as spatial units (grid squares), with integer row and column attributes, they would be interpreted as particles with floating-point (x,y) coordinates. In fact, the (x,y) coordinates could actually be state variables, allowing the individuals to move through space! So it is quite possible that the use-cases presented by the above group (along with many others) could be handled in Simile.

In the next section, we will see that Simile can go further, and allow the number of particles to change dynamically.

2.4 Package 'dyn': Dynamic structures

2.4.1 Rationale and current status of the proposal

Home page for the proposal

http://sbml.org/Community/Wiki/SBML_Level_3_Proposals/Dynamic_Structures

Rationale

"SBML's structural constructs are currently fixed: it is not possible to define the creation or removal of species, compartments, reactions or other components from within a model definition. To simulate the creation or destruction of compartments, one currently has to use tricks. For example, a model could define all the compartments it could ever need and use variables to indicate which compartments are actually "active" at any given time—but this would only work if the total number needed is known at the beginning of a simulation. In defense of SBML's limitation in this area, it should be pointed out that we only know of perhaps two or three software tools that support dynamic structures today. Still, it is clear that some modeling problems would benefit from this capability. Dynamic structures would be used to encapsulate portions of a model that need to change dynamically during a simulation. Within SBML this may correspond to adding, modifying or removing some collection of compartments, species, reactions, rules, etc. Biologically, this might correspond to cell birth, differentiation, cell death, as well as exocytosis and endocytosis; experimentally this might correspond to the administration of a protocol during a simulation, such as turning a laser or the application of a hormone or toxin."

Status

Currently, there is no proposal for Dynamic Structures, and the above text is all that is available under the SBML Level 3 Extension Packages section.

2.4.2 Implementing the Dynamic Structures package in Simile

It would seem that there are two quite separate concepts being entertained here.

First, there is the concept of changes in the number of mathematically-similar entities. The second half of the first paragraph and the whole of the second paragraph focus on this aspect. In UML terms, we have a class for which the number of instances changes dynamically during the course of a simulation.

The second concept being entertained here is that of changing the mathematical structure of the model dynamically during the course of the simulation. This is possibly suggested by the wording of the first half of the first paragraph above. In Simile, this requirement is addressed using a 'conditional submodel' - a standard submodel which includes a condition symbol. The expression for a condition symbol is a Boolean expression: if it evaluates to 'true', then the equations in the submodel are processed, otherwise not.

The way Simile handles these two different facets of the putative **dyn** package are addressed in the following two sections.

2.4.3 Dynamic changes in the number of mathematically-similar entities

As we have seen, Simile allows modellers to specify that a submodel has a fixed number of instances. However, simply by choosing an alternative option for the submodel, they can specify that the number of instances changes dynamically during the course of a simulation. This is Simile's **population submodel**.

For population submodels, Simile provides symbols for specifying:

- the initial number of instances;
- the rate at which instances are created during the course of the simulation;
- the rate at which each individual creates new instances (this can vary between individuals); and
- the rule for destroying an existing individual.

Fig. 11 is a very simple example of what is possible in Simile. The model specifies a population of entities ("cells"). The **Cell** submodel is a multiple-instance submodel, like the ones in the previous sections. However, unlike those, this is a population submodel, which means that the number of instances (in this case, cells) can vary during the course of the simulation. (Visually, the submodel is rendered differently, with a double boundary all around.)

The symbol:

- **initial number** specifies how many cells there are at time zero.
- **immigration** specifies the rate at which new cells are created externally and enter the population. This is an absolute rate, i.e. number of individuals per unit of time. In the present case, this is perhaps biologically implausible, but Simile allows for this possibility.
- **reproduction** specifies the rate at which each cell produces new cells. This is a specific rate, i.e. number of individuals per individual per unit of time. This rate can vary between individuals, possibly depending on various attributes of each individual. In this case it is shown as being not influenced by cell attributes, but it could still be a value sampled from a random distribution, and thus still differ between cells.
- **death** specifies the rule for destroying an individual instance. In this example, we see that it is a function of age - perhaps a Boolean expression which is true when age exceeds some value.

In addition, we note that each cell has **x** and **y** coordinates, and an age. It also has a state variable for its **size**, with its rate of change being represented by a **growth** process. Finally, we see that we can calculate population level statistics: the number of cells (**total_number**) and the total size (**total_size**), using variables located outside the multiple-instance **Cell** submodel.

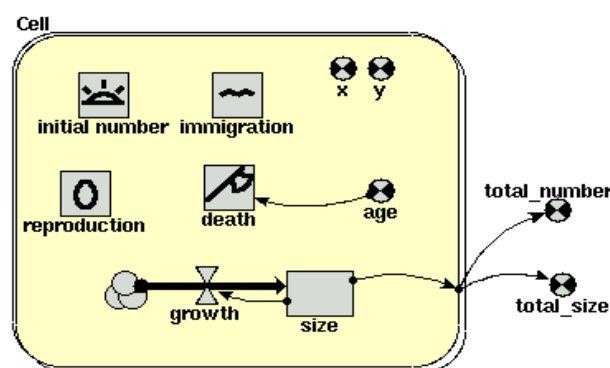


Fig. 11. A simple Simile population submodel

Since the population submodel can be arbitrarily complex (corresponding to SBML concepts such as compartments, reactions, assignment and rate variables, etc), it would seem that Simile does indeed have the capability to address this aspect of this proposed extension package.

2.4.4 Dynamic change in model structure

In this Section I will consider two ways in which the mathematical structure of the model can be changed

dynamically. The first involves switching between alternative ways of modelling some part of the model. The other involves changing which instances of a collection of entities are active and which are not.

Switching between alternative ways of modelling some part of the model

In Section 2.1 (**comp**: Hierarchical model composition), we discussed one aspect of modular modelling, namely the need to be able to replace one model component with another. Perhaps the alternative components implement alternative hypotheses about how to model a particular process. In that case, we are actually changing the model, by removing one submodel and replacing it with another.

Sometimes, however, it is convenient to be able to have both components in a model, and switch between them. The switching can either take place at model initialisation time, by setting a parameter acting as a switch, or dynamically during the course of the simulation. In Simile, both scenarios can be handled by the same mechanism, so I will not treat them separately.

Simile enables a 'condition' symbol to be inserted into any submodel. We have already seen this in the association submodel (Sections 2.2 and 2.3), in which case it is used to specify whether the association exists between pairs of individual entities. When placed in any other submodel, it indicates whether the calculations associated with that submodel should be performed or not, through the evaluation of a Boolean expression in the 'condition' symbol. If the expression evaluates to 'true', then the submodel 'exists' and the calculations are performed at run time. If not, then it doesn't, and they aren't. In the latter case, all outputs from the submodel return a null value (actually, zero), and any downstream calculations can then handle them appropriately.

The following diagram shows an example. The model has two components, which are considered to be alternative ways of calculating **h** from **a**. Which one is used depends on the value of the variable **switch**, the value of which is tested in the **condition** element of each of the two submodels. (For example, in **Component 1a**, the Boolean expression for **condition** could be `switch==1`.) As mentioned above, the value of **switch** could be either an input parameter, or calculated dynamically while the simulation is preceding, as a function of other variables in the model.

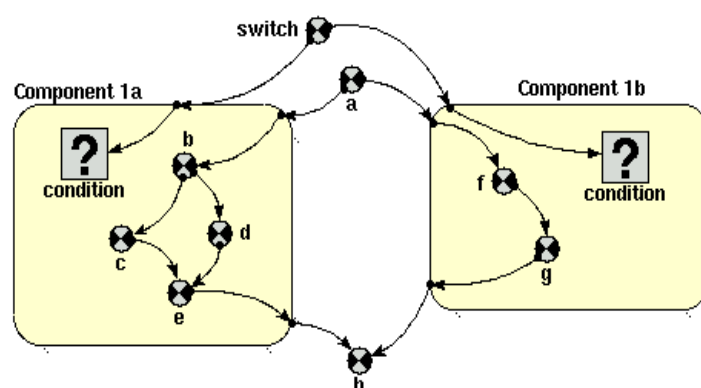


Fig. 12. An example of the use of the conditional submodel in Simile to switch between two alternative methods of calculating the value for a variable.

Note that there is nothing in Simile itself which obliges these two components to be mutually exclusive - that is entirely dependent on the modeller providing the correct expressions for the two **condition** symbols. Therefore, it is quite possible for the modeller to arrange for both submodels to be executed in some circumstances - for example, if one wanted to average the results of the two different methods for calculating **h**.

Selecting instances in a collection of instances

The problem statement for the dynamic structure requirement, quoted above, refers to a 'trick' of setting some compartments to be active, and others not. Sometimes this trick may actually be useful, and a good way of tackling a certain class of problem.

Consider a model with a certain number of molecular species, with various reactions between them. This is of course the basic use-case for SBML, and one can model it using named species and reactions. However, we could also approach the problem by thinking in terms of a collection of species, a possibility envisaged by the authors of the **arrays** package. In that case, it could well be desirable to set a flag against each species, indicating whether it is present or not in a particular version of the model, or indeed at a particular stage during the

course of a simulation run. This would allow us to have a general-model which could be configured for a particular collection of species via an external data file, rather than a separate model for different sets of species.

This can be achieved in Simile simply by having a multiple-instance submodel - called say **Species** - with a **condition** symbol, as shown below. **Species** is a multiple-instance submodel (though it is hard to tell this - look for the small dots right-hand and bottom corners). The condition tests the ID of each instance (an integer going from 1 up to the number of instances) against a selector value, to decide if that instance is selected. As before, this could happen dynamically, since the selector could depend on the state of the rest of the system. The **amount** compartment and associated flows is simply shorthand to represent the fact that there is a state variable associated with each species, which changes through processes (reaction and/or transport). In this simple example, I have not included the association submodel for reactions between species, since that is not relevant to the issue under discussion.

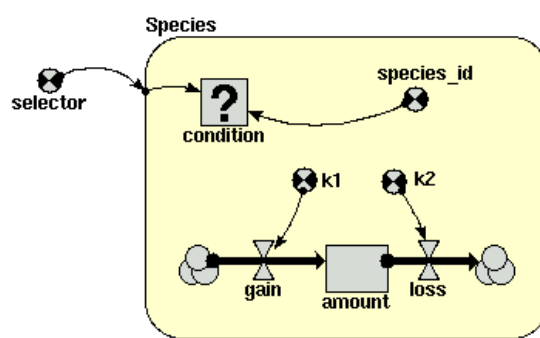


Fig. 13. Use of the Simile condition symbol to activate or de-activate instances of a multiple-instance submodel.

2.4.5 Assessment of Simile's ability to handle the requirements of the 'dyn' package

In this section, we have seen that Simile is able to handle the two main requirements of the **dyn** package:. The ability to dynamically create and destroy instances of some type of entity can be handled using the Simile population submodel. The ability to dynamically change the mathematical structure of the model can be handled by making some submodels conditional.

2.5 Package 'multi': Multi-state, multi-component species

2.5.1 Rationale and current status of the package

Home page

http://sbml.org/Community/Wiki/SBML_Level_3_Proposals/Multistate_and_Multicomponent_Species

Rationale

"Many models and modelers want to represent biochemical species that have internal structure or state properties. These may involve molecules that have multiple potential states, such as a protein that may be covalently modified, and molecules that combine to form heterogeneous complexes located among multiple compartments. The resulting system of reactions needs to be generated based on the combinatorial possibilities inherent in the definition of the species and on user-specified rules. The purpose of the SBML Level 3 Multistate and Multicomponent Species package is to provide a means for doing this in SBML."

Current status

There is one active proposal, dated March 2010.

This proposal links to various discussions and alternative proposals dating back to 2001.

2.5.2 A use-case example

Figs. 14 and **15** are taken from a presentation by Michael Hucka at the Genomes to Systems 2008 Conference in Manchester, U.K. (pers. comm.). They were presented as a typical (if simple) use-case for the proposed SBML Level 3 **multi** package. I have therefore taken this scenario as the precise basis for an implementation of the problem in Simile, though I also show how simple changes to the scenario can be easily implemented in the Simile representation. There is also a worked example in the package proposal, but I found this one easier to understand and use for communication.

The scenario is based on a type of molecule (CaMKII) which can exist in 6 possible states, representing all possible combinations of two 'features' (to use the term used by the authors of the proposed **multi** package). The first feature is **camkii_activity**, which can have 3 possible values (**inactive**, **active** and **inhib**). The other is **T286**, which can have 2 possible values (**phos** and **unphos**). 3x2 gives the 6 possible states the molecule can be in, which can be considered as 6 distinct molecular species.

- Species type CaMKII
 - feature "camkii_activity" with possible values {"active", "inactive", "inhib"}
 - feature "T286" with possible values {"unphos", "phos"}
- A species of type CaMKII could exist in 6 different states:
 - (camkii_activity → inactive, T286 → unphos)
 - (camkii_activity → inactive, T286 → phos)
 - (camkii_activity → active, T286 → unphos)
 - (camkii_activity → active, T286 → phos)
 - (camkii_activity → inhib, T286 → unphos)
 - (camkii_activity → inhib, T286 → phos)

Fig. 14. The six possible states for the CaMKII molecule (from a presentation by M. Hucka, 2008).

One reaction that the **unphos** form of CaMKII can participate in is with a phosphate molecule, which causes it to change state to **phos**, as illustrated by the following example of a 'selector' rule from Michael Hucka's presentation. The figure also shows that this reaction can proceed for any value of the **camkii_activity** feature.

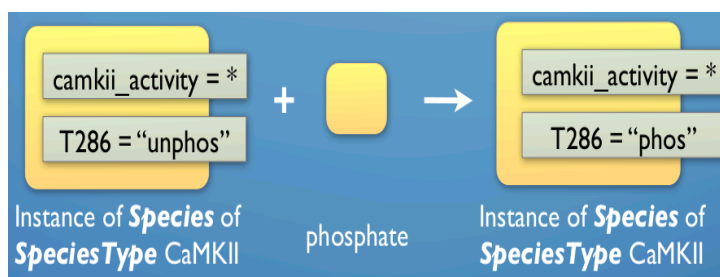


Fig. 15. An example of a 'selector' rule for a reaction involving CaMKII (from a presentation by M. Hucka, 2008)

So, the problem that the **multi** package is intended to address is two-fold. First, it should enable the modeller to specify concisely the existence of a potentially large number of similar molecules, which differ only with respect to one or more 'features', without having to list each form as a separate species. Second, it should allow the modeller to specify that some reaction takes place for some subset of the features, without having to list each equation separately.

2.5.3 Implementing the use-case example in Simile

One way in which the problem can be addressed in Simile is shown in **Fig. 16** (alternative approaches are possible).

The model consists of two submodels. The submodel **CaMKII_species** is a multiple-instance submodel which is specified as having 6 instances. (In fact, the model allows this number to be pruned - this will be discussed below) The submodel **Reaction** defines an association between a CaMKII species which is a substrate in a reaction, and a CaMKII species which is a product in a reaction. It will have as many instances as there are forms of the CaMKII molecule which can participate in the reaction: in the above case, that is 3 (the 3 values for **camkii_activity**).

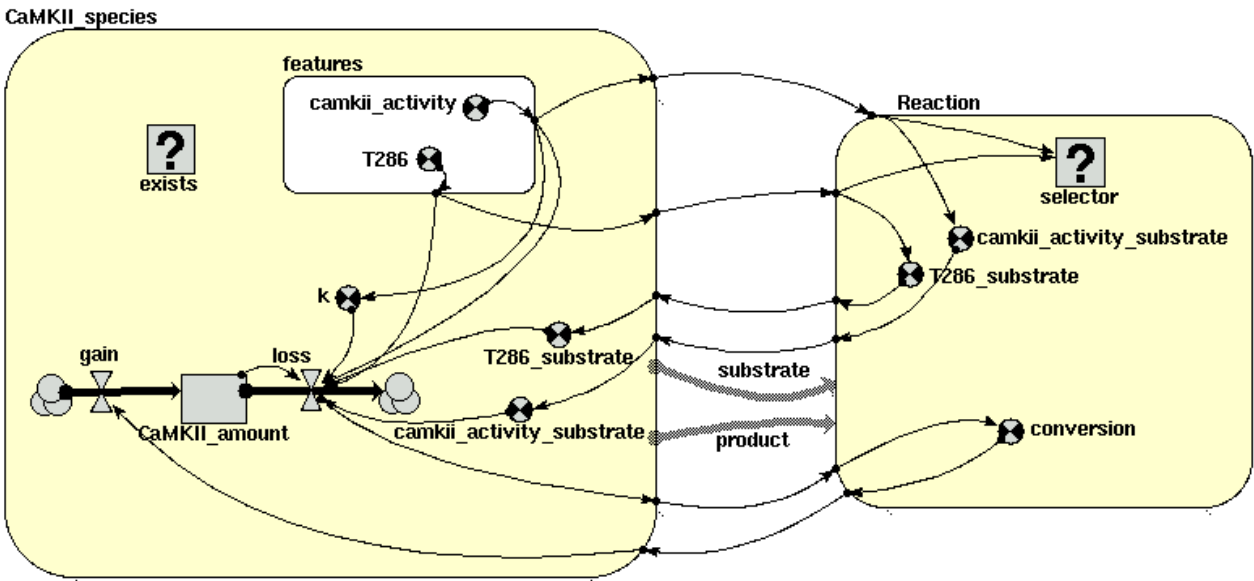


Fig. 16. Simile model diagram for the CaMKII example. The left-hand submodel represents the (up to) 6 different forms the molecule can take. The right-hand submodel handles a reaction involving form(s) of the CaMKII molecule as specified in the **selector** condition.

The submodel 'CaMKII_species'

Each of the 6 instances has a unique combination of values for the two 'features'. The variables **camkii_activity** and **T286** in the submodel features gives the values for these two features for each of the 6 instances of the submodel.

Value for the feature camkii_activity	Value for the feature T286
inactive	unphos
inactive	phos
active	unphos
active	phos
inhib	unphos
inhib	phos

Table 2. Table giving the **camkii_activity** and **T286** values for all 6 instances of the **CaMKII_species** submodel.

The reaction dynamics themselves are represented very basically, since the point of the exercise is not to show how submodel SBML-style reactions are implemented within a stock-and-flow paradigm, but to show how Simile handles the additional requirements introduced in the **multi** package. So, for example, the phosphate species has been left out (but could easily be included). What follows is a brief description of the stock-and-flow structure

that has been introduced to show that reaction dynamics can be included within the suggested framework.

There is a single state variable, for **CaMKII_amount**. Since this is inside a 6-instance submodel, there are in fact up to 6 values for this (dependent on whether any pruning of the number of combinations has been specified in the **exists** condition symbol), once for each combination of **camkii_activity** and **T286**. The rate of change of each **CaMKII_amount** is governed by a loss (when it is used up in a reaction), and a gain (when it is created by a reaction). For any one CaMKII species, the loss flow will be positive and the gain flow zero, or vice-versa, depending on its role in the reaction.

The loss term (in this simple example) is a simple function of **CaMKII_amount** and a rate constant, **k**. The role of the other 4 influences (from **camkii_activity**, **T286**, **camkii_activity_substrate** and **T286_substrate**) are to determine whether the flow actually operates for a given species: this depends on the allowable forms of CaMKII substrate defined in the **Reaction** submodel.

The **exists** conditional symbol is used to (potentially) prune the number of species of CaMKII which actually exist in the model. If that symbol was not present, then all 6 species would be present. If a conditional symbol is present in a submodel, then the actual number of instances can be less than that declared in the submodel properties box. For example, if we take an influence across from **camkii_activity** and enter the boolean expression

```
not(camkii_activity=="inactive")
```

then only 4 instances would exist (the last 4 rows in the above table).

Note that this provides a simple mechanism for addressing one of the requirements of the **multi** package, that it should be possible to describe a molecule with a number of features and perhaps billions of combinations, without actually having to create space in memory for representing each one.

The submodel 'Reaction'

The submodel **Reaction** is a Simile association submodel, defining an association (in the UML sense) between pairs of instances: in this case, between pairs of CaMKII species. Each pair consists of the CaMKII species which plays the role of the substrate and the CaMKII species which plays the role of the product of the same reaction. In this case, we know (from Michael Hucka's original outlining of the problem) that the **camkii_activity** feature will have the same value for substrate and product in the reaction, and the **T286** feature will have a different value (**unphos** for the substrate, **phos** for the product).

The conditional symbol, **selector**, defines the rule which defines whether a particular reaction takes place. It corresponds precisely to the use of the term 'selector' in the **multi** proposals. For example, the following is the selection rule used by Michael Hucka in his example expressed in Simile terms:

```
substrate_camkii_activity==product_camkii_activity and substrate_T286=="phos" and  
product_T286=="unphos"
```

This states that the reaction can proceed between two CaMKII species which have the same value for the **camkii_activity** feature, and, for the **T286** feature, a value of **phos** for the substrate and a value of **unphos** for the product.

There are two things to note about this. The first is that Simile supports enumerated types, which enables variables to have values taken from a defined vocabulary. This makes this rule quite readable. The second is that the first part of the rule ("substrate_camkii_activity==product_camkii_activity") makes explicit something which is implicit in the example given by Michael Hucka, namely that the * operator in a selector means "for all possible values, with the same value being used on each side of the reaction". This explicitness would seem desirable, since there may be occasions when the modeller does not actually want the implicit meaning to be imposed.

An alternative rule could be:

```
substrate_camkii_activity==product_camkii_activity and  
substrate_camkii_activity=="active" and substrate_T286=="phos" and  
product_T286=="unphos"
```

which further constrains the reaction to CaMKII molecules with value **active** for the **camkii_activity** feature. It

is thus pretty straightforward to tailor the selector rule for the requirements of a particular reaction.

How does it work?

When the model is initialised, Simile creates the specified number of instances for the **CaMKII_species** submodel, based on the number specified in the submodel's properties dialogue, pruned by the conditional rule in the **exists** symbol. It also creates as many instances of the **Reaction** submodel as are specified by the rule in the **selector** element.

During the course of the simulation, the loss flow for the **CaMKII_amount** state variable is calculated. This value is used to reduce the amount in the substrate state variable. It is also transferred, via the **conversion** variable in the **Reaction** submodel, to the gain flow for the product state variable.

The following graph shows a sample run. In this case, the **exists** rule was set to true for all 6 instances of **CaMKII_species**, while the **selector** rule was set to restrict the reaction to the **active** and **inhib** types of CaMKII. Also, the value for the rate constant, **k**, was set to be different for different values of the **camkii_activity** feature. All 6 initial values for **CaMKII_amount** are set to 100.

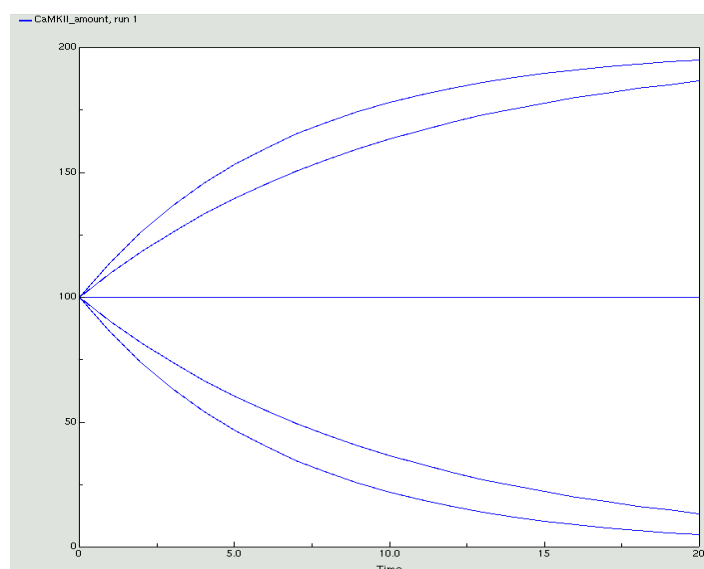


Fig. 17. Result of running the Simile version of the CaMKII model. See text for details.

The bottom and top line are for the **phos** and **unphos** forms of CaMKII with the value **inhib** for the **camkii_activity** feature; the inbetween lines are for the **active** form (assuming a different rate constant); and the horizontal line is in fact two lines, for the two **inactive** forms, which do not participate in the reaction.

2.5.4 Assessment of Simile's ability to handle the requirements of the 'multi' package

My guess is that the reaction of most members of the SBML community, including those involved in developing proposals for Level 3 packages, will be "Yes, but...". Yes, the approach presented here will handle at least some proportion of the use cases which have motivated the proposal of a multi-species, multi-compartment package. But, the way that the problem needs to be formulated within Simile (and/or within a markup language which is based on the same underlying data model) seems unfamiliar and strange.

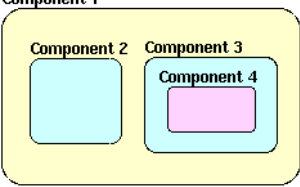
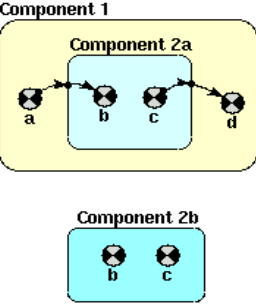
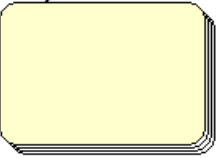
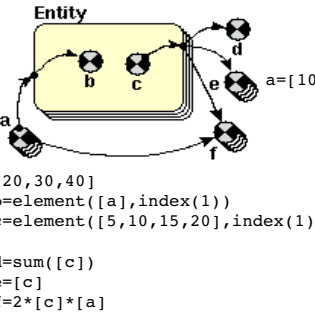
Whether these concerns are sufficient to cause this approach to be rejected outright is, of course, up to the community to decide. All I can do is to demonstrate that (a) the problem is expressible within Simile's representational formalism, and (b) that it is computable - i.e. that a tool (in fact, a code-generator) already exists which knows how to compute with a model which includes a **multi** specification.

Of all the exercises described in this document, showing how requirements addressed by the various SBML Level 3 packages can be represented in Simile, this is the most interesting, and revealing of Simile's representational

flexibility. The reason for this is that this is a class of problem that (at least for this Simile user) has not cropped up before. And yet, the solution is reasonably straightforward, is reasonably understandable, and readily adapted to alterations in the problem description. This demonstrates the flexibility which results from the generic submodel mechanism which Simile provides.

3 Summary of SBML Level 3 packages and how they would be handled in Simile

The following table summarises the ways in which the various SBML Level 3 packages could be handled in Simile. Its main purpose is to show the uniformity of the approach which comes from the Simile data model, with a small number of constructs being used for achieving the various solutions.

SBML Level 3 package	Description of package or sub-topic	Simile approach	Skeleton Simile diagram	Comments
comp	Hierarchical nesting of components	Standard Simile feature: submodels can be nested to any depth.		Nesting of Simile submodels is a generalised concept of containership. Simile only supports inclusion of a component (submodel) by reference if it is a compiled model (DLL), so in this case it is only possible for (e.g.) Component 2 and Component 4 to be two instances of the same, externally-defined component if it has been compiled into a DLL.
	Modular modelling	Simile submodels can be readily swapped in and out. Links with the rest of the model are automatically re-made, using a separate 'interface specification file', which defines the input/output bindings of a submodel in the context of a particular model.		Simile models/submodels are not 'black boxes' with a defined interface (unless they have been compiled into a DLL). Rather, any input can be connected to some other part of the model, and any variable can be treated as an output. In this example, the Interface Spec File contains the links from a-to-b, and c-to-d. This enables Component 2a and Component 2b to be swapped with each other, and all links automatically re-made.
arrays	Arrays of objects	Standard Simile feature: the 'multiple-instance submodel'.		A Simile model can be defined as 'multiple-instance'.
	Array variables	Rich support for arrays in Simile, including the equation language.	 <pre> a=[10, 20, 30, 40] b=element([a],index(1)) c=element([5,10,15,20],index(1)) d=sum([c]) e=[c] f=2*[c]*[a] </pre>	Array variables can be created explicitly by the user (for example, a), or implicitly by exporting a variable from a multiple-instance submodel (e.g. e). f is calculated from both types of array: every element is 2 times the corresponding elements of [c] and [a].

SBML Level 3 package	Description of package or sub-topic	Simile approach	Skeleton Simile diagram	Comments
spatial/geom	Representation of spatial aspects and geometry.	Simile has no built-in spatial or geometrical concepts, <u>but</u> a wide range of spatial configurations and geometries can be modelled.		<p>The upper figure shows, in a generic fashion, how multiple spatial units (e.g. grid squares, hexagons, circles, polygons) which do <u>not</u> interact with each other might be represented, using a multiple-instance submodel.</p> <p>'location' stands for variables which represent locational aspects, e.g. [row, column], [x,y,x], etc.</p> <p>'geometry' stands for variables which represent geometrical attributes, e.g. [radius] for a circle, [height,width] for a rectangle, etc.</p>
dyn	Dynamically changing number of individuals	Standard Simile feature: the 'population submodel' (a form of 'multiple-instance submodel').		'a' represents an arbitrarily-complex set of state, rate and intermediate variables. 'init'a and 'immigrate' are population attributes (initial number, and number created per unit of time). 'reproduce' and 'destroy' are individual attributes, and can thus vary between individuals depending on their other attributes.
	Dynamically-changing structure of model	Any submodel, or instance(s) of a multiple-instance submodel, can be made conditional.		<p>The upper model shows how the structure of the model can be changed dynamically, by using the variable 'a' to determine which of two submodels (Component 2a or Component 2b) are used.</p> <p>The lower model contains a multiple-instance submodel, Component 3, whose individual instances can exist or not, depending on the value of 'condition' for that instance.</p>
multi	Multiple forms of a given (chemical) species type, with a reaction (controlled by a selector), applying to a subset of forms.	Model using a (possibly conditional) multiple-instance submodel for the different forms, and an association submodel for the conversion of one form to another.		In this example, the two 'features' are a and b, with each instance of 'Species type' having a unique pair of values. 'selector' is used to define substrate-product pairs for a particular reaction, and 'convert' is used to transfer the 'loss' of one form of the species to become the 'gain' of the other form.

4 Discussion and conclusions

In this section, I will assess the strength of the claims made for the Simile data model, and the potential advantages and disadvantages of this approach. I will then explore the implications of adopting this approach within SBML Level 3.

4.1 Assessment

This paper has described an approach for representing structural complexity in simulation models based on a small number of constructs: the multiple-instance submodel, the association submodel, and array variables. I have tried to show that this data model is capable of handling most of the requirements of the SBML Level 3 packages that deal with model structure. These include:

- nesting of components;
- arrays of objects;
- array variables;
- spatial modelling;
- representation of geometry;
- dynamic objects;
- conditional components;
- rule-based modelling for multiple forms of a given molecular species type.

Simile does not support model composition by reference to externally-defined models. However, this should be seen as a limitation of Simile itself, rather than as undermining the generic, unified approach advocated in this document. The current **comp** package proposal already proposes a notation for allowing this, and that should be easily factored into any developments based on a unified approach. Also, Simile has no notation for allowing partial differential equations to be directly expressed, but I am not sure to what extent this is considered to be an important requirement in SBML Level 3.

The most obvious advantage of the Simile data model is that it removes the need to devise a number of separate packages. This in turn has benefits for modellers, for SBML software tool developers, and for the community as a whole:

For modellers, it means that they need learn only one formalism, and can then apply this to a variety of apparently different requirements. This applies both to the people constructing models, and to those who need to be able to 'read' a given model in order to understand it. It is true that the effort needed to understand this formalism is probably greater than for any one of the currently-proposed packages (since it is more abstract), but the investment in mastering this knowledge can then be applied across a broad range of SBML models.

For tool developers, it means that they need only implement a mechanism for handling the construct described here (and Simile proves that this is indeed feasible), rather than invest time in coding up mechanisms for a variety of separate packages.

For the community as a whole, it means that the ability to handle most models in most software tools will be greatly increased. The SBML Level 3 package concept was introduced in order to allow tool developers to state which packages they supported. This is an understandable aim in itself, but it opens the door to the situation where models which make use of certain packages cannot be handled by the tools which do not support those packages. This seems to go against the initial goals of SBML, which is to foster the interchange of models between various tools. By adopting a unified approach, this fragmentation should be greatly reduced.

The main disadvantage of the approach advocated here is the same as that associated with most attempts to work at a more generic level, namely that a particular concept has to be expressed using a number of more abstract, low-level constructs, rather than a smaller number of higher-level, domain-specific constructs which relate more closely to the modeller's conceptual universe. This is most evident when considering spatial and geometry modelling: it is easier to incorporate grid-space-with-four-cell-neighbourhood into a model if that is available as a defined term in the language rather than if one has to capture the semantics of that particular spatial arrangement using more basic elements (as one has to do in Simile). This applies both to the person constructing a model, and to those who are trying to understand it. This can be seen as the price one has to pay for the advantages of a generic way of expressing things.

There are, however, ways of mitigating this effect. First, modellers will tend to find that they are using fairly standard concepts, and can draw on template patterns that others have developed. So, for example, the above form of spatial representation can usually be quickly recognised by a particular pattern of Simile symbols and a particular conditional expression specifying the neighbourhood association between grid squares. Second, it is possible to envisage a library of model fragments which are available for anyone to download and incorporate in their model. Such fragments can then be used in ways similar to a set of built-in concepts, but without the restrictions that come with having them hard-wired into the language.

Another potential disadvantage of Simile's generic approach for representing model structure relates to computational efficiency. If a model-representation language has a built-in term for (say) a grid representation of space with a gradient-controlled transport process between cells, then a software tool developed for handling models expressed in this language can have code built in specifically designed for handling this construct. One would therefore expect the software tools developed for a language with domain-specific constructs to be computationally more efficient than software tools developed for a more generic (abstract) model-representation language.

However, it is possible that the more generic approach can be just as efficient, and possibly even more efficient, than one based on domain-specific constructs. It can be just as efficient if the simulator is able to recognise certain patterns (e.g. grid-square space with 4-cell neighbourhood), and invoke specific code for handling this. It can potentially be more efficient because the more generic representation should encourage code developers to invest more in developing computationally-efficient algorithms - the investment will be re-paid over a much larger number of models.

4.2 What are the implications for SBML Level 3?

This document has concentrated exclusively on how the requirements of the various SBML Level 3 packages can be met using the Simile abstract data model. I have done this by using examples of Simile diagrams, rather than SimileXML, in order to avoid confusing things with details of Simile's own XML notation (which in any case is under revision).

This document is not, as it stands, an SBML Level 3 package proposal. Given the radical nature of making such a proposal - in effect, side-lining 6 of the current proposals - it seems appropriate to open up a discussion on the general desirability of adopting a generic data model, before investing the time and effort in developing a specific proposal. This discussion should concentrate on general principles, rather than on the specific details of an extension to the SBML Level 3 Core specification. If the SBML community agrees that it is a good idea to go down this route, then the development of an actual proposal could take place within the community.

A couple of comments are relevant, if things get that far. First, I think that it would be quite straightforward to develop an SBML Level 3 extension package incorporating the generic data model presented here. After all, the **comp** package has a `<listOfSubmodels>` element and has proposals for linking variables in different submodels, and the **arrays** package specifies a way for representing array variables in SBML-MathML, so it should be easy to incorporate the small number of extra constructs required.

Second, as mentioned in Section 1.2.4, there is a strong analogy between Simile's data model and the UML class diagram. UML has an XML syntax (XMI), so one possibility could be to leverage this rather than coming up with one's own notation (in much the same way that MathML is used for encoding mathematical expressions). It is in general considered to be a good idea to build on existing standards where appropriate. Going further, consideration could be given to using SysML, a subset and extension of UML for engineering models, since that provides extra notation for describing parameters and links between components.

4.3 What next?

Discussion of the ideas raised in this document should take place on **[sbml-discuss]**, the mailing list/web forum for discussing SBML and its future. Select this forum from the list of SBML forums at <http://sbml.org/Forums>, to see past postings and to register for submitting your own comments. Alternatively, please feel free to email me directly, at r.muettzelfeldt@ed.ac.uk: I very much welcome any feedback.