Keck Graduate Institute

Human-readable Model Definition Language

First Draft
Revision 1

Frank Bergmann (fbergman@kgi.edu)
Herbert Sauro (hsauro@kgi.edu)

12/17/2006

Table of Contents

Human-readable Model Definition Language

This document describes a first draft of a human readable model definition language. While SBML has become the de-facto standard to exchange models between applications, it was never intended to be human readable. A human readable language will be less verbose, while at the same time map to SBML. This will ensure that existing software tools can take advantage of models created with the new language. SBML remains, however, a key technology enabling easy of computer readability and inclusion of additional annotation such as visual layout information and ontology vocabulary.

The new language should be able to deal with modularity so that models can be combined from sub models that can later be assembled to form a full model. We envision that model construction from this new language would be transformed into standard SBML for reading into simulation and analysis software (Figure 1).

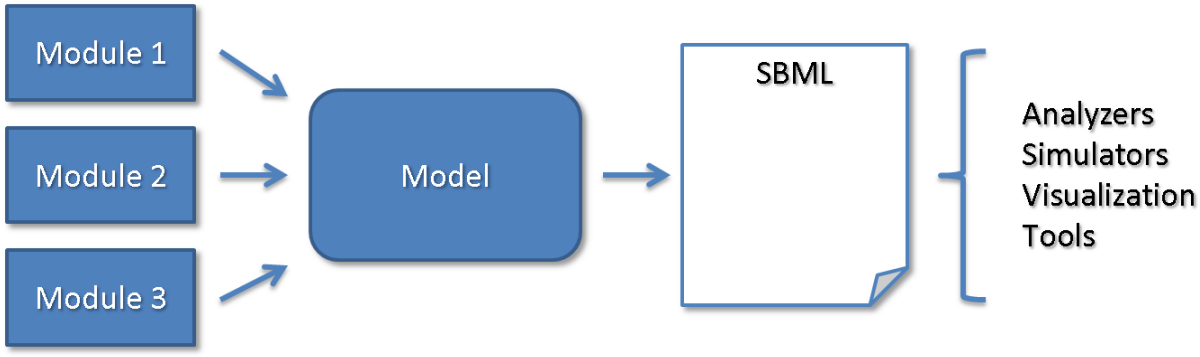

Figure 1: Modules define a Model. This model will be translated to SBML, which then is accepted by existing software tools.

Changes in this Revision

This revised document additionally details how we envision modules, which have been stored in a module library, will be accessed. (c.f. Importing Module Definitions)

We also included the Feedback we have received. We would like to thank Mark Poolman, Brett Olivier, Johann Rohwer and Jannie Hofmeyr for this feedback.

Language Features

The following section will describe suggestions for language elements. It should be noted that for now we decided to have the language features occurring in a predefined order. While this might be strange at first, we believe that a convention over configuration approach will help the language to be more readily understood. The order is as follows:

- Model tag
  - User Defined Function Definitions
  - Compartment Definitions
  - Parameters (constant species (e.g. Boundary Species) and non-species types)
  - Variables (species and non-species types)
  - Module Import statements
  - Module Definitions (sub-models)
  - Assignment Rules
  - Rate Rules
  - Algebraic Rules (Non-conservation law constraints)
  - Kinetic Laws
  - Events
- Initializations

Model

The model tag will define the root element for the language. A script will have exactly one model element, followed by model initializations. The suggested syntax for the model tag is:

```
// model definitions
model <ID>
{
      // full model definition including other language elements
}
// initializations
```

Where `<ID>` is to be replaced with a unique identifier describing the model (e.g.: `MAPK`).

Thus, the model tag is similar to the model element of SBML.

User Defined Functions

Similar to the SBML user-defined functions, these are meant to provide shorthand for wieldy mathematical expressions in rate law expressions. They have the same restrictions as user defined functions in SBML: they can only reference the parameters passed into them, they may access no model variables, they are not allowed to be recursive, and they call only previously defined user-defined functions. Thus as in SBML they can be seen as simple Macros.

Here is the suggested syntax:

```
// User defined function
function <FunctionIdentifier>(<ListOfArguments>)
{
      return <Mathematical Expression>;
}
```

Where `<FunctionIdentifier>` will be the name of the function. `<ListOfArguments>` is a comma-separated list of argument identifiers that can be referenced in the function body. Finally, `<Mathematical Expression>` defines the mathematical expression that the function will provide. As for valid mathematical expressions, we would suggest that we allow predefined functions as described in the MathML subset as allowed by SBML.

Note that the implied return type of the user defined function is of type double.

Compartments

Similar to its counterpart in SBML, a compartment defines a bounded space containing species. It does not need to refer to actual structures of biological cells. Here compartments are placeholders defining where species reside. The proposed syntax is:

```
compartment <ListOfCompartmentIdentifiers>;
```

Where `<ListOfCompartmentIdentifiers>` is a comma-separated list of compartment identifiers. To specify nested compartments, a compartment identifier can be followed by the 'in' keyword followed by a compartment identifier. In this case, the outer compartment has to be defined first:

```
compartment c1, c2 in c1;
```

To assign a volume to these compartments they can be initialized in two ways. Either they are initialized in the `compartment` statement:

```
compartment c1 = 1, c2 in c1 = 1.5;
```

Or they are initialized in the initialization section at the end of the model definition:

```
// model definition
model CompartmentExample
{
   compartment c1 = 1, c2 in c1;
   // rest of model definition
}
// Initialization section
CompartmentExample.c2 = 1.5;
```

Note: If no compartment was declared, a default compartment named 'compartment' with a volume of 1 is assumed.

Parameters (constant species (e.g. Boundary Species) and non-species types)

A corresponding construct to the parameter element in SBML is either a species (with constant and boundary condition defined) or a parameter. As such, they are used to represent constant values to be used in other parts of the model. They are declared similarly to compartments:

```
ext <ListOfParameterIdentifiers>;
```

Where `<ListOfParameterIdentifiers>` is a coma-separated list of parameter identifiers. Parameter identifiers are prefixed by the dollar symbol '$'. The parameter identifier can also be followed by the keyword 'in' and then a compartment identifier. This specifies a boundary species in a designated compartment. To assign a volume to these parameters they can be initialized in two ways.

Either they are initialized in the `ext` statement:

```
compartment c1 = 1;                    // compartment definition
ext $ext1 in c1 = 1, $ext2 = 1.5;      // parameter definition
```

Or they are initialized in the initialization section at the end of the model definition:

```
// model definition
model ParameterExample
{
   ext $ext1 = 1, $ext;
   // rest of model definition
}
// Initialization section
ParameterExample.ext2 = 1.5;
```

We chose the prefix '$' because this will allow us to discern the type of an element (parameter / variable) and then add the corresponding statement automatically.

Note: Parameters can only be modified by assignment rules / rate rules or events.

Discussion: Maybe separate boundary species parameters from kinetic law parameters.

Variables (species and non-species types)

Model Variables are defined similarly to the parameters:

```
var <ListOfVariableIdentifiers>;
```

Where `<ListOfVariableIdentifiers>` is a comma-separated list of variable identifiers. The variable identifier can also be followed by the keyword 'in' and then a compartment identifier. This specifies a floating species in a designated compartment. To assign a volume to these variables they can be initialized in two ways. Either they are initialized in the `var` statement:

```
compartment c1 = 1;                    // compartment definition
var S1 in c1 = 1, S2 = 1.5;            // variable definition
```

Or they are initialized in the initialization section at the end of the model definition:

```
// model definition
model VariableExample
{
   var S1 = 1, S2;
   // rest of model definition
}

// Initialization section
VariableExample.S2 = 1.5;
```

Note: If no compartment has been specified for a variable, a default compartment named 'compartment' with volume 1 will be created in which the species will exist.

Importing Module Definitions

The languages main feature should be the re-use of modules (as defined in the next section) that have been defined previously. At this point, we have not decided on a format of library files. For simplicity's sake, the import definition of a model should be made after the definition of model compartments, variables, and parameters, but before the definition of other modules. This is to avoid any trouble with compartments from the start as well as to avoid name clashes with other modules.

A tentative syntax could look like:

```
import <moduleName> from <source>;
```

where `<moduleName>` is either a name of a module in `<source>`, or a wildcard indicating that the user wants to import all modules from `<source>`. `<source>`, on the other hand, is a reference to an existing library file. It would probably make sense to allow previously defined models also to act as library files.

Module Definitions

A module permits a user to define a sub-model that can be used to compose larger models. The structure of a module is very similar to a model. It can have its own function-, parameter-, variable-, rules-, kinetic-law- and event definitions. These definitions are considered local to the module. A module can also access elements from the global model structure. In particular, it can access modules *previously* defined in the model. At the same time it also specifies a formal input-output relationship.

A module cannot contain other modules.

The suggested syntax is as follows:

```
module <ModuleIdentifier> (<ListOfInputElements>)
{
   // optional local function definitions
   // optional local parameter / variable definitions
   // optional local rule definitions
   // optional ratelaw definitions
   return <ModuleElement>;
}
```

Where `<ModuleIdentifier>` stands for the name of the module, `<ListOfInputElements>` for a comma-separated list of model variable / parameter identifiers, and `<ModuleElement>` for a designated return value of the module. The return value can be thought of as currents.

Perhaps a simple example is in order to define how the module is meant to be used:

```
// model definition
model pathway
{
   // declaration of parameters / variables
   var S1; ext $Xo, $X1;

   // module definitions
   module enzyme(S1, S2)
   {
      J0: S1 + E -> ES; k1*k2*S1*E - k2*ES;
      J1: ES -> S2 + E; k3*ES;
      return J1;
   }

   // rate law definition
   J2: $Xo ->  S1; enzyme($Xo,   S1);
   J3:  S1 -> $X1; enzyme( S1,  $X1);
}

// initialization
pathway.S1 = 1.2;
pathway.enzyme.J0.k1 = 0.2;
```

Having '`enzyme`' defined as a module makes it easy to replace this part of the model. Similarly taking it out of the model provides an easy way to study it in more detail.

Assignment Rules

Assignment rules specify, similar to its counterpart in SBML, mathematical expressions that set the values of parameters. We suggest the following syntax:

```
<ParameterIdentifier> = <Mathematical Expression>;
```

Where `<ParameterIdentifier>` stands for a parameter name and `<Mathematical Expression>` for a mathematical expression (or a call to a user defined function or module).

For example, assignment rules could look like this:

```
// Assignment rules:
// ...to external variables
$Xo = sin (time);

// ...to internal quantities
ph = 7;
k1 = -log10 (pH);
```

Rate Rules

Rate rules specify, similar to its counterpart in SBML, mathematical expressions that determine the rates of change of variables. We suggest the following syntax:

```
d(<VariableIdentifier>) = <Mathematical Expression>;
```

Where `<ParameterIdentifier>` stands for a parameter name and `<Mathematical Expression>` for a mathematical expression (or a call to a user defined function or module).

An example for a rate rule would be:

```
// differential equations
d(S5) = ko*S4;
```

Note: It is important to realize that although a model can be entirely described by rules, we favor the use of reactions / kinetic laws. This makes it easier to import the model into existing software tools.

Algebraic Rules (Non-conservation law constraints)

Algebraic rules define expressions of the form $f(x) = 0$, where x may be a vector. Such expressions may arise in situations that involve computing solutions to systems that have equilibrated. Algebraic rules are generally used when the functions are nonlinear; linear constraints such as conservation laws should be handled internally by the modeling software. In order for algebraic rules to have unique solutions, the number of equations in an algebraic system should be independent and equal to the number of dependent variables. In addition, a user should specify the list of independent variables to solve in any algebraic system. We suggest the following syntax:

```
{ <ListOfEquations> ; <ListOfIndependentVariables> }
```

Where <ListOfEquations> stands for a comma-separated list of equations of the form:

```
f(x) = 0
```

and <ListOfIndependentVariables> is a comma-separated list of model compartments / parameters / variables.

The following single algebraic rule computes the concentration of an equilibrated dimerization reaction:

```
{ Keq*S1^2 + S1 - T = 0; S1 }
```

More details on algebraic rules are in the SBML L2V2 specification.

Kinetic Laws

A kinetic law describes a chemical process (e.g. transformation, transport or binding process) and the speed at which this process takes place. We suggest the following syntax:

```
<ReactantStoichiometry>-><ProductStoichiometry>; <Expression>;
```

Here <ReactantStoichiometry> / <ProductStoichiometry> refers to lists of species (i.e. variables or boundary species) together with their stoichiometries. Here is a basic example defining the chemical reactions for the brusselator reaction scheme:

```
$A       ->       X; k_1*A;
2 X + Y -> 3     X; k_2*X*X*Y;
X + $B   -> Y + $D; k_3*X*B;
X        ->      $E; k_4*X;
```

The <Expression> used to define the speed of the reaction can be either a mathematical expression or a user defined function or a call to a module.

If the model needs access to the fluxes, the kinetic law can be prefixed by a unique identifier followed by a colon. Here are some examples:

```
J:$Xo      -> S1; k1*S1;            //named kinetic law
  S1 + S2 -> S3; k2*S1*S2;
  S3       -> S4; myFunc (k3, S3); //user-defined function/module
```

Inside modules it is recommended to name the kinetic laws, as often the return value of a module will be one of the fluxes.

Events

Events are used to describe instantaneous, discontinuous state changes in the model. The suggested syntax is:

```
@(<BooleanExpression>) : <EventAssignments>;
```

With <EventAssignments> being a comma-separated list of <ElementIdentifier>s and corresponding <Math Expression>s:

```
<ElementIdentifier> = <Math Expression>,
```

<BooleanExpression> is an expression testing values of time or model variables / parameters. <ElementIdentifier> stands for an identifier of a variable / parameter / compartment that will be changed by the expression <Math Expression>. An event should be applied whenever the <BooleanExpression> changes from *false* to *true*. A simple example would be:

```
@(time > 20) : Vmax1 = Vmax1 * 2, Vmax2 = Vmax2 * 2;
```

For more details, please see the event-element description in the SBML L2V2 spec.

Discussion: Alternatively we consider using 'at' instead of the symbol ('@').

Initializations

The previous examples already showed the two methods for initialization of compartments, parameters, and variables. Either the initialization is done immediately during the definition of the element:

```
ext $var2 = 1.5;
```

or the initialization follows after the model definition in a separate block. In this case, every element has to be fully specified with:

- modelIdentifier for compartments, parameters or variables
- modelIdentifier.kineticLawIdentifier for parameters of a selected reaction
- modelIdentifier.moduleIdentifier in order to set values within a module
- modelIdentifier.moduleIdentifier.kineticLawIdentifier for parameters of a reaction in a module

A simple example here would be:

```
pathway.S1 = 1.2;            // model species identifier
pathway.enzyme.J0.k1 = 0.2; // parameter of reaction in module
```

We will also allow the initial assignments to be functions, but it should be noted that these functions are NOT part of the actual model.

Examples

Simple model definition of the Brusselator

```
model brusselator
{
   J0: $A      ->      X; J0k_1*A;
   J1: 2 X + Y ->    3 X; J1k_2*X*X*Y;
   J2: X + $B  -> Y + $D; J2k_3*X*B;
   J3: X       ->     $E; J3k_4*X;
}

brusselator.A = 0.5;
brusselator.B = 3;
brusselator.D = 0;
brusselator.E = 0;
brusselator.X = 3;
brusselator.Y = 3;
brusselator.J0k_1 = 1;
brusselator.J1k_2 = 1;
brusselator.J2k_3 = 1;
brusselator.J3k_4 = 1;
```

model definition of a compartmental model

```
model ce
{
   compartment v1, v2;
   var S1 in v1, S2 in v1, S3 in v2, S4 in v2;

   J0: S1 -> S2; k1*S1*v1;
   // Define the flux through the membrane with given area.
   J1: S2 -> S3; P*Area*(S3-S2);
   J2: S3 -> S4; k2*S3*v2;
}

ce.v1 = 1.0;
ce.v2 = 2.0;
// other variables initialized accordingly
```

Simple modular example

```
// model definition
model pathway
{
   // declaration of parameters / variables
   var S1, S2; ext $Xo, $X1;

   // module definitions
   module enzyme(S1, S2)
   {
      JO: S1 + E -> ES; k1*k2*S1*E - k2*ES;
      J1: ES -> S2 + E; k3*ES;
      return J1;
   }

   // rate law definition
   J1: $Xo ->  S1; enzyme($Xo,  S1);
   J2:  S1 ->  S2; enzyme( S1,  S2);
   J3:  S2 -> $X1; enzyme( S2, $X1);
}

// initialization
pathway.S1 = 1.2;
pathway.enzyme.J0.k1 = 0.2;
```

Similarly, the `enzyme` module could have been defined in a library file and only been imported into `pathway`:

```
// model definition
model pathway
{
   // declaration of parameters / variables
   var S1, S2; ext $Xo, $X1;

   // import enzyme module definition
   import enzyme from myLibrary;

   // rate law definition
   J1: $Xo ->  S1; enzyme($Xo,  S1);
   J2:  S1 ->  S2; enzyme( S1,  S2);
   J3:  S2 -> $X1; enzyme( S2, $X1);
}

// initialization
pathway.S1 = 1.2;
pathway.enzyme.J0.k1 = 0.2;
```

Simple Assignment Example

```
model GMO
{
   compartment cell;
   var C in cell, M in cell, X in cell;

   // assignments
   V1 = C * VM1 * pow(C + Kc, -1);
   V3 = M * VM3;

   J1: $SRC  -> C  ; cell * J1vi;
   J2: C -> $WASTE ; C * cell * J2kd;
   J3: C -> $WASTE ; C * cell * J3vd * X * pow(C + J3Kd, -1.0);
   J4: $SRC-> M    ; cell * (1 - M) * V1 * pow(J4K1 -M + 1, -1);
   J5: M -> $WASTE ; cell * M * J5V2 * pow(J5K2 + M, -1);
   J6: $SRC  -> X  ; cell * V3 * (1 - X) * pow(J6K3  -X + 1, -1);
   J7: X -> $WASTE ; cell * J7V4 * X * pow(J7K4 + X, -1);
}
```

Loose Ends

- Unit definitions: So far, it is assumed that the modeler uses consistent units. At one point we should include unit definitions.
- Vector definitions: It might prove useful to include arrays to define sets of Species / Parameters.
- As indicated before, we will separate boundary species from kinetic parameters by introducing a “par” statement similar to the “ext” statement.
- The current draft specifies the use of the dollar (‘$’) symbol wherever a boundary species is used. This requirement could be relaxed, by requiring the dollar symbol only in the reaction specification, but not in the math expression or the boundary species declaration. So instead of:

```
var S1; ext $X1;
J1: $X1 -> S1; k1*$X1*S1;
```

We would have:

```
var S1; ext X1;
J1: $X1 -> S1; k1*X1*S1;
```

This would still give the visual cue, that a boundary species was used (hence the declaration could be done implicitly by a software tool), while it would be more friendly to users of the language (and less error prone).

- In conversations it has become clear that the topic of annotations (e.g. for compartment, species) will eventually become an issue. We are still thinking about how to best tie annotations into the system. Of course, it will always be possible to use all existing annotation tools, for the SBML view of the model, but that might not be sufficient. Possible ideas include a reserved vocabulary with identifiers in comments.

Feedback

As we strive to develop this draft further, we are looking forward for feedback from other groups. So far two groups have responded in detail with interesting points, which are listed for completeness.

Mark Poolman (ScrumPy)

In general the outline you have proposed looks reasonable, but not beyond improvement. In particular there I think that there are potential redundancies in the proposed syntax which will give rise to classes syntax errors when implemented, that could be eliminated by a simpler language specification. This would also make implementation of parsers somewhat simpler.

A related point is that, to my way of thinking, this looks _much_ too like a conventional programming language, and while it looks OK to people who spend most of their time using such languages, it won't be any too obvious to non-geeks. Admittedly, defining a model is a non-trivial task, but it is considerably simpler than programming, and the syntax of a definition should be comparably simpler, especially if you want it to appeal to the broadest possible audience.

There are also a number of features that, if I've read this correctly, that appear mandatory when they should be optional.

In what follows the term MDL is an abreviation of "Model Description Language", "directive" denotes an optional MDL construct that specifies how some part of the model is to be interpreted, and has the syntax and semantics of a procedure in a conventional programing language, e.g. the ScrumPy MDL directive "Structural()" specifies that any kinetic information is to be ignored. Semantic equivalents of directives exist in other MDLs.

Some specific points:

1) Get rid of, or at least *absolutely* minimise, the need for delimiters, it's perfectly possible to write a parser for this sort of thing without them, they are a distraction both when writing and reading the model, and contain no information about the model itself.
2) The same goes for pre-declaration of metabolites, it contributes nothing but a potential source of error and confusion.
3) There are two approaches to the specifying metabolites as external: either tag the identifier in some way, or by the use of a directive. After a LOT of discussion, I've come to the conclusion that both are equally meretricious, and the choice depends on the context of the problem under consideration. Therefore an MDL should allow both. The potential problem of a metabolite being declared external by both mechanisms is trivially overcome by specifying that a metabolite declared external by any means is external. Specifying that both must be used simply introduces a potential, and unnecessary source of inconsistency.
    a) I'm unconvinced by the use of $ to denote externals. The character itself is never otherwise used in a biochemical context (an MDL is for biochemists, not geeks !), which makes it less intuitive to use. If you read (and therefore think) to yourself e.g. ext_Glc there is less mental effort needed to translate the symbol to the concept of

external glucose, than if you think to yourself "dollar glucose". The mental effort thus saved, small though it may be, might be profitably employed elsewhere. I'd advocate the avoidance of "special" characters in other contexts for the same reasons.

4) If I've read this correctly you seem to be suggesting that all values must be initialised. I think this should be optional, there may be circumstance in which it useful to defer initialisation of some or all parameters and variables to runtime. Fro a more conceptual point of view, I'd argue that initialisation is not part of a model definition, but is an action applied to a model once it has been defined, although it's obviously convenient include initialisations in an MDL.
5) Functions are an obviously good idea. A couple of questions: how is scope resolved ? and can functions contain multiple statements ?
6) Compartments are obviously necessary, but the way they are specified here looks a little confusing. It might be simpler to use a directive like:

```
Compartment(Name, Volume)
// list of reactions continues until the next Compartment()
directive
```

The grammar does not need to define this recursively in order to allow indefinite "nesting" of compartments, rather, the compartmental structure is defined by the presence of transport reactions that can (I think) be defined at any level, e.g. an outline of a leaf, neglecting kinetics:

```
Compartment(Interstitial)

Stomata: x_CO2 + H20 <> HCO3 + x_Proton
#etc.

Compartment(Cytosol)

#bits of glycolysis

Compartment(Chloroplast)

Fixation: HCO3 + RuBP -> 2 PGA
#etc.
```

Then at any point, but most sensibly at the top level, we could specify:

```
Int_Cyt_CO2_tx: Interstitial.CO2 <> Cytosol.CO2

Cyt_Chl_CO2_tx: Cytosol.CO2 <> Choroplast.CO2

TPT: Chloroplast.PGA + Cytosol.Pi <> Cytosol.PGA +
Chloroplast.Pi
# chloroplast triose-phosphate - phosphate translocator
```

However, if HCO3 is not involved in any cytosolic reactions, we might simply replace the first two transporters with:

```
Int_Chl_CO2_tx: Interstitial.CO2 <> Choloroplast.CO2
```

i.e. we are not restricted to a rigidly hierarchical compartmental structure, but we can have one if we want.

Minor points:

Model tags should be optional, if a piece of software really needs one it can default to the something derived from the input file name, or some other obvious strategy.

Reaction stoichiometries should be able to use <> to denote reversible and -> reversible reactions. This is essential if the MDL is to be used for structural modelling.

I'll probably think of more later, but I expect that that's enough for now.

Brett Olivier, Johann Rohwer, Jannie Hofmeyr (PySCeS)

We like the ideas proposed in this draft and are very enthusiastic about the idea of a comprehensive MDL. The following are general comments that should be seen in the context of using this as an input format for PySCeS (while avoiding the SBML translation).

For now I'll avoid discussing scope delimiters and line terminators, as this is more of an implementation issue. In general we like "human readable" to also be "human writable" and therefore less is better. One idea that might be considered is to define keywords as <keyword> : If something like this is implemented with an ordered input file it should be possible to eliminate virtually all scope delimiters.

In general I like the idea of an ordered file, unique model tag and user defined macro functions and compartments. The "in" operator is a very interesting idea and is definitely worth trying out. As far as compartments go , while I agree that initialisation should be able to happen everywhere, personally, I feel they should be intialised where they are defined.

Boundary species/kinetic parameters

First of all, as alluded to in your discussion these should be split, to me, kinetic parameters are reaction properties whereas fixed species are model properties. Following on from this you didn't mention whether the <var> and <par> definitions would be optional or not? I realise that

with the introduction of modules this issue gets more complicated but in principle is it not still the case that with <ext>, a reaction stoichiometry and a rate equation one can automatically generate the <var> and <par> list (which has the useful side of effect of saving the user from getting it wrong)

Finally, the $ prefix, is this still necessary? Fixed species are global to a model and the cost of "identification," as such, should be weighed up against the cost of fixing/unfixing them. I happen to do this quite a lot of this and would ultimately like changing them to be simply a matter of adding/subtracting them from <ext>. My colleagues have stronger feelings about this with the general opinion being that it is redundant to have both a dollar operator and a fixed list.

The module idea is excellent, one question though: will you be able to use previously defined module attributes in a subsequent module. For example, say I defined a set of reactions as a module with its own internal variables, would I be able to access such a local variable in another module?

Assignment/Rate/Algebraic rules

I have a rather limited experience using these rules. While assignment rules make sense, aren't rate rules pretty much made redundant by modules? Depending on the target market for this model description I'd be very much in favour of not using rate rules, while I must admit I haven't fully explored algebraic rules yet.

Events / Loose ends

I would definitely support the use of "at" rather than "@" for no better reason than the latter is the Python decorator syntax. I agree 100% with the idea of assuming units are internally consistent and can be defined somewhere out of the way.

Thanks for letting us know your ideas, I was actually playing with different ideas on how to expand the PySCeS MDL and if there is a common standard (that avoids SBML) so much the better. We are looking forward to future developments in this regard.