# Supporting the Evolutionary Development of Simulation Tools with MathML

Michael Weitzel Katharina Nöh Wolfgang Wiechert
{weitzel,noeh,wiechert}@simtec.mb.uni-siegen.de
Department of Simulation, University of Siegen
Paul-Bonatz-Str. 9-11, D-57068 Siegen, Germany

## Abstract

Platform independent data exchange plays a central role in the development of modular and hybrid simulation tools. During the construction phase, concept finding and verification of simulation results typically involves analysis in computer algebra systems, comparison with existing prototypes, and visualization or animation. Besides the requirement that data exchange must not harm data integrity, the chosen data format should not be obsoleted by future releases of commercial software packages. W3C's MathML language fits into these requirements and is well suited to link proprietary developed software components and commercial software packages for a moderate programming effort. This contribution discusses the use of MathML and illustrates the concepts by development of a simulation system for Metabolic Flux Analysis.

## 1 Introduction

Simulation tool development for research projects is typically an evolutionary process: in subsequent episodes of research, documentation and programming the outcome of the dialog between developer and user often causes changes in system requirements and even the model structures and simulation algorithms may change from time to time.

Because of limited manpower, and algorithm development in test bed environment, the nascent software is typically a modular and hybrid system – possibly programmed in different programming languages. In spite of these conditions it is desirable to have operating prototypes all the time. The MathML standard [1] supports rapid prototyping and the evolutionary, modular design approach by providing a widely supported interface language for the exchange of scientific mathematical data for a moderate programming effort.

The rest of this paper is organized as follows: the next section introduces the W3C's MathML standard and summarizes its most important aspects. Section 3 is a brief introduction into *Metabolic Flux Analysis* (MFA) as an example application which massively relies on the exchange of algebraic and numerical data between proprietary developed applications and established scientific software.

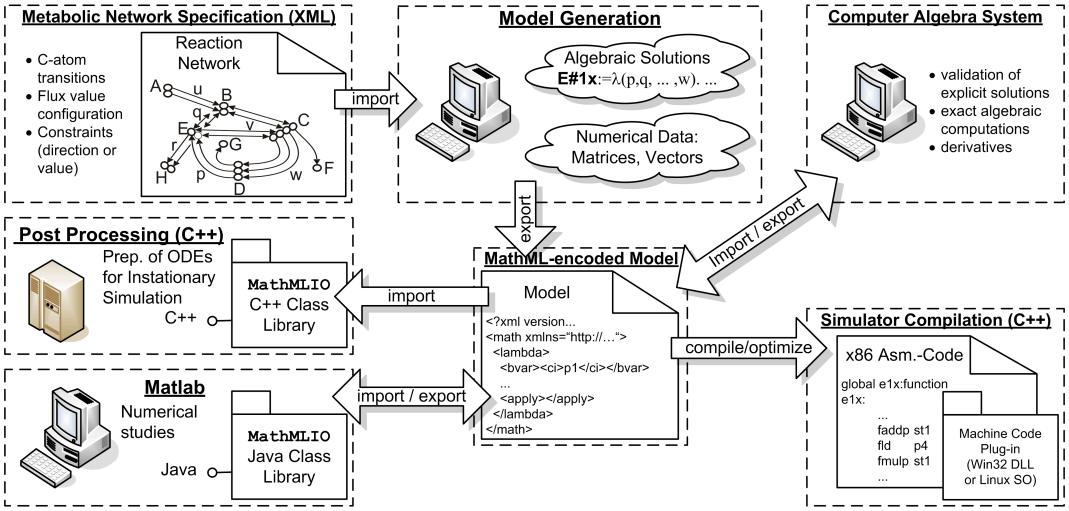


Figure 1: MathML as universal interface language for model exchange between commercial and self-developed application components

## 2 Mathematical Markup Language

MathML, the mathematical markup language specified by the *World Wide Web Consortium* (W3C), is a XML language intended to facilitate encoding, platform-independent exchange, and re-use of mathematical content and notation. The MathML specifications from version 1.0 (1998) through 2.0 $2^{nd}$ ed. (2003) show a fundamental dichotomy: because MathML aims to encode mathematical semantics as well as mathematical typesetting, it is divided into the two independent parts *Content-MathML* and *Presentation-MathML*. While Presentation-MathML aims at a human readable representation, and different types of scientific documentation, Content-MathML is designed to encode mathematical semantics and to facilitate automatic processing [1].

Because Content-MathML's elements show precise semantics, MathML qualifies as an interface language for computer algebra systems (CAS), and other applications dealing with symbolic mathematics – the most popular examples are MAPLE and MATHEMATICA.

Thus, MathML can be very useful for coupling proprietary developed applications and commercial software products from different vendors. Instead of specifying and implementing a proprietary XML language, which can be a complex task and results in an incompatible data format, the programming researcher benefits from the existing MathML framework. The straightforward way to exchange model data between CAS, proprietary developed applications and MATLAB (see section 2.2) prevents time-consuming, error-prone conversion of data formats and alleviates implementation and rapid prototyping.

### 2.1 Storing numerical Data in MathML

Although it might surprise, storing numerical data in a textual format like MathML is not a trivial task. MathML discriminates real, rational, integer and complex number types. Real numbers have to be encoded as decimal fractions and there is no way to store a floating-point number in its lossless binary representation. The programmer has to take care that the textual representation in the MathML file is as short as possible, and represents a correctly rounded, decimal version of the binary floating-point number, from which the original number can be reconstructed. A classical algorithm solving this problem is presented in [3].

### 2.2 Integrating MathML into Software Projects

**Mathematica & Maple** Since version 4.2 in 2002, Mathematica provides full support for MathML 2.0. Maple supported an evolving MathML support since version 7 (2001). In their current versions, both systems provide a convenient support for MathML. MathML structures can be loaded just by calling a library routine of the development environment.

**Matlab** Loading MathML structures into Matlab is not as easy as it is in CAS, because there is no built-in support for MathML. One option is to use the MathML features of the Maple-kernel available in the optional *Symbolic Math Toolbox*. Another option is to install freely available `MathMLIO` library, developed for the ongoing MFA project, discussed in section 3. The `MathMLIO` class library is written in Java (a C++ implementation is also available). MathML import and export in Matlab is handled by two additional wrapper-scripts which use Matlab's excellent Java interface. See [5] for more information.

**Proprietary developed software** Integrating MathML into proprietary developed software generally involves some programming work. If a MathML library like `MathMLIO` is applicable this work can be reduced to a minimum. As there is a Java and C++ version of `MathMLIO` available, it can be used to equip proprietary developed simulation tools with MathML interfaces [5].

## 3 Case Study: Metabolic Flux Analysis

A micro organism's metabolism can be seen as a tiny reactor where enzymes are working as highly efficient catalysts. In *Metabolic Engineering*, scientists seek to employ the metabolism of micro organisms for the economically efficient production of raw materials and drugs.

$^{13}$C Metabolic Flux Analysis (MFA) merges biological knowledge with measured $^{13}$C labeling data to form a mathematical model of the carbon atom flow in a micro organism's metabolism – the *Carbon Labeling System* (CLS). The model can be used to simulate intra-cellular pool's $^{13}$C labeling enrichment. The metabolic flux maps resulting from MFA subserve to compare different strains of micro organisms for productivity and to evaluate and predict the effects of genetic manipulations [6, 4].

### 3.1 Hybrid Simulation Frameworks

MFA is a typical example for a hybrid simulation framework, where highly specialized programs written in C, C++ or Java are mixed with scripts written in Matlab or a CAS. Each of these development environments has its justification by the field is is used for:

**C / C++ / Java** Because compiled code is usually much faster than equivalent programs run by the Matlab interpreter, it is preferred for high-performance computations and simulations of large systems. The iterated solution of CLS during an optimization loop for flux estimation is a typical example. Other examples are parallel cluster applications, where a cluster's specific communication libraries have to be used, and experimental design, a computationally expensive statistical analysis, where an optimization algorithm searches the specific input substrate mixture which results in the most expressive measurement data

**Matlab** Matlab's strengths are a simple and powerful syntax and a huge library of highly elaborated algorithms and toolboxes. In the context of MFA, Matlab is typically used for prototypical implementations – e.g. the test of new numerical algorithms or the simulation and validation of small scale metabolic models – like the instationarity simulation of small models using ODEs. Other typical application fields are one-time computations, like statistical preprocessing of measured data. Matlab is usually avoided when computations rely on elaborate data structures, because there is no replacement for the concept of the *data reference*, available in other languages. Together with the shortcoming that the language is interpreted, Matlab is unsuitable for the development of larger software where high performance is of importance.

**Computer Algebra Systems** CAS are used for statistics like error propagation analysis and integer computations for identifiability analysis. Other topics are computation and validation of steady state CLS, symbolic derivatives (for validation of C++ simulator results), the analytic computation of the Jordan Normal Form for explicit solution of instationary CLS, analytical eigen-analysis for modal analysis and diverse other functional transformations.

### 3.2 Encoding Algebraic Solutions of CLS in MathML

Being a widely accepted standard for the exchange of algebraic and numerical mathematical data, the MathML language can be used to couple the applications found in stationary MFA. This section discusses a typical example for the usefulness of MathML.

In [2] a new algorithm is presented for the analytic solution of CLS by *path-tracing*. This algorithm is part of a model generator implemented in C++ and produces algebraic solutions for CLS which are stored internally in form of expression trees. Along with this algebraic solutions also some numerical data is generated:

- the stoichiometric matrix $N$, stored in a sparse matrix data structure
- a vector $\boldsymbol{v}$ containing the flux quantities of the network
- a vector $\boldsymbol{b}$ containing labeling fractions of input pools

- cascaded systems of linear equations (the *cumomer cascade* [7]) of type:

$$\mathbf{0} = {}^{i}\boldsymbol{A}(\boldsymbol{v}) \cdot {}^{i}\boldsymbol{x} + {}^{i}\boldsymbol{b}\left(\boldsymbol{v}, \boldsymbol{x}^{inp}, {}^{0}\boldsymbol{x}, {}^{1}\boldsymbol{x}, \ldots, {}^{i-1}\boldsymbol{x}\right) \qquad i = 1, 2, \ldots \tag{1}$$

  This cascade can be used to solve the system numerically, and to obtain ODEs for investigating the instationary behavior of the CLS.

Instead of using proprietary formats in the serialization of internal data structures for algebraic and numerical data, the whole solution data set (i.e. ${}^{i}\boldsymbol{A}$, ${}^{i}\boldsymbol{x}$, ${}^{i}\boldsymbol{b}$ and algebraic solutions) is exported into platform independent MathML. The data produced in this step is the starting material for different processing steps (cf. fig 1):

- Using a self-developed MathML lambda-expression compiler, the algebraic solutions can be compiled into optimized assembler code. After assembly and linkage the user obtains a highly efficient machine code plug-in for the simulator (a shared object under Linux or a DLL under Win32) which can be loaded at run-time and establishes the computation of labeling fractions with the ultimately fastest possible evaluation speed.
- Algebraic solutions, matrices and vectors can be imported into a CAS for advanced studies.
- Although Matlab without the optional *Symbolic Math Toolbox* does only numerical computations, not only the matrices and vectors stored in the MathML can be imported, but also the algebraic solutions, which are imported as function handles – optionally directly into the users workspace. Other symbolic expressions, e.g. found in matrix elements, are imported as strings in infix notation into Matlab's cell array type.

### 3.3 Compiling Algebraic Solutions into a highly efficient Simulator

Probably the most exciting application of symbolic mathematics is code generation. Due to the limited number of arithmetical operations used in the expressions generated by the algorithm presented in [2], machine code generation is possible with little effort and results in a highly efficient simulator suitable for the Intel x86 FPU. The compiler is written in C++ and uses the C++ implementation of the `MathMLIO` library.

Algebraic solutions are encoded in MathML in the form of lambda expressions which are bound to names using declarations. A lambda expression consists of a list of *bound variables* (i.e. the parameter list) and an algebraic expression built from these variables. Because MathML files may contain arbitrary many declarations, it is possible to encode a complete network solution (i.e. eq. (1) with algebraic solutions) into a single MathML file. The MathML lambda expression compiler is an isolated software component available as a library or stand-alone application. This is a clear advantage because, for the compiler, there is no need to handle decoding and integrity checking of XML network specifications or to solve a CLS.

Compilation techniques used in the compiler include algebraic simplifications and the

reduction of expensive floating-point divisions to a minimum. Because the obtained expressions are typically still highly redundant, as they contain many identical or arithmetical equivalent subexpressions, expression trees are converted into non-redundant DAGs (directed acyclic graphs) – the actual basis for code generation.

# 4 Conclusion

This contribution demonstrated the usefulness of MathML for the flexible and platform independent exchange of symbolic and numerical data in a hybrid development environment. As MathML is a publicly available standard of the W3C it is resistant against proprietary extensions or version changes in commercial software packages. Its simplicity eases implementation of software interfaces as it was done for the open-source library `MathMLIO`. The Java and C++ version of this library can be used to add limited MathML support to Matlab, as well as for exchange of symbolic and numerical mathematical content between proprietary developed simulation tools.

Because `MathMLIO` is currently used only for data exchange between the application components shown in fig. 1, it does not implement the full MathML standard. Future will bring a more complete implementation of the standard and a cooperation with the open-source community.

# References


[1] D. Carlisle, P. Ion, R. Miner, and N. Poppelier et al. Mathematical markup language (mathml) version 2.0 (second edition). *World Wide Web Consortium (W3C)*, 2003. URL: `http://www.w3.org/TR/2003/REC-MathML2-20031021/`.

[2] N. Isermann, M. Weitzel, and W. Wiechert. Kleene's theorem and the solution of metabolic carbon labeling systems. In *German Conference on Bioinformatics 2004, Bielefeld*, 4.–6. October 2004.

[3] Guy L. Steele Jr. and Jon L. White. How to print floating-point numbers accurately. In *SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25(6), pages 112–126. ACM, 1990.

[4] W. van Winden. *$^{13}$C labeling technique for metabolic network and flux analysis*. PhD thesis, Kluver Laboratory for Biotechnology: TU Delft, 2002.

[5] M. Weitzel. MathMLIO – integrating mathml in matlab. *Dept. of Simulation, University of Siegen*, 2005. URL: `http://www.simtec.mb.uni-siegen.de/~weitzel/mathml.php`.

[6] W. Wiechert. $^{13}$C metabolite flux analysis. *Metab. Eng.*, 3:195–206, 2001.

[7] W. Wiechert, M. Möllney, N. Isermann, M. Wurzel, and Albert A. de Graaf. Bidirectional reaction steps in metabolic networks. Part III: Explicit solution and analysis of isotopomer labeling systems. *Biotechnology and Bioengineering*, 1999.