SBML Level 3 Package Specification

Spatial Processes

James C. Schaff
schaff@neuron.uchc.edu
Center for Cell Analysis and Modeling
University of Connecticut
Farmington, Connecticut, US

Anu Lakshminarayana
anu@uchc.edu)
Center for Cell Analysis and Modeling
University of Connecticut
Farmington, Connecticut, US

Lucian P. Smith
lpsmith@u.washington.edu
Computing and Mathematical Sciences
California Institute of Technology
Seattle, Washington, US

Version 1, Release 0.88 (Draft)

September 2014

DRAFT

This is a draft specification for the SBML Level 3 package called "spatial". It is not a normative document. Please send feedback to the Package Working Group mailing list at sbml-spatial@lists.sourceforge.net.

The latest release, past releases, and other materials related to this specification are available at http://sbml.org/Documents/Specifications/SBML_Level_3/Packages/spatial

This release of the specification is available at TBD

Contents

1 Introduction

A set of biochemical process in cellular physiology may be modeled using different choices of spatial and temporal scales depending on the questions to be addressed. SBML Level 3 Core has explicit support for multi-compartmental modeling where cellular organization is approximated by a set of compartments (e.g. membrane-bound organelles) containing well-stirred populations of molecules. However, the coupling between localized biochemical reactions and diffusive molecular transport within the constraints of cellular geometry often results in important nonuniform molecular distributions. While it is often possible to approximate the influence of spatial organization and localization within a compartmental model using altered parameters and additional species, it is sometimes simpler and always more mechanistic to directly model these spatial processes.

An increasing number of modeling and simulation tools include direct support for modeling with explicitly defined cellular geometry. These models generally include heterogeneous molecular distributions, diffusive transport, and spatially localized reactions. These spatial models generally belong to two different mathematical frameworks, stochastic (where each molecule is tracked in space and time) and deterministic (where time varying species concentration fields are described by partial differential equations).

There are a sufficient number of spatial modeling tools and spatial models to justify the effort of creating a spatial modeling extension of SBML. All such models must describe the cellular geometry, map molecular species to spatial locations, map reactions to spatial locations, and specify molecular transport within geometric compartments and at boundaries of these compartments.

It is the purpose of this SBML Level 3 extension to define a common representation for cellular geometry, spatial mappings of species and reactions, and explicit species transport.

1.1 Proposal corresponding to this package specification

This specification for Spatial in SBML Level 3 Version 1 is based on the proposal located at the following URL:

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

The tracking number in the SBML issue tracking system (SBML Team, 2010) for Spatial package activities is 188 (`http://sourceforge.net/p/sbml/sbml-specifications/188/`).

1.2 Package dependencies

The Spatial package has no dependencies on other SBML Level 3 packages.

1.3 Document conventions

UML 1.0 (Unified Modeling Language; Eriksson and Penker 1998; Oestereich 1999) notation is used in this document to define the constructs provided by this package. Colors in the diagrams carry the following additional information for the benefit of those viewing the document on media that can display color:

- *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.

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

AbstractClass: Abstract classes are never instantiated directly, but rather serve as parents of other 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 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 tokens *other* than SBML class names, are printed in an upright typewriter typeface.

`[elementName]`: In some cases, an element may contain a child of any class inheriting from an abstract base class. In this case, the name of the element is indicated by giving the abstract base class name in brackets, meaning that the actual name of the element is the de-capitalized form of whichever subclass is used.

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

2 Background and context

2.1 Problems with current SBML approaches

There is no standard way of specifying spatial models in SBML short of introducing an explicit spatial discretization in the form of a large number of compartments with duplicate species and reactions and additional reactions for coupling due to transport. This approach hard-codes the numerical methods which destroys portability and is not practical beyond a few compartments. Tools have been forced to resort to proprietary extensions (e.g. MesoRD custom annotations) to encode geometry.

2.2 Past work on this problem or similar topics

There are many standards for the exchange of geometric information of engineered parts in Computer Aided Design and Manufacturing. These formats are designed for geometric shapes which are directly specified by a designer rather than the data driven, freeform biological structures encountered in cell biology.

There also exist standards for the representation of unstructured computational meshes that can encode these freeform biological structures more readily. However, it is important to note that while a computational mesh necessarily encodes an approximation to the shapes of geometric objects, the particular form will be algorithm dependent.

To ensure model interoperability, we must encode the geometric shapes in a way that is independent of the numerical methods and even the mathematical framework. The representation of a spatial model within SBML should be largely invariant of the particular encoding of the geometry definition within that model. For example, a spatial model represented in SBML that encodes geometry as a set of geometric primitives (e.g. spheres, cylinders) should be easily portable to a tool that only supports polygonal surface tessellation. It is expected that a geometry translation library will be very useful for interoperability the same way that libSBML greatly improved model interchange by solving similar implementation problems in a standard way.

2.3 Prior work

The first version of the Spatial proposal was written [fill in history]

☞ Lucian: We probably do need something here.

3 Package syntax and semantics

This section contains a definition of the syntax and semantics of the Spatial package for SBML Level 3 Version 1 Core. The Spatial package involves several new object classes, and extends the existing **Model**, **Compartment**, **Species**, **Reaction**, and **Parameter** object class. Section 4 on page 39 contains complete examples of using the constructs in SBML models.

☞ Lucian: Periodically when I have comments, I'll put them in sections that look like this–in red, with the pointy-hand icon off to the side. They tend to be design questions I had when creating this document for parts I thought were not clear, or are suggestions for changes that could be made.

3.1 Overview of spatial extension

The SBML **Compartment**, **Reaction** and **Species**, and molecular transport mechanisms (**DiffusionCoefficient**, **AdvectionCoefficient**, **BoundaryCondition**) are mapped to geometric domains to describe spatial models within SBML. The primary mechanism to accomplish this mapping is to simply map **Compartments** to collections of geometric Domains called **DomainTypes**. Each **Domain** is a contiguous patch of volumetric space or a contiguous surface patch that is ultimately described by a single system of equations (whichever mathematical framework is used). In analogy with initial conditions, the mathematical system defined within a domain often needs a definition of what happens at the domain boundary (e.g. boundary conditions) to complete the specification. Because of this, the boundaries between adjacent domains need to be identified so that appropriate boundary conditions can be specified. For compactness of representation, rather than map to each individual **Domain**, **Compartments** are mapped to **DomainTypes**, along with the corresponding **Species** and **Reactions** (with the new compartment attribute).

3.1.1 Geometry

The **Geometry** object within a model is completely modular and does not reference the rest of the model, promoting reuse of the same geometry in different models. The geometry separately defines a coordinate system, a list of domain types, a list of domains and their adjacency relationships, and a list of alternate geometric representations.

3.1.2 Alternative GeometryDefinitions

Modeling and simulation tools will each natively support some subset (often just one) of the possible ***GeometryDefinitions*** (analytic, sampled field, constructive solid geometry, and parametric shapes). Interoperability will be enhanced if tools write as many geometry definitions as they are able. Upon reading the model, a tool will typically choose the most convenient geometry definition, i.e. the one that it natively supports. If a tool does not edit the geometry, it has the ability to preserve the alternate representations during model editing (because the mapping of the model to the geometry is not stored in the geometry).

There are two general classes of geometric representation specification: those that explicitly specify surfaces and those that implicitly specify surfaces. For example, a level set is a field where a specific isosurface of the field specifies a geometric surface. A geometry described using constructive solid geometry of geometric primitives (e.g. spheres, cylinders) specifies directly which points are "inside" an object. Alternatively, explicit surface representations explicitly declare the set of points belonging to surfaces (e.g. polygonal tessellations).

3.2 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 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 Spatial package for SBML Level 3 Version 1 Core:

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

In addition, SBML documents using a given package must indicate whether the package can be used to change the mathematical interpretation of a model. This is done using the attribute `required` on the `<sbml>` element in the SBML document. For the Spatial package, the value of this attribute must be "`true`", because the use of the Spatial package can change the mathematical meaning of a model.

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

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

3.3 Primitive data types

The Spatial package uses a number of the primitive data types described in Section 3.1 of the SBML Level 3 Version 1 Core specification, and adds several additional primitive types described below.

3.3.1 *Type* SpId

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

3.3.2 *Type* SpIdRef

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

3.3.3 *Type* BoundaryConditionKind

The `BoundaryConditionKind` primitive data type is used in the definition of the **BoundaryCondition** class. The type `BoundaryConditionKind` is derived from type `string` and its values are restricted to being one of the following possibilities: "`Robin_valueCoefficient`", "`Robin_inwardNormalGradientCoefficient`", "`Robin_sum`", "`Neumann`", and "`Dirichlet`". Attributes of type `BoundaryConditionKind` cannot take on any other values. The meaning of these values is discussed in the context of the **BoundaryCondition** class's definition in Section 3.12 on page 16.

3.3.4 *Type* CoordinateKind

The `CoordinateKind` primitive data type is used in the definition of the **CoordinateComponent** class. `CoordinateKind` is derived from type `string` and its values are restricted to being one of the following possibilities: "`cartesianX`", "`cartesianY`", and "`cartesianZ`". Attributes of type `CoordinateKind` cannot take on any other values. The meaning of these values is discussed in the context of the **CoordinateComponent** class's definition in Section 3.15 on page 19.

Other `CoordinateKind` types are held in reserve for future versions of this specification, and may include "`spherical-Radius`", "`sphericalAzimuth`", "`sphericalElevation`", "`cylindricalRadius`", "`cylindricalAzimuth`", "`cylin-dricalHeight`", "`polarRadius`", and "`polarAzimuth`".

3.3.5 *Type* `DataKind`

The `DataKind` primitive data type is used in the definition of the **SampledField** class. `DataKind` is derived from type `string` and its values are restricted to being one of the following possibilities: "`double`", "`float`", "`uint8`", "`uint16`", and "`uint32`". Attributes of type `DataKind` cannot take on any other values. The meaning of these values is discussed in the context of the **SampledField** class's definition in Section 3.46 on page 37.

3.3.6 *Type* `DiffusionKind`

The `DiffusionKind` primitive data type is used in the definition of the **DiffusionCoefficient** class. `DiffusionKind` is derived from type `string` and its values are restricted to being one of the following possibilities: "`isotropic`", "`anisotropic`", and "`tensor`". Attributes of type `DiffusionKind` cannot take on any other values. The meaning of these values is discussed in the context of the **DiffusionCoefficient** class's definition in Section 3.10 on page 15.

3.3.7 *Type* `CompressionKind`

The `CompressionKind` primitive data type is used in the definition of the **SampledField** class. `CompressionKind` is derived from type `string` and its values are restricted to being one of the following possibilities: "`uncompressed`" and "`deflated`". Attributes of type `CompressionKind` cannot take on any other values. The meaning of these values is discussed in the context of the **SampledField** class's definition in Section 3.46 on page 37.

3.3.8 *Type* `FunctionKind`

The `FunctionKind` primitive data type is used in the definition of the **AnalyticVolume** class. The type `FunctionKind` is derived from type `string` and its values are restricted to being one of the single possibility "`layered`". Attributes of type `FunctionKind` cannot take on any other values. The meaning of these values is discussed in the context of the **AnalyticVolume** class's definition in Section 3.23 on page 24.

3.3.9 *Type* `GeometryKind`

The `GeometryKind` primitive data type is used in the definition of the **Geometry** class. `GeometryKind` is derived from type `string` and its values are restricted to being the single possibility "`cartesian`". Other `GeometryKind` types are held in reserve for future versions of this specification, and may include "`cylindrical`", "`spherical`", and "`polar`". Attributes of type `GeometryKind` cannot take on any other values. The meaning of these values is discussed in the context of the **Geometry** class's definition in Section 3.14 on page 18.

3.3.10 *Type* `InterpolationKind`

The `InterpolationKind` primitive data type is used in the definition of the **SampledField** class. `InterpolationKind` is derived from type `string` and its values are restricted to being one of the following possibilities: "`nearestNeighbor`" and "`linear`". Attributes of type `InterpolationKind` cannot take on any other values. The meaning of these values is discussed in the context of the **SampledField** class's definition in Section 3.46 on page 37.

3.3.11 *Type* `PolygonKind`

The `PolygonKind` primitive data type is used in the definition of the **ParametricObject** class. `PolygonKind` is derived from type `string` and its values are restricted to being one of the following possibilities: "`triangle`" and "`quadrilateral`". Attributes of type `PolygonKind` cannot take on any other values. The meaning of these values is discussed in the context of the **ParametricObject** class's definition in Section 3.41 on page 34.

3.3.12 *Type* `PrimitiveKind`

The `PrimitiveKind` primitive data type is used in the definition of the **CSGPrimitive** class. `InterpolationKind` is derived from type `string` and its values are restricted to being one of the following possibilities: "`sphere`", "`cube`", "`cylinder`", "`cone`", "`circle`", "`square`", and "`rightTriangle`". Attributes of type `PrimitiveKind` cannot take on any other values. The meaning of these values is discussed in the context of the **CSGPrimitive** class's definition in

Section 3.46 on page 37.

3.3.13 *Type* `SetOperation`

The `SetOperation` primitive data type is used in the definition of the **CSGSetOperator** class. The type `SetOperation` is derived from type `string` and its values are restricted to being one of the following possibilities: "`union`", "`intersection`", and "`difference`". Attributes of type `SetOperation` cannot take on any other values. The meaning of these values is discussed in the context of the **CSGSetOperator** class's definition in Section 3.32 on page 30.

3.3.14 *Type* `doubleArray`

The `doubleArray` primitive data type is a space-delimited list of `double` values in a single string.

3.3.15 *Type* `arrayData`

The `arrayData` primitive data type is used in the definition of the **SampledField** class. It consists of a possibly-compressed whitespace-delimited list of numerical values in a single string. The meaning and possible content of these values is discussed in the context of the **SampledField** class's definition in Section 3.46 on page 37.

3.4 The extended Model object

The **Model** object is extended in the spatial package to contain a new **Geometry** child, as seen in Figure 1. The **Geometry** element is contained in the Model element in the 'spatial' namespace. In order to specify a spatial geometry, some of the existing SBML elements need to be extended (**Compartment**, **Species**, **Parameter**, and **Reaction**). These extensions to the SBML elements are discussed in the sections that follow.

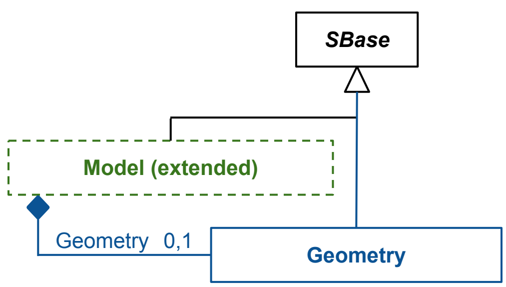

Figure 1: *The definition of the extended **Model** object from the Spatial package. The **Geometry** object and its children are defined in their own sections.*

3.5 The extended Compartment object

The **Compartment** in the SBML core is extended while defining a spatial model. An SBML model with spatial geometry defines domain types (classes of domains that are anatomically and physiologically similar). These domain types need to be mapped to a compartment in the SBML model. **Compartments** are extended to define **CompartmentMappings** that map compartments to **DomainTypes** such that each corresponding **DomainType** is assigned the same biological and mathematical function. Within SBML L3 Core, the compartment Sid refers to the size of that compartment and is specified by the size attribute or may be set by a rule. For spatial models, the compartment size is calculated as the product of the unit size specified in the compartment mapping and the size of the current domain. The definition for the extension of the Compartment element is shown in Figure 2 on the following page.

The **Compartment** element has an optional **CompartmentMapping** child which indicates the **DomainType** to which the **Compartment** is mapped. If there is no **CompartmentMapping** for a **Compartment** in a spatial model, then that

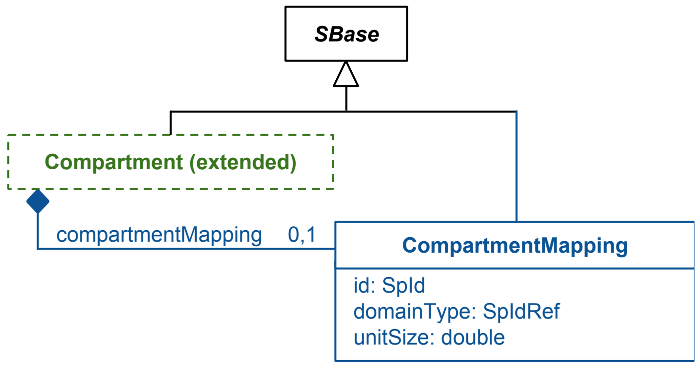

Figure 2: *The definition of the extension to the **Compartment** element, and the definition of the **CompartmentMapping** class. The SBML core attributes of **Compartment** are not displayed.*

Compartment is excluded from the spatial version of the model. In the same way, if a **DomainType** is not mapped to one or more **Compartments**, then the corresponding **Domains** in the geometry have no assigned function.

3.6 The CompartmentMapping class

Each **Compartment** in a model that defines a spatial geometry may contain an optional **CompartmentMapping**. A **CompartmentMapping** is defined as part of the model rather than part of the geometry so that the geometry is modular and may be readily shared between models and reused. A **CompartmentMapping** maps a **Compartment** defined in the model to a **DomainType** defined in the geometry such that each corresponding **DomainType** is assigned the same biological and mathematical function described by the set of **Compartments** that are mapped to that **DomainType**.

This mapping need not be one-to-one. In fact, it is common to map er-lumen, er-membrane, and cytosol to the same cell interior volume or 3D **DomainType**. The `unitSize` attribute specifies the relative quantity of each **Compartment** that is mapped to the **DomainType**.

3.6.1 The `id` attribute

The `id` attribute is a mandatory attribute of type `SpId` that is used to uniquely identify a **CompartmentMapping** in the model. All identifiers of type `SpId` must be unique within the **Geometry**. The mathematical value of a **CompartmentMapping** is its `unitSize` attribute, and can be bound to a **Parameter** by using a **SpatialSymbolReference**.

3.6.2 The `domainType` attribute

The mandatory `domainType` attribute is of type `SpIdRef` that indicates a **DomainType** defined in the **Geometry** element.

3.6.3 The `unitSize` attribute

The `unitSize` attribute is of type `double` and represents the relative size of the **Compartment** with respect to the size of the **Domains** to which they are mapped. Thus for any infinitesimal subset of the **Domain** with size S, there exists an amount of Compartment_i of size $(S{*}\text{unitSize}_i)$ for i=1..N compartments mapped to that **DomainType**. For example, a 3D **Compartment** (and **DomainType**) which is mapped to a 3D **DomainType** has a `unitSize` which is a volume fraction of dimensionless unit. The total set of all such volume fractions mapped to a particular **DomainType** will typically sum to one.

If the `spatialDimensions` attribute of the parent **Compartment** is different than the `spatialDimensions` attribute of referenced **DomainType**, the `unitSize` attribute is a conversion factor between the two. The most common example of this would be a 2D **Compartment** being mapped to a 3D **DomainType**, such as an ER-membrane being mapped to a volumetric cell interior. In this case, the `unitSize` is a surface-to-volume ratio.

If connected to a **Parameter** via a **SpatialSymbolReference**, an **InitialAssignment** may override the value of the `unitSize` attribute. It is theoretically possible to have this value change in time through the use of a **Rule** or **Event**, but some (if not all) software tools may not support this setup. If the value is set to change, and the dimensionality of the parent **Compartment** and referenced **DomainType** is the same, the other **CompartmentMapping** elements for the same **DomainType** will typically change in concert, so that they continue to sum to one. Also note that **Species** amounts are preserved in SBML, so a changing `unitSize` may induce a corresponding change in any correlated **Species** concentration.

Any bound **Parameter**'s units should be equivalent to the units of the parent **Compartment** divided by the units of the referenced **DomainType**.

3.7 The extended Species object

The SBML core **Species** is extended when a spatial geometry is defined in the model with the addition of a single new required boolean "`isSpatial`" attribute. The extension to the **Species** element is shown in Figure 3.

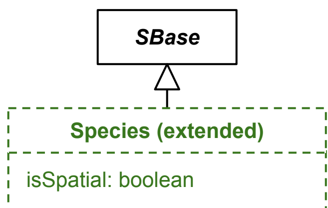

Figure 3: *The extension to the **Species** element. The attributes of **Species** from SBML Level 3 Version 1 Core are not displayed.*

3.7.1 The `isSpatial` *attribute*

The `isSpatial` attribute is of data type boolean. If it is set to true, the **Species** is spatially distributed in a possibly nonhomogeneous manner within the **Domains** of the same type as the mapped **DomainType**.

For continuous deterministic models (described by partial differential equations), a spatial **Species** will result in a concentration field described by a partial differential equation which incorporates contributions from **Reactions**, diffusion (**DiffusionCoefficient**) and advection (**AdvectionCoefficient**) and are subject to boundary conditions (**BoundaryCondition**) and initial conditions (**InitialAssignment** and **Rule**). All of these quantities can be explicit functions of the spatial coordinates as well as spatial and nonspatial **Parameters** and **Species**.

For stochastic models, the **Species** is represented as a collection of particles that are distributed throughout the **Domains** and are subject to reactions, diffusion and advection. Simulation algorithms either track individual particles (e.g. Particle-based methods) or use spatial discretization to track a large number of well stirred pools (e.g. Next-Subvolume Method).

The `compartment` of any **Species** set `isSpatial` = "`true`" must have a child **CompartmentMapping**: if it did not, its compartment would not actually be a part of the spatial model.

3.8 The extended Parameter object

When an SBML model defines a spatial geometry, the SBML core **Parameter** is used to define the diffusion coefficient, transport velocity (advection) and boundary conditions for species and the coordinate components defined in the **Geometry**. One **Parameter** is created for each quantity, by adding a child **DiffusionCoefficient**, **AdvectionCoefficient**, or **BoundaryCondition**. Conversely, some elements defined in the spatial package may need to be referenced by mathematics in core constructs, or even have their value set by core constructs such as **InitialAssignment** or

Rule. These spatial elements can be semantically linked to a **Parameter** by giving it a child **SpatialSymbolReference** pointing to that element.

A **Parameter** that has been extended for the Spatial package can have only one of the above listed objects. For example, if a **Parameter** is extended to represent the diffusion coefficient of a species, the existing attributes of the **Parameter** (id, name, value, units, constant) are defined according to SBML core specifications, along with a **DiffusionCoefficient** child that contains the information about the species it represents. Figure 4 represents the extension to the **Parameter** element.

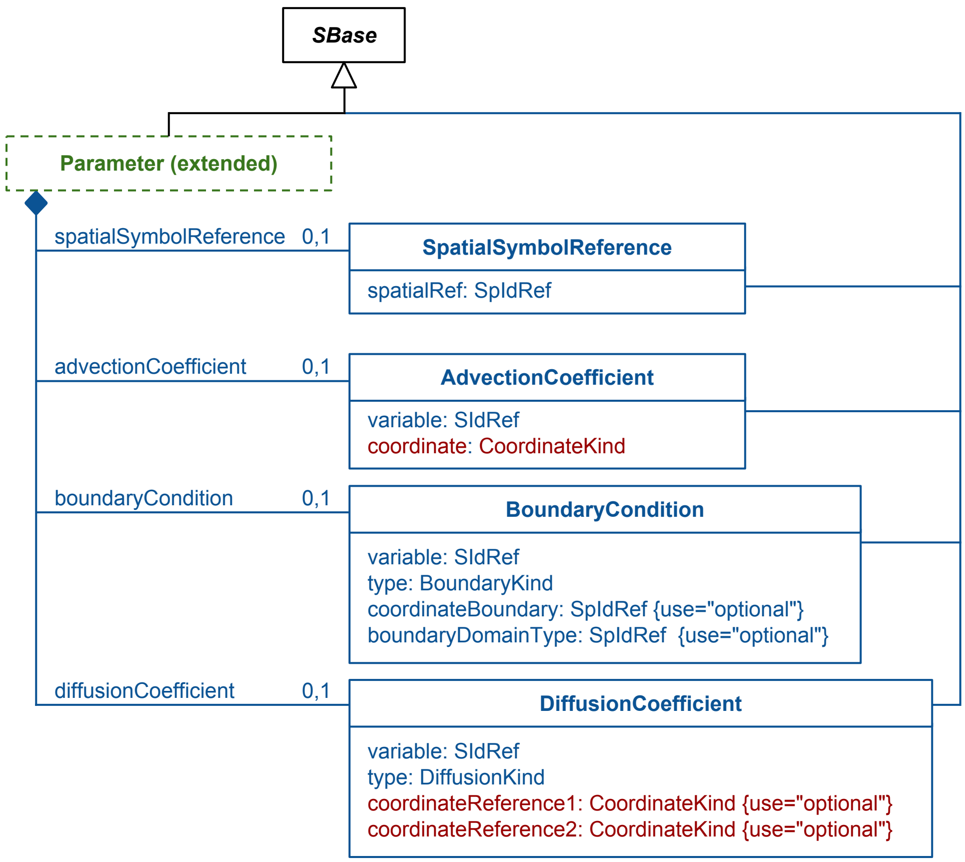

Figure 4: *The **Parameter** element extension for spatial package. The SBML Level 3 Version 1 Core attributes for **Parameter** are not displayed in this figure.*

3.9 The SpatialSymbolReference class

A **Parameter** is extended with a **SpatialSymbolReference** element, when a symbol from the defined spatial geometry (`id` of any element contained in **Geometry**) is required to be used in the SBML core model. Typically, the **SpatialSymbolReference** is used to represent the coordinate components defined in the **Geometry**'s listOfCoordinateComponents. For example, if the **Geometry** is defined in a 2-dimensional Cartesian coordinate system with X and Y defined as coordinate components, two **Parameters** (one each for **CoordinateComponents** X and Y) are created in the model. The value of the parameter is not required to be set. For each of these parameters, a **SpatialSymbolReference** object is created.

3.9.1 The `spatialRef` *attribute*

The `spatialRef` attribute of **SpatialSymbolReference**, is of type `SpIdRef` and refers to the `SpId` of any element defined in the **Geometry** of the model.

3.10 The DiffusionCoefficient class

When a species in a spatial model has a diffusion rate constant, a **Parameter** for this diffusion constant is created in the SBML model with a **DiffusionCoefficient** child, which is used to identify the **Species** whose diffusion rate the **Parameter** represents. The diffusion coefficient can then be set like any other variable: its initial value can be set using the **Parameter**'s `value` attribute or through an **InitialAssignment**, and if the diffusion coefficient changes in time, this can be defined with a **Rule** or **Event**. If set, the units of this **Parameter** should be $\text{length}^2/\text{time}$. If left unset, the **DiffusionCoefficient** will inherit the model units of $\text{length}^2/\text{time}$ (typically cm^2s^{-1} or um^2s^{-1}).

It is possible to define both diffusion and advection for the same **Species**.

3.10.1 The `variable` *attribute*

The required `variable` attribute of **DiffusionCoefficient** is of type `SIdRef` and is the `SId` of the **Species** or **Parameter** in the model whose diffusion coefficient is being set.

3.10.2 The `type` *and* `coordinateReference` *attributes*

The required `type` attribute of **DiffusionCoefficient** is of type `DiffusionKind` and indicates whether the diffusion coefficient is "`isotropic`" (i.e. applies equally in all dimensions/directions), "`anisotropic`" (i.e. applies only for a single coordinate), or "`tensor`" (i.e. applies only for a particular pair of coordinates). Coefficients of type "`isotropic`" may not have any `coordinateReference` attributes defined, since diffusion is defined for all axes. Coefficients of type "`anisotropic`" must define the `coordinateReference1` attribute and not the `coordinateReference2` attribute, and applies in the direction of that axis. Coefficients of type "`tensor`" must define both the attributes `coordinateReference1` and `coordinateReference2`, defining diffusion in relation to the direction due to a gradient in the diagonal term of the diffusion tensor for the two coordinates. In no case may `coordinateReference2` be defined but not `coordinateReference1`.

3.10.3 DiffusionCoefficient uniqueness

Only one **DiffusionCoefficient** may be defined per **Species** per axis or pair of valid axes in the **Compartment** in which it resides. Since isotropic diffusion is defined for all axes at once, this means that if an isotropic **DiffusionCoefficient** is defined for a **Species**, it may have no other diffusuion coefficients.

3.11 The AdvectionCoefficient class

The **AdvectionCoefficient** is the extension to **Parameter** in SBML core that is used to represent transport velocity of a species, if it exists. The transport velocity for the species is defined in a manner similar to the diffusion constant with a unit of length/time (regardless of the units of the corresponding **Species**' "`compartment`" attribute). A **Parameter** is created in SBML code for the velocity with an **AdvectionCoefficient** child to identify the **Species** whose velocity is represented by the **Parameter**; its initial value is set either through the `value` attribute or an **InitialAssignment**. If the advection coefficient changes in time or space, this can be modeled with a **Rule** or **Event**.

If defined, the units of the parent **Parameter** should be in length/time; if not defined, it inherits from the model-wide units of length divided by the model-wide units of time.

It is possible to define both diffusion and advection for the same **Species**.

3.11.1 The variable attribute

The `variable` attribute of **AdvectionCoefficient** is of type `SIdRef` and is the SId of the **Species** or **Parameter** in the model whose advection coefficient (transport velocity) is being set.

3.11.2 The coordinate attribute

The `coordinate` is of type `CoordinateKind` and represents the coordinate component of the velocity. For example, if the **Geometry** is defined in the Cartesian coordinate system and is 2-dimensional, the species can have velocity terms for both X and Y. If the **Parameter** represents the transport velocity of the species in the X-coordinate, the `coordinate` attribute will take a value of "`cartesianX`", and if it represents the velocity in the Y-coordinate, the attribute will take a value of "`cartesianY`". Only one **AdvectionCoefficient** may be defined per **Species** per valid `coordinate`.

3.12 The BoundaryCondition class

A **Species** in a spatial model that has a diffusion rate or an advection velocity needs to have specified boundary conditions. A boundary condition is either the concentration of the species or the flux density of the species at a boundary. The boundary refers to either an internal membrane boundary or a face of the box defined by the minimum and maximum coordinates of the geometry (the geometries bounding box).

When creating a spatial SBML model, species boundary conditions are created as parameters, one for each boundary condition, by adding a child **BoundaryCondition** that points to the corresponding **Species** and boundary, depending on the coordinate system.

For Cartesian Geometries, there are two boundaries for every axis being modeled. For example, in a 2D cartesian geometry for the external boundaries, there could be up to four parameters or parameter sets created for each spatial **Species** whose **Compartments** abut the minimum and maximums of the X and Y axes).

For internal boundaries, one may either define a **BoundaryCondition** for a **Species** at that boundary, or one may define one or more transport reactions that describe how the physical entities that **Species** represent are moved (or converted) from one side of the boundary to the other. One may not define both a **BoundaryCondition** and a **Reaction** that describes the same phenomenon, as this would result in the equivalent of an overdetermined system, not dissimilar from the reason that the change in a **Species** may not be defined by both a **Reaction** and a **RateRule**. A **Species** set `boundaryCondition` = "`true`" may have a defined **BoundaryCondition** and also appear in a transport **Reaction**, but its change in time and space will only be determined by the **BoundaryCondition**.

If neither a **BoundaryCondition** nor a **Reaction** is defined for a particular **Species**/boundary pair, the flux of that **Species** at that boundary is zero.

The **Parameter**'s value is set either through the `value` attribute or an **InitialAssignment**. If the boundary condition changes in time, it can be set with a **Rule** or **Event**. If set, the **Parameter** unit must be equal to the appropriate unit for its `type` (see below). Only one **BoundaryCondition** may be defined per **Species** per boundary (regardless of type).

3.12.1 The variable attribute

The `variable` attribute of **BoundaryCondition** is of type `SIdRef` and is the SId of the **Species** or **Parameter** in the model whose boundary condition is being set.

3.12.2 The type attribute

The `type` attribute is of type `BoundaryConditionKind` and indicates the type of boundary condition. The boundary condition types come in three groups: for Neumann boundaries, "`Neumann`" (the inward normal flux) is used. For Dirichlet boundaries, "`Dirichlet`" (the value) is used. For Robin boundaries, three different Parameters must be defined, with **BoundaryCondition** elements of type "`Robin_valueCoefficient`", "`Robin_inwardNormalGradientCoefficient`", and “Robin_sum”.

The unit of the boundary condition is determined by the type, and the unit for density and velocity. For “Dirichlet”, the unit would be the unit of concentration. For “Neumann”, the unit would be concentration*length/time. For Robin boundaries, for a variable “u” with an inward pointing normal vector “n”, Robin value coefficient “a”, Robin inward normal gradient coefficient “b”, and Robin sum “g”, the condition is defined by the equation “a*u + b*inner_product(grad(u),n) = g”, with appropriate units. The suggested set of units that satisfy this condition is to have units of nondimensional for “a”, 1/length for “b”, and the same units as the referenced variable for “c”.

3.12.3 The `coordinateBoundary` *attribute*

The `coordinateBoundary` attribute is of type `SpIdRef` and refers to the `SpId` of either the `boundaryMin` or `boundaryMax` object of the **CoordinateComponent** defined in **Geometry**. This `SpId` indicates the boundary condition (minimum or maximum) in the **CoordinateComponent**. A **Parameter** that is extended with a **BoundaryCondition** object can only define the `coordinateBoundary` attribute or the `boundaryDomainType` attribute, but not both.

3.12.4 The `boundaryDomainType` *attribute*

The `boundaryDomainType` attribute is of type `SpIdRef` and refers to the `SpId` of the **DomainType** of the location of the species whose boundary condition is being defined. A **Parameter** that is extended with a **BoundaryCondition** object can only define the `coordinateBoundary` attribute or the `boundaryDomainType` attribute, but not both.

3.13 The extended Reaction object

The SBML core **Reaction** is extended when a spatial geometry is defined in the model with the addition of a single new required boolean `isLocal` attribute. Figure 5 displays the definition of the extension of the **Reaction** element.

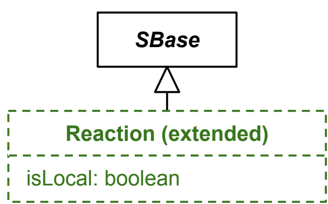

Figure 5: *The extension to the* ***Reaction*** *element. The SBML Level 3 Version 1 Core atrributes and children for* ***Reaction*** *are not displayed in the figure.*

3.13.1 The `isLocal` *attribute*

The `isLocal` attribute for a **Reaction** is of type `Boolean`. The attribute is set to true if the reaction is to be considered a local description of the reaction in terms of concentration/time defined at each point in space rather than substance/time over an entire **Compartment** or "pool". Note that this means that the units of the **KineticLaw** are different depending on whether the **Reaction** is local or not.

If a **Reaction** is defined to be a local (having an `isLocal` value of “`true`”), the value of the `compartment` attribute of the **Reaction** must be defined. This is because the interpretation of the **Reaction** is very different if the reaction takes place at the boundary of the **Compartment** of the **Species** (where the reaction rate units are flux densities) than if it takes place within the interior of that **Compartment** (where the reaction rate units are concentration/time define throughout the volume). The first will give you gradients in the solution, while the other will not.

If the referenced **Species** come from multiple compartments, the `compartment` of the **Reaction** must be a **Compartment** that makes physical sense for the individual **Species** to meet.

3.14 The Geometry class

A single geometry must be defined within the model if the spatial extension is to be used. Figure 6 shows the definition of the **Geometry** element.

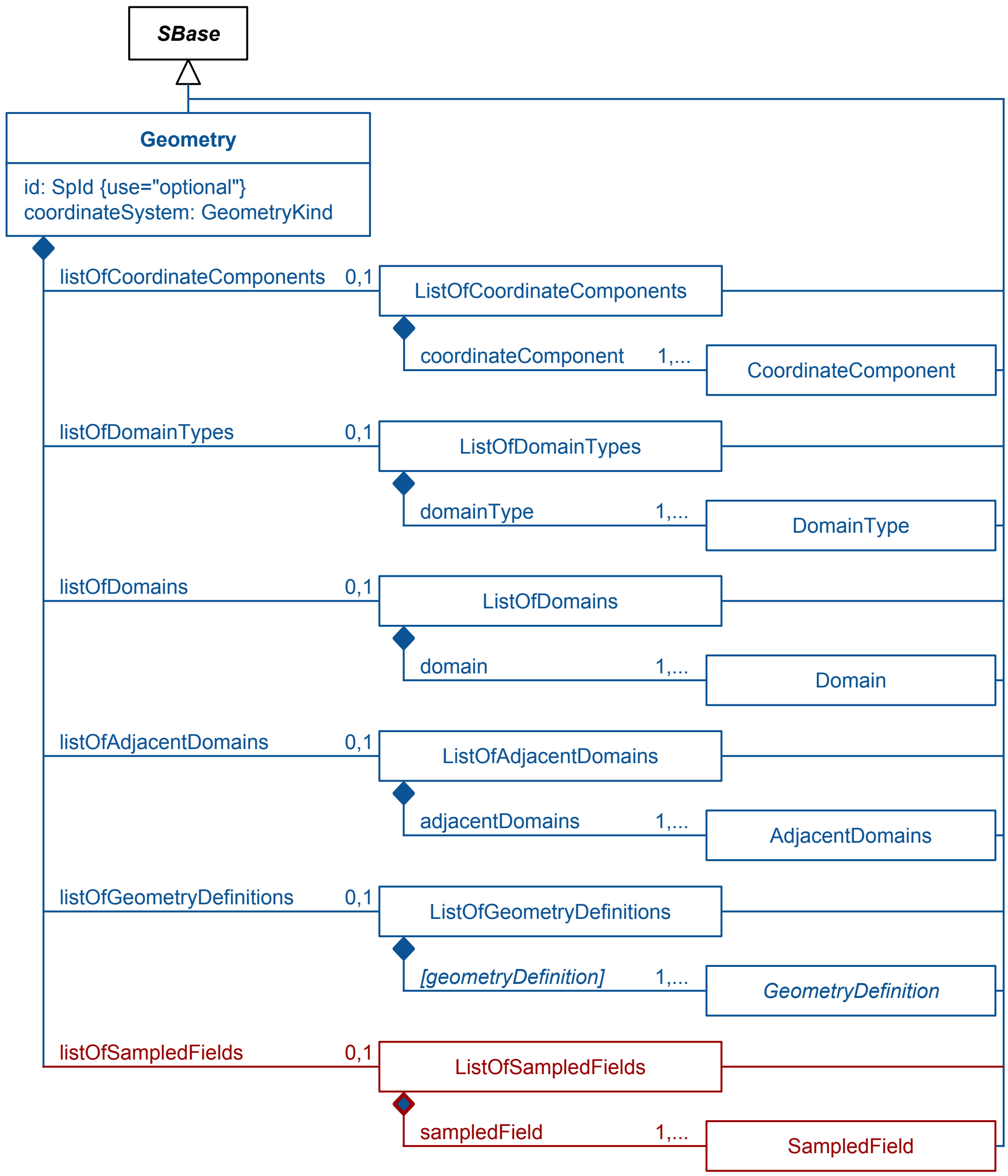

Figure 6: *The definition of the* ***Geometry***, ***ListOfCoordinateComponents***, ***ListOfDomainTypes***, ***ListOfDomains***, ***ListOfAdjacentDomains***, *and* ***ListOfGeometryDefinitions*** *classes from the Spatial package. The various children of the ListOf-classes are defined in their own sections.*

3.14.1 The `id` attribute

The `id` attribute is of type `SpId`, uniquely identifies the **Geometry** element, and is optional. It has no mathematical meaning, and cannot be connected to a **Parameter** via a **SpatialSymbolReference** element.

3.14.2 The `coordinateSystem` *attribute*

The `coordinateSystem` attribute is a required attribute and is of type `GeometryKind`. It represents the coordinate system used by the **Geometry**. A value of "`cartesian`" indicates that the geometry is a cartesian coordinate system, with the coordinate components corresponding to the x, y, and z components of that system (which could be 1-, 2-, or 3-dimensional). This is the only coordinate system defined in this version of the specification–in the future, if necessary, "`cylendrical`", "`spherical`", and "`polar`" may be added as possibilities, along with n-dimensional cartesian modeling, should there be interest in the modeling community to exchange these types of models.

3.14.3 The listOf container classes

The **Geometry** has listOfCoordinateComponents, listOfDomainTypes, listOfDomains, and listOfAdjacentDomains, and listOfGeometryDefinitions that help define the geometry. The **ListOfCoordinateComponents** is a list of **CoordinateComponent** objects, the **ListOfDomainTypes** is a list of **DomainType** objects, the **ListOfDomains** is a list of **Domain** objects, **ListOfAdjacentDomains** is a list of **AdjacentDomains** objects, and the **ListOfGeometryDefinitions** is a list of alternative ***GeometryDefinitions*** (**ParametricGeometry**,**CSGeometry**, **SampledFieldGeometry**, **AnalyticGeometry**). None of these lists are technically required, but, if present, none of them may be empty.

Note that the children of the **ListOfGeometryDefinitions** object are not called `geometryDefinition` but rather take the name of the derived class, decapitalized. Thus, they may be called `parametricGeometry`, `sampledFieldGeometry`, `csGeometry`, or `analyticGeometry`.

3.15 The CoordinateComponent class

A **CoordinateComponent** object explicitly defines a coordinate component of the coordinate axes and gives them names, units, and formally associates them with a coordinate system. The **CoordinateComponent** also defines the minimum and maximum values of the coordinate axis it represents. The definition of **CoordinateComponent** is shown in Figure 7.

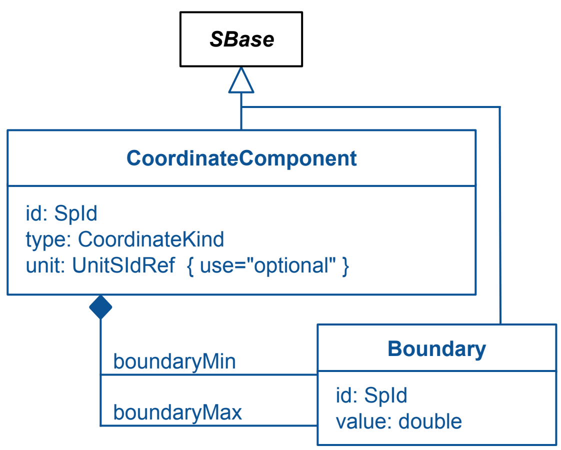

Figure 7: *The* ***CoordinateComponent*** *object definition. One or more instances of* ***CoordinateComponent*** *objects in a* ***ListOfCoordinateComponents*** *can be present in* ***Geometry****.*

3.15.1 The `id` *attribute*

A **CoordinateComponent** is identified with the `id` attribute which is of type `SpId`. The mathematical value of a **CoordinateComponent** is its coordinate value `unitSize` attribute, and can be bound to a **Parameter** by using a **SpatialSymbolReference**.

Because a **CoordinateComponent** represents an entire axis, it is not appropriate, should it be connected to a

Parameter via a **SpatialSymbolReference**, for that **Parameter** to be set via an **InitialAssignment** or **Rule**. Rather, it is treated like the SBML core `csymbol` "`time`", and can be used as an independent variable in other calculations.

3.15.2 The `type` attribute

The `type` attibute of type `CoordinateKind` represents the type of the coordinate component, and may take one of a subset of all possible `CoordinateKind` values depending on its parent **Geometry**, as defined in Table 1. For Cartesian geometries, one-dimensional geometries may be defined by having a single "`cartesianX`" **CoordinateComponent**; two-dimensional geometries may be defined by having two **CoordinateComponent** children with `type` values of "`cartesianX`" and "`cartesianY`", and three-dimensional geometries may be defined by having three **CoordinateComponent** children with `type` values of "`cartesianX`", "`cartesianY`", and "`cartesianZ`".

GeometryKind	Dimensions	CoordinateKinds
`cartesian`	1	"`cartesianX`" (`coord1`)
`cartesian`	2	"`cartesianX`" (`coord1`) and "`cartesianY`" (`coord2`)
`cartesian`	3	"`cartesianX`" (`coord1`), "`cartesianY`" (`coord2`), and "`cartesianZ`" (`coord3`)

Table 1: *Correspondance between the* `type` *of a* ***Geometry*** *and the possible* `type`*s of its child* ***CoordinateComponent*** *elements. Also noted is the corresponding attribute (*`coord1`*,* `coord2`*, or* `coord3`*) corresponding to each axis when defining* ***InteriorPoint*** *elements (see Section 3.19 on page 22).*

3.15.3 The `unit` attribute

The unit of a **CoordinateComponent** is represented by the `unit` attribute, of type `UnitSIdRef`. If not specified, the unit of a **CoordinateComponent** inherits from the `lengthUnits` attribute of the **Model** object, and if that in turn is not specified, the **CoordinateComponent** units cannot be determined.

3.16 The Boundary class

The minimum and the maximum for a **CoordinateComponent** represent the bounds in each coordinate. For example, for three dimensional Cartesian coordinate system with x, y, and z coordinates, the minimum and maximum limits for each coordinates define planes orthogonal to each coordinate axis and passing through the minimum or maximum. If max-min is the same for each x,y,z then the bounds on the geometry is a cube. The **Boundary** class interacts with the **BoundaryCondition** class, allowing modelers to define how model elements behave at the boundary of the model. For species defined within volumes adjacent to these surfaces, **BoundaryCondition** elements must be introduced.

☞ Lucian: Just a note to coordinate the text here to sync with the text in the **BoundaryCondition** class, if that changes.

The minimum limit of a **CoordinateComponent** is represented by the `boundaryMin` object and the maximum limit is represented by the `boundaryMax` object, and apply to **CoordinateComponent** elements. Both are **Boundary** objects, and have the following attributes:

3.16.1 The `id` attribute

The `id` attribute of the **Boundary** object identifies the object. The attribute is required and is of type `SpId`. This attribute is used when specifying the **BoundaryCondition** for a species as an extension of an SBML core **Parameter**. The mathematical value of a **Boundary** is its `value` attribute, and can be bound to a **Parameter** by using a **SpatialSymbolReference**. The units are the same as its parent **CoordinateComponent**, and are not set separately.

3.16.2 The value attribute

The value attribute is of type double. In a boundaryMin object, it represents the minimum limit of the **CoordinateComponent**. In a boundaryMax object, it represents the maximum limit of the **CoordinateComponent**.

If connected to a **Parameter** via a **SpatialSymbolReference**, this value may be overridden by an **InitialAssignment**. It is theoretically possible to have this value change in time through the use of a **Rule** or **Event**, but some (if not all) software tools may not support this setup.

3.17 The DomainType class

A **DomainType** is a class of domains that are identified as being anatomically and physiologically similar. For example, a **DomainType** "cytosol" may be defined in a **Geometry** as identifying the structure and function of the cell interior. If there is one cell, then there is one domain, if there are multiple cells, then there are multiple disjoint domains ("cytosol1","cytosol2", etc.) identified with the **DomainType** "cytosol". **CompartmentMappings**, defined as an extension to an SBML core **Compartment**, map compartments to domain types such that each corresponding domain is assigned the same biological and mathematical function. Figure 8 shows the **DomainType** object.

Each SBML **Compartment** maps to a single **DomainType**, meaning that the initial condition of each **Species** in each **Compartment** will be the same across all **Domains** that map to a given **DomainType**. If those **Species** are spatially distributed, they will subsequently evolve independently from each other. However, if modeling two **Domains** that are similar but whose **Species** have different initial conditions, those **Domains** should be modeled as separate **DomainTypes**.

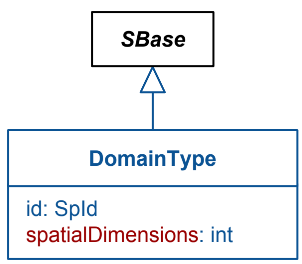

*Figure 8: The **DomainType** object. One or more instances of **DomainType** in a **ListOfDomainTypes** instance can be present in a **Geometry** object.*

3.17.1 The id attribute

Each **DomainType** is identified with a id of type SpId. The mathematical value of a **CompartmentMapping** is the sum of the sizes of all domains associated with this **DomainType**, and can be bound to a **Parameter** by using a **SpatialSymbolReference**.

As a derived quantity, if connected to a **Parameter** via a **SpatialSymbolReference**, this value may *not* be overridden by an **InitialAssignment**, nor by the use of a **Rule** or **Event**. Its value is always connected to the size of its component **Domains** instead. The units of a **DomainType** are the units of the corresponding base units of the SBML **Model** for length (for one-dimensional domains), area (for two-dimensional domains), or volume (for three-dimensional domains). It is required to define the corresponding base units for every **DomainType** in the **Model**.

3.17.2 The spatialDimensions attribute

The spatialDimensions attribute of the **DomainType** is of type int and can take on a value of 0, 1, 2, or 3. The spatial dimension is specified for a **DomainType**, rather than being repeated for each **Domain** that is represented by the **DomainType**.

3.18 The Domain class

Domains represent contiguous regions identified by the same **DomainType**. One, two and three dimensional domains are contiguous linear regions, surface regions, and volume regions (respectively), bounded by the limits of the coordinate system (e.g. min/max of x,y,z) and adjacent domains corresponding to different domain types. **Domain** is shown in Figure 9.

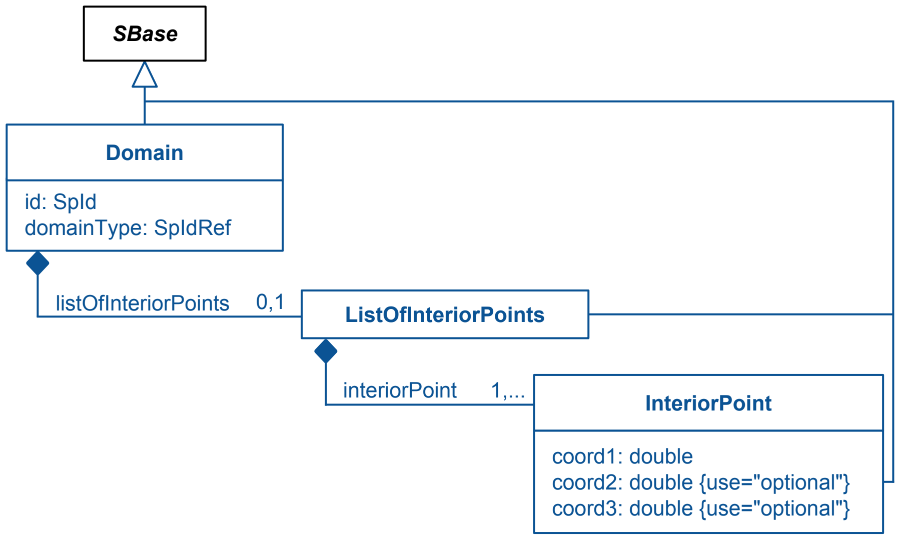

Figure 9: *The definition of the* ***Domain***, ***ListOfInteriorPoints***, *and* ***InteriorPoint*** *classes. A* ***ListOfDomains*** *instance in* ***Geometry*** *can contain one or more* ***Domain*** *object instances.*

3.18.1 The `id` attribute

A **Domain** is identified with an `id` attribute of type `SpId`. This `id` may be used within a **SpatialSymbolReference** object that is extended from an SBML core **Parameter** and can be used in an expression. The mathematical value of a **Domain** is the absolute size of that domain as used by the simulator (the meshed size), and can be bound to a **Parameter** by using a **SpatialSymbolReference**.

As a derived quantity, if connected to a **Parameter** via a **SpatialSymbolReference**, this value may *not* be overridden by an **InitialAssignment**, nor by the use of a **Rule** or **Event**. Its value is always connected to the size of the corresponding **Geometry** instead. The units of the **Domain** are the same as the units of the corresponding **DomainType**.

3.18.2 The `domainType` attribute

The `domainTpe` attribute refers to the `SpId` of the **DomainType** that describes the anatomy and physiology of this domain. The attribute is of type `SpIdRef`. It is through this association that compartments, and hence the whole SBML model, gets mapped to the individual domains.

3.19 The InteriorPoint class

Each **Domain** can contain a **ListOfInteriorPoints**. The list of spatial points for a domain is interior to that domain. This list is optional for a **Domain** if it is the only **Domain** defined for its **DomainType**, but is required otherwise.

For those geometric descriptions that can describe multiple disjoint domains belonging to the same `domainType`, these interior points allow unambiguous identification of each domain. Formally, a single point would suffice, but in practice some tools (e.g. Smoldyn) require multiple points to handle non-convex volumes bounded by explicit surfaces. For discontinuous surfaces with the same `domainType`, the interior point identifies which domain is associated with which surface patch defined in the geometry definition.

Each **InteriorPoint** has three attributes: `coord1`, `coord2`, and `coord3`.

3.19.1 The `coord1`*,* `coord2`*, and* `coord3` *attributes*

An InteriorPoint element represents a single point within the defined coordinate system and should be in the interior of the domain that contains it. It has three attributes, `coord1`, `coord2`, and `coord3`, of type `double`, representing the position along each of the up to three coordinate axes defined by the **CoordinateComponents** (with `type` "`cartesianX`", "`cartesianY`", and "`cartesianZ`", respectively, for each `coord_` attribute; see Table 1 on page 20).

Each **InteriorPoint** must define the same number of attributes as there are dimensions of the corresponding **Geometry** to which it belongs.

In the case of surfaces, interior points are sometimes required to make unambiguous identification of multiple surfaces (e.g multiple plasma membranes for multiple cells present in a geometry). Due to roundoff error and finite word lengths, it is difficult to find a three dimensional point that lies on a surface. In this case, the distance from the surface will be used to provide unambiguous identification.

3.20 The AdjacentDomains class

AdjacentDomains (or domain adjacencies) captures the topological relationships within the **Geometry**. Consider that the **Domains** are nodes in a graph. The **AdjacentDomains** objects are the edges that specify the spatial connectivity of these nodes. Armed with the topology and the domain sizes, one can readily perform a compartmental approximation. Figure 10 shows the definition of the **AdjacentDomains** object.

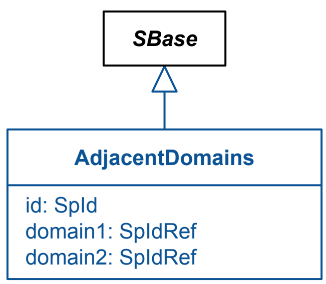

Figure 10: *The definition of the* ***AdjacentDomains*** *class.* ***Geometry*** *can contain one instance of* ***ListOfAdjacentDomains*** *that can have one or more instances of* ***AdjacentDomains*** *objects.*

3.20.1 The `id` *attribute*

This attribute identifies an **AdjacentDomains** object. The attribute is of type `SpId`. It has no mathematical meaning, and cannot be connected to a **Parameter** via a **SpatialSymbolReference** element.

3.20.2 The `domain1` *and* `domain2` *attributes*

The `domain1` and `domain2` attributes, of type `SpIdRef`, are required attributes. They are the `SpId`'s of two domains that touch each other (spatially adjacent). These are typically surface-volume contacts.

3.21 The GeometryDefinition class

A **Geometry** can specify a list of ***GeometryDefinitions***. The ***GeometryDefinition*** is an abstract class that is the general term for the container which defines the concrete geometric constructs represented by the **Geometry**. Four types of ***GeometryDefinitions*** have been identified - **AnalyticGeometry**, **SampledFieldGeometry**, **ParametricGeometry**, **CSGeometry** (Constructed Solid Geometry) - and are elaborated in the following sections. The definition of the

GeometryDefinition element is displayed in Figure 11. The spatial dimension of the ***GeometryDefinition*** must match the `spatialDimensions` of the **DomainType** defined for the associated **Domain**.

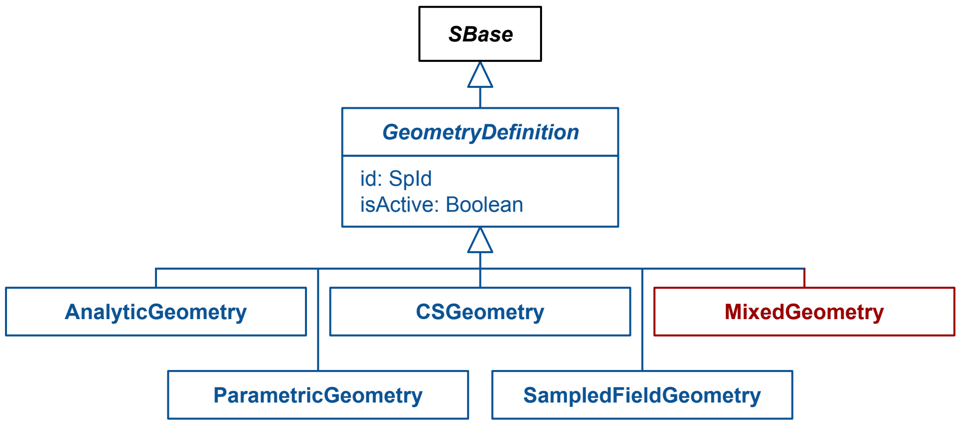

Figure 11: *The* ***GeometryDefinition*** *element.* ***Geometry*** *contains one instance of listOfGeometryDefinitions that can contain one or more instances of* ***GeometryDefinition*** *(one of* ***AnalyticGeometry****,* ***SampledFieldGeometry****,* ***CSGeometry****,* ***ParametricGeometry****, defined below).*

3.21.1 The `id` *attribute*

The `id` attribute is common to all the ***GeometryDefinition*** types and is used to uniquely identify the ***GeometryDefinition***. The attribute is of type `SpId`. It has no mathematical meaning, and cannot be connected to a **Parameter** via a **SpatialSymbolReference** element.

3.21.2 The `isActive` *attribute*

The `isActive` attribute that is common to all the ***GeometryDefinition*** types is used to identify the ***GeometryDefinition*** that is considered the active ***GeometryDefinition*** for the document. When multiple ***GeometryDefinition*** elements define the same underlying geometry, each may set their `isActive` attribute to "`true`". At least one ***GeometryDefinition*** in a **Model** must have an `isActive` attribute of "`true`", and any other ***GeometryDefinition*** that does not describe that same underlying physical geometry must have an `isActive` value of "`false`".

3.22 The AnalyticGeometry class

The **AnalyticGeometry** is a class of ***GeometryDefinition*** where the geometry of each domain is defined by an analytic expression. An **AnalyticGeometry** is defined as a collection of **AnalyticVolumes**, one **AnalyticVolume** for each volumetric domain in the geometry. In this representation, the surfaces are treated as the boundaries between dissimilar **AnalyticVolumes**. The **AnalyticGeometry** object contains a **ListOfAnalyticVolumes**. Figure 12 on the following page shows the definition of the **AnalyticGeometry** object.

3.23 The AnalyticVolume class

The **AnalyticVolume** is used to specify the analytic expression of a volumetric (3-dimensional) domain. The analytic expression for the **AnalyticVolume** is defined in the Math element.

3.23.1 The `id` *attribute*

The `id` attribute uniquely identifies the **AnalyticVolume**. The attribute is required and is of type `SpId`. It has no mathematical meaning, and cannot be connected to a **Parameter** via a **SpatialSymbolReference** element.

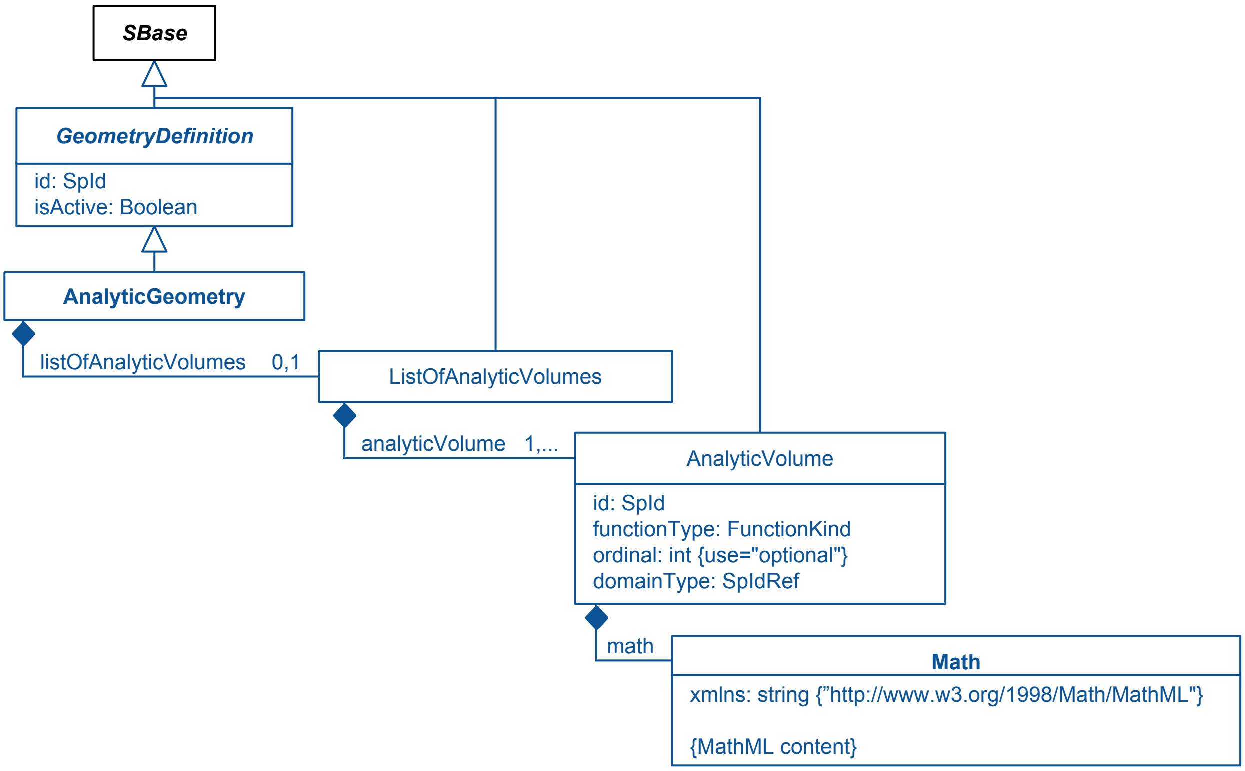

Figure 12: *The definition of the* ***AnalyticGeometry****,* ***ListOfAnalyticVolumes****, and* ***AnalyticVolume*** *classes.*

3.23.2 The `functionType` *attribute*

The `functionType` attribute is of type `FunctionKind` and is currently limited to just "`layered`" (a possibility for future versions of the specification is to allow the value "`R-function`"). A "`layered`" function type implies that the Math child element contains an inequality in the spatial dimensions (e.g. x,y,z) such that evaluation to "`true`" indicates that the point (x,y,z) is within that shape, and "false" indicates that it is not covered by that shape.

☞ Lucian: If you want to rename 'layered', now is the time to do it ;-)

3.23.3 The `domainType` *attribute*

The `domainType` attribute of type `SpIdRef` is a required attribute. It represents the `SpId` of the **DomainType** of the **Domain** that is represented by this **AnalyticVolume**.

3.23.4 The `ordinal` *attribute*

The `ordinal` attribute is of type `int`, and is required. It is used to represent the order of the **AnalyticVolume**. The `ordinal` is useful while reconstructing the geometry in the specific software tool - it represents the order in which the **AnalyticVolumes** representing geometric domains have to be evaluated.

Rather than struggle with the task of preventing overlapping regions of space from different **AnalyticVolumes**, the **AnalyticVolumes** are to be considered to be evaluated in the reverse order of their ordinals. In this way, any **AnalyticVolumes** that have already been processed will cover those with a smaller ordinal, thus resolving any ambiguities and removing the constraint that all **AnalyticVolumes** be disjoint and cover the entire geometric domain. The **AnalyticVolume** with `ordinal` 0 can be the "background" layer (typically the extracellular space).

No two **AnalyticVolume** elements should have the same `ordinal` value, even if they should not overlap, because some tools may not calculate the geometries to the same level of precision as other tools, and may end up with overlap due to rounding errors, and will still need to resolve the ambiguity for their own purposes. If a software tool discovers two overlapping **AnalyticVolume** elements with the same `ordinal` value, it may resolve the situation however it sees fit.

Lucian: To match the **CSGPrimitive** class, relaxed the 'no two may be equal' rule to be a suggestion.

3.24 The Math class

The Math element is a required element for an **AnalyticVolume**. The Math element contains a MathML expression that defines the analytic expression for the **AnalyticVolume** referencing the coordinate components that are specified in the **ListOfCoordinateComponents** in the **Geometry**, according to the `functionType`.

3.25 The SampledFieldGeometry class

SampledFieldGeometry is a type of ***GeometryDefinition*** that defines a sampled image-based geometry or a geometry based on samples from a level set. **SampledFieldGeometry** is defined using a **SampledField** that specifies the sampled image and a list of **SampledVolumes** that represent the volumetric domains as sampled image regions. Figure 13 shows the definition of the **SampledFieldGeometry** object. It may be used for geometries of any dimension.

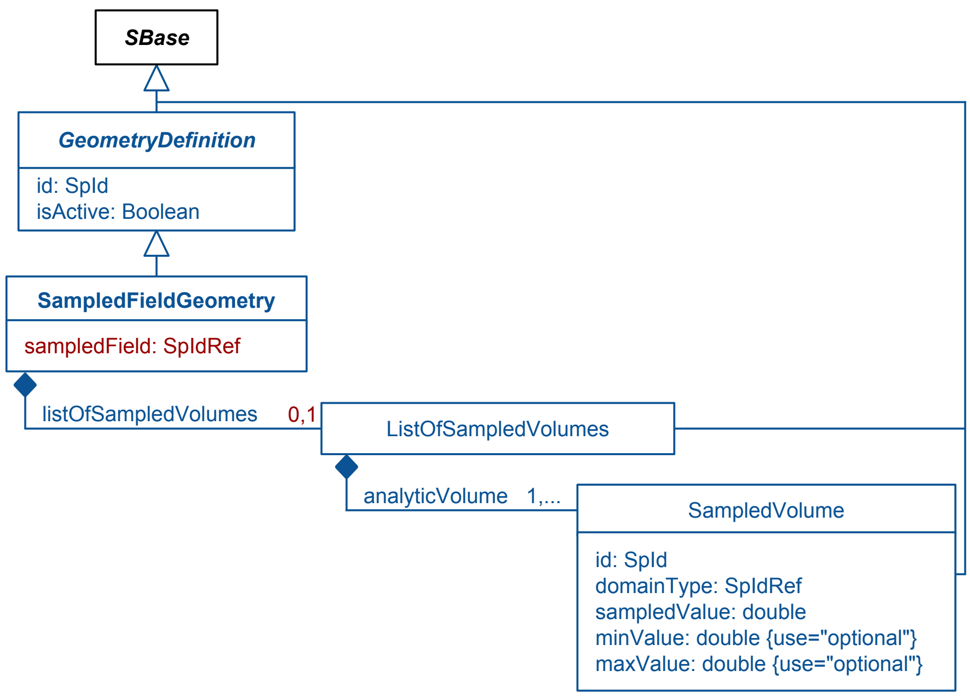

Figure 13: *The definition of the **SampledFieldGeometry**, **ListOfSampledVolumes**, and **SampledVolume** classes.*

3.25.1 The `sampledField` attribute

The `sampledField` attribute is of type SpIdRef, and must reference a **SampledField** from the same **Geometry**. That referenced field is to be used to set up the different spatial areas in the geometry of the **Model**, according to the **SampledVolume** child elements.

3.26 The SampledVolume class

A **SampledVolume** represents an interval of the sampled field that constitutes one or more contiguous regions. A **SampledVolume** is defined for each volumetric (3-dimensional) **Domain** in the **Geometry**. It has the following attributes.

3.26.1 The `id` *attribute*

The `id` attribute identifies a **SampledVolume** object. The attribute is of type `SpId` and is required when specifying a **SampledVolume**. It has no mathematical meaning, and cannot be connected to a **Parameter** via a **SpatialSymbolReference** element.

3.26.2 The `domainType` *attribute*

The required `domainType` attribute is of type `SpIdRef`. It is the `SpId` of the **DomainType** that represents this class of anatomical features. If there are more than one contiguous regions, then more than one domain will be defined corresponding to each **SampledVolume**.

3.26.3 The `sampledValue` *attribute*

The required `sampledValue` attribute is of type `double`. It represents the pixel value of a **SampledVolume**.

3.26.4 The `minValue` *attribute*

The optional `minValue` attribute is of type `double`. It represents the minimum of the pixel value (`sampledValue`) range.

3.26.5 The `maxValue` *attribute*

The optional `maxValue` attribute is of type `double`. It represents the maximum of the pixel value (`sampledValue`) range.

3.27 The CSGeometry class

CSGeometry (Constructed Solid Geometry) is a type of ***GeometryDefinition*** that defines a combined, solid, volumetric object from a number of primitive solid volumes by the application of set operations such as union, intersection and difference and affine transformations such as rotation, scaling, translation, etc. The **CSGeometry** element is defined by a `listOfCSGObjects` element that contains a collection of **CSGObjects**. Figure 14 on the following page shows the definition of the **CSGeometry** object.

3.28 The CSGObject class

Each **CSGObject** is a scene graph representing a particular geometric object using constructed solid geometry. A node in a tree (scene graph) is made up of **CSGPrimitives**, **CSGSetOperators**, and ***CSGTransformations***. Note that the **CSGPrimitives** are always leaves in this tree. The **CSGObject** is analogous to an **AnalyticVolume** element in the sense that it is a constructed geometry (from primitives) used to specify a volumetric (3-dimensional) domain. The **CSGObject** element has three attributes : `id`, `domain`Type and `ordinal`. The definition of the **CSGObject** is completed by defining a ***CSGNode*** which is the root of the **CSGObject** scene graph.

3.28.1 The `id` *attribute*

The `id` attribute uniquely identifies the **CSGObject** element. The attribute is required and is of type `SpId`. It has no mathematical meaning, and cannot be connected to a **Parameter** via a **SpatialSymbolReference** element.

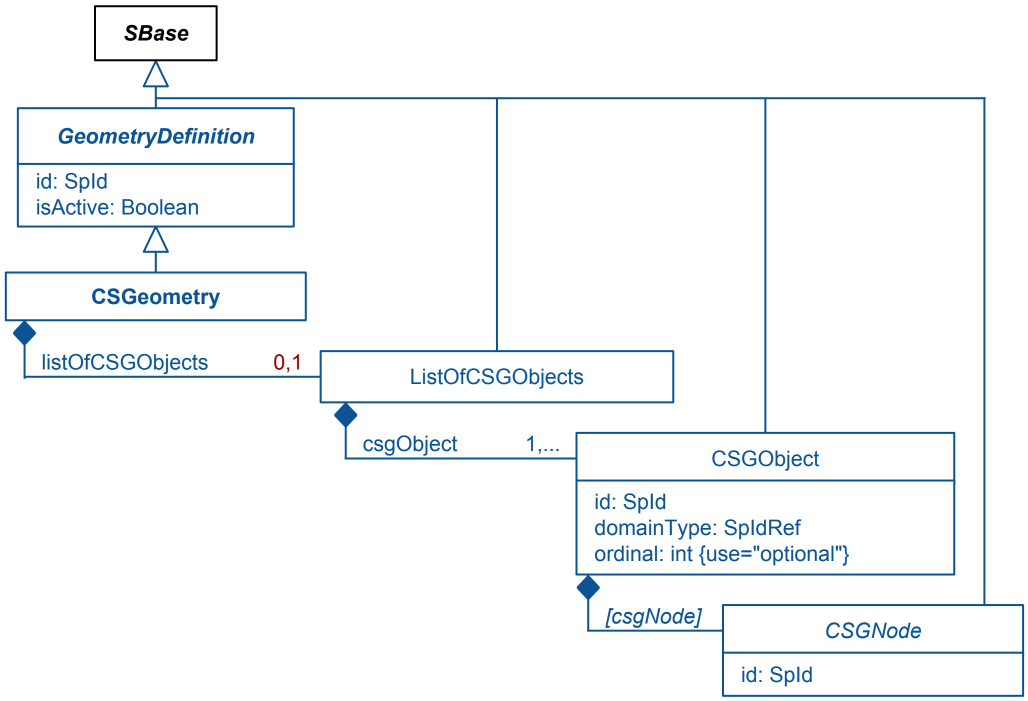

Figure 14: *The definition of the* ***CSGeometry***, ***ListOfCSGObjects***, *and* ***CSGObject*** *classes.*

3.28.2 The `domainType` *attribute*

The `domainType` attribute is of type `SpIdRef` and is a required attribute. It is a reference to the `id` of the **DomainType** that this **CSGObject** represents.

All **InteriorPoints** of the corresponding **DomainType** must be points inside the geometry this **CSGObject** describes.

☞ Lucian: Jim clarified that all of these attributes are supposed to be domainType references, not domain references. In this case, it was always 'domainType' in the UML.

3.28.3 The `ordinal` *attribute*

The `ordinal` attribute is of type `int`. It is used to represent the order of the **CSGObject**. The `ordinal` is useful while reconstructing the geometry in the specific software tool - it represents the order in which the **CSGObjects** representing geometric domains have to be placed.

No two **CSGObject** elements should have the same `ordinal` value, even if they should not overlap, because some tools may not calculate the geometries to the same level of precision as other tools, and may end up with overlap due to rounding errors, and will still need to resolve the ambiguity for their own purposes. If a software tool discovers two overlapping **CSGObject** elements with the same `ordinal` value, it may resolve the situation however it sees fit.

☞ Lucian: Made this required, but relaxed the 'no two may be equal' rule to be a suggestion.

3.28.4 The [`csgNode`] *child*

The child [`csgNode`] element represents the geometry that is to be linked to the `domainType` of the **CSGObject**. Note that the child of the **CSGObject** element is not called "`csgNode`" but rather takes the name of the derived class, decapitalized. Thus, a **CSGObject** may have a `csgPrimitive`, `csgPseudoPrimitive`, `csgSetOperator`, `csgTranslation`, `csgRotation`, `csgScale`, or `csgHomogeneousTransformation` child.

3.29 The CSGNode class

The operators and operands used to construct a constructed solid geometry are generalized as a ***CSGNode***, defined in Figure 15 as an abstract base class. The classes that inherit from ***CSGNode*** can be one of the following: **CSGSetOperator**, ***CSGTransformation*** (operators; itself another abstract base class), **CSGPrimitive**, or **CSGPseudoPrimitive** (operands). The ***CSGNode*** has one attribute: `id`. The **CSGObject** contains a ***CSGNode*** object which is the root of the **CSGObject** scene graph (representing one constructed solid geometry domain).

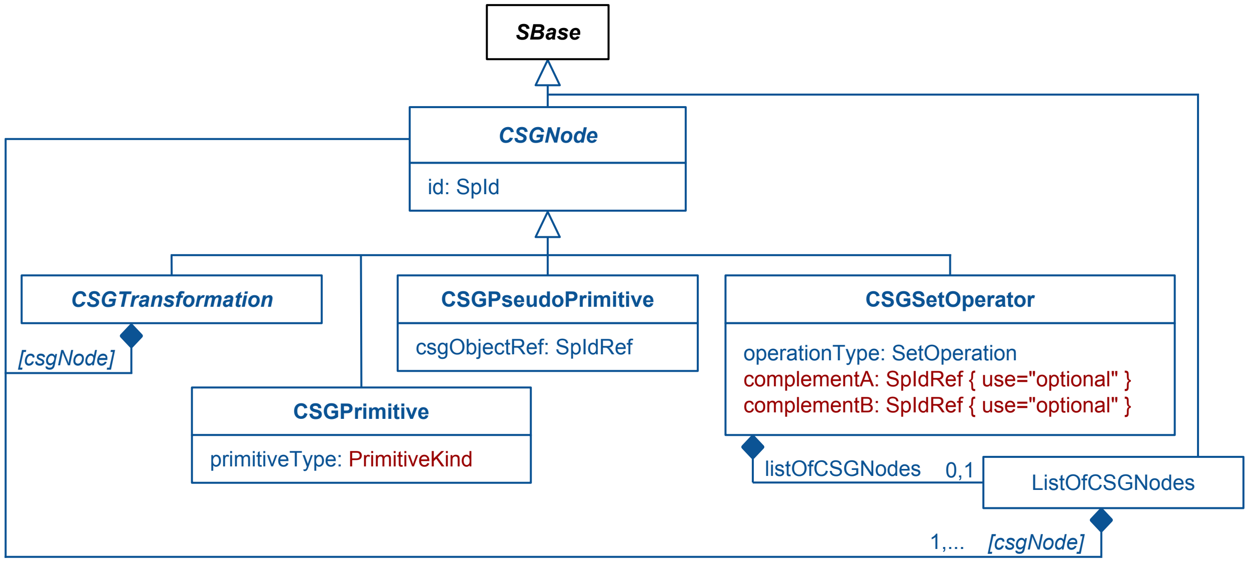

Figure 15: *The definition of the abstract base class **CSGNode**, and its subclasses **CSGPrimitive**, **CSGPseudoPrimitive**, and **CSGSetOperator**. The abstract base class **CSGTransformation** (also a subclass of **CSGNode**) is defined in Section 3.34 on page 31.*

3.29.1 The `id` *attribute*

The `id` attribute uniquely identifies the ***CSGNode*** element. The attribute is required and is of type `SpId`. It has no mathematical meaning, and cannot be connected to a **Parameter** via a **SpatialSymbolReference** element.

PrimitiveKind	Dimensions	Definition
`sphere`	3	A sphere with radius 1, centered at the origin.
`cube`	3	A cube with sides of length 2, centered at the origin.
`cylinder`	3	A cylinder with a base circle of radius 1, centered at (0,0,-1), and top circle of radius 1, centered at (0,0,1). The height is 2.
`cone`	3	A cone with a base circle of radius 1, centered at (0,0,-1), and top vertex at (0,0,1).
`circle`	2	A circle with radius 1, centered at the origin.
`square`	2	A square with sides of length 2, centered at the origin
`rightTriangle`	2	A triangle with vertexes at (-1,-1), (-1,1), and (1,-1).

Table 2: *Definitios of the possible values of the* `primitiveType` *attribute of the **CSGPrimitive** class.*

☞ Lucian: Does this work?

3.30 The CSGPrimitive class

CSGPrimitive element represents the primitive geometric shapes that can be represented by the **CSGeometry**. These shapes are defined in Table 2 on the preceding page with a predefined orientation and fitting within the unit cube (+/- 1 in x, y, and z) or unit square (+/- 1 in x and y). This element has one required attribute : `primitiveType` of type `PrimitiveKind`.

3.30.1 The `primitiveType` *attribute*

The `primitiveType` attribute is a required attribute that is of type `PrimitiveKind`. It represents one of the predefined primitive shapes. The meaning and definition of those types is listed in Table 2 on the previous page.

3.31 The CSGPseudoPrimitive class

CSGPseudoPrimitive element is used to reference a pre-defined **CSGObject** object while defining a **CSGObject** (geometric domain). This allows the re-use of one constructed **CSGObject** in another. It has one attribute of type `SpIdRef`.

3.31.1 The `csgObjectRef` *attribute*

The `csgObjectRef` attribute identifies a pre-defined **CSGObject** in the **CSGeometry** The attribute is required and is of type `SpId`. A **CSGObject** may not reference itself, nor its parent, nor its parent's parent, etc.

3.32 The CSGSetOperator class

The **CSGSetOperator** element represents the set operations (union, intersection, difference) that can be performed on a set of primitive geometric shapes (**CSGPrimitives**) or on a set of ***CSGNodes*** (a transformation or set operation on one or a set of **CSGPrimitives**). This element has one attribute of type `string`. It also contains a required child **ListOfCSGNodes** that represents the set of nodes on which the set operation is performed.

3.32.1 The `operationType` *attribute*

The `operationType` attribute is of type `SetOperation` and represents an operation that can be performed on a set of ***CSGNodes***. The possible values that the `operationType` attribute can take are "`union`", "`intersection`", or "`relativeComplement`". The values "`union`" and "`intersection`" are n-ary, meaning they are defined for any number of child nodes of this **CSGSetOperator**. The intersection or union of the empty set (zero children) is defined as the empty set, and the intersection or union of a single child is defined as that child. The union of multiple sets is defined as including any element that appears in any of those component sets, and the intersection of multiple sets is defined as including only those elements that appear in all of the component sets.

The value "`relativeComplement`" is binary, meaning that it must have exactly two children. Its meaning is defined according to the `complement` attributes, below.

3.32.2 The `complementA` *and* `complementB` *attributes*

The `complement` attributes are of type `SpIdRef`. If the `operationType` of the **CSGSetOperator** is "`relativeComplement`", they both must be set to indicate the order in which the complement is to be carried out, and must refer, respectively, to the two `csgNode` children of this **CSGSetOperator**. The relative complement of the children (and thus the meaning of this node) is defined as the set of elements in `complementB`, but not in `complementA`.

If the `operationType` of the **CSGSetOperator** is not "`relativeComplement`", neither `complement` attribute may be set.

3.33 The ListOfCSGNodes class

The **ListOfCSGNodes** must contain one or more `csgNode` children that are to be combined according to the set operation of the parent **CSGSetOperator**. While having a single child is legal, this is semantically equivalent to simply putting that child in the model instead of the **CSGSetOperator**, and therefore has limited modeling benefit. Note that the children of the **ListOfCSGNodes** object are not called `csgNode` but rather take the name of the derived class, decapitalized. Thus, they may be called `csgPrimitive`, `csgPseudoPrimitive`, `csgSetOperator`, `csgTranslation`, `csgRotation`, `csgScale`, or `csgHomogeneousTransformation`.

3.34 The CSGTransformation class

The ***CSGTransformation*** represents a generalization for the type of transformation that can be performed on a primitive geometric shape (**CSGPrimitive**) or on a ***CSGNode*** (a transformation or set operation on one or a set of **CSGPrimitives**). The types of possible transformations are 'rotation', 'translation', 'scaling', and 'homogeneous transformation', defined below. The ***CSGTransformation*** element contains a ***CSGNode*** element upon which the transformation is performed.

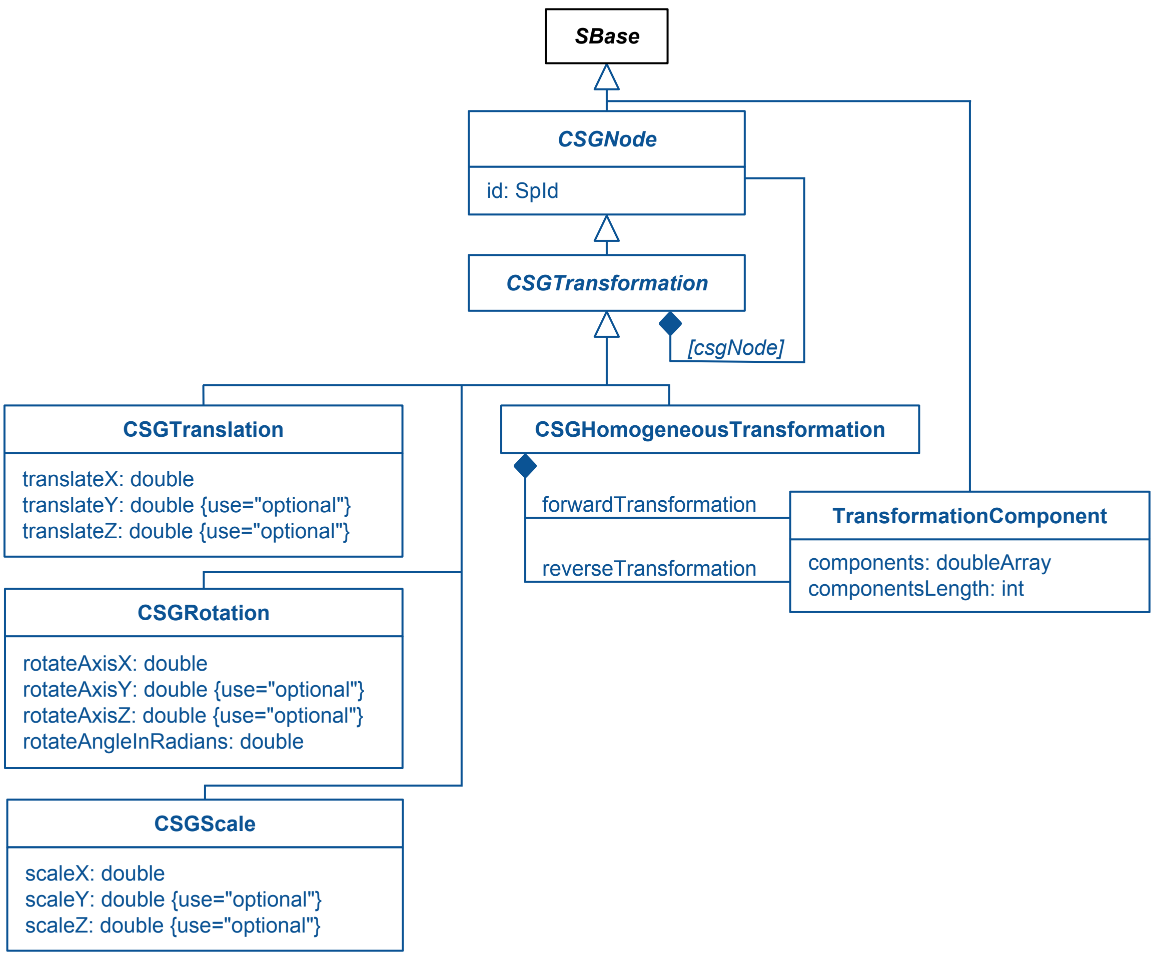

Figure 16: *The definition of the abstract base class **CSGTransformation**, its subclasses, **CSGRotation**, **CSGScale**, and **CSGHomogeneousTransformation**, and the **TransformationComponent** class.*

☞ Lucian: 'transformationType' attribute removed (it was never in the UML).

3.34.1 *The* `csgNode` *child*

The child `csgNode` element represents the geometry that is to be transformed by the ***CSGTransformation*** element. Note that this child is not called `csgNode` but rather takes the name of the derived class, decapitalized. Thus, it may be called `csgPrimitive`, `csgPseudoPrimitive`, `csgSetOperator`, `csgTranslation`, `csgRotation`, `csgScale`, or `csgHomogeneousTransformation`.

3.35 The CSGTranslation class

The **CSGTranslation** element represents a translation transformation on a ***CSGNode*** (a transformation or set operation on one or a set of **CSGPrimitives**) or a **CSGPrimitive** along the axes defined in the **Geometry**. This element has 3 attributes:

3.35.1 *The* `translateX` *attribute*

The `translateX` attribute is of type `double`. It represents the translation of the ***CSGNode*** along the x-axis (the **CoordinateComponent** with the `type` of "`cartesianX`").

3.35.2 *The* `translateY` *attribute*

The `translateY` attribute is of type `double`. It represents the translation of the ***CSGNode*** along the y-axis (the **CoordinateComponent** with the `type` of "`cartesianY`").

3.35.3 *The* `translateZ` *attribute*

The `translateZ` attribute is of type `double`. It represents the translation of the ***CSGNode*** along the z-axis (the **CoordinateComponent** with the `type` of "`cartesianZ`").

3.36 The CSGRotation class

The **CSGRotation** element represents a rotation transformation on a ***CSGNode*** (a transformation or set operation on one or a set of **CSGPrimitives**) or a **CSGPrimitive** about the axes defined in the **Geometry**. This element has 4 attributes:

3.36.1 *The* `rotateX`*,* `rotateY`*, and* `rotateZ` *attributes*

The `rotate` attributes are of type `double`, and must be defined for the same number of **CoordinateComponents** as are present in the parent **Geometry**. If all three are defined, they define a point in space that determines the vector (from the origin) of the axis about which the rotation is to occur in three-dimensional space. They must therefore not all be equal to zero. If two are defined (`rotateX` and `rotateY`), they define the point in two-dimensional space about which the rotation is to occur. (In this case, (0,0) would be legal.) Nothing can be rotated in 1-dimensional space, and therefore defining only `rotateX` is meaningless.

3.36.2 *The* `rotationAngleInRadians` *attribute*

The `rotationAngleInRadians` attribute is of type `double`. It represents the rotation angle of the ***CSGNode***, in radians, along the defined axis. In three-dimensional space, this rotation is defined as clockwise looking down the vector from the origin. In two-dimensional space, this rotation is defined as being clockwise.

3.37 The CSGScale class

The **CSGScale** element represents a scale transformation on a ***CSGNode*** (a transformation or set operation on one or a set of **CSGPrimitives**) or a **CSGPrimitive** along the axes defined in the **Geometry**. All scaling occurs individually for all component primitive shapes, relative to those shapes' origins when originally defined in Table 2 on page 29. This makes a significant difference when using (for example) the 'set' operations: two adjacent circles may not touch, and their intersection may therefore be zero. But if scaled to be larger, those two circles may now overlap, creating an intersection where none existed before. Even if the child of the **CSGScale** class is a **CSGSetOperator**, then, it is the children of that set operator that are scaled, and not the calculated set itself. It is therefore perhaps easier (and more clear to model) if the only children of the **CSGScale** class are **CSGPrimitives**.

☞ Lucian: So, I just made up the above based on what I imagined would be easier to implement. The other option is that the scale operator acts on the *center of mass* of its child, meaning that the center of mass must be calculated for anything not already a **CSGPrimitive**. Whoever implements this first, let me know what turns out to be easier, and we can do it that way.

This element has 3 attributes:

3.37.1 The `scaleX` attribute

The `scaleX` attribute is of type `double`. It represents the amount of scaling of the ***CSGNode*** along the x-axis (the **CoordinateComponent** with the `type` of "`cartesianX`").

3.37.2 The `scaleY` attribute

The `scaleY` attribute is of type `double`. It represents the amount of scaling of the ***CSGNode*** along the y-axis (the **CoordinateComponent** with the `type` of "`cartesianY`").

3.37.3 The `scaleZ` attribute

The `scaleZ` attribute is of type `double`. It represents the amount of scaling of the ***CSGNode*** along the z-axis (the **CoordinateComponent** with the `type` of "`cartesianZ`").

☞ Lucian: Is 'the amount of scaling' here relative? In other words, if I have a `scaleX` value of '1.0', is it not scaled along the X axis at all, with '0.5' halving it and '2.0' doubling it? And '0.0' making it disappear? Are negative values allowed, inverting the object, i.e. inverting a cone by giving it a 'z' scaling factor of '-1'?

3.38 The CSGHomogeneousTransformation class

The **CSGHomogeneousTransformation** element represents a homogeneous transformation on a ***CSGNode***: a transformation or set operation on one or more **CSGPrimitives**. This element contains two TransformationComponent elements : a `forwardTransformation` and a `reverseTransformation`, both of type **TransformationComponent**.

☞ Lucian: We probably don't need both a forwardTransformation and a reverseTransformation: if it turns out to be difficult to implement both, just implement 'forwardTransformation'. If nobody ever implements 'reverseTransformation', we'll remove it from the spec.

3.39 The TransformationComponent class

The **TransformationComponent** element represents an affine transformation that can be applied to a ***CSGNode***. This element has the following two attributes:

3.39.1 The `components` attribute

The `components` attribute is of type `doubleArray`, whose values represent the affine transformation. This attribute is required.

An affine transformation is essentially a method to transform a shape's scale, rotation, and translation all at once instead of breaking it down into its component tranformations.

☞ Lucian: If someone can concisely explain the algorithm at work here, and can send it to me, I will put it into the spec, because despite reading up on affine transformations on Wikipedia, I don't understand it well enough to explain how you get from a vector of components to a transformed shape.

3.39.2 The `componentsLength` attribute

The `componentsLength` attribute is of type `int` and is required. It represents the array length of the `components` attribute (number of values in the `components` array).

3.40 The ParametricGeometry class

ParametricGeometry is a type of ***GeometryDefinition*** that parametrically defines geometric strucutures/domains. The **ParametricGeometry** element is defined with a `listOfObjects` that is a collection of **ParametricObjects** and a `listOfSpatialPoints` that is a collection of **SpatialPoints**. Figure 17 shows the definition of the **ParametricGeometry** object.

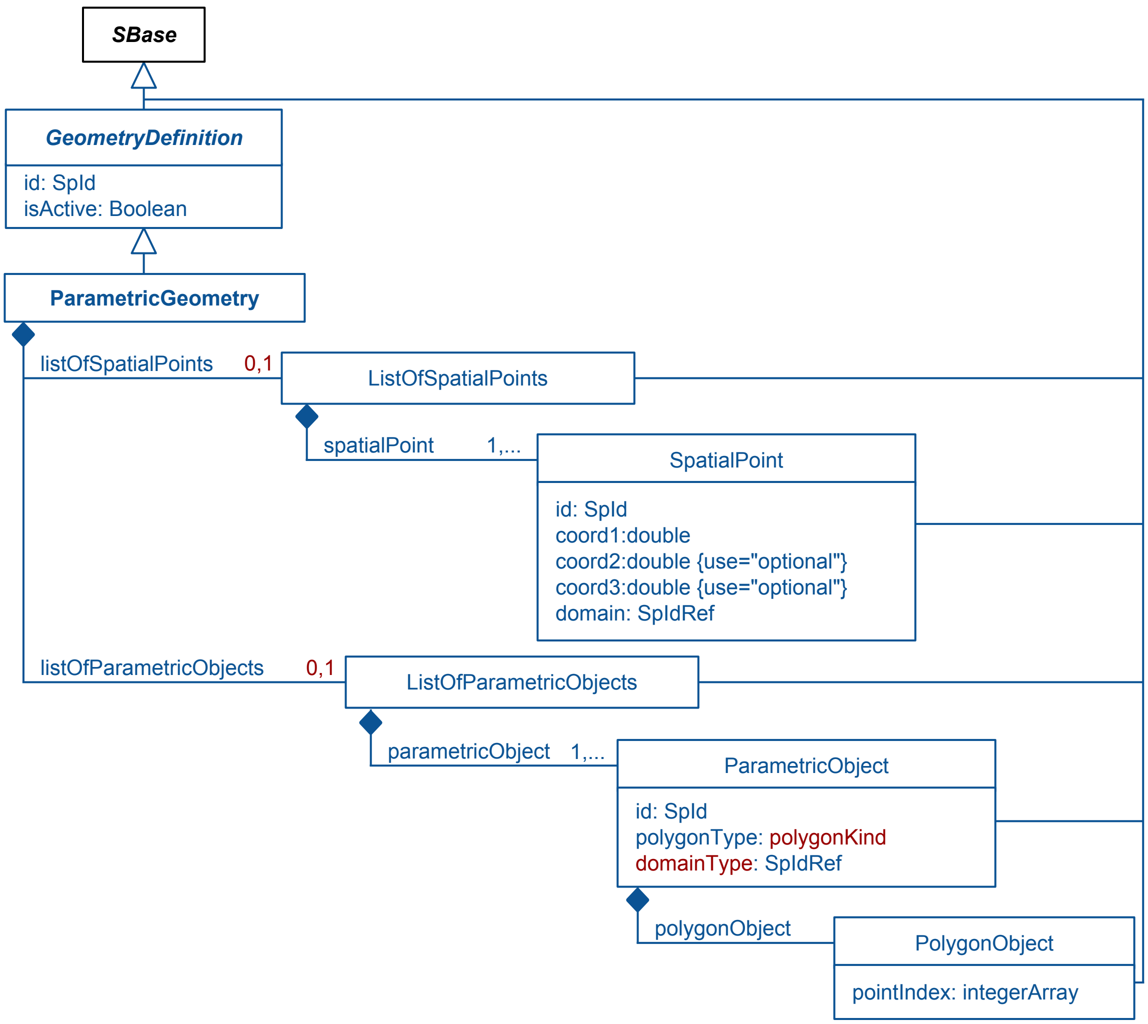

*Figure 17: The definition of the **ParametricGeometry**, **ListOfSpatialPoints**, **SpatialPoint**, **ListOfParametricObjects**, **ParametricObject**, and **PolygonObject** classes.*

3.41 The ParametricObject class

The **ParametricObject** element represents a parametric geometry object.

3.41.1 The `id` attribute

The `id` attribute is a required attribute of type `SpId`. It uniquely identifies the **ParametricObject** element. It has no mathematical meaning, and cannot be connected to a **Parameter** via a **SpatialSymbolReference** element.

3.41.2 The `polygonType` attribute

The `polygonType` attribute is of type `PolygonKind` and is a required attribute. It represents the type of polygon that describes the **ParametricObject**. **ParametricObject** elements of type "`triangle`" have three points, and those of type "`quadrilateral`" have four.

3.41.3 The `domainType` attribute

The `domainType` attribute is of type `SpIdRef` and is a required attribute. It is a reference to the `id` of the **DomainType** that this **ParametricObject** represents.

3.42 The PolygonObject class

The polygonObject represents an ordered list of indices that refer to elements in the SpatialPoints array and are interpreted by considering the `polygonType` attribute of the **ParametricObject**. The data must follow VTK polydata conventions for shapes and vertex orderings, defined at `http://www.vtk.org/VTK/img/file-formats.pdf`.

☞ Lucian: Is that a reasonable reference for the VTK format? Is there a better one?

3.42.1 The `pointIndex` attribute

The `pointIndex` attribute is an array of integers holding the indices of the points in the SpatialPoints array.

3.43 The SpatialPoint class

The **SpatialPoint** element represents a point used as a vertex in the **ParametricGeometry**.

3.43.1 The `id` attribute

The `id` element uniquely identifies a SpatialPoint element. It is a required attribute and is of type `SpId`. It has no mathematical meaning, and cannot be connected to a **Parameter** via a **SpatialSymbolReference** element.

3.43.2 The `coord1`, `coord2`, and `coord3` attributes

The `coord1`, `coord2`, `coord3` attributes are of type `double`. They represent the 3-dimensional coordinate of the SpatialPoint. Depending on the dimension of the **Geometry**, one, two or all the three attributes are required.

3.43.3 The `domain` attribute

The `domain` attribute is of type `SpIdRef` and is a required attribute. It is a reference to the `id` of the **Domain** which contains this SpatialPoint.

☞ Lucian: This is the single place where a Geometry refers to a **Domain** and not a **DomainType**. Is that correct? Should it actually point to a **DomainType**? Do we need the attribute at all; is it redundant with the 'domainType' attribute on ParametricObject?

3.44 The MixedGeometry class

A **MixedGeometry** defines a **Geometry** constructed from a collection of various ***GeometryDefinition*** objects that together define the complete geometry for the **Model**. It has a child **ListOfGeometryDefinitions** object that behaves exactly the same as the **ListOfGeometryDefinitions** child of the **Geometry**, but instead of that collection of geometry definitions defining alternate geometries, or alternate ways to define one geometry, the collection of geometry definitions in a **MixedGeometry** together define a single space. For example, a **MixedGeometry** may contain a **ParametricGeometry** that defines the contours of a cell membrane, plus a **CSGeometry** that defines a sphere that models the nucleus of that cell. The definition of a **MixedGeometry** is shown in Figure 18. Its **OrdinalMapping** children define how those geometries overlap one another.

Note that every child ***GeometryDefinition*** of a **MixedGeometry** must have an `isActive` value of "`false`". 'Active' geometries are a concept that applies only to the **Model** and its direct children, not to component geometries of a **MixedGeometry**.

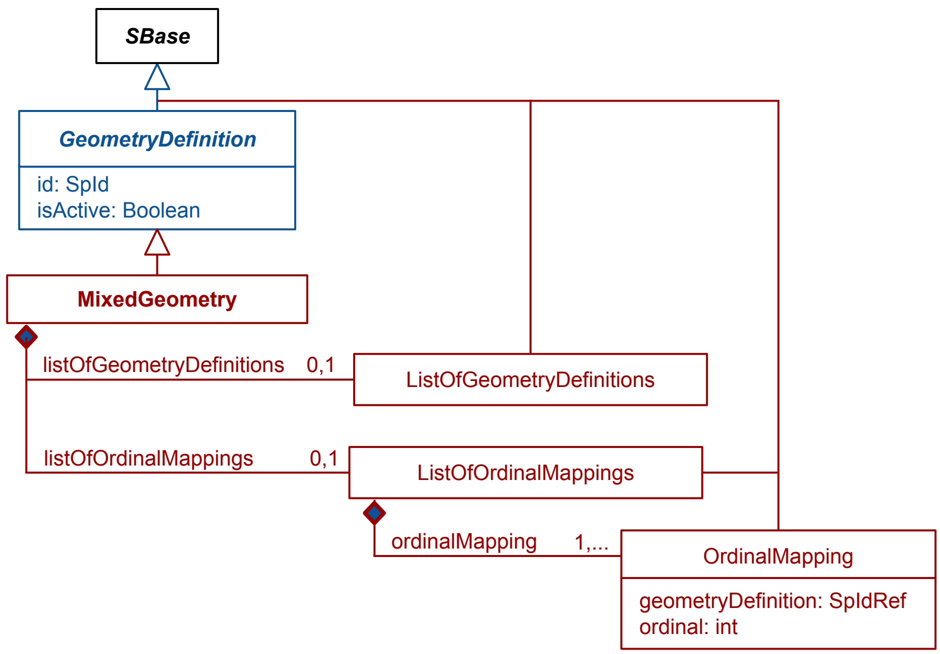

Figure 18: *The definition of the **MixedGeometry**, **ListOfOrdinalMappings**, and **OrdinalMapping** classes from the Spatial package. The **ListOfGeometryDefinitions** class is defined in Section 3.14 on page 18*

3.45 The OrdinalMapping class

A **OrdinalMapping** defines an ordinal level for the various geometries that comprised this **MixedGeometry**. In this way, the overlap between them can be resolved cleanly. There must be exactly one **OrdinalMapping** for each child ***GeometryDefinition*** of a **MixedGeometry**.

☞ Lucian: Or we could make that a 'should'? 'ordinal' is required in the other places it is used, so 'must' is parallel to that.

3.45.1 The `geometryDefinition` attribute

The tokengeometryDefinition attribute is of type `SpIdRef`, and is required. It must reference a direct child ***GeometryDefinition*** of this **MixedGeometry**. The `ordinal` value is then taken to refer to that geometry.

3.45.2 The `ordinal` attribute

The `ordinal` attribute is of type `int`, and is required. It is used to represent the order of the corresponding ***GeometryDefinition*** within this **MixedGeometry**. The `ordinal` is useful while reconstructing the geometry in the specific software tool - it represents the order in which each ***GeometryDefinition*** have to be evaluated.

Rather than struggle with the task of preventing overlapping regions of space from each different ***GeometryDefinition***, they are to be considered to be evaluated in the reverse order of their ordinals. In this way, any ***GeometryDefinition*** that has already been processed will cover those with a smaller ordinal, thus resolving any ambiguities and removing the constraint that each ***GeometryDefinition*** be disjoint and cover the entire geometric domain. The ***GeometryDefinition*** with `ordinal` 0 can be the "background" layer (typically the extracellular space).

No two ***GeometryDefinition*** elements should have the same `ordinal` value, even if they should not overlap, because some tools may not calculate the geometries to the same level of precision as other tools, and may end up with overlap due to rounding errors, and will still need to resolve the ambiguity for their own purposes. If a software tool discovers two overlapping ***GeometryDefinition*** elements with the same `ordinal` value, it may resolve the situation however it sees fit.

Note that these ordinals only apply at the level of the immediate parent **MixedGeometry**. Any `ordinal` attribute from any **AnalyticVolume** or **CSGObject** applies only at the level of those volumes or objects, and serve to distinguish within-***GeometryDefinition*** layout order, not between-***GeometryDefinition*** layout order, defined here. Likewise, if a ***GeometryDefinition*** child of a **MixedGeometry** it itself a **MixedGeometry**, those ordinals also only apply at the level of that **MixedGeometry**, and not at the level of the parent **MixedGeometry**.

☞ Lucian: Realized we needed ordinals for the mixed geometries, and figured this was the cleanest solution.

3.46 The SampledField class

A **SampledField** is a sampled scalar field such as an image or samples from a level set. The attributes of **SampledField** represent the specification of a sample dataset (the number of samples in x, y, z coordinates, data type of the sample representation, etc.) and the text child of the **SampledField** is the actual sampled data.

3.46.1 The `id` attribute

The `id` attribute identifies a **SampledField**. It is of type `SpId` and is a required attribute. The mathematical value of a **SampledField** is the value and dimensionality of the field itself, and can be bound to a **Parameter** by using a **SpatialSymbolReference**. If used in conjunction with the SBML Level 3 "`arrays`" package, it can be used and manipulated as if it was an array of the appropriate dimensions, even though its *meaning* is the value of the field at all points within its borders, not just those at the lattice points. However, even without the use of the "`arrays`" package, it can be used in SBML Level 3 Version 1 Core MathML to set the value of a spatially-distributed SBML symbol such as a **Species** or **Parameter**, such as through an **InitialAssignment**, **Rule**, or **EventAssignment**.

The size of the field is assumed to match the axes (the **CoordinateComponent** children) of the parent **Geometry**, and is assumed to be regularly spaced in each dimension, but is not required to be spaced the same way in all dimensions. In other words, if the Geometry defines a 10 cm by 10 cm square, and a SampledField is a 10x5 array, the "`[0,0]`" entry in the array will correspond to the point "`0 cm, 0 cm`" in the Geometry, and the "`[10,5]`" entry in the array will correspond to the point "`10 cm, 10 cm`" in the Geometry. Off-latice points (such as the value at "`9 cm, 9 cm`" in this example) have no direct corresponding value in the **SampledField**, and are determined according to the `interpolationType` attribute, defined below.

When tied to a **SpatialSymbolReference**, regardless of its useage, each **SampledField** still must represent values across the entire **Geometry**. If used in an **InitialAssignment** to assign values to a **Species** that only exists in a particular **DomainType** within the **Geometry**, entries in the **SampledField** that correspond to areas of space not covered by that **DomainType** will simply be ignored. Those values may be set to zero, or could be used in other contexts. For example, a **SampledField** could represent 'the concentration of ATP in the Geometry', and one **InitialAssignment** could be used to apply the field to the species 'ATP in the cytosol' and a second **InitialAssignment** could be used to apply the same field to the species 'ATP in the nucleus', with different values being examined and used in each case.

☞ Lucian: Is the above description reasonable?

3.46.2 The `numSamples1`*,* `numSamples2`*,* `numSamples3` *attributes*

The `numSamples1`, `numSamples2`, and `numSamples3` attributes represent the number of samples in each of the coordinate components. (e.g. numX, numY, numZ) in an image dataset. These attributes are of type `int` and are required to specify the **SampledField**. The samples are assumed to be uniformly sampled.

☞ Lucian: Removed the `samplesLength` attribute, since that information is redundant with the `numSamples` attributes, above (and represents those values multiplied by each other). We could put it back if people thought it was a good check, I mostly see it as a confusing way to possibly produce invalid SBML.

3.46.3 The `dataType` *attribute*

The `dataType` attribute is of type `DataKind` and is required. It is used to specify the type of the data being stored. The value "`double`" is used to indicate double-precision (64-bit) floating point values; "`float`" to indicate single-precision (32-bit) floating point values, and "`uint8`", "`uint16`", and "`uint32`" to indicate 8-bit, 16-bit, and 32-bit unsigned integer values, respectively.

3.46.4 The `compression` *attribute*

The `compression` attribute is of type `CompressionKind`. It is used to specify the compression used when encoding the data, and can have the value "`uncompressed`" if no compression was used, or "`deflated`" if the deflation algorithm was used.

3.46.5 The `interpolationType` *attribute*

The `interpolationType` attribute is type `InterpolationKind`. It is used to specify how values at off-lattice locations are to be calculated. A value of "`nearestNeighbor`" means that the nearest lattice point value is to be returned. A value of "`linear`" means that the value to be returned is the linear interpolation from nearby lattice points, either simple linear in the case of one-dimensional interpolation, bilinear in the case of two-dimensional interpolation, or trilinear in the case of three-dimensional interpolation.

3.46.6 The `Samples` *data*

The text node child of the **SampledField** is where the data for the **SampledField** resides. It is of type `arrayData`, which is defined as whitespace-delimited, possibly-compressed numerical values. Whether or not the data is compressed (and how, if so) is stated with the `compression` attribute, and the type of numerical values included is stated in the `dataType` attribute. The total number of entries in the array can be derived from the `numSamples` attributes, by multiplying them together (if present).

☞ Lucian: I'm once again confused about this–if the data is a space-delimited string, surely its encoding should be 'string' and not 'uint8' or 'double' or whatever? Why do we define things like 'uint8' and 'double' and then not use them? Or am I missing something?

☞ Lucian: Merged the **SampledField** class with its child class, to make things simpler. Does this work for people?

4 Examples

This section will hopefully contain examples employing the Spatial package for SBML Level 3.

5 Interaction with the Required Elements package

The Required Elements package is designed to create a way for a modeler to denote which specific elements of an SBML model have changed due to interactions with a package. The Spatial Processes package can change the mathematical meaning of the SBML core elements **Compartment**, **Species**, **Reaction**, and **Parameter**. If the Required Elements namespace and the Spatial Processes namespace are declared in the same SBML document, the following restrictions apply. Note that these do not apply in a document without the Required Elements namespace declared.

5.1 Compartments

When a spatial geometry is defined in the SBML model, the **Compartment** element may be extended for the spatial package to represent a spatially-defined area with particular boundaries whose mathematical value is defined as its `unitSize` multiplied by the size of the **DomainType** to which it maps. Any **Compartment** with a child **CompartmentMapping** element must therefore have a **ChangedMath** child pointing to the Spatial Processes namespace. Its `viableWithoutChange` attribute may be set to "`true`" if the compartment's size is set with the `size` attribute (setting it through a **Rule** or **InitialAssignment** is illegal for the purposes of the spatial package). For example, the modeling package may precalculate the compartment size and store it in the Compartment size attribute. However, due to slight differences in mesh generation between simulators, the domain size actually used within computations may vary slightly from simulator to simulator, and will thus introduce small errors that may break mass conservation.

A **Compartment** without a child **CompartmentMapping** element remains unaffected by the Spatial package, and must therefore not have a **ChangedMath** child that points to the Spatial Processes namespace.

5.2 Species

Any **Species** with the `isSpatial` attribute set to "`true`" must have a **ChangedMath** child that points to the Spatial Processes namespace, as the model assumes a spatially-distributed level of that **Species**, instead of considering it to be a well-mixed pool. Its `viableWithoutChange` attribute may be set to "`true`" if the model specifies the initialAmount or initialConcentration attributes of the Species or if the initial condition is specified by a Rule or an InitialAssignment.

A **Species** with the `isSpatial` attribute set to "`false`" remains unaffected by the Spatial package, and must therefore not have a **ChangedMath** child that points to the Spatial Processes namespace.

5.3 Reactions

Any **Reaction** with an `isLocal` attribute set to "`true`" must have a **ChangedMath** child that points to the Spatial Processes namespace, as the model treats such a **Reaction** as defining a change in local substrate concentrations over time, instead of as a change in global substrate amounts over time. Its `viableWithoutChange` attribute will therefore almost always be set to "`false`", as the units of the **KineticLaw** have been changed. However, concentration over time can be numerically identical to amount over time in **Compartments** of unit volume; in this situation, the value of that attribute may be set to "`true`". However, this practice is discouraged.

A **Reaction** with the `isLocal` attribute set to "`false`" remains unaffected by the Spatial package, and must therefore not have a **ChangedMath** child that points to the Spatial Processes namespace.

5.4 Parameters

A **Parameter** object with a **SpatialSymbolReference** child does not take its value from its `value` attribute, but rather from the Spatial object with which it is linked. Therefore, all **Parameter** objects with a **SpatialSymbolReference** child must have a **ChangedMath** child that points to the Spatial Processes namespace. Its `viableWithoutChange` attribute may be set to "`true`" if the **Parameter**'s `value` is set, and/or if there is an **InitialAssignment** or **Rule** that sets that value.

Parameter objects with **DiffusionCoefficient**, **AdvectionCoefficient**, or **BoundaryCondition** children, on the other hand, still take their values from the `value` attribute and/or other SBML Level 3 Version 1 Core elements, and remain unchanged by any Spatial construct. Therefore, these and any other **Parameter** elements without **SpatialSymbolReference** children may not be given a **ChangedMath** child that points to the Spatial Processes namespace.

5.5 General

No other SBML Level 3 Version 1 Core element is affected by the Spatial Processes package, and none may therefore have a **ChangedMath** child that points to the Spatial Processes namespace.

A Validation of SBML documents using Spatial constructs

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 Spatial package. We use the same conventions that 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 Spatial package 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 Spatial package 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 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 Spatial package.

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

General rules about the Spatial package

spatial-10101 ☑ To conform to Version 1 of the Spatial package specification for SBML Level 3, an SBML document must declare the use of the following XML Namespace: "`http://www.sbml.org/sbml/level3/version1/spatial/version1`". (References: SBML Level 3 Package Specification for Spatial, Version 1, Section 3.2 on page 8.)

spatial-10102 ☑ Wherever they appear in an SBML document, elements and attributes from the Spatial package must be declared either implicitly or explicitly to be in the XML namespace "`http://www.sbml.org/sbml/level3/version1/spatial/version1`". (References: SBML Level 3 Package Specification for Spatial, Version 1, Section 3.2 on page 8.)

Rules for the extended SBML class

spatial-10201 ☑ In all SBML documents using the Spatial package, the **SBML** object must include a value for the attribute `spatial:required` attribute. (References: SBML Level 3 Version 1 Core, Section 4.1.2.)

spatial-10202 ☑ The value of attribute `spatial:required` on the **SBML** object must be of the data type `boolean`. (References: SBML Level 3 Version 1 Core, Section 4.1.2.)

spatial-10203 ☑ The value of attribute `spatial:required` on the **SBML** object must be set to "`true`" (References: SBML Level 3 Package Specification for Spatial, Version 1, Section 3.2 on page 8.)

General rules about attributes

☞ Lucian: Obviously this will need to be filled out.

Acknowledgments

Work such as this does not take place in a vacuum; many people contributed ideas and discussions that shaped the Groups proposal that you see before you. We particularly thank Frank Bergmann, and the members of the *sbml-discuss* and *sbml-spatial* mailing lists for suggestions and comments.

This work was funded by the National Institutes of Health (USA) under grant [Something?]

References

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

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

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