BIOINFORMATICS DISCOVERY NOTE

Vol. 30 no. 7 2014, pages 903–907
doi:10.1093/bioinformatics/btt641

Systems biology

Advance Access publication November 8, 2013

SBML and CellML translation in Antimony and JSim

Lucian P. Smith*, Erik Butterworth, James B. Bassingthwaighte and Herbert M. Sauro

Department of Bioengineering, University of Washington, Seattle, WA, USA

Associate Editor: Olga Troyanskaya

ABSTRACT

Motivation: The creation and exchange of biologically relevant models is of great interest to many researchers. When multiple standards are in use, models are more readily used and re-used if there exist robust translators between the various accepted formats.

Summary: Antimony 2.4 and JSim 2.10 provide translation capabilities from their own formats to SBML and CellML. All provided unique challenges, stemming from differences in each format's inherent design, in addition to differences in functionality.

Availability and implementation: Both programs are available under BSD licenses; Antimony from http://antimony.sourceforge.net/ and JSim from http://physiome.org/jsim/.

Contact: lpsmith@u.washington.edu

Received on February 26, 2013; revised on September 15, 2013; accepted on November 3, 2013

1 INTRODUCTION

Biologists who want to write models for their own use tend to use proprietary procedural-based languages such as MATLAB or C to encode their models. But these models are rarely suitable for exchange or re-use. To more readily disseminate the model itself beyond the biological conclusions reached by running the model, a common model exchange format is used. Two popular formats are the Systems Biology Markup Language (SBML) (Hucka *et al*. 2003) and CellML (Lloyd *et al*., 2004).

SBML grew out of the systems biology community, where it was important to encode semantic information in the core structures of the language. As a result, it is particularly well suited to encoding models of processes such as biological reactions. The mathematics may then be derived from these process descriptions as a series of ordinary differential equations (ODEs) by simulation software when desired. ODEs may be encoded directly in alternative constructs, but encoding the model using processes preserves more of the semantics.

CellML started with a different goal—that of encoding all mathematical models, not just process ones. Instead of encoding processes, it encodes all mathematics (such as ODEs) explicitly. As a result, it is not inherently semantically rich in the same way as SBML, but covers a wider scope of models.

SBML and CellML were both designed to be created, read and exchanged by computers and chose the same basic structure (XML) to accomplish this. JSim (Bassingthwaighte *et al*., 2006) and Antimony (Smith, 2009), in contrast, were designed to be created and read by humans, and therefore use human-readable infix notation as the fundamental storage unit.

*To whom correspondence should be addressed.

JSim is a complete modeling creation and simulation tool, and stores its models in the Mathematical Modeling Language (MML) format. Its scope is similar to that of CellML in that it is a way to encode a wide variety of mathematical models. As a whole, it goes beyond the scope of existing CellML simulators in that it can encode models with partial differential equations (PDEs) as well as ODEs with multiple domains (i.e. both time and space).

Antimony's scope is more focused. It was developed solely as a modeling language, and comes only with a standalone editor (QTAntimony), translator (sbtranslate) and a cross-platform programming library (libAntimony) to parse Antimony models and translate them to SBML (in version 1.0) and CellML (in version 2.1). Like SBML, it not only concentrates on modeling processes but also allows you to encode ODEs directly.

CellML and Antimony were both designed to be modular from the beginning, and both allow the user to define and use submodels from a single file or spanning multiple files. JSim's modularity facilities are currently limited and not in common use. SBML was recently extended to become modular, through the Hierarchical Model Composition package (SBML-comp) (http://sbml.org/Documents/Specifications/SBML_Level_3/Packages/comp) that adds modularity to core SBML.

Translators have been developed for both JSim and Antimony to and from SBML and CellML. These translators allow models created in one tool to be used in a wide variety of software tools that can import SBML and CellML (Fig. 1). The issues we had to solve during the development of these tools are inherent to the problem of translating these formats. Other programs that include converters to and from SBML and CellML such as Copasi (Hoops *et al*., 2006), the Systems Biology Workbench (Hucka

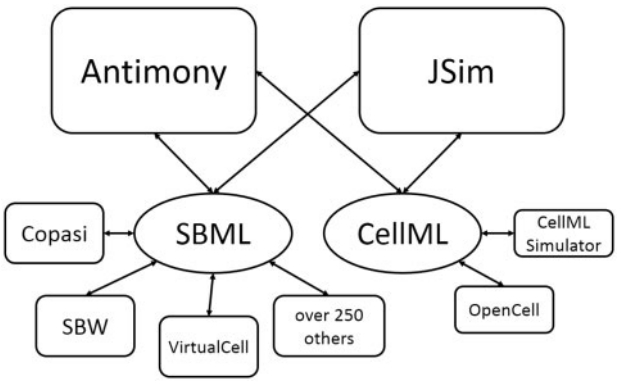

Fig. 1. Flowchart of model development. Models can be developed in a variety of tools such as JSim or Antimony, then translated to SBML and/or CellML for use in other tools

et al., 2002), Virtual Cell (Moraru *et al.*, 2002) and OpenCell (Reeve *et al.*, 2010), as well as direct converters such as CellML2SBML (Schilstra *et al.*, 2006) and SBMLToolbox (Keating *et al.*, 2006), do and will continue to encounter the same issues detailed here as the formats and the software develop.

2 METHODS

Both the JSim program and the Antimony library have internal data models to and from which all translations are accomplished. In addition, JSim can export models to an intermediary XML-based model format called XMML, which contains a processed version of the mathematics from the MML model. XMML was used as an intermediary format for both SBML and CellML export.

Translation to and from SBML in both Antimony and JSim used libsbml (Bornstein, 2008). Antimony was written in C++, and thus was able to use the C++ library directly, whereas JSim, written in Java, used the Java bindings to the library.

Translation to and from CellML in Antimony used the CellML API (Miller *et al.*, 2010). Translation to and from CellML in JSim did not use this API, and instead simply used the native Java XML DOM.

Both CellML and SBML models can be translated to Antimony by importing them into the QTAntimony editor or by using the 'sbtranslate' command-line tool. In addition, the libantimony library can be used to add translation capabilities to other software tools. Cross-platform source code is provided for all tools, as are pre-compiled binaries for Windows.

Similarly, CellML, SBML and Antimony (version 2.1) models can be translated to JSim by importing them using the JSim GUI interface or by invoking JSim from the command line. Source code and pre-compiled binaries are available for Windows, MacOS and UNIX.

Translation between SBML and CellML is possible with both tools by using the native format of each as an intermediary: one can use the GUI or the command-line tools for both JSim and Antimony to import one format and export the other.

The translation algorithms themselves import the original model, then take each individual element and create a corresponding element in the target format. If possible, when no such corresponding element exists, the original elements are processed to better match the target format (such as converting processes to ODEs or flattening any modularity). If this is not possible, the element is omitted and a warning is produced.

3 RESULTS

There were two main obstacles to translating models between different languages: differences in capabilities and differences in design philosophies. Differences in capabilities are usually insurmountable: the bits of the model that have no equivalent construct in the target language simply get lost along the way. Differences in design philosophies can more often be overcome, but the model must often change along the way, sometimes in surprising directions.

3.1 Processes versus math

The most obvious design difference between the various languages is the decision to encode processes as core elements in SBML and Antimony (the 'Reaction' construct) versus the decision to only encode math in CellML and JSim (where everything is an equation). Models in SBML and Antimony can be converted to CellML or JSim, but the semantics of the processes are lost: the overall rate of change for any particular symbol is retained, but the information about which processes it might have participated in that contributed to that rate of change is no longer present.

For translation from CellML and JSim to SBML or Antimony, the math is retained (when compatible) but no processes are created. Only CellML has the possibility of including this information at all (via XML annotations), but even in that language, this capability is rarely, if ever, actually used by modelers. It is also theoretically possible to programmatically analyze rates of change to discover and group like terms, and build up possible coupled processes from this: if one term of 'dB/dt' is –k1*B, and one term of 'dC/dt' is +k1*B, it is not unlikely that there is a single process that converts B to C at a rate of k1*B. But such methods are not foolproof, and usually need to be vetted by a human with first-hand knowledge of the system being modeled. For that reason, we did not attempt to recreate processes from math-only descriptions.

3.2 Modularity

A modular model is one composed of submodels created directly for organizational purposes, collected from other sources or both.

Neither core SBML nor JSim have modularity, so translations to and from those languages were relatively straightforward: the models were flattened (collapsed into a single model with one copy of each entity) before translation to SBML or JSim and translated into a single module when translated into Antimony or CellML.

The one tricky part of flattening a modular model is ensuring that the resulting symbols are unique: in the original model, it is possible to have the same symbol in different submodels without overlap, but when flattened, they need to have unique names. In the Antimony to SBML translator, all names were given hierarchical translations: if a submodel 'A' contained the symbol 's1', it would be always be given the id 'A_s1' in the flattened model. In the CellML to JSim translator, 's1' would be used even in the flattened model unless some other 's1' was already present: only then would 'A_s1' be used instead.

Translations between CellML, Antimony and SBML-comp (SBML plus the hierarchy package) retained the modularity, but with two major differences. The first difference is that CellML has a 'black box' structure, such that it is illegal to connect variables between modules that have not been explicitly exposed as interface variables. Antimony and SBML-comp, in contrast, allow all variables to be connected whether or not they were tagged as interface variables. Although these designs are different philosophically, it was simple to pragmatically translate from one system to the other: translations from CellML were straightforward as all connected variables were obviously already in the interface. Translations to CellML were also straightforward: all connected variables simply needed to be explicitly marked as such.

A more interesting difference was seen when translating Antimony to CellML. Because Antimony may encode processes from which the ODEs are derived, the elements in the interfaces often cannot be used in an explicit-math system like CellML, which contains no such derived ODEs. As Figure 2 illustrates,

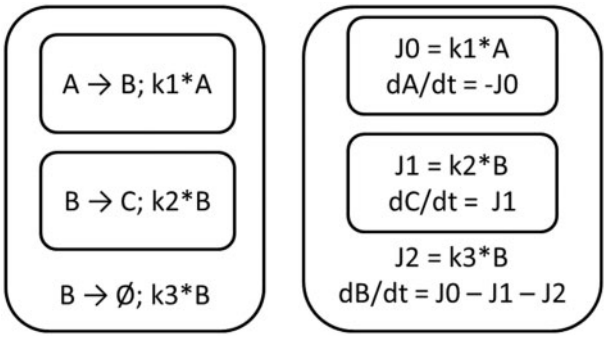

Fig. 2. Schematics of a model encoded in Antimony (left) and in CellML (right). The Antimony model divides processes into submodels; the CellML model divides mathematical terms into submodels

a modular structure is possible in Antimony where a single variable participates in a variety of processes that may span multiple submodels. The rate of change of that variable is then derived from its participation in all processes regardless of the submodel the process came from. When translating this to CellML, it is necessary to connect not the variable itself, but the rates of the processes instead.

This results in a modular structure that encodes the same mathematical model as a whole, but whose submodels actually contain different information than the submodels from which they were derived. In Antimony, the information in a single submodel can be 'here is a rate of change that affects variables A and B'. In CellML, if there are other factors that affect the rate of change of B, you cannot encode that information in a submodel. Instead, it can only encode the information 'here is rate of change J0', and it is up to the containing model to store 'and here is how it affects the variable B'.

3.3 Events

JSim, SBML and Antimony all have constructs called 'Events': moments in the model, established by a trigger, where a discontinuous change occurs to one or more variables in the model (assignments). Events can be used to model phenomena like cell division (Novak and Tyson, 1997), input signals (Shen-Orr *et al*., 2002) and neural spiking (Izhikevich, 2004), all of which can be found as Biomodels (Le Novère *et al*., 2006).

CellML has no event constructs, so models with events cannot be translated fully to CellML. JSim has events that are semantically different from the events in SBML and Antimony, so it is impossible to translate between them: in JSim, events are elements with a trigger and one or more assignments. For every time step at which the trigger is true, the assignments are carried out, meaning that several assignments may occur before the trigger transitions to 'false' again. In SBML and Antimony, events are also composed of a trigger and one or more assignments, but the assignments are only carried out at each moment within the simulation that the trigger transitions from false to true. As SBML does not encode within the model the idea of a 'time step', a JSim event is impossible to encode. Similarly, as JSim does not yet have the concept of an assignment that happens exactly once, SBML events are impossible to automatically translate to JSim. Events of this type are planned for future releases of JSim.

Even translating events between SBML and Antimony proved somewhat difficult when the semantics of events were greatly expanded in SBML Level 3. In the end, several new constructs were added to Antimony to preserve this information. These include:

- *Delay*: an expression that indicates how much simulation time is to pass before carrying out the event assignments.
- *Assignment formulas*: The formulas used in event assignments either use values from the moment the event was triggered or from the moment the event assignment is carried out.
- *Persistence*: Once triggered, the event assignments either must be carried out, or, if the trigger transitions to 'false' again, must not be carried out. This allows event cancellation, particularly when used in conjunction with delays.
- *Firing when t = 0:* Events may be considered to have 'transitioned from false to true' at the start of the simulation, or it may only transition after the simulation start.
- *Priority:* When multiple event assignments are being carried out at once, the event with the highest priority is to be carried out first. Events with identical priorities must be carried out in a random order with respect to each other.

Antimony was unique in that it targeted SBML specifically, so adding special constructs for all of these different nuances was both feasible and desirable. For modeling languages that develop independently, however, it is hard to imagine them coming up with the exact same set of constructs that would translate to and from that system. It makes SBML an expressive language for models that rely heavily on different kinds of events, but simultaneously makes those models less likely to be able to be translated to other languages.

3.4 Units

In Antimony and SBML, units are expected to be consistent. They can be used in model validation, but cannot affect the mathematical calculations in the model. CellML and JSim both have automatic unit conversion (which can be turned on or off in JSim). This means that adding 0.01 l + 10 ml gives you 20 ml in JSim and CellML, but has a unit validation error in SBML and Antimony. The different expectations come from the former assuming that models may have been assembled from several sources, making unit discrepancies expected, versus the latter assuming a model came from a single creator, making unit discrepancies likely to be mistakes.

Translations between JSim and SBML are most affected by this difference. The JSim to SBML translator must pre-convert the equations into a consistent set of units before exporting to SBML. Fortunately, this routine already existed in JSim as part of the MML parsing routines, so XMML output is already unit-consistent. The SBML to JSim translator must be sure to turn off automatic unit conversion in JSim, to ensure that if the SBML model had unchecked units, the JSim version will still produce the same mathematical results.

The CellML to Antimony translator is deficient here—because the CellML API had no automatic unit conversion routines provided, the Antimony translation simply drops the unit definitions, effectively changing the mathematics of any model that relied on automatic unit conversion.

3.5 Math handling

Differences also exist in the different languages as to what math each can handle. SBML explicitly defines a subset of MathML2.0 (http://www.w3.org/TR/MathML2/) it allows in its models, so anything in JSim that uses math outside of that subset (such as <int> (integral), <sum>, and <diff>) is not translatable to SBML, and must be dropped from the model. Conversely, a few elements of that subset have no current equivalent in JSim (such as <factorial> and <xor>), so SBML equations with these elements cannot be expressed in JSim. CellML officially accepts the entirety of MathML, making it and JSim more closely compatible. However, there are a few built-in mathematical constructs in JSim like 'random' that have no MathML equivalent and must be dropped from both SBML and CellML. As these languages develop, more capabilities are planned: JSim will add factorial() and xor() functions in future releases, and the 'distributions and ranges' package for SBML (http://sbml.org/Community/Wiki/SBML_Level_3_Proposals/Distributions_and_Ranges) will add distribution functions such as 'random()' to SBML.

Antimony, due to not having to include a simulator and being solely a model definition language, is at an advantage here, as it can borrow the MathML-to-infix routines of libSBML and the CellML API and simply use the result verbatim in the model.

Beyond the use of individual constructs, there are more basic differences in how the languages approach the math at all. JSim includes a robust multidomain simulator that allows the construction and simulation of PDEs. CellML does allow multiple domains to be used, and can store many parts of a PDE model, but has no way of specifying the boundary conditions, making translations of such models incomplete.

SBML and Antimony have only a single pre-defined domain (time). This makes models with multiple domains impossible to translate fully to SBML or Antimony. Partial translation is accomplished by choosing the single domain most likely to represent time (e.g. the one having base units of 'seconds') translating that, and dropping references to any other domains.

It is also possible in JSim to encode models with references to actual computational code (Java or Fortran routines, for example). These models are obviously not computationally translatable to any other language. Even if one had access to the source code, it would still be necessary to translate procedural code to something declarative, which is often problematic at best: there is no accepted universal standard for exchanging arbitrary data structures or language embedding.

3.6 Model versus experiment

CellML, Antimony and SBML all define models and allow various software tools to define the simulation experiment desired. JSim defines both a model and a simulation experiment within MML. It is, therefore, necessary when translating a model to MML to know what kind of experiment is desired. In our translations, we assume the experiment is a simple deterministic time series with a user-defined end point, but it would be possible to translate to other experimental scenarios such as reaching a steady state or performing a stochastic analysis.

Conversely, some simulation experiments involve parameterization of variables, which may be stored separately from the model (as they are in JSim), and therefore not exported during translation. In the future, SED-ML, the Simulation Experiment Description Language (Köhn and Le Novère, 2008) could be used to translate this part of JSim models.

3.7 Structural rules

CellML, Antimony, JSim and SBML-comp all provide ways of referencing other files from within a single file. This is done in fairly idiosyncratic ways in each system, and is best handled by the native libraries for each. In our case, Antimony and JSim handle their own 'include' statements (and JSim's exported XMML models already include any information from externally referenced files), and because the Antimony/CellML translations used the CellML API, the complications of finding and parsing the referenced files were handled by that library, though the translator did have to handle submodels from the base file differently than submodels from external files.

For the CellML to JSim translator, however, the CellML API was deemed too awkward to use and distribute in Java, so the Java XML DOM was used instead. For all other CellML features, this was adequate, but finding and parsing the extra files from multiple-file models proved to be overly complicated, was pushed to a design goal of a later version of JSim.

For SBML-comp, libSBML handles import routines for externally defined models on the same file system, so these routines were used to create single modular Antimony files.

4 CONCLUSIONS

Models translated to JSim and Antiony from the originals at http://models.cellml.org/and from the curated branch of http://biomodels.net/ (Le Novère *et al.*, 2006), together representing the vast majority of all publically available models in the CellML and SBML formats, can be found at http://antimony.sf.net/antimony-translations. Table 1 summarizes the results. Despite the differences discussed earlier in text, >90% of the models were fully translatable, and >98% were partially translatable. The differences in the number of CellML models reflects a different starting set of models unrelated to the translation efforts

Table 1. Translation results

Language	CellML Models	Full	Partial	Biomodels	Full	Partial
Antimony	787	719	787	366	366	366
Jsim	910	840	884	366	284[a]	366

Note: The total number of models translated from each source, with the number of fully and partially translated models to Antimony and JSim.
[a]All biomodels that were not fully translatable to JSim included the SBML event construct.

Table 2. Language feature comparison

Language	Math	Processes	Modularity	Events	Unit	Text	XML
Antimony	+	+	+	+	+	+	import
CellML	+	import	+	−	+	import	+
JSim	+	+	import	−	+	+	import
SBML	+	+	import	+	+	import	+
SBML-comp	+	+	+	+	+	import	+

Note: Each language feature is designated supported ('+'), not supported ('−') or whether a model with that feature can be imported to that language ('import').

above: of the full set of 1055 hosted models, 787 were parsable by the CellML API (used by Antimony) and 910 were CellML version 1.0 models, paresable by JSim.

Table 2 summarizes the differences in features of the four languages. When there is clear overlap of features in different modeling languages, it is relatively straightforward to translate concepts between them. But any modeling language is going to have areas of specialty or concepts it handles in idiosyncratic ways, and those areas present unique challenges for those wanting to translate models between languages.

One of the most interesting differences between the languages that we found when developing these translators was how different modular design looked in the math-only CellML versus the process-based Antimony and SBML-comp. As the latter languages develop, and the research community begins to build models with them, we predict that those models will be fundamentally different in design from those written in CellML, and their modular design will end up being almost entirely incompatible with each other. This could be mitigated by curated hand-translations of the models between the different systems.

But automatically translated models from one system will be hard to use when translated to the new system without adopting the original design scheme, re-writing the model or using some sort of interface layer that converts one design to the other. It could be argued that as the computational results are the same in modular and flattened models, the main use of a modular design is the design itself, and how well it affords a more comprehensible analysis by human readers. As such, a translation of a process-based design to a math-based design loses its most desirable feature: the thought that went into presenting the original design to other researchers.

In the future, we expect the design of most modular models to be entirely either process-based or math-based, and that the exchange and re-use of models between these two paradigms will be low. To promote the exchange and re-use of a particular model, it may become necessary to hand-design both a process-based and a math-based version of the model, so that researchers in both communities will be able to use the version that best fits with the rest of the models they use.

ACKNOWLEDGEMENTS

The authors would like to thank the developers of libsbml and the CellML API for many productive conversations on how best to use their software.

Funding: NIH (grants 1R01EB08407 and 1R01GM081070-01).

Conflict of Interest: none declared.

REFERENCES

Bassingthwaighte,J.B. *et al.* (2006) GENTEX, a general multiscale model for *in vivo* tissue exchanges and intraorgan metabolism. *Phil. Trans. R. Soc. A*, **1843**, 1423–1442.

Bornstein,B.J. *et al.* (2008) LibSBML: an API Library for SBML. *Bioinformatics*, **24**, 880.

Hucka,M. *et al.* (2003) The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, **19**, 524–531.

Hoops,S. *et al.* (2006) COPASI: a complex pathway simulator. *Bioinformatics*, **22**, 3067–3074.

Hucka,M. *et al.* (2002) The ERATO Systems Biology Workbench: enabling interaction and exchange between software tools for computational biology. *Pac. Symp. Biocomput.*, **2002**, 450–461.

Izhikevich,E.M. (2004) Which model to use for cortical spiking neurons? *IEEE Trans. Neural Netw.*, **15**, 1063–1070.

Keating,S.M. *et al.* (2006) SBMLToolbox: an SBML toolbox for MATLAB users. *Bioinformatics*, **22**, 1275–1277.

Köhn,D. and Le Novère,N. (2008) SED-ML—An XML Format for the Implementation of the MIASE Guidelines. *Comput. Methods Syst. Biol.*, **5307**, 176–190.

Le Novère,N. *et al.* (2006) BioModels Database: a free, centralized database of curated, published, quantitative kinetic models of biochemical and cellular systems. *Nucleic Acids Res.*, **34**, D689–D691.

Lloyd,C.M. *et al.* (2004) CellML: its future, present and past. *Prog. Biophys. Mol. Biol.*, **85**, 433–450.

Miller,A.K. *et al.* (2010) An overview of the CellML API and its implementation. *BMC Bioinformatics*, **11**, 178.

Moraru,I.I. *et al.* (2002) The virtual cell: an integrated modeling environment for experimental and computational cell biology. *Ann. N. Y. Acad. Sci.*, **971**, 595–596.

Novak,B. and Tyson,J.J. (1997) Modeling the control of DNA replication in fission yeast. *Proc. Natl Acad. Sci. USA*, **94**, 9147–9152.

Reeve,A. *et al.* (2010) Biological modelling using CellML and MATLAB. *Open Pacing Electrophysiol. Ther. J.*, **3**, 7.

Schilstra,M.J. *et al.* (2006) CellML2SBML: conversion of CellML into SBML. *Bioinformatics*, **22**, 1018–1020.

Shen-Orr,S.S. *et al.* (2002) Network motifs in the transcriptional regulation network of *Escherichia coli*. *Nature genetics*, **31**, 64–68.

Smith,L.P. *et al.* (2009) Antimony: a modular model definition language. *Bioinformatics*, **25**, 2452–2454.