SBML Level 3 Package Specification

Hierarchical Model Composition

Lucian P. Smith
lpsmith@u.washington.edu
Department of Bioengineering
University of Washington
Seattle, WA, US

Michael Hucka
mhucka@caltech.edu
Computing and Mathematical Sciences
California Institute of Technology
Pasadena, CA, US

23 December 2011

Version 1 (Draft)

This is a working draft of the specification for the SBML Level 3 package "comp". It is not a normative document. Please send comments and other feedback to the Package Working Group mailing list, sbml-comp@lists.sourceforge.net.

The latest release, past releases, and other materials related to this specification are available at
http://sbml.org/Documents/Specifications/Packages/Hierarchical_Model_Composition

This release of the specification is available at
http://sbml.org/Documents/Specifications/Packages/Hierarchical_Model_Composition/XXXXX

Contents

1 Introduction

In the context of SBML, "hierarchical model composition" refers to the ability to include models as submodels inside another model. The goal is to support the ability of modelers and software tools to do such things as (1) decompose larger models into smaller ones, as a way to manage complexity; (2) incorporate multiple instances of a given model within one or more enclosing models, to avoid literal duplication of repeated elements; and (3) create libraries of reusable, tested models, much as is done in software development and other engineering fields.

SBML Level 3 Version 1 Core (Hucka et al., 2010), by itself, 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) could use annotations to augment a plain SBML Level 3 model with the necessary information to allow a software tool to compose a model out of submodels. However, such solutions would be proprietary and tool-specific, and not conducive to interoperability. There is a clear need for an official SBML language facility for hierarchical model composition.

This document describes a specification for an SBML Level 3 package that provides exactly such a facility. Figure 1 illustrates some of the scenarios targeted by this package.

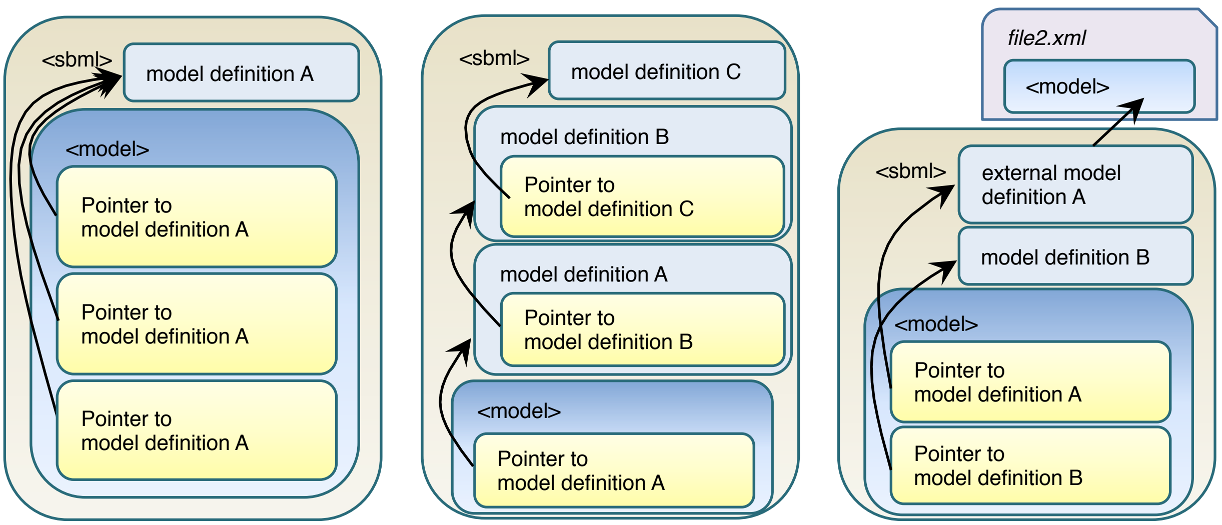

Figure 1: *Three different examples of model composition scenarios. From left to right: (1) a model composed of multiple instances of a single, internally-defined submodel definition; (2) a model composed of a submodel that is itself composed of submodels; and (3) a model composed of submodels, one of which is defined in an external file.*

The effort to create a hierarchical model composition mechanism in SBML has a long history, which we summarize in Section 2. It has also been known by different names. In the beginning, it was called *modularity* because it allows a model to be divided into structural and conceptual modules. It was renamed *model composition* when it became apparent that the name "modularity" was easily confused with other notions modularity, particularly XHTML 1.1 (Pemberton et al., 2002) modularity (which concerns decomposition into separate files). To make clear that the purpose is structural *model* composition, regardless of whether the components are stored in separate files, the SBML community adopted the name SBML *Hierarchical Model Composition.*

To support a variety of composition scenarios, this package provides for optional black-box encapsulation by means of defined data communication interfaces (here called *ports*). In addition, it also separates model *definitions* (i.e., blueprints, or templates) from *instances* of those definitions, it supports optional external file storage, and it allows recursive model decomposition with arbitrary submodel nesting.

1.1 Proposal corresponding to this package specification

This specification for Hierarchical Model Composition in SBML Level 3 Version 1 is based on the proposal by the same authors, located at the following URL:

https://sbml.svn.sf.net/svnroot/sbml/trunk/specifications/sbml-level-3/version-1/comp/proposal

The tracking number in the SBML issue tracking system (SBML Team, 2010) for Hierarchical Model Composition package activities is 2404771. The version of the proposal used as the starting point for this specification is the version of August, 2011.

1.2 Package dependecies

The Hierarchical Model Composition package has no dependencies on other SBML Level 3 packages. It is also designed with the goal of being able to work seamlessly with other SBML Level 3 packages. For example, one can create a set of hierarchical models that also use Groups or Spatial Geometry features. (If you find incompatibilities with other packages, please contact the authors. Contact information is shown on the front page of this document.)

1.3 Document conventions

Following the precedent set by the SBML Level 3 Core specification document, we use UML 1.0 (Unified Modeling Language; Eriksson and Penker 1998; Oestereich 1999) class diagram notation to define the constructs provided by this package. We also use color in the diagrams to carry additional information for the benefit of those viewing the document on media that can display color. The following are the colors we use and what they represent:

- *Black*: Items colored black in the UML diagrams are components taken unchanged from their definition in the SBML Level 3 Core specification document.
- *Green*: Items colored green are components that exist in SBML Level 3 Core, but are extended by this package. Class boxes are also drawn with dashed lines to further distinguish them.
- *Blue*: Items colored blue are new components introduced in this package specification. They have no equivalent in the SBML Level 3 Core specification.

We also use the following typographical conventions to distinguish the names of objects and data types from other entities; these conventions are identical the conventions used in the SBML Level 3 Core specification document:

AbstractClass: Abstract classes are classes that are never instantiated directly, but rather serve as parents of other object classes. Their names begin with a capital letter and they are printed in a slanted, bold, sans-serif typeface. In electronic document formats, the class names defined within this document are also hyperlinked to their definitions; clicking your computer pointer on these items will, given appropriate software, switch the view to the section in this document containing the definition of that class. (However, for classes that are unchanged from their definitions in SBML Level 3 Core, the class names are not hyperlinked because they are not defined within this document.)

Class: Names of ordinary (concrete) classes begin with a capital letter and are printed in an upright, bold, sans-serif typeface. In electronic document formats, the class names are also hyperlinked to their definitions in this specification document. (However, as in the previous case, class names are not hyperlinked if they are for classes that are unchanged from their definitions in the SBML Level 3 Core specification.)

`SomeThing`, `otherThing`: Attributes of classes, data type names, literal XML, and generally all tokens *other* than SBML UML class names, are printed in an upright typewriter typeface. Primitive types defined by SBML begin with a capital letter; SBML also makes use of primitive types defined by XML Schema 1.0 (Biron and Malhotra, 2000; Fallside, 2000; Thompson et al., 2000), but unfortunately, XML Schema does not follow any capitalization convention and primitive types drawn from the XML Schema language may or may not start with a capital letter.

For other matters involving the use of UML and XML, we follow the conventions used in the SBML Level 3 Core specification document.

2 Background and context

The focus of this section is prior work on the topic of model composition in SBML. We also explain how the current specification relates to that prior work.

2.1 Prior work on model composition in SBML

The SBML community has discussed the need to add model composition to SBML since SBML's very beginning, some ten years ago. The formulation of model composition contained in the present document draws substantially from prior work. Before we turn to a narrative of the history that led to the current specification, we want to highlight a number of individuals for their inspirations and past work in the development of precursors to this package. These individuals are listed in Table 1.

Contributor	Affiliation	City and Country
Stefan Hoops	Virginia Bioinformatics Institute	Blacksburg, Virginia, US
Nicolas Le Novère	EMBL-European Bioinformatics Institute	Hinxton, Cambridge, UK
Andrew Finney	(Independent)	Oxford, UK
Martin Ginkel	Max Planck Institute for Dynamics of Complex Technical Systems	Magdeburg, DE
Wolfram Leibermeister	Max Planck Institute for Molecular Genetics	Berlin, DE
Ranjit Randhawa	Dept. of Computer Science, Virginia Tech.	Blacksburg, VA, US
Jonathan Webb	BBN Technologies	Cambridge, MA, US

Table 1: *List of individuals who made significant contributions to the development of prior SBML proposals that influenced the present version of hierarchical model composition.*

The first known written proposal for composition in SBML appeared in an internal discussion document titled *Possible extensions to the Systems Biology Markup Language* (Finney, 2000) principally authored by Andrew Finney (and, notably, written even before SBML Level 1 Version 1 was finalized in March of 2001). The first of the four titular possible extensions in that document concerns "submodels": the main model in a file can contain a list of submodels, each of which are model definitions only, and a list of submodel instantiations, each of which are references to model definitions. Finney's proposal also extends the syntax of SBML identifiers (the `SId` data type) to allow entity references using a dotted notation, in which `X.y` signifies element `y` of submodel instance `X`; the proposal also defines a form of linking model elements through "substitutions". In addition, the proposal also introduces the concept of validation through what it called the "expanded" version of the model (now commonly referred to as the "flattened" form, meaning translation to a plain SBML format that does not use composition features): if the flat version of the model is valid, then the model as a whole must also be valid.

In June of 2001, at the Third Workshop on Software Platforms for Systems Biology, Martin Ginkel and Jörg Stelling presented their proposal titled *XML Notation for Modularity* (Ginkel and Stelling, 2001), complete with an accompanying proposal document and sample XML file, partially in response to deficiencies or missing elements they believed existed in the proposal by Finney. In their proposal, Ginkel and Stelling present a "classic view" of modularity, where models are packaged as black boxes with interfaces. One of their design goals is to support the substitution of one module for another with the same defined interface, thereby supporting the simplification or elaboration of models as needed. Their proposal emphasizes the reuse of models and with the possibility of developing libraries of models.

Martin Ginkel presented an expanded version of that proposal (Ginkel, 2002) at in the July 2002 Fifth Workshop on Software Platforms for Systems Biology, in the hope that it could be incorporated into the definition of SBML Level 2 that was being developed at the time. This proposal clarified the need to separate model definitions from model instantiations, and, further, the need to designate one model per document as the "main" model.

In March of 2003, Jonathan Webb produced an independent proposal (Webb, 2003) and circulated it on the mailing list sbml-discuss@caltech.edu. This proposal included a unified, generic approach to making links and references to elements in submodels using XML XPath (Clark and DeRose, 1999). Previous proposals used separate mechanisms for species, parameters, compartments, and reactions. Webb also raised the issue of how to successfully resolve conflicting attributes of linked elements, debated whether formal interfaces were necessary or even preferable to directly access model elements, discussed type-checking for linkages, and discussed issues with unit incompatibilities. Around this time, Martin Ginkel formed the Model Composition Special Interest Group (Ginkel, 2003), a group that eventually reached 18 members (including Webb).

Model composition did not make it into SBML Level 2 when that specification was released in June of 2003, because the changes between SBML Level 1 and Level 2 were already substantial enough that software developers at the time expressed a desire to delay the introduction of composition to a later revision of SBML. Andrew Finney (now the co-chair of the Model Composition SIG) presented yet another proposal (Finney, 2003b) in May of 2003, even before SBML Level 2 Version 1 was finalized, that aimed to add model composition to SBML Level 3. With only two years having passed between SBML Level 1 and Level 2, the feeling at the time was that Level 3 was likely to be released in 2005 or 2006, and the model composition proposal would be ready when it was. However, Level 2 ended up occupying the SBML community longer than expected, with four versions of Level 2 produced to adjust features in response to user feedback and developers' experiences.

In the interim, the desire to develop model composition features for SBML continued unabated. Finney revised his 2003 proposal in October 2003 Finney (2003c); this new version represented an attempt to synthesize the earlier proposals by Ginkel and Webb, supplemented with his own original submodel ideas, and was envisioned to exist in parallel with another proposal by Finney, for arrays and sets of SBML elements (including submodels) (Finney, 2003a). Finney attempted to resolve the differences in the two basic philosophies (essentially, black-box versus white-box encapsulation) by introducing optional "ports" as interfaces between a submodel and its containing model, as well as including an XPath-based method to allow referencing model entities. The intention was that a modeler who wanted to follow the classic modularity (black-box) approach could do so, but other modelers could still use models in ways not envisioned by the original modeler simply by accessing a model's elements directly via XPath-based references. In both schemes, elements in the submodels were replaced by corresponding elements of the containing model. Finney's proposal also provided a direct link facility that allows a containing model to refer directly to submodel elements without providing placeholder elements in the containing model. For example, a containing model could have a reaction that converts a species in one submodel to a species in a different submodel, and in the direct-link approach, it would only need to define the reaction, with the reactant and product being expressed as links directly to the species defined in the submodels.

After Finney's last effort, activities in the SBML community focused on updates to SBML Level 2, and since model composition was slated for Level 3, not much progress was made for several years, apart from Finney including a summary of his 2003 proposal and of some of the unresolved issues in a poster (Finney, 2004) at the 2004 Intelligent Systems for Molecular Biology (ISMB) conference held in Glasgow.

Finally, in June of 2007, unplanned discussions at the Fifth SBML Hackathon (SBML Team, 2007) prompted the convening of a workshop specifically to revitalize the model composition package, and in September of 2007, the SBML Composition Workshop (Various, 2007) was held at the University of Connecticut Health Center, hosted by the Virtual Cell group and organized by Ion Moraru and Michael Blinov. The event produced several artifacts, still available online:

1. Martin Ginkel provided a list of goals for model composition (Ginkel, 2007), including use cases, and summarized many of the issues described above, including the notion of definition versus instantiation, linking, referencing elements that lack SBML identifiers, and the creation of optional interfaces. The list of goals also mentioned the need of allowing parameterization of instances (i.e., setting new numerical values that override the defaults), and the need to be able to "delete" or elide elements out of submodels. (He also provided a summary of ProMoT's model composition approach and a summary of other approaches.)

2. Andrew Finney wrote a list of issues and comments, recorded on the meeting wiki page (Finney, 2007); these included some old issues as well as some new ones:
   - There should perhaps be a flag for ports to indicate whether a given port must be overloaded.
   - There should be support for N-to-M links, when a set of elements in one model are replaced as a group, conceptually, with one or more elements from a different model.
   - The proposal should be generic enough to accommodate future updates and other Level 3 packages.

3. Wolfram Liebermeister presented his group's experience with SBMLMerge (Leibermeister, 2007), dealing with the pragmatics of merging multiple models. He also noted that the annotations in a composed model need to be considered, particularly since they can be crucial to successfully merging models in the first place.

4. On behalf of Ranjit Randhawa, Cliff Shaffer summarized Ranjit's work in the JigCell group on model fusion, aggregation, and composition (Randhawa, 2007). Highlights of this presentation and work include the following:

   - A description of different methods which all need some form of model composition, along with the realization that model fusion and model composition, though philosophically different, entail exactly the same processes and require the same information.
   - A software application (the JigCell Composition Wizard) that can perform conversion between types. The application can, for example, promote a parameter to a species, a concept which had been assumed to be impossible and undesirable in previous proposals.
   - The discovery that merging of SBML models should be done in the order Compartments → Species → Function Definitions → Rules → Events → Units → Reactions → Parameters. If done in this order, potential conflicts are resolved incrementally along the way.

5. Nicolas Le Novère created a proposal for SBML modularity in Core (Novère, 2007). This is actually unrelated to the efforts described above; it is an attempt to modularize a "normal" SBML model in the sense of divvying up the information into modules or blocks stored in separate files, rather than composing a model from different chunks. It was agreed at the workshop that this is a completely separate idea, and while it has merits, should be handled separately.

6. As a collective, the group produced an "Issues to Address" document (Various, 2007a), with several conclusions:

   - It should be possible to "flatten" a composed model to produce a valid SBML Level 3 Core model, and all questions of validity can then be simply applied to the flattened model. If the Core-only version is valid, the composed model is valid.
   - The model composition proposal should cover both designed-ahead-of-time as well as ad-hoc composition. (The latter refers to composing models out of components that were not originally developed with the use of ports or the expectation of being incorporated into other models.)
   - The approach probably needs a mechanism for deleting SBML model elements. The deletion syntax should be explicit, instead of being implied by (e.g.) using a generic replacement construct and omitting the target of the replacement.
   - It should be possible to link any part of a model, not just (e.g.) compartments, species and parameters.
   - The approach should support item "object overloading" (Various, 2007b) and be generally applicable to all SBML objects. However, contrary to what is provided in the JigCell Composition Wizard, changing SBML component types is not supported in object overloading.
   - A proposition made during the workshop is that elements in the outer model always override elements in the submodels, and perhaps that sibling linking be disallowed. This idea was hotly debated.
   - Interfaces (ports) are indeed considered helpful, but should be optional. They do not need to be directional as in the electrical engineering "input" and "output" sense—the outer element always overrides the inner element, but apart from that, biology does not tend to work in the directional way that electrical components do.
   - The ability to refer to or import external files may need a mechanism to allow an application to check whether what is being imported is the same as it was when the modeler created the model. The mechanism offered in this context was the use of MD5 hashes.
   - A model composition approach should probably only allow whole-model imports, not importing of individual SBML elements such as species or reactions. The reason is that model components are invariably defined within a larger context, and attempting to pull a single piece out of a model is unlikely to be safe or desirable.

- The model composition approach must provide a means to handle the conversion of units, so that the units of entities defined in a submodel can be made congruent with the entities that refer to them in the enclosing model.

During the workshop, the attendees worked on a draft proposal. Stefan Hoops acted as principal editor. The proposal for the SBML package (which was renamed *Hierarchical Model Composition* (Hoops, 2007)), was issued one day after the end of the workshop. It represented an attempt to summarize the workshop as a whole, and provide a coherent whole, suitable as a Level 3 package. It provided a brief overview of the history and goals of the proposal, as well as several UML diagrams of the proposed data structures. Hoops presented (Hoops, 2008) the proposal in August, 2008, at the 13th SBML Forum, and again at the 7th SBML Hackathon in March of 2009 as well as the 14th SBML Forum in September of 2009, in a continuing effort to raise interest.

Roughly concurrently, Herbert Sauro, one of the original developers of SBML, received a grant to develop a modular human-readable model definition language, and hired Lucian Smith in November of 2007 to work on the project. Sauro and Frank Bergmann, then a graduate student with Herbert, had previously written a proposal (Bergmann and Sauro, 2006) for a human-readable language that provided composition features, and this was the design document Smith initially used to create a software system that was eventually called *Antimony*. Through a few iterations, the design eventually settled on was very similar in concept (largely by coincidence) to that developed by the group at the 2007 Connecticut workshop: namely, with model definitions placed separately from their instantiations in other models, and with the ability to link (or "synchronize", in Antimony terminology) elements of models with each other. Because Antimony was designed to be "quick and dirty", it allowed type conversions much like the JigCell Composition Wizard, whereby a parameter could become a species, compartment, or even reaction. Synchronized elements could end up with aspects of both parent elements in their final definitions: if one element defined a starting condition and the other how it changed in time, the final element would have both. If both elements defined the same aspect (like a starting condition), the one designated the "default" would be used in the final version. Smith developed methods to import other Antimony files and even SBML models, which could then be used as submodels of other models and exported as flattened SBML.

At the SBML-BioModels.net Hackathon in 2010, in response to popular demand from people who attended the workshop, Smith put together a short presentation (Smith, 2010a) about model composition and some of the limitations he found with the 2007 proposal. He proposed separating the replacement concept (where old references to replaced values are still valid) from the deletion concept (where old references to replaced values are no longer valid). Smith wrote a summary of that discussion, added some more of thoughts, and posted it to the `sbml-discuss@caltech.edu` mailing list (Smith, 2010b). In this posting, he proposed and/or reported several possible modifications to the Hoops et al. 2007 proposal, including the following:

- Separation of *replacement* from *deletion*.
- Separation of model definition from instantiation.
- Elimination of ports, and the use of annotations instead.
- Annotation for identifying N-to-M replacements, instead of giving them their own construct.

The message to `sbml-discuss@caltech.edu` was met with limited discussion. However, it turns out that several of the issues raised by Smith were brought up at the 2007 meeting, and had simply been missed in the generation of the (incomplete) proposal after the workshop. The meeting attendees had, for example, originally preferred to differentiate deletions from replacements more strongly than by simply having an empty list of replacements, but omitted this feature because no better method could be found. Similarly, the separation of definitions from instantiations had been in every proposal up until 2007, and was mentioned in the notes for that meeting. The decision to merge the two was a last-minute design decision brought about when the group noted that if the XInclude (Marsch et al., 206) construct was used, the separation was not strictly necessary from a technical standpoint.

Smith joined the SBML team in September of 2010, and was tasked with going through the old proposals and synthesizing from them a new version that would work with the final incarnation of SBML Level 3. That version (the first version of this document) was presented at COMBINE in October 2010 (Smith and Hucka, 2010), and further discussed on the `sbml-discuss@caltech.edu` mailing list. At HARMONY in April of 2011, consensus was reached on a way forward for resolving the remaining controversies surrounding the specification, resulting in the current version of the document you are reading.

2.2 Genesis of the current formulation of the package

The present specification for Hierarchical Model Composition is an attempt to blend features of previous efforts into a concrete, Level 3-compatible syntax. The specification has been written from scratch, but draws strongly on the Hoops 2007 and Finney 2003 proposals, as well as, to some degree, every one of the sources mentioned above. Some practical decisions are new to this proposal, sometimes due to additional design constraints resulting from the final incarnation of SBML Level 3, but all of them draw from a wealth of history and experimentation by many different people over the last decade. Where this proposal differs from the historical consensus, the reasoning is explained, but for the most part, the proposal follows the road most traveled, and focuses on being clear, simple, only as complex as necessary, and applicable to the largest number of situations.

2.3 Design goals for the Hierarchical Model Composition package

The following are the basic design goals followed in this package:

- *Allow modelers to build models by aggregation, composition, or modularity.* These methods are so similar to one another, and the process of creating an SBML Level 3 package is so involved, that we believe it is not advantageous to create one SBML package for aggregation and composition, and a separate package for modularity. Users of the hierarchical model composition package should be able to use and create models in the style that is best suited for their individual tasks, using any of these mechanisms, and to exchange and reuse models from other groups simply and straightforwardly.

- *Interoperate cleanly with other packages.* The rules of composition should be such that they could apply to any SBML element, even unanticipated elements not defined in SBML Level 3 Core and introduced by some future Level 3 package.

- *Allow models produced with these constructs to be valid SBML if the constructs are ignored.* As proposed by Nicolas Le Novère (Novère, 2003) and affirmed by the SBML Editors (The SBML Editors, 2010), whenever possible, ignoring elements defined in a Level 3 package namespace should result in syntactically-correct SBML models that can still be interpreted to some degree, even if it cannot produce the intended simulation results of the full (i.e., interpreting the package constructs) model. For example, inspection and visualization of the Core model should still be possible.

- *Ignore verbosity of models.* We assume that software will deal with the "nuts and bolts" of reading and writing SBML. If there are two approaches to designing a mechanism for this hierarchical composition package, where one approach is clear but verbose and the other approach is concise but complex or unobvious, we prefer the clear and verbose approach. We assume that software tools can abstract away the verbosity for the user. (However, tempering this goal is the next point.)

- *Avoid over-complicating the specification.* Apart from the base constructs defined by this specification, any new element or attribute introduced should have a clear use case that cannot be achieved in any other way.

- *Allow modular access to files outside the modeler's control.* In order to encourage direct referencing of models (e.g., to models hosted online on sites such as BioModels Database (`http://biomodels.net/database`), whenever possible, we will require referenced submodels only to be in SBML Level 3 Core format, and not require that they include constructs from this specification.

- *Incorporate most, if not all, of the desirable features of past proposals.* The names may change, but the aims of past efforts at SBML model composition should still be achievable with the present specification.

3 Package syntax and semantics

In this section, we define the syntax and semantics of the Hierarchical Model Composition package for SBML Level 3 Version 1. We expound on the various data types and constructs defined in this package, then in Section 4, we provide complete examples of using the constructs in example SBML models.

3.1 Namespace URI and other declarations necessary for using this package

Every SBML Level 3 package is identified uniquely by an XML namespace URI. For an SBML document to be able to use a given SBML Level 3 package, it must declare the use of that package by referencing its URI. The following is the namespace URI for this version of the Hierarchical Model Composition package for SBML Level 3 Version 1:

"`http://www.sbml.org/sbml/level3/version1/comp/version1`"

In addition, SBML documents using a given package must indicate whether understanding the package is required for complete mathematical interpretation of a model, or whether the package is optional. This is done using the attribute `required` on the `<sbml>` element in the SBML document. For the Hierarchical Model Composition package, the value of this attribute must be set to "`true`".

The following fragment illustrates the beginning of a typical SBML model using SBML Level 3 Version 1 and this version of the Hierarchical Model Composition package:

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level3/version1/core" level="3" version="1"
      xmlns:comp="http://www.sbml.org/sbml/level3/version1/comp/version1" comp:required="true">
```

3.2 Primitive data types

Section 3.1 of the SBML Level 3 specification defines a number of primitive data types and also uses a number of XML Schema 1.0 data types (Biron and Malhotra, 2000). We assume and use some of them in the rest of this specification, specifically `boolean`, `ID`, `SId`, `SIdRef`, `UnitSId`, `UnitSIdRef`, and `string`. The Hierarchical Model Composition package also makes use of or defines other primitive types; they are described below.

3.2.1 Type `anyURI`

Type `anyURI` is defined by XML Schema 1.0. It is a character string data type whose values are interpretable as URIs (*Universal Resource Identifiers*; Harold and Means 2001; W3C 2000) as described by the W3C document RFC 3986 (Berners-Lee et al., 2005).

3.2.2 Type `PortSId`

The type `PortSId` is derived from `SId` (SBML Level 3 Version 1 Core specification Section 3.1.7) and has identical syntax. The `PortSId` type is used as the data type for the identifiers of **Port** objects (Section 3.4.3) in the Hierarchical Model Composition package. The purpose of having a separate type for such identifiers is to enable the space of possible port identifier values to be separated from the space of all other identifier values in SBML. The equality of `PortSId` values is determined by an exact character sequence match; i.e., comparisons of these identifiers must be performed in a case-sensitive manner.

3.2.3 Type `PortSIdRef`

Type `PortSIdRef` is used for all attributes that refer to identifiers of type `PortSId`. This type is derived from `PortSId`, but with the restriction that the value of an attribute having type `PortSIdRef` must match the value of a `PortSId` attribute in the relevant model; in other words, the value of the attribute must be an existing port identifier in the referenced model. As with `PortSId`, the equality of `PortSIdRef` values is determined by exact character sequence match; i.e., comparisons of these identifiers must be performed in a case-sensitive manner.

3.3 The extended SBML class

The top level of an "SBML document" is a container whose structure is defined by the object class **SBML** in the SBML Level 3 specification. In Level 3 Core, this container can contain only one model, an object of class **Model**. The Hierarchical Model Composition package allows SBML documents to contain *more* than one model.

To explain how this is accomplished, we first need to introduce some terminology. In the approach taken here, we make a distinction between (a) the definition of a model, before it is actually used anywhere, and (b) its actual use:

- The term *model definition* refers to the former case; that is, the definition of a model, before it is used. A model definition is akin to a Platonic ideal: it may be a complete model in and of itself, but until it is instantiated, it exists only as a concept.
- The term *submodel* refers to actual use of a model definition. A submodel is an instantiation or instance of a previously-defined model: it is the realization of that model inside another model. From the perspective of the model that contains this submodel, the model definition has come into being, and now exists as something that can be used (and possibly modified and adapted).

It may be helpful to contrast these terms with those in other approaches to model composition. Some approaches call the model definitions themselves the "submodels". We avoid that usage because, in the present formulation, model definitions must be valid **Model** objects in and of themselves, and might never appear inside other models in the SBML document where they are defined. (This might be the situation, for example, if the document defines multiple models purely to serve as a sort of component library used by other files.) We reserve the term "submodel" specifically for the *instance of a model inside a containing model*. Another term used in other schemes is "model template", which is close to what is intended by "model definition" here, but "template" implies something that is incomplete and needs to be filled in. While this is possible in the approach described here, it is not required; for example, in model aggregation, several complete working models may be integrated to form a larger whole. We therefore eschew the term "model template" in favor of *model definition*.

Figure 2 gives the definition of the extended **SBML** class that ties these different components together; it also provides the definition of **ExternalModelDefinition**. Readers familiar with the **SBML** class in SBML Level 3 Core will

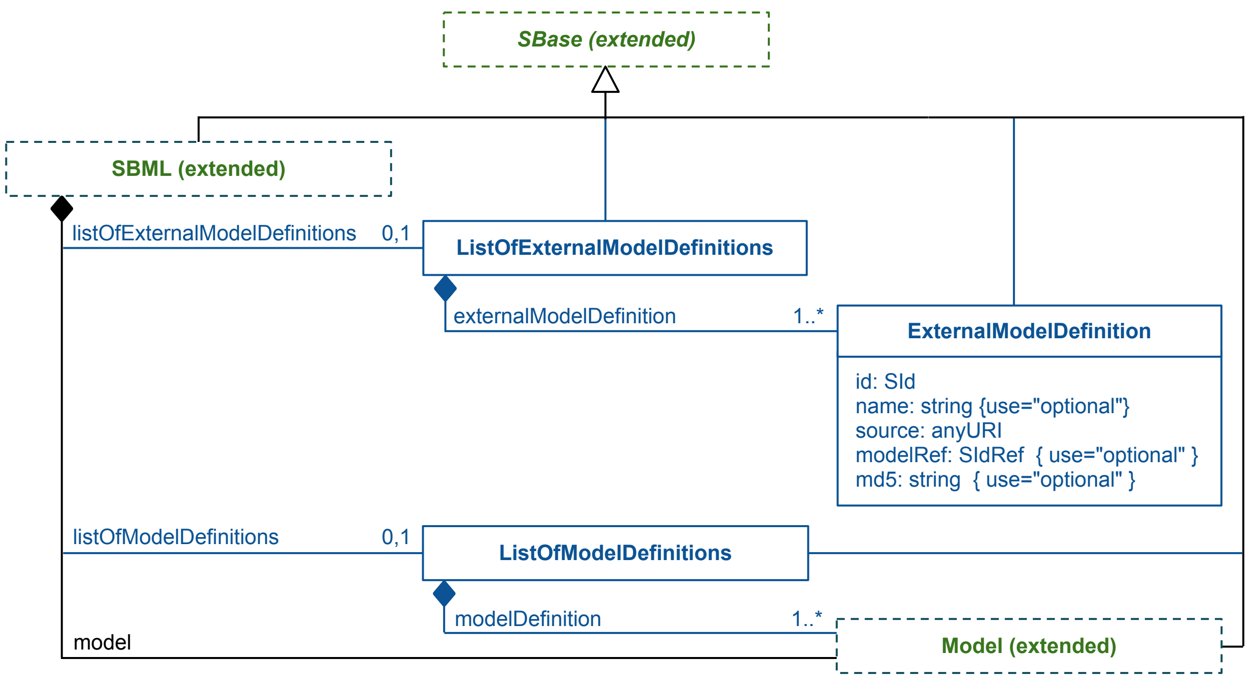

Figure 2: *The definitions of the extended* ***SBML*** *class as well as the new classes* ***ListOfModelDefinitions***, ***ListOfExternalModelDefinitions****, and* ***ExternalModelDefinition****. The extended* ***Model*** *class is defined in Section 3.4.*

notice that this package adds two new lists, `listOfModelDefinitions` and `listOfExternalModelDefinitions`, of class **ListOfModelDefinitions** and **ListOfExternalModelDefinitions**, respectively. The class diagram also makes concrete the notions described above, that model definition objects are not "owned" by any other model (they can be instantiated anywhere, even by models in other files) and that they exist outside the **Model** class entirely.

Figure 2 on the previous page also makes clear how model definitions *are* **Model** objects. At the same time, the scheme preserves the aspect of core SBML Level 3 in which a single **Model** object appears at the top level of an SBML document. As will become when we define **Model**, submodels appear inside **Model** objects. This is a key feature of the design described above; namely, when the top-level model references submodels, the submodels are instantiated, whereas when a model definition references submodels, the submodels are simply part of that model definition—they are not instantiated until the model definitions themselves are instantiated.

Finally, to give a more intuitive sense for how the pieces fit together, Figure 3 below shows a template structure of an SBML document with both a `listOfModelDefinitions` and `listOfExternalModelDefinitions`, as well as a list of **Submodel** objects inside the **Model** object.

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level3/version1/core" level="3" version="1"
      xmlns:comp="http://www.sbml.org/sbml/level3/version1/comp/version1" comp:required="true">
  <model id="My_Model">
    <comp:listOfSubmodels>
      one or more <comp:submodel> ... </comp:submodel> elements                    } optional
    </comp:listOfSubmodels>
  </model>
  <comp:listOfModelDefinitions>
    one or more <comp:modelDefinition> ... </comp:modelDefinition> elements          } optional
  </comp:listOfModelDefinitions>
  <comp:listOfExternalModelDefinitions>
    one or more <comp:modelExternalDefinition> ... </comp:modelExternalDefinition> elements  } optional
  </comp:listOfModelDefinitions>
</sbml>
```

Figure 3: *Example of a skeleton SBML Level 3 document containing the possible top-level constructs defined by the Hierarchical Model Composition package.*

3.3.1 The lists of internal and external model definitions

As shown in Figure 2 on the previous page, `listOfModelDefinitions` and `listOfExternalModelDefinitions` are children of an extended **SBML** object. Like other **ListOf___** classes in SBML, the **ListOfModelDefinitions** and **ListOfExternalModelDefinitions** are derived from ***SBase*** (more specifically, the extended ***SBase*** class defined in Section 3.6). They inherit ***SBase***'s attributes `metaid` and `sboTerm`, as well as the subcomponents for **Annotation** and **Notes**, but they do not add any especial attributes of their own.

If a model from an external SBML document is needed, it can be referenced with an **ExternalModelDefinition** object (Section 3.3.2). The **ListOfExternalModelDefinitions** container gathers all such references. It is derived from ***SBase*** but adds no especial attributes of its own. Like the other **ListOf___** classes, it inherits the attributes `metaid` and `sboTerm`, as well as the subcomponents for **Annotation** and **Notes**, that most SBML components have.

3.3.2 The ExternalModelDefinition class

To describe model definitions contained in external documents, the Hierarchical Model Composition package uses a separate object class (**ExternalModelDefinition**), defined in Figure 2 on the preceding page. In the sense being used here, **ExternalModelDefinition** objects are model definitions—in and of themselves, they are definitions of models but not *uses* of those models. The **ExternalModelDefinition** provides a way to declare and identify them so that the **Model** object in the present SBML document can use them in **Submodel** objects.

ExternalModelDefinition contains two required attributes (`source` and `id`) and three optional attributes (`modelRef`, `md5` and `name`). These attributes are explained below.

The `id` and `name` attributes

The `id` attribute serves to provide a handle for the external model reference so that **Submodel** objects can refer to it. (Crucially, it is *not* the identifier of the model being referenced; rather, it is an identifier for this **ExternalModelDefinition** object within the current SBML document.) The `id` attribute takes a required value of type `SId`.

ExternalModelDefinition also has an optional `name` attribute, of type `string`. The `id` attribute must be used as described in Section 3.9. The `name` attribute may be used in the same manner as other `name` attributes on SBML Level 3 Core objects; see Section 3.3.2 of the SBML Level 3 Version 1 Core specification for more information.

The `source` attribute

The required attribute `source` is used to locate the SBML document containing an external model definition. The value of this attribute must be of type `anyURI` (see Section 3.2.1). Since URIs may be either URLs, URNs, or relative or absolute file locations, this offers flexibility in referencing SBML documents. In all cases, the `source` attribute value must refer specifically to an SBML Level 3 Version 1 document; prior Levels/Versions of SBML are not supported by this package. The entire file at the given location is referenced. The `source` attribute must have a value for every **ExternalModelDefinition** instance.

Here are some examples of different `source` attribute values for the different cases. Suppose we have a model with the identifier “`glau`” located in a file named “`firefly.xml`”. The following fragment defines an external model definition and gives it the identifier “`m1`”, the latter being valid for use within the containing SBML document:

```
<comp:listOfExternalModelDefinitions>
    <comp:externalModelDefinition comp:source="firefly.xml" comp:modelRef="glau" comp:id="m1"/>
</comp:listOfExternalModelDefinitions>
```

(In the above, we assume the XML namespace prefix “`comp`” has been assigned to the correct XML namespace URI for the Hierarchical Model Composition package, but we do not show that part of the SBML document here. Section 3.1 explains this in more detail.) On the other hand, suppose that we wanted to reference the model defined as model “`BIOMD0000000002`” in BioModels Database. Looking inside the text of that SBML document, it becomes evident that the model identifier (its `id` value) is set to “`mod`”. Thus, using a URN to reference that model, we can write the following:

```
<comp:listOfExternalModelDefinitions>
    <comp:externalModelDefinition comp:id="m2" comp:modelRef="mod"
                                  comp:source="urn:miriam:biomodels.db:BIOMD0000000002"/>
</comp:listOfExternalModelDefinitions>
```

Finally, we can imagine the situation where a model is made accessible from a location accessible on the Internet, say via the HTTP protocol. The following is a (fake) example:

```
<comp:listOfExternalModelDefinitions>
    <comp:externalModelDefinition comp:id="m3" comp:modelRef="the_model_id"
                                  comp:source="http://www.cds.caltech.edu/~mhucka/sbmlmodel.xml"/>
</comp:listOfExternalModelDefinitions>
```

The `modelRef` attribute

ExternalModelDefinition’s optional attribute `modelRef`, of type `SIdRef`, in is used to identify a **Model** object within the SBML document located at `source`. The object referenced may be the main model in the document, or a model definition contained in the SBML document’s `listOfModelDefinitions` list.

In standard SBML, `id` on **Model** is an optional attribute, and therefore, it is possible that the **Model** object in a given SBML document does *not* have an identifier. In that case, there is no value to give to the `modelRef` attribute in **ExternalModelDefinition**. If `modelRef` does *not* have a value, then the main model (i.e., the `<model>` element within the `<sbml>` element) in the referenced file is interpreted as being the model referenced by this **ExternalModelDefinition** instance.

The md5 *attribute*

The optional md5 attribute takes a string value. If set, it must be an MD5 checksum value computed over the document referenced by source. This checksum can be used as a data integrity check over the contents of the source. Applications may use this to verify that the contents have not changed since the time that the **ExternalModelDefinition** reference was constructed. The procedure for using the md5 attribute is described in Table 2.

Case	Procedure
Creating and writing an SBML document	1. Compute the MD5 hash for the document located at source. 2. Store the hash value as the value of the md5 attribute.
Reading an SBML document	1. Read the value of the md5 attribute. 2. Read the document at the location indicated by the source attribute value. 3. Compute the MD5 hash for the document. 4. Compare the computed MD5 value to the value in the md5 attribute. If they are identical, assume the document has not changed since the time the **ExternalModelDefinition** object was defined; if the values are different, assume that the document indicated by source has changed.

Table 2: *Procedures for using the* md5 *attribute on* ***ExternalModelDefinition****.*

Software tools encountering a difference in the MD5 checksums should warn their users that a discrepancy exists, because a difference in the documents may imply a difference in the mathematical interpretation of the models.

Note that the MD5 approach is not without limitations. An MD5 hash is typically expressed as a 32-bit hexadecimal number. If a difference arises in the checksum values, there is no way to determine the cause of the difference without an component-by-component comparison of the models. (Even a difference in annotations, which cannot affect a models' mathematical interpretations, will result in a difference in the MD5 checksum values.) On the other hand, it is also not impossible that two different documents yield the *same* MD5 hash value, although it is extremely unlikely in practice. In any event, the MD5 approach is intended as an optional, simple and fast data integrity check, and not a final answer.

3.4 The extended Model class

The extension of SBML Level 3 Core's **Model** class is relatively straightforward: the Hierarchical Composition Package adds two lists, one for holding submodels (listOfSubmodels, of class **ListOfSubmodels**), and the other for holding ports (listOfPorts, of class **ListOfPorts**). Figure 4 on the following page provides the UML diagram. The rest of this section defines the extended **Model** class and the **Port** class; **Submodel** is defined in Section 3.5.

3.4.1 The list of submodels

The optional listOfSubmodels subcomponent in **Model** holds a **ListOfSubmodels** container object. If present, it must contain one or more **Submodel** objects (see Section 3.5).

3.4.2 The list of ports

The optional listOfPorts subcomponent in **Model** holds a **ListOfPorts** container object. If present, it must contain one or more **Port** objects (see Section 3.5). Ports are described below.

3.4.3 The Port class

In Hierarchical Model Composition, the *port* concept allows a modeler to design a submodel such that other models interact with the submodel through designated interfaces. The intention is that a modeler can indicate explicitly the intended points of interaction between a given model and other models that include or otherwise interact with it. The **Port** class is defined in Figure 4 on the next page. It is derived from **SBaseRef**, a class whose definition we leave to Section 3.7; for now, it is worth mentioning that **SBaseRef** provides attributes portRef, idRef, unitRef and metaIdRef, and a recursive subcomponent, sBaseRef.

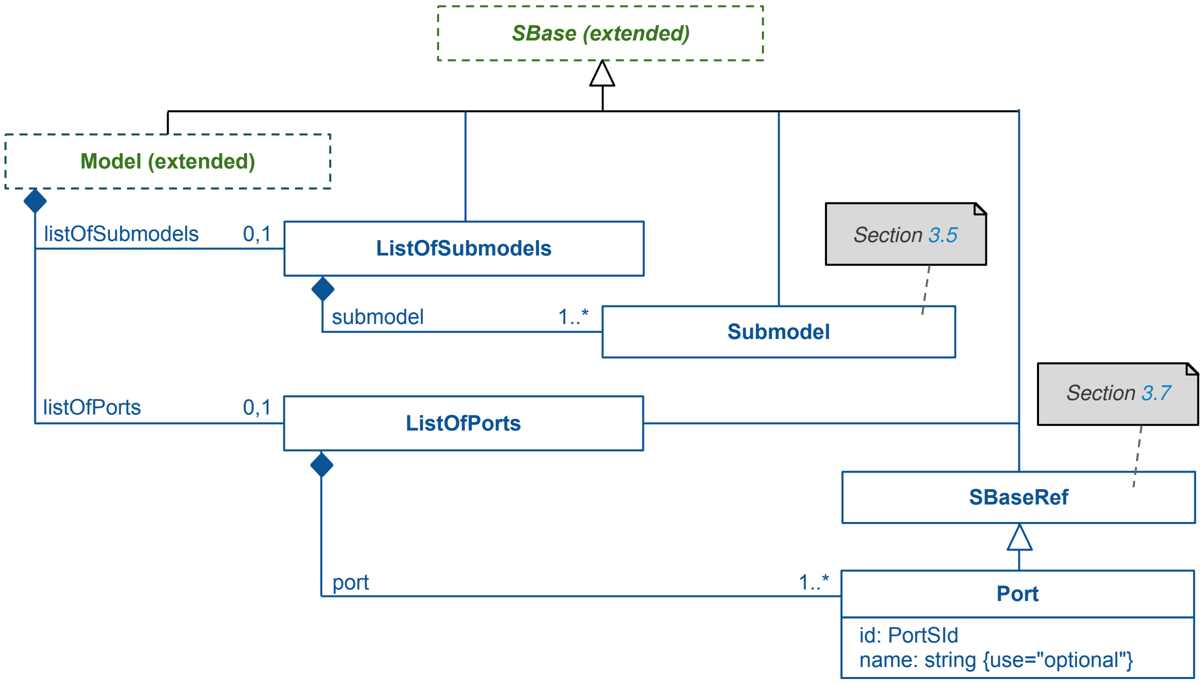

***Figure 4:** The extensions of the **Model** class and the definitions of the classes **Port**, **ListOfPorts**, and **ListOfSubmodels**. **Submodel** is defined in Section 3.5 and **SBaseRef** is defined in Section 3.7. In other respects, **Model** remains defined as in the SBML Level 3 Core specification.*

We say that a **Port** object instance *defines a port* for a component in a model. As will become clear in Section 3.7, the facilities of the **SBaseRef** parent class from which **Port** is derived are what provides the means for the component to be identified. For example, a port could be created by using the `metaIdRef` attribute to identify the object for which a given **Port** instance is the port; then the question "what does this port correspond to?" would be answered by the value of the `metaIdRef` attribute.

In the present formulation of the Hierarchical Model Composition package, the use of ports is not *enforced*, nor is there any mechanism to restrict which ports may be used in what ways—they are only an advisory construct. Future versions of this SBML package may provide additional functionality to support explicit restrictions on port use. For the present definition of Hierarchical Model Composition, users of models containing ports are encouraged to respect the modeler's intention in defining ports, and use the port definitions to interact with components through their ports (when they have ports defined) rather than interact directly with the components.

The `id` *attribute*

The required attribute `id` is used to give an identifier to a **Port** object so that other objects can refer to it. The attribute has type `PortSId` and is essentially identical to the SBML primitive type `SId`, except that its namespace is limited to the identifiers of **Port** objects defined within a **Model** object. In parallel, the `PortSId` type has a companion type, `PortSIdRef`, that corresponds to the SBML primitive type `SIdRef`; the value space of `PortSIdRef` is limited to `PortSId` values. (See also Figure 7 on page 22.)

Note the implication of the separate namespaces of port identifiers (values of type `PortSId`) and other identifiers (values of `SId` or `UnitSId`): since `PortSId` values are in their own namespace within the parent **Model**, it is possible for a `PortSId` value to be the same as some `SId` value in the model, without causing an identifier collision.

The `name` *attribute*

The optional `name` attribute is provided on **Port** so that port definitions may be given human-readable names. Such names may be useful in situations when port names need to be displayed to modelers.

Additional restrictions on Port objects

Several additional restrictions exist on the use of ports. It will immediately become apparent that these restrictions are common-sense rules, but they are worth making explicit:

1. The model to which a **Port** object refers with its **SBaseRef** constructs must be the parent **Model** object containing the **Port** object itself.
2. Each port in a model must refer to a unique component of that model; that is, no two ports in a model may both refer to the same model component.
3. A port cannot refer to another port of the same model.
4. A port cannot refer to itself.

3.5 The Submodel class

In the SBML Hierarchical Model Composition package, submodels are instantiations of models contained within other models. Submodel instances are expressed using objects of class **Submodel**, defined in Figure 5. Objects of this class represent submodels contained within **Model** objects, as depicted in Figure 4 on the preceding page.

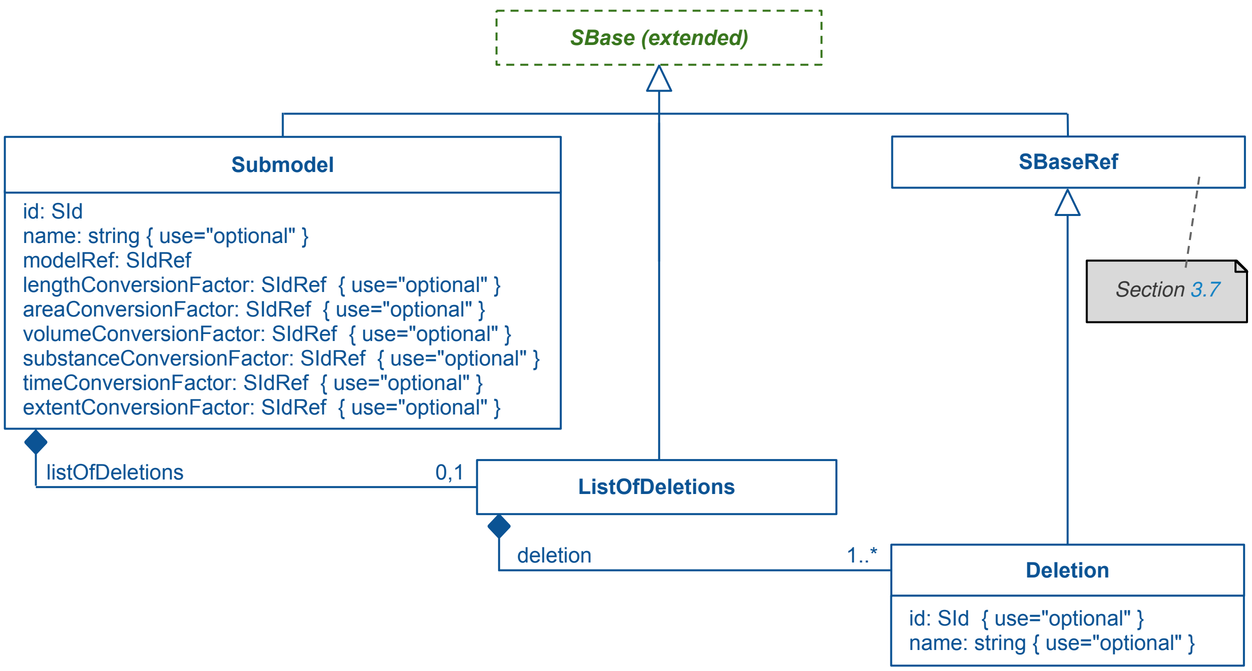

Figure 5: *The definition of the **Submodel**, **Deletion** and **ListOfDeletions** classes.*

A **Submodel** object must say which **Model** object it instantiates, and may additionally define how the **Model** object is to be modified before it is instantiated in the enclosing model. With respect to the latter capability, this SBML package provides two possible types of direct modifications: conversion factors, and deletions. We describe these two mechanisms in more detail in the subsections below, but the following informal summary may serve as a useful guide to get a general sense for how it things work:

- If numerical values in the referenced model must be changed in order to fit them into their new context as part of the submodel, the changes can be handled through conversion factors.
- If one or more structural features in the referenced model are undesirable and should be removed, the changes can be handled through deletions. (For example, an initial assignment or reaction not be relevant in its new context, and may need to be removed.)

3.5.1 The attributes of Submodel

Figure 5 on the previous page shows that **Submodel** has numerous attributes, as well as a single subcomponent, `listOfDeletions`. We describe the attributes below, then turn to `listOfDeletions` in Sections 3.5.2–3.5.3.

The `id` attribute

The `id` is a required attribute of type `SId` that gives an identifier to the **Submodel** instance. It is required so that other references may always have a means through which a parent model may refer to this submodel instance's elements (e.g., to link and replace them). The identifier has no mathematical meaning.

This identifier must follow the normal restrictions on SBML `SId` values for uniqueness within **Model** objects. In addition, the `id` value may not be referenced by SBML Level 3 Core components in a model. For example, the identifier may not appear as the `<ci>` value in a mathematical formula of a component defined by SBML Level 3 Core (e.g., rules, initial assignments, etc.). This restriction is necessary so that if a software package does not have support for the Hierarchical Model Composition package, it can ignore the package constructs and still end up with a syntactically valid (though perhaps diminished) SBML document.

The `name` attribute

The optional `name` attribute is provided on **Submodel** so that submodel definitions may be given human-readable names. Such names may be useful in situations when submodel names are displayed to modelers.

The `modelRef` attribute

The whole purpose of a **Submodel** object is to instantiate a model definition, which is to say, either a **Model** object defined in the same enclosing SBML document, or a model defined in an external SBML document. The `modelRef` attribute is the means by which that model is identified. This required attribute of type `SIdRef`, must refer to the identifier of a **Model** or **ExternalModelDefinition** object within the enclosing SBML document (i.e., in the model namespace of the document).

It is perfectly legitimate for the model referenced by `modelRef` to have its own submodels. The composite model defined in that case is simply the composed model that results from following the change of inclusion. However, there is one restriction: loops are not allowed. In other words, the referenced model may not refer to its parent model, nor may it refer to a model which in turn instantiates its parent model, etc.

It is also legal for the model referenced by `modelRef` to refer to the `<model>` child of the enclosing SBML document, i.e., the main **Model** object in the **SBML** object where it is itself located. This would mean that the document contains a model definition that itself contains (and perhaps modifies) the model it presents to the world as the main or top-level model in the document. A possible use for this might be to define a common scenario in the main model, then create alternate scenarios with different initial conditions or parameter sets using the list of model definitions (Figure 2 on page 11) in the **SBML** object. Because the model namespace is defined per document, this means that it is possible to define and include a new model namespace by creating a new document, then importing one or more of those models using the **ExternalModelDefinition** class. (Section 3.9 discusses identifier scoping in more detail.)

The conversion factor attributes

Conversion factors enable the matching up of mathematical values and units of measurement between submodels and models. There are six possible conversion factors, corresponding to the six conversion factors defined by SBML Level 3 Core's basic **Model** object class. The six attributes representing these conversion factors on **Submodel** are `lengthConversionFactor`, `areaConversionFactor`, `volumeConversionFactor`, `substanceConversionFactor`, `timeConversionFactor`, and `extentConversionFactor`.

All of these optional attributes have type `SIdRef`. If set, the value must be the identifier of a **Parameter** object in the parent **Model** object. The parameter will be used to convert the value of the submodel quantities of the indicated type (e.g., volume, time, etc.) to the units used in the parent model. The procedures are involved, and a separate section (Section 3.8) is devoted to explaining them.

3.5.2 The list of deletions

The `listOfDeletions` subcomponent on **Submodel** holds an optional **ListOfDeletions** container which, if present, must contain one or more **Deletion** objects. This list of deletions specifies objects to be removed from the submodel when composing the overall model. (This "removal" of course does not involve physically editing the files; rather, it is mathematical and conceptual.)

3.5.3 The Deletion class

The **Deletion** object class is used to define a *deletion* operation to be applied when a submodel instantiates a model definition. Deletions may be useful in hierarchical model composition scenarios for various reasons. For example, some components in a submodel may be redundant in the composed model, perhaps because the same features are implemented in a different way in the new model.

Deletions function as follows. When the **Model** to which the **Submodel** object refers (via the `modelRef` attribute) is read and processed for inclusion into the composed model, each **Deletion** object identifies an object to "remove" from that **Model** instance. The resulting submodel instance consists of everything in the **Model** object instance *minus* the entities referenced by the list of **Deletion** objects. We discuss the implications of deletions further below.

The definition of the **Deletion** object class is shown in Figure 5 on page 16. It is subclassed from **SBaseRef**, described in detail in Section 3.7, and reuses all the machinery provided by **SBaseRef**. In addition, it defines two optional attributes, `id` and `name`.

The `id` and `name` attributes

The optional attribute `id` on **Deletion** can be used to give an identifier to a given deletion operation. The identifier has no mathematical meaning, but it may be useful for creating submodels that can be manipulated more directly by other submodels. (Indeed, it is legitimate for an enclosing model definition to delete a deletion!)

The optional `name` attribute is provided on **Deletion** for the same reason it is provided on other elements that have identifiers; viz., to provide for the possibility of giving a human-readable name to the object. The name may be useful in situations when deletions are displayed to modelers.

Implications of a deletion

As might be expected, deletions can have wide-ranging implications, especially when the object deleted has substantial substructure, as in the case of reactions. The following are rules regarding deletions and their effects.

1. An object that has been "deleted" is considered inaccessible. Any element that has been deleted (or replaced, as discussed in Section 3.6) may not be referenced by an **SBaseRef** object.

2. If the deleted object has child objects and other structures, the child objects and substructure are also considered to be deleted.

3. It is not an error to delete explicitly an object that is already deleted by implication (for example as a result of point number 2 above). The resulting model is the same.

We leave additional comments about best practices surrounding deletions to Section 5.2.

3.6 Replacements

The model definition, submodel and port constructs defined so far allow a modeler to identify and aggregate models, as well as (possibly) delete features from model definitions before the definitions are used create a final, composed model. In this section, we turn to two final capabilities needed for effective model composition: linking and substituting components. Both are implemented as *replacements*, which are the glue used to connect submodels together with each other and with a containing model.

Replacements are implemented by extending the SBML Level 3 Core ***SBase*** class as shown in Figure 6. This extension provides the means to define replacements in two directions: either the current object replaces one or more others, or the current object is itself replaced by another. These two possibilities are captured by the `listOfReplacedElements` and `replacedBy` subcomponents shown in Figure 6.

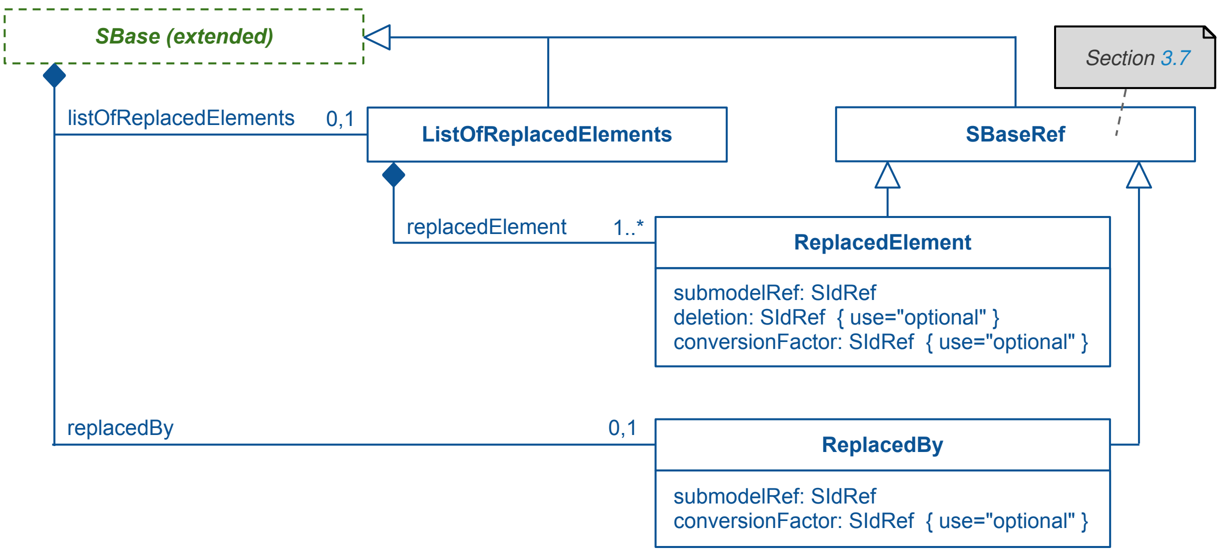

Figure 6: *The extension of **SBase** and the definition of the **ListOfReplacedElements** and **ReplacedElement** classes. The **SBaseRef** class is defined in Section 3.7.*

Since ***SBase*** in SBML is the abstract base class of all other SBML object classes, the extension of ***SBase*** means all SBML objects gain the ability to define how they replace (or are replaced by) components in submodels. Replacements are a general mechanism that serve multiple purposes. At their most basic, they allow a model builder to make a statement of the form "entity W in this model replaces entity X in submodel Y". In the final composed model, all references to X in Y are replaced with references to W. The same approach is used as the mechanism for linking or gluing entities from different models together. Thus, to connect entity X and some other entity Z located in another submodel, a model could make W define multiple replacements simultaneously: one for X and another for Z. Entity W will then act as an intermediary at the level of the containing model.

3.6.1 The list of replaced elements

Figure 6 shows that the extension of ***SBase*** by the Hierarchical Model Composition package adds an optional `listOfReplacedElements` subcomponent for holding a **ListOfReplacedElements** container object. If present, it must contain at least one **ReplacedElement** object (Section 3.6.2).

3.6.2 The ReplacedElement class

A **ReplacedElement** object is essentially a pointer to a submodel object that is should be considered "replaced". The object *holding* the **ReplacedElement** instance is the one *doing the replacing*; the object pointed to by the **ReplacedElement** object is the object being replaced. A replacement implies that dependencies involving the replaced object must be updated: all references to the replaced object elsewhere in the model are taken to refer to the replacement object. For example, if one species replaces another, then any reference to the original species in mathematical formulas, or lists of reactants or products or modifiers in reactions, or initial assignments, or any other SBML construct, are taken to refer to the replacement species instead (with its value possibly modified by either this object's `conversionFactor` attribute or the relevant submodel's conversion factors—see Section 3.8). Moreover, any annotations that refer to the replaced species' `metaid` value must be made to refer to the replacement species' `metaid` value instead; and anything else that referred either to the object identifier (i.e., the `id` attribute) or meta identifier (i.e., the `metaid` attribute) must be made to refer to the replacement species object instead.

ReplacedElement inherits from **SBaseRef** and adds three attributes: one required (`submodelRef`) and two optional (`deletion` and `conversionFactor`), as described below.

Attributes inherited from SBaseRef

The **ReplacedElement** class, being derived from **SBaseRef**, inherits all of that class' attributes and its subelement. This means that **ReplacedElement** has the `portRef`, `idRef`, `unitRef` and `metaIdRef` attributes, as well as the subcomponent `sBaseRef` and the recursive structure that it implies.

It is the properties of **SBaseRef** that allow a **ReplacedElement** object to refer to what is being replaced. For example, if the object being replaced has a port identifying it, the instance of **ReplacedElement** would have its `portRef` attribute value set to the identifier of the **Port** pointing to the object being replaced. If there is no corresponding port for the object being replaced, but the object has a regular identifier (typically an attribute named `id`), then the **ReplacedElement** object would set `idRef` instead. If there is no identifier, the replacement would use the next available option defined by **SBaseRef**, and so on. Section 3.7 describes the alternatives in more detail.

The `submodelRef` attribute

The required attribute `submodelRef` takes a value of type `SIdRef`. This value must be the identifier of a **Submodel** object in the containing model. The **Model** object referenced by the **Submodel** object establishes the object namespaces for the `portRef`, `idRef`, `unitRef` and `metaIdRef` attributes: only objects within the **Model** object may be referenced by those attributes.

The `deletion` attribute

The optional attribute `deletion` takes a value of type `SIdRef`. The value must be the identifier of a **Deletion** object in the parent **Model** of the **ReplacedElement** (i.e., the value of some **Deletion** object's `id` attribute). When `deletion` is set, it means the **ReplacedElement** object is actually an annotation to indicate that the replacement object replaces something *deleted* from a submodel. The use of the `deletion` attribute overrides the use of the attributes inherited from **SBaseRef**: instead of using, e.g., `portRef` or `idRef`, the **ReplacedElement** instance sets `deletion` to the identifier of the **Deletion** object.

The use of **ReplacedElement** objects to refer to deletions has no effect on the composition of models or the mathematical properties of the result. It serves instead to help record the decision-making process that lead to a given model. It can be particularly useful for visualization purposes, as well as to serve as scaffolding where other types of annotations can be added using the normal **Annotation** subcomponents available on all ***SBase*** objects in SBML.

The `conversionFactor` attribute

The **ReplacedElement** class' `conversionFactor` attribute may be used to define how to transform or rescale the replaced object's value so that it is appropriate for the new contexts in which it appears. The value of this attribute must be of type `SIdRef` and refer to a **Parameter** object instance defined in the model. The conversion factor identified by the `conversionFactor` attribute overrides any automatic conversion that may have been performed based on the submodel's relevant conversion factors. The details of this are left to Section 3.8.

3.6.3 The `replacedBy` subcomponent

As mentioned above, the extension of ***SBase*** defined in Figure 6 on the previous page introduces a new optional subcomponent, `replacedBy`. Its value, if present on a given ***SBase***-derived object, must be a single object of the **ReplacedBy** class (described in Section 3.6.4 below).

3.6.4 The ReplacedBy class

Whereas a **ReplacedElement** object indicates that the containing object replaces another, a **ReplacedBy** object indicates the converse: the present object is to replaced by another object. As is the case with **ReplacedElement**, the **ReplacedBy** class inherits from **SBaseRef**. It adds one required attribute (`submodelRef`) and one optional attribute (`conversionFactor`).

Attributes inherited from SBaseRef

The **ReplacedBy** class, being derived from **SBaseRef**, inherits the latter class' attributes `portRef`, `idRef`, `unitRef` and `metaIdRef`, as well as the subcomponent `sBaseRef` and the recursive structure that it implies.

The `submodelRef` attribute

The required attribute `submodelRef` takes a value of type `SIdRef`, which must be the identifier of a **Submodel** object in the containing model. This attribute is analogous to the corresponding attribute on **ReplacedElement**; that is, the model referenced by the **Submodel** object establishes the object namespaces for the `portRef`, `idRef`, `unitRef` and `metaIdRef` attributes: only objects within the **Model** object may be referenced by those attributes.

The `conversionFactor` attribute

The **ReplacedBy** class' `conversionFactor` attribute may be used to define how to transform or rescale this object's value so that it is appropriate for the new contexts in which it appears. The value of this attribute must be of type `SIdRef` and refer to a **Parameter** object instance defined in the model. The conversion factor identified by the `conversionFactor` attribute overrides any automatic conversion that may have been performed based on the submodel's relevant conversion factors. The details of this are left to Section 3.8.

3.6.5 Additional requirements and implications

Replacements are a powerful mechanism and carry significant implications. We focus on crucial ones below:

- *Identifier references must be updated.* Both **ReplacedElement** and **ReplacedBy** introduce the need to change identifier references that refer to the object being replaced. This identifier-redirection process has some important implications. First, when anything refers to the replaced object's `id` and/or `metaid` value, the replacement object must itself define its own `id` and/or `metaid` value, or else the step of adjusting references will be impossible to perform because there will not be new `id` or `metaid` values to use in place of the old ones. Second, if other SBML Level 3 packages attach identifiers in their own namespaces to an object being replaced, those identifiers must also be likewise redirected. (And again, this implies that the SBML document must put suitable identifier attributes from those package namespaces on the *replacement* object, so that the replacement object's identifiers can be substituted for those of the object being replaced.)

- *Subcomponents of a replaced object become inaccessible.* An important and far-reaching consequence of replacements is that if the object being replaced contains other objects, then *those other objects are considered deleted.* For example, replacing a **Reaction** or an **Event** object means all of the substructure of those entities in a model are deleted, and references to the identifiers of those deleted entities are made invalid. (This has implications for such things as **SpeciesReference** identifiers that may be used in a model outside of the **Reaction** objects where they are defined.)

- *Replacements must be unique.* Unless the `deletion` attribute on the **ReplacedElement** object is "`true`", a given object being replaced may only appear in exactly one **ReplacedElement** object anywhere in a model; otherwise, it would imply multiple entities replace the same object, and this would lead to ambiguities (e.g., in old references to the entities being replaced). A **ReplacedElement** object with `deletion`="`true`" is the only type of entity that may be listed in more than one **ListOfReplacedElements**.

This scheme does not provide for direct "horizontal replacements" involving only subelements. An example might be when one species in one submodel is the conceptual replacement for a second species in a second submodel. The lack of a direct mechanism for horizontal replacements is not a true limitation; the same effect can be achieved by creating an intermediate object in the containing model and linking it to the other objects via replacements.

Finally, there is no requirement that replaced objects must be of the same type as the replacing object. The only restriction is that all old references to the replaced object must now point to the replacing object. Of course, the normal rules of SBML validity apply, so the resulting model must still make sense and be valid SBML. There is no requirement for like-kind replacements, however, because errors do not *necessarily* result. For example, replacing a **Species** object that appeared in a **Reaction** object would lead to an invalid SBML document if the replacement was a **Parameter** object, but replacing a **Species** object by a **Parameter** could be valid if that **Species** never appeared in any **Reaction** object or if all the **Reaction** objects were deleted.

3.7 The SBaseRef class

Defining ports, deletions, and replacements requires a way to refer to specific components within enclosed models or even external models located in other files. The machinery for constructing such references is embodied in the **SBaseRef** class. This class is the parent class of the **Port**, **Deletion**, **ReplacedElement** and **ReplacedBy** classes described in previous sections.

Figure 7 shows the definition of **SBaseRef**. This class includes several attributes used to implement alternative approaches to referencing a particular component, and it also has a recursive structure, providing the ability to refer to elements buried within (say) a sub-submodel configuration.

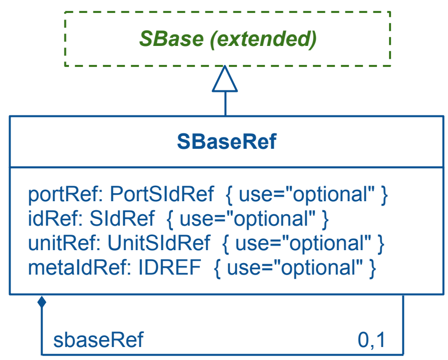

Figure 7: *The extensions of the* ***SBaseRef*** *class. The four attributes* `portRef`, `idRef`, `unitRef` *and* `metaIdRef` *are mutually exclusive; only one can have a value in a given object instance. The recursive structure also allows referencing entities in submodels of submodels, to arbitrary depths, as described in the text.*

3.7.1 The attributes of SBaseRef

The four different attributes on **SBaseRef** are mutually exclusive: only one of the attributes can have a value at any given time, and exactly one must have a value in a given **SBaseRef** object instance. (Note that this is true of the basic **SBaseRef** class; however, derived classes such as **ReplacedElement** may add additional attributes and extend or override the basic attributes and mechanisms.)

All of the attributes face the following common namespace issue. As discussed in more detail in Section 3.9, attributes of type `SId`, `UnitSId`, and `PortSId` need only be unique on a per-**Model** basis. This means that an interpreter of the SBML document must be able to determine the model to which `idRef`, `unitRef`, and `portRef` attributes refer. In addition, even though `ID` values are unique on per-document level, the same SBML element may be instantiated in multiple submodels, in any number of **Model** objects, and therefore the `metaIdRef` attribute must also know to which **Model** instantiation it is referring. Just exactly how this is done is something that is left up to the classes that subclass **SBaseRef**. The sections that describe **Port**, **Deletion**, **ReplacedElement** and **ReplacedBy** describe the approach taken in each case.

Separately, readers may wonder why so many different alternatives are necessary. The reason is that in a given scenario, the referenced model may be located in an external file beyond the direct control of the modeler, and so the preferred methods of referencing the subobjects may not be available. **SBaseRef** provides multiple alternatives so that a variety of modeling scenarios can be supported.

The `portRef` *attribute*

The optional attribute `portRef` takes a value of type `PortSIdRef`. As its name implies, this attribute is used to refer to a port identifier, in the case when the reference being constructed with the **SBaseRef** is intended to refer to a port on a submodel. The namespace of the `PortSIdRef` value is the set of identifiers of type `PortSId` defined in the submodel, not the parent model.

The `idRef` *attribute*

The optional attribute `idRef` takes a value of type `SIdRef`. As its name implies, this attribute is used to refer to a regular identifier (i.e., the value of an `id` attribute on some other object), in the case when the reference being constructed with the **SBaseRef** is intended to refer to an object that does not have a port identifier. The namespace of the `SIdRef` value is the set of identifiers of type `SId` defined in the submodel, not the parent model.

The `unitRef` *attribute*

The optional attribute `unitRef` takes a value of type `UnitSIdRef`. This attribute is used to refer to the identifier of a **UnitDefinition** object. The namespace of the `UnitSIdRef` value is the set of unit identifiers defined in the submodel, not the parent model.

Note that even though this attribute is of type `UnitSIdRef`, the reserved unit identifiers that are defined by SBML Level 3 (see Section 3.1.10 of the SBML Level 3 Version 1 core specification) are *not* permitted as values of `unitRef`. Reserved unit identifiers may not be replaced or deleted.

The `metaIdRef` *attribute*

The optional attribute `metaIdRef` takes a value of type XML `IDREF`. This attribute is used to refer to a `metaid` attribute value on some other object, in the case when the reference being constructed with the **SBaseRef** is intended to refer to an object that does not have a port identifier. Since meta identifiers are optional attributes of ***SBase***, all SBML objects have the potential to have a meta identifier value.

3.7.2 Recursive SBaseRef structures

An **SBaseRef** object may have up to one subcomponent named `sBaseRef`, of type **SBaseRef** (see Figure 7 on the previous page). This arrangement permits recursive structures to be constructed so that objects inside submodels can be referenced.

The form of such recursive references must be as follows. The highest-level **SBaseRef** object of such a chain (which will necessarily be an object of class **Port**, **Deletion**, **ReplacedElement** or **ReplacedBy**, because they are the only classes derived from the class **SBaseRef**) must refer to a **Submodel** object in the containing model. All child **SBaseRef** objects in the chain must refer to components inside the **Model** instance to which the **Submodel** refers.

The following example may help clarify this. Suppose that we want to delete an object with the identifier "`p1`" inside the **Submodel** object identified as "`m1`". Figure 5 on page 16 shows that **Deletion** objects are placed inside a **ListOfDeletions**. The following XML fragment illustrates how the constructs will look:

```
<listOfDeletions>
  <deletion idRef="m1">
    <sBaseRef idRef="p1" />
  </deletion>
</listOfDeletions>
```

Suppose instead that the submodel "`m1`" from the previous example is actually nested inside another submodel "`m2`". Then, we would have the following:

```
<listOfModelDefinitions>
  <modelDefinition id="m1">
    <listOfParameters>
      <parameter id="p1" value="3"/>
```

```
      </listOfParameters>
    </modelDefinition>
    <modelDefinition id="m2">
      <listOfSubmodels>
        <submodel id="sub_m1" modelRef="m1"/>
      </listOfSubmodels>
    </modelDefinition>
    <modelDefinition id="m3">
      <listOfSubmodels>
        <submodel id="sub_m2" modelRef="m2">
          <listOfDeletions>
            <deletion idRef="sub_m1">
              <sBaseRef idRef="p1" />
            </deletion>
          </listOfDeletions>
        </submodel>
      </listOfSubmodels>
</listOfModelDefinitions>
```

Finally, what if we needed to go further down into nested submodels? The following example illustrates this:

```
<listOfModelDefinitions>
  <modelDefinition id="m1">
    <listOfParameters>
      <parameter id="p1" value="3"/>
    </listOfParameters>
  </modelDefinition>
  <modelDefinition id="m2">
    <listOfSubmodels>
      <submodel id="sub_m1" modelRef="m1"/>
    </listOfSubmodels>
  </modelDefinition>
  <modelDefinition id="m3">
    <listOfSubmodels>
      <submodel id="sub_m2" modelRef="m2"/>
    </listOfSubmodels>
  </modelDefinition>
  <modelDefinition id="m3">
    <listOfSubmodels>
      <submodel id="sub_m2" modelRef="m3">
        <listOfDeletions>
          <deletion idRef="sub_m1">
            <sBaseRef idRef="sub_m2">
              <sBaseRef idRef="p1" />
            </sBaseRef>
          </deletion>
        </listOfDeletions>
      </submodel>
    </listOfSubmodels>
</listOfModelDefinitions>
```

Although this use of nested **SBaseRef** objects allows a model to refer to components buried inside submodels, it is considered inelegant model design. It is better to promote any element in a submodel to a local element if it can be predicted that containing models may need to reference it. However, if the submodel is fixed (e.g., if is in an external file), then no other option may be available except to use nested references.

3.8 Conversion factors

[MH] I have not looked at this section.

In SBML Level 3 Version 1 Core, units of measurement are optional information. Modelers are required to write their models in such a way that all conversions between units are explicitly incorporated into the quantities, so that nowhere do units need to be understood and values implicitly converted before use. Given the Hierarchical Model Composition package's design goal of compatibility with existing models and files that may not be changeable, it is not an option to require that all included models must be written in such a way that they are numerically compatible with each other. For example, if one submodel defines how a species amount changes in time, and a second submodel defines an initial assignment for that same species in terms of concentration, something must be done to make the model as a whole coherent without editing the submodels directly. That is the purpose of the conversion factor attributes on the **Submodel** and **ReplacedElement** classes.

There are many situations to account for, so unfortunately, the topic of conversion factors is rather involved. We begin with the relatively straightforward case of **ReplacedElement**.

3.8.1 Conversion factors involving ReplacedElement

As explained in Section 3.6.2, the various conversion factor attributes on **ReplacedElement** override any conversion factors defined on the **Submodel** object that the **ReplacedElement** references via its `modelRef` attribute.

If the submodels of the merged model retain any mathematical formulas (that is, if there are any reactions, assignments, or any other construct with a MathML `<math>` element that has not been replaced or deleted from the submodel), then those formulas may be subject to different scales and units than the mathematical formulas of the containing model. In that case, the entities and formulas should be converted to the new units. If a replaced element has a defined conversion factor, then any time a calculation is performed within the math of the **Submodel** object where the replaced element's identifier is found on the left-hand side of an equation, the right-hand side is multiplied by that conversion factor before assignment to that variable. For example, if a species has an initial assignment of $4x+3$, and has a conversion factor of c, the initial assignment formula become $c(4x+3)$. The same is true for assignment rules, rate rules, kinetic laws, event assignments, and the implied rates of change of species as calculated from kinetic laws, as described in section 4.11.7 of the SBML Level 3 Version 1 Core specification.

Conversely, wherever the identifier of a replaced element appears on the right-hand side of an equation in its original submodel, its appearance in that equation should be *divided* by the conversion factor. In our previous example of an initial assignment of $4x+3$, if the x had been replaced and given a conversion factor of c_x, that initial assignment formula would become $4(x/c_x)+3$. This holds true for any mathematical equation in the model, including algebraic rules. This also means that if a value appears on the right and left-hand sides of an equation, you must apply the conversion factor twice: if the rate rule of x is $4x+3$, it becomes $c_x(4(x/c_x)+3)$. (This simplifies to $4x+3c_x$, as you would expect—the x is already in the correct scale; it is only the 3 that must be converted.)

3.8.2 Conversion factors involving Submodel

The six attributes on **Submodel** with the names of the form ___`ConversionFactor` dictate how any submodel mathematics whose unit types are defined by the Level 3 Core specification are to be converted whether or not that element was replaced, in the absence of an explicit conversion factor for that element. To understand the rules, it is first helpful to have a summary of the conversion factors implied by the SBML Level 3 Version 1 Core specification. We provide a summary in Table 3 on the next page.

The procedures for using the conversion factor attributes on **Submodel** are based on the conversion factors defined by the Core. Thus, all compartments that set `spatialDimension`="1" in the submodel must be converted according to the `lengthConversionFactor`, with all assignments to that compartment multiplied by the conversion factor, and that compartment's identifier divided by it wherever it appears inside a math element. All rates of change of species amounts (defined in section 4.11.7 of the Level 3 Version 1 Core specification) are converted by the `substanceConversionFactor` divided by the `timeConversionFactor`, after being converted (if necessary) by any internal conversion factors, as described. All species concentrations from compartments of dimension 2 are converted by the `substanceConversionFactor` divided by the `areaConversionFactor`. Non-replaced elements with defined unit types are still converted, so that the output of any simulation will be on the same scale as elements from the containing model.

In the core specification for SBML Level 3, if the conversion factor attributes for **Model** and **Species** are undefined, the rate of change of species amounts over time is defined to be equal to the rate of extent of the reaction over time, arguably creating a default conversion of extent to amount of 1. Similarly, all conversion factors here effectively default to '1' as well, so that if (for example) 'substanceConversionFactor' is defined but 'areaConversionFactor' is not, species concentrations from compartments of dimension 2 are still converted according to the substance-

Component	Attribute value	Automatic conversion factor
AlgebraicRule	*(All)*	1
AssignmentRule	*(All)*	*Conversion factor for referenced object*
Compartment	`spatialDimensions`=“1”	`lengthConversionFactor`
Compartment	`spatialDimensions`=“2”	`areaConversionFactor`
Compartment	`spatialDimensions`=“3”	`volumeConversionFactor`
Compartment	`spatialDimensions` not equal to “1”, “2”, or “3”	1
Constraint	*(All)*	*(None needed)*
Delay	*(All)*	`timeConversionFactor`
EventAssignment	*(All)*	*Conversion factor for referenced object*
FunctionDefinition	*(All)*	1
InitialAssignment	*(All)*	*Conversion factor for referenced object*
KineticLaw	*(All)*	$\frac{\texttt{extentConversionFactor}}{\texttt{timeConversionFactor}}$
Implied rate of change of a species	*(All)*	$\frac{\texttt{substanceConversionFactor}}{\texttt{timeConversionFactor}}$
Parameter	*(All)*	1
Priority	*(All)*	1
RateRule	*(All)*	$\frac{\textit{Conversion factor for referenced object}}{\texttt{timeConversionFactor}}$
Species	`hasOnlySubstanceUnits`=“true”	`substanceConversionFactor`
Species	`hasOnlySubstanceUnits`=“false”	$\frac{\texttt{substanceConversionFactor}}{\textit{Conversion factor for referenced object}}$
Species	`hasOnlySubstanceUnits`=“true” replaced by a **Species** having `hasOnlySubstanceUnits`=“false”	$\frac{\texttt{substanceConversionFactor}}{\textit{Compartment size}}$
Species	`hasOnlySubstanceUnits`=“false” replaced by a **Species** having `hasOnlySubstanceUnits`=“true”	$\frac{\texttt{substanceConversionFactor} \cdot \textit{(Compart. size)}}{\textit{Conversion factor for compartment}}$
SpeciesReference	*(All)*	1
Trigger	*(All)*	*(None needed)*
(Unknown)	*(All)*	1

Table 3: *Conversion factors used for the different components defined by SBML Level 3 Core.*

ConversionFactor, ’divided by 1’.

Critically, if an element’s unit type cannot be determined, it has no default conversion factor, and one must be set explicitly for the element in question. All **Parameter** objects fall into this category, as parameters may have any unit at all, and have no defined unit type as a class. Similarly, compartments with no `spatialDimension` set, or set to a fractal `spatialDimension` such as 2.6 should not be converted automatically. This means that if a parameter is internal to a submodel and not replaced, there is no way to convert it, and it will remain in its original scale. This will not affect the math of the converted elements, as the rules above first convert all math to the original scale, and only convert it to the new scale when assigning it to a variable. However, if it is displayed as output, these values may be in a different scale from other displayed output. The only way to correct this situation is to replace the parameter in question, and give it an explicit conversion factor.

Some math may use a combination of conversion factors defined on the **Submodel** object with the conversion factors defined explicitly on an element’s replacement construct. The simplest example is that of a rate rule that defines how a parameter changes with time. If the **Parameter** object has been replaced and given a conversion factor, the parameter’s explicit conversion factor is divided by the submodel’s `timeConversionFactor` to act as the overall conversion factor for the rate rule’s formula. As a slightly more complicated example, a species concentration that has no explicit conversion factor set for its replacement, and which is contained in a compartment that does have an explicit conversion factor, will be converted according to the `substanceConversionFactor` from the **Submodel** object divided by the conversion factor defined by the compartment replacement construct.

Species concentrations of species from compartments with undefined unit types will be converted according to the `substanceConversionFactor` alone, if no conversion factor is defined for its compartment. An odd potential situation arises here in the case where the species' compartment has been actually deleted instead of replaced, the replacement species being put into a new compartment in the containing model. In this case, no automatic conversion factor is possible, and if one is needed, it must be set explicitly on the species' replacement itself. Another complication is the situation where a species is set `hasOnlySubstanceUnits`="`true`" in the submodel, but is set `hasOnlySubstanceUnits`="`false`" in its replacement, or vice versa. In this case, the species must be converted according to the actual value of its compartment. If an explicit conversion factor is set, it is assumed that the modeler took this into consideration, and created an assignment rule (or similar) such that the conversion parameter would function appropriately. If not, the automatic conversion must use the value for the compartment of the replacement species to convert the species values to amounts from concentrations, and back again. Unreplaced species are still converted, but if they were in amounts before, they remain in amounts afterwards and likewise when in concentrations.

Any math not directly associated with a replaced element and that does not have a defined unit type is assumed to exist in the same scale as all other similar elements across all submodels. The only example of this in the Level 3 Core is the math associated with the **Priority** subcomponent of **Event** objects. A **Priority** element may be replaced directly by a **ReplacedElement** construct or by replacing its parent **Event**, but when comparing priority expressions from submodels with priority expressions from the containing model or from other submodels, they are all assumed to be on the same scale relative to each other. (If one model had priorities set on a scale of 0 to 10 and another had priorities set on a scale of -100 to 100, that is just the way it is, and to fix it, all incompatible priorities would have to be replaced.) The same would be true of math defined in any other Level 3 package without a defined unit type, or with a newly-defined unit type: none of it would be converted automatically, and all such elements would have to be converted explicitly by being replaced, or that package would have to extend this Hierarchical Model Composition package to define a new attribute on Submodel that could be used to automatically convert all such elements in the submodel with that unit type.

3.9 Namespace scoping rules for identifiers

In the Hierarchical Model Composition package, as in SBML Level 3 Version 1 Core, the **Model** object contains the main components of an SBML model, such as the species, compartments and reactions. The package adds the ability to put *multiple* models inside an SBML document, and therefore must define the scope of identifiers in such a way that identifier collisions are prevented. In this section, we describe the scoping rules for all of the types and classes defined in Sections 3.2–3.7.

1. A shared namespace exists for `SId` values defined at the level of the SBML document. This namespace applies to the identifiers (i.e., values of `id` attributes) of all **Model** and **ExternalModelDefinition** objects within the SBML document. (An **ExternalModelDefinition** object references a model elsewhere, so in this sense, both **Model** and **ExternalModelDefinition** are types of models.) The identifier of every **Model** and **ExternalModelDefinition** object must be unique across the set of all such identifiers in the document. The namespace is limited to that SBML document, and is not shared with any other SBML document, even if that document is referenced via an **ExternalModelDefinition**. This namespace is known as *the model namespace of the document*.

2. The namespace for `SId` identifiers defined *within* a **Model** object used in Hierarchical Model Composition follows the same rules as those defined in SBML Level 3 Core for plain **Model** objects. That is, the scope of the identifiers is limited to the enclosing **Model** object. This means that two or more **Model** objects in the same document may reuse the same object identifiers inside of them; the identifiers do not need to be unique at the level of the SBML document. (For example, two model definitions could use the same `SId` value for **Parameter** objects within their respective contents. Note, however, that this does *not* imply that the two objects are equated with each other!) This is known as the *object namespace of the model*. An implication of this rule is that to fully locate an object, one must know not only the object's identifier, but also the identifier of the model in which it is located.

3. The identifier of every **UnitDefinition** object must be unique across the set of all such identifiers in the **Model** to which they belong. (This is the same as in SBML Level 3 Version 1 Core.) This namespace is referred to as the *unit namespace of the model*. Similar to the case above, an implication of this rule is that to fully locate a user-defined unit definition when there are multiple models in an SBML document, one must know not only the unit definition's identifier, but also the identifier of the model in which it is located.

4. The Hierarchical Model Composition package defines a new kind of component: the *port*, represented by **Port** objects. The identifier of every **Port** object must be unique across the set of all such identifiers in the **Model** object to which they belong. Again, an implication of this rule is that to fully locate a port when there are multiple models in an SBML document, one must know not only the port's identifier, but also the identifier of the model in which it is located.

5. **Reaction** objects introduce a local namespace for **LocalParameter** objects. These objects cannot be referenced from outside a given reaction definition. For the Hierarchical Model Composition package, the implication is that the the **SBaseRef** class (Section 3.7) cannot reference reaction local parameters simply by their identifiers (i.e., the value of their `id` attribute). However, the **LocalParameter** objects *can* be given meta identifiers (a value for their ***SBase***-derived `metaid` attribute), and be referenced using those.

The following example may clarify some of these rules. Suppose a given SBML document contains a **Model** object having the identifier "`mod1`". This **Model** cannot contain another object with the same identifier (e.g., it could not have a **Parameter** object with the identifier "`mod1`"), nor can there be any other **Model** or **ExternalModelDefinition** objects identified as "`mod1`" within the same SBML document. The first restriction is simply the regular SBML rule about uniqueness of identifiers throughout a **Model** object; the second restriction is due to point (1) above. On the other hand, there could be a second **Model** object in the same document containing a component (e.g., a **Parameter**) with the identifier "`mod1`". This would not conflict with the first **Model** identifier (because the **Parameter** would be effectively hidden at a lower level within the second **Model**).

4 Examples

This section contains a variety of examples of SBML Level 3 Version 1 documents employing the Hierarchical Model Composition package.

4.1 Simple aggregate model

The following is a simple aggregate model, with one defined model being instantiated twice:

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level3/version1/core" level="3" version="1"
      xmlns:comp="http://www.sbml.org/sbml/level3/version1/comp/version1" comp:required="true">

  <model id="aggregate">
    <comp:listOfSubmodels>
      <comp:submodel comp:id="submod1" comp:modelRef="enzyme"/>
      <comp:submodel comp:id="submod2" comp:modelRef="enzyme"/>
    </comp:listOfSubmodels>
  </model>

  <comp:listOfModelDefinitions>
    <comp:modelDefinition id="enzyme" name="enzyme">
      <listOfCompartments>
        <compartment id="comp" spatialDimensions="3" size="1" constant="true"/>
      </listOfCompartments>
      <listOfSpecies>
        <species id="S"  compartment="comp" hasOnlySubstanceUnits="false"
                         boundaryCondition="false" constant="false"/>
        <species id="E"  compartment="comp" hasOnlySubstanceUnits="false"
                         boundaryCondition="false" constant="false"/>
        <species id="D"  compartment="comp" hasOnlySubstanceUnits="false"
                         boundaryCondition="false" constant="false"/>
        <species id="ES" compartment="comp" hasOnlySubstanceUnits="false"
      </listOfSpecies>
      <listOfReactions>
        <reaction id="J0" reversible="true" fast="false">
          <listOfReactants>
            <speciesReference species="S" stoichiometry="1" constant="true"/>
            <speciesReference species="E" stoichiometry="1" constant="true"/>
          </listOfReactants>
          <listOfProducts>
            <speciesReference species="ES" stoichiometry="1" constant="true"/>
          </listOfProducts>
        </reaction>
        <reaction id="J1" reversible="true" fast="false">
          <listOfReactants>
            <speciesReference species="ES" stoichiometry="1" constant="true"/>
          </listOfReactants>
          <listOfProducts>
            <speciesReference species="E" stoichiometry="1" constant="true"/>
            <speciesReference species="D" stoichiometry="1" constant="true"/>
          </listOfProducts>
        </reaction>
      </listOfReactions>
    </comp:modelDefinition>
  </comp:listOfModelDefinitions>
</sbml>
```

In the model above, we defined a two-step enzymatic process, with species "`S`" and "`E`" forming a complex, then dissociating to "`E`" and "`D`". The aggregate model instantiates it twice, so the resulting model (the one with the identifier "`aggregate`") has two parallel processes in two parallel compartments performing the same reaction. The compartments and processes are independent, because there is nothing in the model that links them together.

4.2 Example of importing definitions from external files

Now suppose that we have saved the above SBML content to a file named "enzyme_model.xml". The following example imports the model "enzyme" from that file into a new model:

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level3/version1/core" level="3" version="1"
      xmlns:comp="http://www.sbml.org/sbml/level3/version1/comp/version1" comp:required="true">
  <model>
    <listOfCompartments>
      <compartment id="comp" spatialDimensions="3" size="1" constant="true">
        <comp:listOfReplacedElements>
          <comp:replacedElement comp:idRef="comp" comp:submodelRef="A"/>
          <comp:replacedElement comp:idRef="comp" comp:submodelRef="B"/>
        </comp:listOfReplacedElements>
      </compartment>
    </listOfCompartments>
    <listOfSpecies>
      <species id="S" compartment="comp" hasOnlySubstanceUnits="false"
                      boundaryCondition="false" constant="false">
        <comp:listOfReplacedElements>
          <comp:replacedElement comp:idRef="S" comp:submodelRef="A"/>
          <comp:replacedElement comp:idRef="S" comp:submodelRef="B"/>
        </comp:listOfReplacedElements>
      </species>
    </listOfSpecies>
    <comp:listOfSubmodels>
      <comp:submodel comp:id="A" comp:modelRef="ExtMod1"/>
      <comp:submodel comp:id="B" comp:modelRef="ExtMod1"/>
    </comp:listOfSubmodels>
  </model>
  <comp:listOfExternalModelDefinitions>
    <comp:externalModelDefinition comp:id="ExtMod1" comp:source="enzyme_model.xml"
                                  comp:model="enzyme"/>
  </comp:listOfExternalModelDefinitions>
</sbml>
```

In the model above, we import "enzyme" twice. We then create a compartment and species local to the parent model, but use that compartment and species to replace "comp" and "S", respectively, from the two instantiations of "enzyme". The result is a model with a single compartment and two reactions; the species "S" can either bind with enzyme "E" (from instance "A") to form "D" (from instance "A"), or with enzyme "E" (from instance "B") to form "D" (from instance "B").

4.3 Example of using ports

In the following, we define one model ("simple") with a single reaction involving species "S" and "D", and ports, and we again import model "enzyme":

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level3/version1/core" level="3" version="1"
      xmlns:comp="http://www.sbml.org/sbml/level3/version1/comp/version1" comp:required="true">
  <model id="complexified">
    <listOfCompartments>
      <compartment id="comp" spatialDimensions="3" size="1" constant="true">
        <comp:listOfReplacedElements>
          <comp:replacedElement comp:idRef="comp" comp:submodelRef="A"/>
          <comp:replacedElement comp:portRef="comp_port" comp:submodelRef="B"/>
        </comp:listOfReplacedElements>
      </compartment>
    </listOfCompartments>
    <listOfSpecies>
      <species id="S" compartment="comp" initialConcentration="5" hasOnlySubstanceUnits="false"
               boundaryCondition="false" constant="false">
```

```
        <comp:listOfReplacedElements>
          <comp:replacedElement comp:idRef="S" comp:submodelRef="A"/>
          <comp:replacedElement comp:portRef="S_port" comp:submodelRef="B"/>
        </comp:listOfReplacedElements>
      </species>
      <species id="D" compartment="comp" initialConcentration="10" hasOnlySubstanceUnits="false"
               boundaryCondition="false" constant="false">
        <comp:listOfReplacedElements>
          <comp:replacedElement comp:idRef="D" comp:submodelRef="A"/>
          <comp:replacedElement comp:portRef="D_port" comp:submodelRef="B"/>
        </comp:listOfReplacedElements>
      </species>
    </listOfSpecies>
    <comp:listOfSubmodels>
      <comp:submodel comp:id="A" comp:modelRef="ExtMod1"/>
      <comp:submodel comp:id="B" comp:modelRef="simple">
        <comp:listOfDeletions>
          <comp:deletion comp:portRef="J0_port"/>
        </comp:listOfDeletions>
      </comp:submodel>
    </comp:listOfSubmodels>
  </model>
  <comp:listOfModelDefinitions>
    <comp:modelDefinition id="simple">
      <listOfCompartments>
        <compartment id="comp" spatialDimensions="3" size="1" constant="true"/>
      </listOfCompartments>
      <listOfSpecies>
        <species id="S" compartment="comp" initialConcentration="5" hasOnlySubstanceUnits="false"
                 boundaryCondition="false" constant="false"/>
        <species id="D" compartment="comp" initialConcentration="10" hasOnlySubstanceUnits="false"
                 boundaryCondition="false" constant="false"/>
      </listOfSpecies>
      <listOfReactions>
        <reaction id="J0" reversible="true" fast="false">
          <listOfReactants>
            <speciesReference species="S" stoichiometry="1" constant="true"/>
          </listOfReactants>
          <listOfProducts>
            <speciesReference species="D" stoichiometry="1" constant="true"/>
          </listOfProducts>
        </reaction>
      </listOfReactions>
      <comp:listOfPorts>
        <comp:port comp:idRef="S" comp:id="S_port"/>
        <comp:port comp:idRef="D" comp:id="D_port"/>
        <comp:port comp:idRef="comp" comp:id="comp_port"/>
        <comp:port comp:idRef="J0" comp:id="J0_port"/>
      </comp:listOfPorts>
    </comp:modelDefinition>
  </comp:listOfModelDefinitions>
  <comp:listOfExternalModelDefinitions>
    <comp:externalModelDefinition comp:id="ExtMod1" comp:source="enzyme_model.xml"
                                  comp:modelRef="enzyme"/>
  </comp:listOfExternalModelDefinitions>
</sbml>
```

In model “`simple`” above, we give ports to the compartment, the two species, and the reaction. Then, in model “`complexified`”, we import both “`simple`” and the model “`enzyme`” from the file “`enzyme_model.xml`”, and replace the simple reaction with the more complex two-step reaction by deleting reaction “`J0`” from model “`simple`” and replacing “`S`” and “`D`” from both models with local replacements. Note that it is model “`simple`” that defines the initial concentrations of “`S`” and “`D`”, so our modeler set the attribute `identical` to “`true`” for those elements, faithfully reproducing the values “`5`” and “`10`” in the local copy, and set the attribute `identical` to “`false`” for the replacement of those elements from model “`enzyme`”. Also note that since “`simple`” defines ports, the `port` attribute is used for the subelements that referenced "`simple`" model elements, but "`symbol`" still had to be used for subelements referencing "`enzyme`".

In the resulting model, "`S`" is converted to "`D`" by a two-step enzymatic reaction defined wholly in model "`enzyme`", with the initial concentrations of "`S`" and "`D`" set, in effect, in "`simple`" (through the appropriate setting of the attribute `identical`). If "`simple`" had other reactions that created "`S`" and destroyed "`D`", "`S`" would be created, would bind with "`E`" to form "`D`", and "`D`" would then be destroyed, even though those reaction steps were defined in separate models.

4.4 Example of replacement

In reference to the previous example, what if we would like to annotate that the deleted reaction had been *replaced* by the two-step enzymatic process? To do that, we must move those reactions to the parent model, and, since those reactions involve "`E`" and "`ES`", we must also move those species as well. The following SBML fragment demonstrates one way of doing that.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level3/version1/core" level="3" version="1"
      xmlns:comp="http://www.sbml.org/sbml/level3/version1/comp/version1" comp:required="true">
  <model id="complexified">
    <listOfCompartments>
      <compartment id="comp" spatialDimensions="3" size="1" constant="true">
        <comp:listOfReplacedElements>
          <comp:replacedElement comp:idRef="comp" comp:submodelRef="A"/>
          <comp:replacedElement comp:portRef="comp_port" comp:submodelRef="B"/>
        </comp:listOfReplacedElements>
      </compartment>
    </listOfCompartments>
    <listOfSpecies>
      <species id="S" compartment="comp" initialConcentration="5" hasOnlySubstanceUnits="false"
                      boundaryCondition="false" constant="false">
        <comp:listOfReplacedElements>
          <comp:replacedElement comp:idRef="S" comp:submodelRef="A"/>
          <comp:replacedElement comp:portRef="S_port" comp:submodelRef="B"/>
        </comp:listOfReplacedElements>
      </species>
      <species id="D" compartment="comp" initialConcentration="10" hasOnlySubstanceUnits="false"
                      boundaryCondition="false" constant="false">
        <comp:listOfReplacedElements>
          <comp:replacedElement comp:idRef="D" comp:submodelRef="A"/>
          <comp:replacedElement comp:portRef="D_port" comp:submodelRef="B"/>
        </comp:listOfReplacedElements>
      </species>
      <species id="E" compartment="comp" hasOnlySubstanceUnits="false"
                      boundaryCondition="false" constant="false">
        <comp:listOfReplacedElements>
          <comp:replacedElement comp:portRef="E_port" comp:submodelRef="A"/>
          <comp:replacedElement comp:portRef="D_port" comp:submodelRef="B"/>
        </comp:listOfReplacedElements>
      </species>
      <species id="ES" compartment="comp" hasOnlySubstanceUnits="false"
                       boundaryCondition="false" constant="false">
        <comp:listOfReplacedElements>
          <comp:replacedElement comp:portRef="ES_port" comp:submodelRef="A"/>
          <comp:replacedElement comp:portRef="D_port" comp:submodelRef="B"/>
        </comp:listOfReplacedElements>
      </species>
    </listOfSpecies>
    <listOfReactions>
      <reaction id="J0" reversible="true" fast="false">
        <listOfReactants>
          <speciesReference species="S" stoichiometry="1" constant="true"/>
          <speciesReference species="E" stoichiometry="1" constant="true"/>
```

```
        </listOfReactants>
        <listOfProducts>
          <speciesReference species="ES" stoichiometry="1" constant="true"/>
        </listOfProducts>
        <comp:listOfReplacedElements>
          <comp:replacedElement comp:submodelRef="B" comp:deletion="oldrxn"/>
          <comp:replacedElement comp:portRef="J0_port" comp:submodelRef="A"/>
        </comp:listOfReplacedElements>
      </reaction>
      <reaction id="J1" reversible="true" fast="false">
        <listOfReactants>
          <speciesReference species="ES" stoichiometry="1" constant="true"/>
        </listOfReactants>
        <listOfProducts>
          <speciesReference species="E" stoichiometry="1" constant="true"/>
          <speciesReference species="D" stoichiometry="1" constant="true"/>
        </listOfProducts>
        <comp:listOfReplacedElements>
          <comp:replacedElement comp:submodelRef="B" comp:deletion="oldrxn"/>
          <comp:replacedElement comp:portRef="J1_port" comp:submodelRef="A"/>
        </comp:listOfReplacedElements>
      </reaction>
    </listOfReactions>
    <comp:listOfSubmodels>
      <comp:submodel comp:id="A" comp:modelRef="enzyme"/>
      <comp:submodel comp:id="B" comp:modelRef="simple">
        <comp:listOfDeletions>
          <comp:deletion comp:portRef="J0_port" comp:id="oldrxn"/>
        </comp:listOfDeletions>
      </comp:submodel>
    </comp:listOfSubmodels>
  </model>
  <comp:listOfModelDefinitions>
    <comp:modelDefinition id="enzyme" name="enzyme">
      <listOfCompartments>
        <compartment id="comp" spatialDimensions="3" size="1" constant="true"/>
      </listOfCompartments>
      <listOfSpecies>
        <species id="S"  compartment="comp" hasOnlySubstanceUnits="false"
                         boundaryCondition="false" constant="false"/>
        <species id="E"  compartment="comp" hasOnlySubstanceUnits="false"
                         boundaryCondition="false" constant="false"/>
        <species id="D"  compartment="comp" hasOnlySubstanceUnits="false"
                         boundaryCondition="false" constant="false"/>
        <species id="ES" compartment="comp" hasOnlySubstanceUnits="false"
                         boundaryCondition="false" constant="false"/>
      </listOfSpecies>
      <listOfReactions>
        <reaction id="J0" reversible="true" fast="false">
          <listOfReactants>
            <speciesReference species="S" stoichiometry="1" constant="true"/>
            <speciesReference species="E" stoichiometry="1" constant="true"/>
          </listOfReactants>
          <listOfProducts>
            <speciesReference species="ES" stoichiometry="1" constant="true"/>
          </listOfProducts>
        </reaction>
        <reaction id="J1" reversible="true" fast="false">
          <listOfReactants>
            <speciesReference species="ES" stoichiometry="1" constant="true"/>
          </listOfReactants>
          <listOfProducts>
            <speciesReference species="E" stoichiometry="1" constant="true"/>
            <speciesReference species="D" stoichiometry="1" constant="true"/>
          </listOfProducts>
        </reaction>
      </listOfReactions>
```

```
      <comp:listOfPorts>
        <comp:port comp:idRef="comp" comp:id="comp_port"/>
        <comp:port comp:idRef="S" comp:id="S_port"/>
        <comp:port comp:idRef="E" comp:id="E_port"/>
        <comp:port comp:idRef="D" comp:id="D_port"/>
        <comp:port comp:idRef="ES" comp:id="ES_port"/>
        <comp:port comp:idRef="J0" comp:id="J0_port"/>
        <comp:port comp:idRef="J1" comp:id="J1_port"/>
      </comp:listOfPorts>
    </comp:modelDefinition>
    <comp:modelDefinition id="simple">
      <listOfCompartments>
        <compartment id="comp" spatialDimensions="3" size="1" constant="true"/>
      </listOfCompartments>
      <listOfSpecies>
        <species id="S" compartment="comp" initialConcentration="5" hasOnlySubstanceUnits="false"
                 boundaryCondition="false" constant="false"/>
        <species id="D" compartment="comp" initialConcentration="10" hasOnlySubstanceUnits="false"
                 boundaryCondition="false" constant="false"/>
      </listOfSpecies>
      <listOfReactions>
        <reaction id="J0" reversible="true" fast="false">
          <listOfReactants>
            <speciesReference species="S" stoichiometry="1" constant="true"/>
          </listOfReactants>
          <listOfProducts>
            <speciesReference species="D" stoichiometry="1" constant="true"/>
          </listOfProducts>
        </reaction>
      </listOfReactions>
      <comp:listOfPorts>
        <comp:port comp:idRef="S" comp:id="S_port"/>
        <comp:port comp:idRef="D" comp:id="D_port"/>
        <comp:port comp:idRef="comp" comp:id="comp_port"/>
        <comp:port comp:idRef="J0" comp:id="J0_port"/>
      </comp:listOfPorts>
    </comp:modelDefinition>
  </comp:listOfModelDefinitions>
</sbml>
```

In the example above, we have recreated "enzyme" so as to provide it with ports, then recreated basically the entire model in the parent "complexified" so we can reference the deleted "oldrxn" in the replacements lists. Note that we list the deletion of "oldrxn" both for the two new reactions and for the two new species "E" and "ES", since those species were themselves elided in the simple form of the "S" to "D" reaction in "simple". The attribute identical is used throughout, so that any visualization or manipulation software knows that the only reason those elements exist in the parent model is to create a reference, not to actually change the element itself.

5 Best practices

This section is still unfinished.

In this section, we recommend a number of practices for using and interpreting various constructs in the Hierarchical Model Composition package. These recommendations are non-normative, but we advocate them strongly; ignoring them will not render a model invalid, but may reduce interoperability between software and models.

5.1 Best practices for using SBaseRef for references

5.2 Best practices for deletions and replacements

Note that there may be model composition situations in which a model contains elements that do not have an identifier, nor a meta identifier, nor a port identifier. In that case, there is no way to refer to it using the with the **Deletion** or **ReplacedElement** objects defined in this specification. A viable alternative to use in that case is to copy the original model and modify it, either to perform the desired deletions directly or to add the necessary identifiers so that **Deletion** objects can be defined and used in a submodel. (Presumably, the original model was readable in the first place, or else composition would have been impossible anyway.) Copying a model and making one's own version may have additional benefits, such as the ability to control versions explicitly and references. A second method may be to delete or replace the parent object of the element you wish to replace, assuming that element has an identifier, meta identifier, or port identifier. When this is performed, the errant element will be deleted implicitly, allowing you to create replacements in the containing model without overlapping functionality.

5.3 Best practices for using ports

Software developers who wish to include restrictions are encouraged to experiment here, and add new attributes in a namespace of their own devising.

A Validation of SBML documents

An important issue for software systems is being able to determine the validity of a given SBML document that uses constructs from the Hierarchical Model Composition package. This section describes operational rules for assessing validity.

A.1 Validation procedure

The validation rules below only apply to models defined in the SBML document being validated. Models defined in external files are not required to be valid in and of themselves; the only requirement is that the model containing the instantiation of an externally-defined model must be the one that is valid. That may seem counterintuitive, but the reason is that replacements and deletions can be used to render valid what might otherwise be invalid. For example, an external model that omits required attributes on some objects (which would be invalid according to SBML Level 3 Version 1 Core) may become valid if those objects are replaced by objects that *are* valid, or if they are deleted entirely. Similarly, references to nonexistent objects may themselves be deleted, or illegal combinations of objects may be rectified, etc.

The two-phase validation approach

To understand the validation procedure for models that use Hierarchical Model Composition, it is helpful to think in terms of an analogy to baking. To make a cake, one first assembles specific ingredients in a certain way, and then one bakes the result to produce the final product—the cake. An SBML document using the Hierarchical Model Composition package constructs is analogous to *only a recipe*: it is a description of how to assemble ingredients in a certain way to create a "cake", but it is *not the cake itself.* The cake is only produced after *following* the instructions, which here involves traversing the various model, submodel, deletion, and replacement descriptions.

We decompose validation of such a composite model into two phases:

1. *Validate the "recipe"*. The submodel, deletion, and replacement constructs themselves must be valid.
2. *Validate the "cake"*. The model produced by interpreting the various constructs must be valid SBML.

The first phase involves checking the aggregation, deletion and linkage instructions defined by the Hierarchical Model Composition constructs in an SBML document. The **Submodel**, **Port**, **Deletion**, **ReplacedElement**, **ReplacedBy** and other constructs defined in this specification must be valid according to the rules defined in Section A.2. Passing this phase means that the constructs are well-formed, referenced files and models and other entities exist, ports have identifiers in the relevant namespaces, and so on.

The second validation phase takes place after interpreting the Hierarchical Model Composition constructs. The result of this phase must be a valid SBML model. This verification can in principle be performed in various ways. In this specification, we describe one approach below that involves interpreting the Hierarchical Model Composition constructs to produce a kind of "flattened" version of the model devoid of the Hierarchical Model Composition package constructs. The "flattened" version of the model only exists in memory: the referenced files are not actually modified, but rather, the interpretation of the package constructs leads to an in-memory representation of a final, composite model implied by following the recipe. This generated model can then be tested against the rules for SBML validity defined in the SBML Level 3 Version 1 Core specification. Performing this "flattening" allows for the most straightforward way of testing the validity of the resulting SBML model; however, it is *not part of the requirements for this package*. The requirements are only that the model implied by the package constructs is valid.

Example algorithm for producing a "flattened" model

Figure 8 on the following page presents a possible algorithm for interpreting the Hierarchical Model Composition constructs and creating a "flattened" SBML document. As explained above, this procedure can be used as part of a process to test the validity of an SBML document that uses Hierarchical Model Composition. After performing the steps of the flattening algorithm, the result should be evaluated for validity according to the normal rules of SBML Level 3 Version 1 Core and (if applicable) the rules defined by any other Level 3 packages used in the model.

Step	Procedure
1.	Examine the submodels of the model being validated. If any submodel itself contains submodels, perform this algorithm in its entirety on each submodel before proceeding to the next step on the model.
2.	For every submodel identifier "*M*", verify that no object identifier or meta identifier (i.e., the `id` or `metaid` attribute values of the object) begin with the character sequence "*M*_". For example, if a submodel has identifier "`foo`", change it (for the purposes of this algorithm) to "`foo_`". If an object has identifier "`foo_`" already, add another underscore.
3.	Remove any submodel object that have been replaced or deleted.
4.	Transform the remaining submodel object such that: a) Every identifier (`id` attribute) is changed to a value of the form "*M_id*", where "*id*" is the original `id` value and "*M*" is the submodel identifier. b) Every meta identifier (`metaid` attribute) is changed to a value of the form "*M_metaid*", where "*metaid*" is the original `metaid` value and "*M*" is the submodel identifier.
5.	For every object that has not been removed in the submodel, adjust every reference to the identifiers that were transformed in the previous step as follows. Change every `SIdRef` value to the form form "*M_id*", where "*id*" is the original `SIdRef` value and "*M*" is the submodel identifier, and change every XML `IDREF` value to the form "*M_metaid*", where "*metaid*" is the original `IDREF` value. *However*, if the submodel has been replaced as a consequence of applying either a **ReplacedElement** or **ReplacedBy** construct, the `SIdRef` and `IDREF` values that pointed to the submodel need to be changed to the `SId` or `ID` value, respectively, of the object replacing it.
6.	After performing the tasks above for all remaining objects, merge the objects of the remaining submodels into a single model. The various lists (list of species, list of compartments, etc.) are merged in this step. Annotations must be preserved, as should constructs from other SBML Level 3 packages.

Figure 8: *Example algorithm for "flattening" a model to remove Hierarchical Model Composition package constructs.*

Additional remarks about the validation procedure

☞ When instantiating a model, one does not have to first test the validity of that model. If it is in the same file as the containing model, it will be tested anyway when the result of the "flattening" algorithm is checked for validity in the second phase. If it is in a different file, that file's validity (or lack thereof) should not affect the validity of the file being tested, though a validator may warn the user of this situation if it desires.

A.2 Validation and consistency rules

This section summarizes all the conditions that must (or in some cases, at least *should*) be true of an SBML Level 3 Version 1 model that uses the Hierarchical Model Composition package. We use the same conventions as are used in the SBML Level 3 Version 1 Core specification document. In particular, there are different degrees of rule strictness. Formally, the differences are expressed in the statement of a rule: either a rule states that a condition *must* be true, or a rule states that it *should* be true. Rules of the former kind are strict SBML validation rules—a model encoded in SBML must conform to all of them in order to be considered valid. Rules of the latter kind are consistency rules. To help highlight these differences, we use the following three symbols next to the rule numbers:

☑ A checked box indicates a *requirement* for SBML conformance. If a model does not follow this rule, it does not conform to the Hierarchical Model Composition specification. (Mnemonic intention behind the choice of symbol: "This must be checked.")

▲ A triangle indicates a *recommendation* for model consistency. If a model does not follow this rule, it is not considered strictly invalid as far as the Hierarchical Model Composition specification is concerned; however, it indicates that the model contains a physical or conceptual inconsistency. (Mnemonic intention behind the choice of symbol: "This is a cause for warning.")

★ A star indicates a strong recommendation for good modeling practice. This rule is not strictly a matter of SBML encoding, but the recommendation comes from logical reasoning. As in the previous case, if a model does not follow this rule, it is not strictly considered an invalid SBML encoding. (Mnemonic intention behind the choice of symbol: "You're a star if you heed this.")

The validation rules listed in the following subsections are all stated or implied in the rest of this specification document. They are enumerated here for convenience. Unless explicitly stated, all validation rules concern objects and attributes specifically defined in the Hierarchical Model Composition package.

☞ For convenience and brevity, we use the shorthand "`comp:x`" to stand for an attribute or element name `x` in the namespace for the Hierarchical Model Composition package, using the namespace prefix `comp`. In reality, the prefix string may be different from the literal "`comp`" used here (and indeed, it can be any valid XML namespace prefix that the modeler or software chooses). We use "`comp:x`" because it is shorter than to write a full explanation everywhere we refer to an attribute or element in the Hierarchical Model Composition package namespace.

General rules about this package

comp-10101 ☑ In order to conform to the Hierarchical Model Composition package specification for SBML Level 3 Version 1, an SBML document must declare the use of the following XML Namespace: "`http://www.sbml.org/sbml/level3/version1/comp/version1`". (References: SBML L3 Hierarchical Composition Package V1, Section 3.1.)

comp-10102 ☑ Wherever they appear in an SBML document, elements and attributes from the Hierarchical Model Composition package must be declared either implicitly or explicitly to be in the XML namespace "`http://www.sbml.org/sbml/level3/version1/comp/version1`". (References: SBML L3 Hierarchical Composition Package V1, Section 3.1.)

General rules about identifiers

comp-10301 ☑ (Extends validation rule #10301 in the SBML Level 3 Version 1 Core specification.) Within a **Model** or **ExternalModelDefinition** object, the values of the attributes `id` and `comp:id` on every instance of the following classes of objects must be unique across the set of all `id` and `comp:id` attribute values of all such objects in a model: the **Model** itself, plus all contained **FunctionDefinition**, **Compartment**, **Species**, **Reaction**, **SpeciesReference**, **ModifierSpeciesReference**, **Event**, and **Parameter** objects, plus the **Submodel** and **Deletion** objects defined by the Hierarchical Model Composition package. (References: SBML L3 Hierarchical Composition Package V1 Section 3.9.)

comp-10302 ☑ The values of the attributes `id` and `comp:id` on every instance of all **Model** and **ExternalModelDefinition** objects must be unique across the set of all `id` and `comp:id` attribute values of such identifiers in the SBML document to which they belong. (References: SBML L3 Hierarchical Composition Package V1 Section 3.9.)

comp-10303 ☑ Within a **Model** or **ExternalModelDefinition** object inside an SBML document, the value of the attribute `comp:id` on every instance of a **Port** object must be unique across the set of all `comp:id` attribute values of all such objects in the model. (References: SBML L3 Hierarchical Composition Package V1 Section 3.9.)

comp-10304 ☑ The value of a `comp:id` attribute must always conform to the syntax of the SBML data type `SId`. (References: SBML L3V1 Section 3.1.7.)

comp-10308 ☑ The value of a `comp:submodelRef` attribute on a **ReplacedElement** object must always conform to the syntax of the SBML data type `SId`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.6.2.)

comp-10309 ☑ The value of a `comp:deletion` attribute on a **ReplacedElement** object must always conform to the syntax of the SBML data type `SId`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.6.2.)

comp-10310 ☑ The value of a `comp:conversionFactor` attribute on a **ReplacedElement** object must always conform to the syntax of the SBML data type `SId`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.6.2.)

Rules for the extended SBase abstract class

comp-20101 ☑ Any object derived from the extended ***SBase*** class (as defined in the Hierarchical Model Composition package) may contain at most one instance of a **ListOfReplacedElements** subobject. (References: SBML L3 Hierarchical Composition Package V1 Section 3.6.)

comp-20102 ☑ Apart from the general notes and annotation subobjects permitted on all SBML objects, a **ListOfReplacedElements** container object may only contain **ReplacedElement** objects. (References: SBML L3 Hierarchical Composition Package V1 Section 3.6.)

comp-20103 ☑ A **ListOfReplacedElements** object may have the optional attributes `metaid` and `sboTerm` defined by SBML Level 3 Core. No other attributes from the SBML Level 3 Core namespace or the Hierarchical Model Composition namespace are permitted on a **ListOfReplacedElements** object. (References: SBML L3 Hierarchical Composition Package V1 Section 3.6.)

comp-20104 ☑ The **ListOfReplacedElements** in an ***SBase*** object is optional, but if present, must not be empty. (References: SBML L3 Hierarchical Composition Package V1 Section 3.6.)

Rules for the extended SBML class

comp-20201 ☑ In all SBML documents using the Hierarchical Model Composition package, the **SBML** object must include a value for the attribute `comp:required` attribute. (References: SBML L3V1 Section 4.1.2.)

comp-20202 ☑ The value of attribute `comp:required` on the **SBML** object must be of the data type `boolean`. (References: SBML L3V1 Section 4.1.2.)

need find the section

comp-20203 ☑ The value of attribute `comp:required` on the **SBML** object must be set to "`true`" if the **Model** object within the **SBML** object contains any **Submodel** with **Species**, **Parameter**, **Reaction**, or **Event** objects that have been either directly or indirectly replaced. (References: SBML L3 Hierarchical Composition Package V1 Section **??**.)

comp-20204 ☑ There may be at most one instance of each of the following kinds of subobjects within an **SBML** object that uses the Hierarchical Model Composition package: **ListOfModelDefinitions** and **ListOfExternalModelDefinitions**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.3.)

comp-20205 ☑ The various **ListOf__** subobjects within an **SBML** object are optional, but if present, these container objects must not be empty. Specifically, if any of the following classes of objects is present within the **SBML** object, it must not be empty: **ListOfModelDefinitions** and **ListOfExternalModelDefinitions**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.3.)

comp-20206 ☑ Apart from the general notes and annotation subobjects permitted on all SBML objects, a **ListOfModelDefinitions** container may only contain extended **Model** class objects. (References: SBML L3 Hierarchical Composition Package V1 Section 3.3.)

comp-20207 ☑ Apart from the general notes and annotation subobjects permitted on all SBML objects, a **ListOfExternalModelDefinitions** container may only contain **ExternalModelDefinition** objects. (References: SBML L3 Hierarchical Composition Package V1 Section 3.3.)

comp-20208 ☑ A **ListOfModelDefinitions** object may have the optional attributes `metaid` and `sboTerm`. No other attributes from the SBML Level 3 Core namespace or the Hierarchical Model Composition namespace are permitted on a **ListOfModelDefinitions** object. (References: SBML L3 Hierarchical Composition Package V1 Section 3.3.)

comp-20209 ☑ A **ListOfExternalModelDefinitions** object may have the optional SBML core attributes `metaid` and `sboTerm`. No other attributes from the SBML Level 3 Core namespace or the comp namespace are permitted on a **ListOfExternalModelDefinitions** object. (References: SBML L3 Hierarchical Composition Package V1 Section 3.3.)

Rules for ExternalModelDefinition objects

comp-20301 ☑ An **ExternalModelDefinition** object may have the optional SBML Level 3 Core attributes `metaid` and `sboTerm`. No other attributes from the SBML Level 3 Core namespace are permitted on an **ExternalModelDefinition**. (References: SBML L3V1 Section 3.2.)

comp-20302 ☑ An **ExternalModelDefinition** object may the optional SBML Level 3 Core subobjects for notes and annotation. No other subobjects from the SBML Level 3 Core namespace are permitted in an **ExternalModelDefinition**. (References: SBML L3V1 Section 3.2.)

comp-20303 ☑ An ExternalModelDefinition object must have the required attributes `comp:id` and `comp:source`, and may have the optional attributes `comp:modelRef` and `comp:md5`. No other attributes from the Hierarchical Model Composition namespace are permitted on an **ExternalModelDefinition** object. (References: SBML L3 Hierarchical Composition Package V1 Section 3.3.2.)

comp-20304 ☑ The value of the `comp:source` attribute on an **ExternalModelDefinition** object must reference an SBML Level 3 document. (References: SBML L3 Hierarchical Composition Package V1 Section 3.3.2.)

comp-20305 ☑ The value of the `comp:modelRef` attribute on an **ExternalModelDefinition** object must be the value of an `id` attribute on a **Model** object in the SBML document referenced by the `comp:source` attribute. (References: SBML L3 Hierarchical Composition Package V1 Section 3.3.2.)

comp-20306 ▲ The value of the `comp:md5` attribute on an **ExternalModelDefinition** object should match the calculated MD5 checksum of the SBML document referenced by the `comp:source` attribute. (References: SBML L3 Hierarchical Composition Package V1 Section 3.3.2.)

comp-20307 ☑ The value of a `comp:source` attribute on an **ExternalModelDefinition** object must always conform to the syntax of the XML Schema 1.0 data type `anyURI`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.3.)

comp-20308 ☑ The value of a `comp:modelRef` attribute on an **ExternalModelDefinition** object must always conform to the syntax of the SBML data type `SId`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.3.)

comp-20309 ☑ The value of a `comp:md5` attribute on an **ExternalModelDefinition** object must always conform to the syntax of type `string`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.3.)

need find section

comp-20310 ☑ A **ExternalModelDefinition** object must not reference an **ExternalModelDefinition** in a different SBML document that, in turn, refers back to the original **ExternalModelDefinition** object, whether directly or indirectly through a chain of **ExternalModelDefinition** objects. (References: SBML L3 Hierarchical Composition Package V1 Section **??**.)

Rules for extended Model objects

comp-20501 ☑ There may be at most one instance of each of the following kinds of objects within a **Model** object using Hierarchical Model Composition: **ListOfSubmodels** and **ListOfPorts**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.4.)

comp-20502 ☑ The various **ListOf__** subobjects with an **Model** object are optional, but if present, these container object must not be empty. Specifically, if any of the following classes of objects are present on the **Model**, it must not be empty: **ListOfSubmodels** and **ListOfPorts**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.4.)

comp-20503 ☑ Apart from the general notes and annotation subobjects permitted on all SBML objects, a **ListOfSubmodels** container object may only contain **Submodel** objects. (References: SBML L3 Hierarchical Composition Package V1 Section 3.4.)

comp-20504 ☑ Apart from the general notes and annotation subobjects permitted on all SBML objects, a **ListOfPorts** container object may only contain **Port** objects. (References: SBML L3 Hierarchical Composition Package V1 Section 3.4.)

comp-20505 ☑ A **ListOfSubmodels** object may have the optional `metaid` and `sboTerm` defined by SBML Level 3 Core. No other attributes from the SBML Level 3 Core namespace or the Hierarchical Model Composition namespace are permitted on a **ListOfSubmodels** object. (References: SBML L3 Hierarchical Composition Package V1 Section 3.4.)

comp-20506 ☑ A **ListOfPorts** object may have the optional attributes `metaid` and `sboTerm` defined by SBML Level 3 Core. No other attributes from the SBML Level 3 Core namespace or the Hierarchical Model Composition namespace are permitted on a **ListOfPorts** object. (References: SBML L3 Hierarchical Composition Package V1 Section 3.4.)

Rules for Submodel objects

comp-20601 ☑ A **Submodel** object may have the optional SBML Level 3 Core attributes `metaid` and `sboTerm`. No other attributes from the SBML Level 3 Core namespace are permitted on an **Submodel** object. (References: SBML L3V1 Section 3.2.)

comp-20602 ☑ An **Submodel** object may have the optional SBML Level 3 Core subobjects for notes and annotation. No other elements from the SBML Level 3 Core namespace are permitted on an **Submodel** object. (References: SBML L3V1 Section 3.2.)

comp-20603 ☑ There may be at most one **ListOfDeletions** container object within a **Submodel** object. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.)

comp-20604 ☑ A **ListOfDeletions** container object within a **Submodel** object is optional, but if present, must not be empty. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.)

comp-20605 ☑ Apart from the general notes and annotation subobjects permitted on all SBML objects, a **ListOfDeletions** container object may only contain **Deletion** objects. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.)

comp-20606 ☑ A **ListOfDeletions** object may have the optional SBML core attributes `metaid` and `sboTerm`. No other attributes from the SBML Level 3 Core namespace or the comp namespace are permitted on a **ListOfDeletions** object. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.)

comp-20607 ☑ A **Submodel** object must have the required attributes `comp:id` and `comp:modelRef`, and may also have any or all of the following optional attributes: `comp:lengthConversionFactor`, `comp:areaConversionFactor`, `comp:volumeConversionFactor`, `comp:substanceConversionFactor`, `comp:timeConversionFactor`, and `comp:extentConversionFactor`. No other attributes from the Hierarchical Model Composition namespace are permitted on a **Submodel** object. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.)

comp-20608 ☑ The value of a `comp:modelRef` attribute on a **Submodel** object must always conform to the syntax of the SBML data type `SId`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.1.)

comp-20609 ☑ The value of a `comp:lengthConversionFactor` attribute on a **Submodel** object must always conform to the syntax of the SBML data type `SId`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.1.)

comp-20610 ☑ The value of a `comp:areaConversionFactor` attribute on a **Submodel** object must always conform to the syntax of the SBML data type `SId`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.1.)

comp-20611 ☑ The value of a `comp:volumeConversionFactor` attribute on a **Submodel** object must always conform to the syntax of the SBML data type `SId`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.1.)

comp-20612 ☑ The value of a `comp:substanceConversionFactor` attribute on a **Submodel** object must always conform to the syntax of the SBML data type `SId`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.1.)

comp-20613 ☑ The value of a `comp:timeConversionFactor` attribute on a **Submodel** object must always conform to the syntax of the SBML data type `SId`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.1.)

comp-20614 ☑ The value of a `comp:extentConversionFactor` attribute on a **Submodel** object must always conform to the syntax of the SBML data type `SId`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.1.)

comp-20615 ☑ The value of a `comp:modelRef` attribute on a **Submodel** must be the identifier of a **Model** or **ExternalModelDefinition** object in the same **SBML** object as the **Submodel**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.1.)

comp-20616 ☑ A **Model** object must not contain a **Submodel** which references that **Model** object itself. That is, the value of a `comp:modelRef` attribute on a **Submodel** must not be the value of the parent **Model** object's `id` attribute. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.1.)

comp-20617 ☑ A **Model** object must not contain a **Submodel** which references that **Model** indirectly. That is, the `comp:modelRef` attribute of a **Submodel** may not point to the `id` of a **Model** containing a **Submodel** object that references the original **Model** directly or indirectly through a chain of **Model**/**Submodel** pairs. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.1.)

comp-20618 ☑ The `comp:lengthConversionFactor` attribute on a **Submodel** object must be the identifier of a **Parameter** object in the same **Model** containing the **Submodel**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.1.)

comp-20619 ☑ The `comp:areaConversionFactor` attribute on a given **Submodel** object must be the identifier of a **Parameter** object defined in the same **Model** containing the **Submodel**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.1.)

comp-20620 ☑ The `comp:volumeConversionFactor` attribute on a given **Submodel** object must be the identifier of a **Parameter** object defined in the same **Model** containing the **Submodel**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.1.)

comp-20621 ☑ The `comp:substanceConversionFactor` attribute on a given **Submodel** object must be the identifier of a **Parameter** object defined in the same **Model** containing the **Submodel**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.1.)

comp-20622 ☑ The `comp:timeConversionFactor` attribute on a given **Submodel** object must be the identifier of a **Parameter** object defined in the same **Model** containing the **Submodel**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.1.)

comp-20623 ☑ The `comp:extentConversionFactor` attribute on a given **Submodel** object must be the identifier of a **Parameter** object defined in the same **Model** containing the **Submodel**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.1.)

Rules for the SBaseRef abstract object

comp-20701 ☑ The value of a `comp:portRef` attribute on an **SBaseRef** object must be the identifier of a **Port** object in the **Model** referenced by that **SBaseRef**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.7.1.)

comp-20702 ☑ The value of a `comp:idRef` attribute on an **SBaseRef** object must be the identifier of an object contained in (that is, within the `SId` namespace of) the **Model** referenced by that **SBaseRef**. This includes objects with `id` attributes defined in packages other than SBML Level 3 Core or the Hierarchical Model Composition package. (References: SBML L3 Hierarchical Composition Package V1 Section 3.7.1.)

comp-20703 ☑ The value of a `comp:unitRef` attribute on an **SBaseRef** object must be the identifier of a **UnitDefinition** object contained in the **Model** referenced by that **SBaseRef**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.7.1.)

comp-20704 ☑ The value of a `comp:metaIdRef` attribute on an **SBaseRef** object must be the value of a `comp:metaid` attribute on an element contained in the **Model** referenced by that **SBaseRef**. This includes elements with `metaid` attributes defined in packages other than SBML Level 3 Core or the Hierarchical Model Composition package. (References: SBML L3 Hierarchical Composition Package V1 Section 3.7.1.)

comp-20705 ☑ If an **SBaseRef** object contains an **SBaseRef** child, the parent **SBaseRef** must point to a **Submodel** object. (References: SBML L3 Hierarchical Composition Package V1 Section 3.7.2.)

comp-20706 ☑ The value of a `comp:portRef` attribute on an **SBaseRef** object must always conform to the syntax of the SBML data type `SId`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.7.1.)

comp-20707 ☑ The value of a `comp:idRef` attribute on an **SBaseRef** object must always conform to the syntax of the SBML data type `SId`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.7.1.)

comp-20708 ☑ The value of a `comp:unitRef` attribute on an **SBaseRef** object must always conform to the syntax of the SBML data type `UnitSId`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.7.1.)

comp-20709 ☑ The value of a `comp:metaIdRef` attribute on an **SBaseRef** object must always conform to the syntax of the XML data type `ID`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.7.1.)

Rules for Port objects

comp-20801 ☑ A **Port** object must point to another object; that is, a **Port** object must always have a value for one of the attributes `comp:idRef`, `comp:unitRef`, or `comp:metaIdRef`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.4.3.)

comp-20802 ☑ A **Port** object can only point to *one* other object; that is, a given **Port** object can only have a value for one of the attributes `comp:idRef`, `comp:unitRef`, or `comp:metaIdRef`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.4.3.)

comp-20803 ☑ A **Port** object must have a value for the required attribute `comp:id`, and one, and only one, of the attributes `comp:idRef`, `comp:unitRef`, or `comp:metaIdRef`. No other attributes from the Hierarchical Model Composition namespace are permitted on a **Port** object. (References: SBML L3 Hierarchical Composition Package V1 Section 3.4.3.)

comp-20804 ☑ Port definitions must be unique; that is, no two **Port** objects in a given **Model** may reference the same object in that **Model**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.4.3.)

Rules for Deletion objects

comp-20901 ☑ A **Deletion** object must point to another object; that is, a **Deletion** object must have a value for one of the attributes `comp:portRef`, `comp:idRef`, `comp:unitRef`, or `comp:metaIdRef`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.3.)

comp-20902 ☑ A **Deletion** object can only point to *one* other object; that is, a given **Deletion** object can only have a value for one of the attributes `comp:portRef`, `comp:idRef`, `comp:unitRef`, or `comp:metaIdRef`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.3.)

comp-20903 ☑ A **Deletion** object must have a value for the required attribute `comp:id`, and one, and only one, of the attributes `comp:portRef`, `comp:idRef`, `comp:unitRef`, or `comp:metaIdRef`. No other attributes from the Hierarchical Model Composition namespace are permitted on a **Deletion** object. (References: SBML L3 Hierarchical Composition Package V1 Section 3.5.3.)

Rules for ReplacedElement objects

comp-21001 ☑ A **ReplacedElement** object must point to another object; that is, a given **ReplacedElement** object must have a value for one of the following attributes: `comp:portRef`, `comp:idRef`, `comp:unitRef`, `comp:metaIdRef`, or `comp:deletion`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.6.2.)

comp-21002 ☑ A **ReplacedElement** object can only point to *one* target object; that is, a given **ReplacedElement** can only have a value for one of the following attributes: `comp:portRef`, `comp:idRef`, `comp:unitRef`, `comp:metaIdRef`, or `comp:deletion`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.6.2.)

comp-21003 ☑ A **ReplacedElement** object must have a value for the required attribute `comp:submodelRef`, and one, and only one, of the following attributes: `comp:portRef`, `comp:idRef`, `comp:unitRef`, `comp:metaIdRef`, or `comp:deletion`. It may also have a value for the optional attribute `comp:conversionFactor`. No other attributes from the Hierarchical Model Composition namespace are permitted on a **ReplacedElement** object. (References: SBML L3 Hierarchical Composition Package V1 Section 3.6.2.)

comp-21004 ☑ The value of a `comp:submodelRef` attribute on a **ReplacedElement** object must be the identifier of a **Submodel** present in **ReplacedElement** object's parent **Model**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.6.2.)

comp-21005 ☑ The value of a `comp:deletion` attribute on a **ReplacedElement** object must be the identifier of a **Deletion** present in the **ReplacedElement** object's parent **Model**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.6.2.)

comp-21006 ☑ The value of a `conversionFactor` attribute on a **ReplacedElement** object must be the identifier of a **Parameter** present in the **ReplacedElement** object's parent **Model**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.6.2.)

comp-21007 ☑ The value of a `comp:submodelRef` attribute on a **ReplacedElement** object must always conform to the syntax of the SBML data type `SId`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.6.2.)

comp-21008 ☑ The value of a `comp:deletion` attribute on a **ReplacedElement** object must always conform to the syntax of the SBML data type `SId`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.6.2.)

comp-21009 ☑ The value of a `comp:conversionFactor` attribute on a **ReplacedElement** object must always conform to the syntax of the SBML data type `SId`. (References: SBML L3 Hierarchical Composition Package V1 Section 3.6.2.)

comp-21010 ☑ No two **ReplacedElement** objects in the same **Model** may reference the object unless that object is a **Deletion**. (References: SBML L3 Hierarchical Composition Package V1 Section 3.6.2.)

Acknowledgments

We thank Andrew Finney, Nicolas Le Novère, Stefan Hoops, Martin Ginkel, Wolfram Leibermeister, Ranjit Randhawa, Jonathan Webb, Frank Bergmann, Sarah Keating, Sven Sahle, James Schaff, Chris Myers, and the members of the *sbml-comp* Package Working Group for prior work, suggestions, and comments that helped shape the Hierarchical Model Composition package as you see it today.

We also thank with great enthusiasm the financial support of the National Institutes of Health (USA) under grant R01 GM070923 to M. Hucka.

References

Bergmann, F. and Sauro, H. M. (2006). Human-readable model definition language. Available via the World Wide Web at ttp://www.sys-bio.org/sbwWiki/_media/sbw/standards/2006-12-17_humanreadable_mdl.pdf.

Berners-Lee, T., Fielding, R., and Masinter, L. (2005). Uniform resource identifier (URI): Generic syntax. Available online via the World Wide Web at http://www.ietf.org/rfc/rfc3986.txt.

Biron, P. V. and Malhotra, A. (2000). XML Schema part 2: Datatypes (W3C candidate recommendation 24 October 2000). Available via the World Wide Web at http://www.w3.org/TR/xmlschema-2/.

Clark, J. and DeRose, S. (1999). XML Path Language (XPath) Version 1.0. Available via the World Wide Web at http://www.w3.org/TR/xpath/.

Eriksson, H.-E. and Penker, M. (1998). *UML Toolkit*. John Wiley & Sons, New York.

Fallside, D. C. (2000). XML Schema part 0: Primer (W3C candidate recommendation 24 October 2000). Available via the World Wide Web at http://www.w3.org/TR/xmlschema-0/.

Finney, A. (2000). Internal discussion document: Possible extensions to the Systems Biology Markup Language. Available via the World Wide Web at http://www.cds.caltech.edu/erato/extensions.html.

Finney, A. (2003a). Sbml sets concept. Available via the World Wide Web at http://sbml.org/Forums/index.php?t=msg&th=234&rid=0#msg_683.

Finney, A. (2003b). Systems biology markup language (SBML) level 3 proposal: Model composition features. Available via the World Wide Web at http://www.mpi-magdeburg.mpg.de/zlocal/martins/sbml-comp/model-composition.pdf.

Finney, A. (2003c). Systems biology markup language (SBML) level 3 proposal: Model composition features. Available via the World Wide Web at http://sbml.org/images/7/73/Model-composition.pdf.

Finney, A. (2004). SBML level 3: Proposals for advanced model representations. Available via the World Wide Web at http://sbml.org/images/9/9c/Ismb-2004-sbml-level-3-poster.pdf.

Finney, A. (2007). Andrew 2007 comments about model composition. Available via the World Wide Web at http://sbml.org/Andrew_2007_Comments_about_Model_Composition.

Ginkel, M. (2002). Modular SBML. Available via the World Wide Web at http://sbml.org/images/9/90/Sbml-modular.pdf.

Ginkel, M. (2003). SBML model composition special interest group. Available via the World Wide Web at http://www.mpi-magdeburg.mpg.de/zlocal/martins/sbml-comp.

Ginkel, M. (2007). Martin Goals. Available via the World Wide Web at http://sbml.org/Events/Other_Events/SBML_Composition_Workshop_2007/Martin_Goals.

Ginkel, M. and Stelling, J. (2001). XML notation for modularity. Available via the World Wide Web at http://sbml.org/images/7/73/Vortrag.pdf.

Harold, E. R. and Means, E. S. (2001). *XML in a Nutshell*. O'Reilly.

Hoops, S. (2007). Hierarchical model composition. Available via the World Wide Web at http://sbml.org/Community/Wiki/SBML_Level_3_Proposals/Hierarchical_Model_Composition_(Hoops_2007).

Hoops, S. (2008). Hierarchical model composition. Available via the World Wide Web at http://sbml.org/images/e/e9/HierarchicalModelGothenburg.pdf.

Hucka, M., Bergmann, F. T., Hoops, S., Keating, S. M., Sahle, S., Schaff, J. C., Smith, L. P., and Wilkinson, D. J. (2010). The Systems Biology Markup Language (SBML): Language Specification for Level 3 Version 1 Core. Available via the World Wide Web at http://sbml.org/Documents/Specifications.

Leibermeister, W. (2007). semanticsbml. Available via the World Wide Web at http://sbml.org/images/c/c1/SemanticSBML_SBMLcomposition.pdf.

Marsch, J., Orchard, D., and Veillard, D. (206). XML Inclusions (XInclude) Version 1.0 (Second Edition). Available via the World Wide Web at http://www.w3.org/TR/xinclude/.

Novère, N. L. (2003). On the relationships between L3 packages, including core. Available via the World Wide Web at http://www.sbml.org/Forums/index.php?t=tree&goto=6104.

Novère, N. L. (2007). Modularity in core. Available via the World Wide Web at http://sbml.org/Events/Other_Events/SBML_Composition_Workshop_2007/Modularity_In_Core.

Oestereich, B. (1999). *Developing Software with UML: Object-Oriented Analysis and Design in Practice*. Addison-Wesley.

Pemberton, S., Austin, D., Axelsson, J., Celik, T., Dominiak, D., Elenbaas, H., Epperson, B., Ishikawa, M., Matsui, S., McCarron, S., Navarro, Peruvemba, S., Relyea, R., Schnitzenbaumer, S., and Stark, P. (2002). XHTML 1.0 the Extensible HyperText Markup Language (second edition): W3C Recommendation 26 January 2000, revised 1 August 2002. Available via the World Wide Web at http://www.w3.org/TR/xhtml1/.

Randhawa, R. (2007). Model Composition for Macromolecular Regulatory Networks. Available via the World Wide Web at http://sbml.org/documents/proposals/CCB2007DemoPresentation.pdf.

SBML Team (2007). The 5th SBML Hackathon. Available via the World Wide Web at http://sbml.org/Events/Hackathons/The_5th_SBML_Hackathon.

SBML Team (2010). The SBML issue tracker. Available via the World Wide Web at http://sbml.org/issue-tracker.

Smith, L. P. (2010a). Hierarchical model composition. Available via the World Wide Web at http://sbml.org/images/b/b6/Smith-Hierarchical_Model_Composition-2010-05-03.pdf.

Smith, L. P. (2010b). Hierarchical model composition. Available via the World Wide Web at http://www.sbml.org/Forums/index.php?t=tree&goto=6124.

Smith, L. P. and Hucka, M. (2010). SBML Level 3 hierarchical model composition. Available via the World Wide Web at http://precedings.nature.com/documents/5133/version/1.

The SBML Editors (2010). SBML Editors' meeting minutes 2010-06-22. Available via the World Wide Web at http://sbml.org/Events/SBML_Editors'_Meetings/Minutes/2010-06-22.

Thompson, H. S., Beech, D., Maloney, M., and Mendelsohn, N. (2000). XML Schema part 1: Structures (W3C candidate recommendation 24 October 2000). Available online via the World Wide Web at the address http://www.w3.org/TR/xmlschema-1/.

Various (2007a). Issues to address. Available via the World Wide Web at http://sbml.org/Events/Other_Events/SBML_Composition_Workshop_2007/Issues_To_Address.

Various (2007b). Overloading semantics. Available via the World Wide Web at http://sbml.org/Events/Other_Events/SBML_Composition_Workshop_2007/Overloading_Semantics.

Various (2007). SBML Composition Workshop 2007. Available via the World Wide Web at http://sbml.org/Events/Other_Events/SBML_Composition_Workshop_2007.

W3C (2000). Naming and addressing: URIs, URLs, ... Available online via the World Wide Web at http://www.w3.org/Addressing/.

Webb, J. (2003). BioSpice MDL Model Composition and Libraries. Available via the World Wide Web at http://sbml.org/Forums/index.php?t=msg&th=67&rid=0#msg_111.