A Transformation Tool for ODE Based Models

Ciro B. Barbosa, Rodrigo W. dos Santos, Ronan M. Amorim,
Leandro N. Ciuffo, Fairus Manfroi, Rafael S. Oliveira, and Fernando O. Campos

FISIOCOMP, Laboratory of Computational Physiology
Department of Computer Science - Universidade Federal de Juiz de Fora (UFJF)
PO Box 15.064 - 91.501-970 - Juiz de Fora - MG - Brasil
{ciro, rodrigo}@dcc.ufjf.br,
ronanrmo@ig.com.br, leandro@areaweb.com.br, fayrus@gmail.com,
rsachetto@gmail.com, fernando.ocampos@terra.com.br

Abstract. This paper presents a tool for prototyping ODE (Ordinary Differential Equations) based systems in the area of computational modeling. The models, tailored during the project step of the system development, are recorded in MathML, a markup language built upon XML. This design choice improves interoperability with other tools used for mathematical modeling, mainly considering that it is based on Web architecture. The resulting work is a Web portal that transforms an ODE model documented in MathML to a C++ API that offers numerical solutions for that model.

1 Introduction

This work is within the scope of Computational Modeling of Electrophysiology [1]. Under this area of research, biological cell models are often based on large non-linear systems of Ordinary Differential Equations (ODEs). Nowadays, modern cardiac cell models comprises of ODE systems with tens to near hundred of free variables and hundreds of parameters. Recently, the computational biology community has come out with a XML based standard for the description of cellular models [2]. The CellML standard provides the community with both a human- and computer-readable representation of mathematical relationships of biological components.

In this work we extend the CellML goals with a transformation tool that automatically generates C++ code that allows one to manipulate and numerically solve CellML based models.

The transformation tool described here alleviates several problems inherent to the development, implementation, debugging and use of cellular biophysical models.

The implementation of the mathematical models is a time consuming and error prone process, due mainly to the ever rising size and complexity of the models. Even the setup process of the simulations, where all initial values and parameters are to be set, is time consuming and error prone. In addition, the numerical resolution typically demands high performance computing environments and the programming expertise adds more complexity to this multidisciplinary area of research.

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part I, LNCS 3991, pp. 68 – 75, 2006.

To minimize the above mentioned problems, we have built a systematic transformation process that automatically turns mathematical models into corresponding executable code. The tool is an API (Application Program Interface) generator for ODE Solution (AGOS) [1]. AGOS is an on-line tool that automatically builds up an object-oriented C++ class library that allows its users to manipulate and numerically solve Initial Value Problems based on ODE systems described by the CellML or MathML standard. Manipulation here means to set initial values, parameters and some features of the embedded model and of the numerical solver.

Finally, although the AGOS tool was initially tailored to support models described by the CellML standard, currently it works for any initial value problem based on non-linear system of first-order ODEs documented in the MathML standard. Therefore, AGOS is a powerful and useful transformation tool that aims to support the development of many scientific problems in the most diverse areas of research. Biological, ecological, neural and cardiac prototype models are available at the AGOS web page [1] as examples.

In this paper we present the systematization of the transformation process, showing a compromise with implementation correctness. Some other tools described in the Internet [3][4] pursue similar goals. However, the lack of scientific documentation does not allow a proper evaluation and comparison to the AGOS tool.

The next sections present the transformation process, the tool architecture and its components, and some concluding remarks.

2 Transformation Process

The input data for this process is a CellML [2] or a Content MathML [5] file, i.e., XML-based languages. MathML is a W3C standard for describing mathematical notation. CellML is an open-source mark-up language used for defining mathematical and electrophysiological models of cellular functions. A CellML file includes Content MathML to provide both a human- and a computer-readable representation of mathematical relationships of biological components. Therefore, the AGOS tool allows the submission of a complete CellML file or just its MathML subset.

Once submitted, the XML file is translated to an API. The AGOS application was implemented in C++ and makes use of basic computer data structures and algorithms in order to capture the variables, parameters and equations that are embedded in a MathML file and to translate these to executable C++ code, i.e. the AGOS API.

The transformation process consists of identifying and extracting the ODE elements documented in the XML file and generating the corresponding API classes. The conceptual elements in the ODE are: independent variable, dependent variables, auxiliary variables, equation parameters, differential equations and algebraic equations.

The structural elements in the API are methods that can be classified as private or public. The public ones include methods that: set and get the values of the dependent variables (Set/GetVar), set the number of iteration cycles and the discretion interval

(Setup), set the equation parameters (ParSet), calculate the numerical solution via the Explicit Euler scheme (SolveODE).

In addition, the API offers public reflexive functions used, for example, to restore the number of variables and their names. These reflexive functions allow the automatic creation of model-specific interfaces. This automatic generated interface enables one to set any model initial condition or parameter, displaying their actual names, as documented in the CellML or MathML input file.

The algebraic equation solver (SolveAE) is an example of a AGOS private method that is used by the numerical solution method (SolveODE) to obtain the values of auxiliary variables.

Figure 1 synthesizes the relations between the conceptual elements of the ODEs and the basic methods of the API. ODE elements are presented with circles and API methods with rectangles. Arrow directions define the relationship dependency. For instance, algebraic equations depend on parameters, dependent, auxiliary and independent variables; the SolveAE method depends on the algebraic equations; and in turn it influences the auxiliary variables.

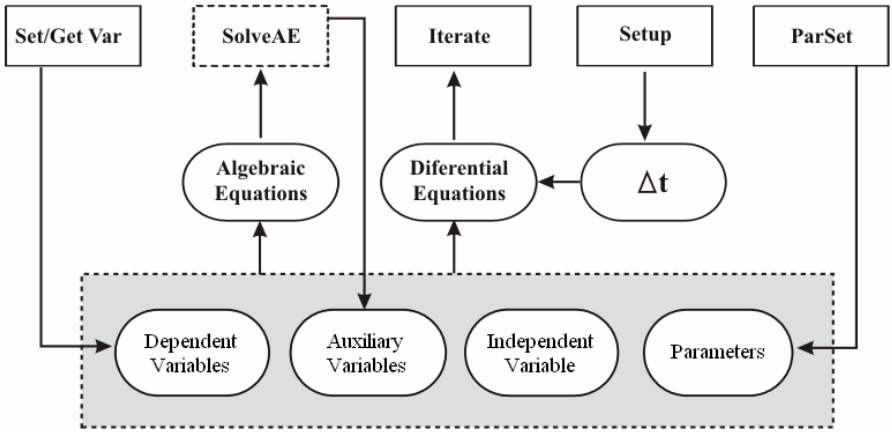

Fig. 1. ODE to API Mapping

The next example better illustrates the transformation process and the relationship between ODE elements and API methods. Consider the following ODE, known as the bistable equation [6]:

$$dVm/dt = – (I\_ion) / Cm, \quad (1)$$

$$I\_ion = a\ (Vm – b)\ (c – Vm)\ Vm. \quad (2)$$

AGOS identifies the ODE elements: Eq. 1 is a differential equation and Eq. 2 is an algebraic equation; Vm, I_ion, and t are dependent, auxiliary and independent variables, respectively; and the ODE parameters are Cm, a, b and c. Using the Forward Euler method a numerical implementation of the above ODE can be written as:

$$Vm^{i} = -\Delta t\, a\, (Vm^{i-1} - b)\, (c - Vm^{i-1})\, Vm^{i-1}/\, Cm + Vm^{i-1}, \quad (3)$$

where Δt is the time step and Vmi is the discretization of Vm(i Δt), for i ≥ 0.

Based on the extracted ODE elements from Eqs. 1 and 2, AGOS generates the following SolveODE and SolveAE methods that implement the numerical solution presented by Eq. 3.

```
void Solveode::solve(int iterations){
   for(i=1; i<iterations; i++)
      Vm[i] = dt* (-calc_I_ion()/Cm) + Vm[i-1];
}
double calc_I_ion(){
          return a*(Vm[i-1]-b)*(c-Vm[i-1])*Vm[i-1];
}
```

3 Tool Architecture

The translator tool comprises of three basic components: a Preprocessor for XML format, an Extractor of ODE conceptual elements, and a Code Generator. The components are organized as a pipeline. The Preprocessor reads an XML-based file (MathML or CellML) and extracts the content into an array of tree data structures. Every tree of this array is processed by the ODE extractor that identifies the ODE elements and stores them in appropriate data formats. At the end of the pipeline, the Code Generator combines the extracted information to a code template and generates the AGOS API. Fig. 2 presents the tool architecture where the relations between the basic components are illustrated.

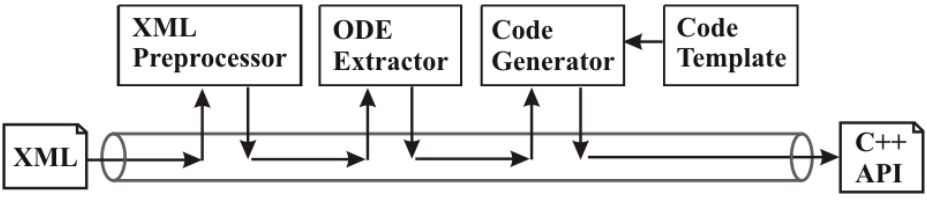

Fig. 2. AGOS Architecture

3.1 XML Preprocessing

The MathML description language uses prefix format on input, i.e. the operators precede the operands. Therefore, a tree is an appropriate structure to store the XML content as it facilitates the identification of the operands and operators. In addition, with the information stored in a tree it is easy to recover the equation formulation with a search in depth procedure. We use the DOM class library [7] to manipulate the XML input files. The Document Object Model (DOM) is an API for HTML and

XML files that provides a structural representation of the document, enabling programs and scripts to access and modify its content [7]. The information is extracted into a tree data structure with equation elements and XML tags. The DOM tree nodes contain information about each operand and operator, besides the equation type (if it is a differential equation or an algebraic one).

To illustrate the preprocessing step, Fig. 3 presents the corresponding Content MathML code and the generated tree of Eq. 1.

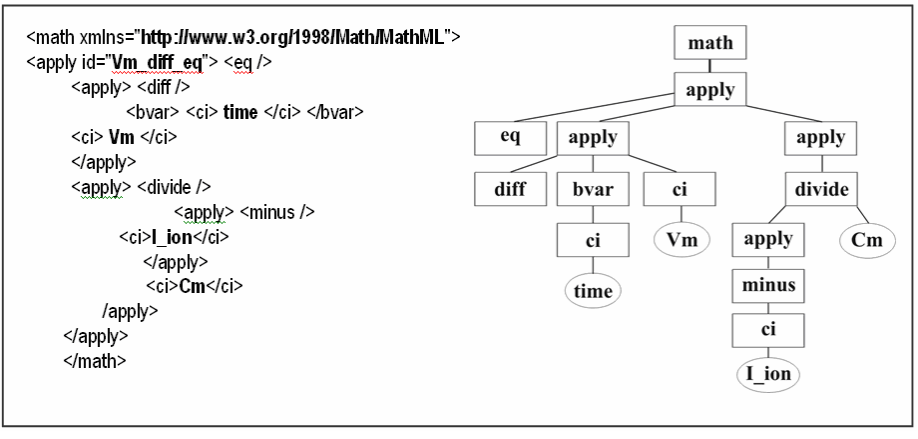

Fig. 3. Content MathML code and tree-like representation

3.2 Extracting ODE Elements

ODE elements are to be used in different parts of the API code. They have to be correctly placed in the code and the corresponding code variables must be properly declared and initialized. Therefore, before the final code can be generated, all the ODE elements must be identified and stored in what we will call here the ODE Element Pool. The identification of all of ODE elements is done with multiple searches in depth in the array of trees. In addition, different ODE elements require different data formats for storage and manipulation. Parameters, dependent and auxiliary variables are each stored in different linked lists. Examples of information stored here are the names, units and default values. The equations are stored in a linked list of trees. This way, the order between elements is preserved as well as information concerning the element type (operand or operator), element characteristic (infixed, prefixed, variable or constant), among others. Figure 5 illustrates the tree that corresponds to Eq. 1. During the creation of this data structures the XML tags are eliminated and the position of operands is standardized. Once the ODE elements are identified and stored in the appropriate data structures, the collection of these structures, i.e. the ODE Element Pool, contains all the necessary information for the Code Generator.

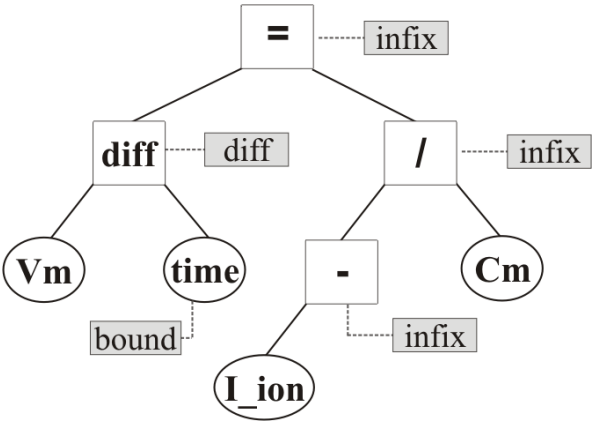

Fig. 4. The tree structure obtained from the MathML

3.3 Generating the AGOS Code

The adopted strategy for code generation is largely based on code templates. The syntactical structure of code templates can be described using formal grammar notation. The algorithm for code generation is inspired in a recursive algorithm for syntax analysis [8]. This algorithm fills in the C++ code template with data contained in the generated Pool of ODE elements. Next we illustrate the AGOS grammar.

```
<api> -> "Class header" "class body" <variables
         declaration>       <solution> <algebraic
         equation set> <GetVar>   <SetVar>    <Setup>
         <ParSet>

<variable declaration> -> "type" <variable> | "type"
         <variable>   <variables declaration>

<solution> ->  "method prototype"  <equation group>

<equation group>  ->  <equation>  |  <equation>
          <equation group>

<equation> ->  <dependent variable (t)> "="
         <discretization> "*"      <expression (t-dt)>
         "+" <dependent variable (t-dt)>

<algebraic equation set> -> <algebraic equation>  |
                       <algebraic equation> <algebraic
         equation set>

<discretization> -> "d"<independent variable>
```

In the grammar, terminal symbols are enclosed by ("). The title of the terminal symbol indicates a piece of the code template. Non-terminal elements are enclosed by (<>). Such elements are defined elsewhere in the grammar or represent functions that

fill in a particular template section. The syntax is recursive, as can be seen in the definition of <equation group>. An example of terminal element is presented below for the terminal "method prototype". This code below is a fixed part of the template code and, therefore, will be used for all APIs.

```
void Solveode::solve(int iterations){
    // solutions' calculation
    for(it_=1; it_ < iterations; it_++){
       // <equation group>
    }
}
```

An example of non-terminal element is presented next.

```
MMLVarListNode *cur = vlVariables;
fprintf(file,"//private variables\n");
fprintf(file,"//private: \n");
while(cur != NULL){
   fprintf(file,"\tdouble *%s;\t //%s \n",cur->name,
           cur->units);
   cur = cur->next;
}
```

The above code shows the implementation of the recursive definition of <variables declaration>. This part of the code generation uses the linked list structure that stores the dependent variables (linked list vlVariables) to dynamically generate the variable declaration of the AGOS API. The resulting code is:

```
//private variables
private:
         double *Vm;
```

4 Conclusions

In this work we described AGOS, a transformation tool that automatically generates executable code that solves and manipulates mathematical models described by initial value problems based on non-linear systems of ODEs and documented in the MathML or CellML standards. The support provided by this systematic transformation process aims on reducing the time during the various phases of scientific model development, implementation, debugging and use.

The AGOS Tool is available at [1], from where it is possible to download the API source-code. The AGOS API can also be used online via a web application, which uses the generated API to solve the ODE system and to visualize the results. Via a dynamic web form, that uses the reflexive AGOS methods, one is able to set up the ODE parameters and initial conditions of the specific submitted ODE system.

Acknowledgements. We thank the support provided the Brazilian Ministry of Science and Technology, CNPq (processes 506795/2004-7).

References

1. Fisiocomp. Laboratory of Computational Physiology. UFJF, Brazil (2005). http://www.fisiocomp.ufjf.br/
2. CellML biology, math, data, knowledge. Internet site address: http://www.cellml.org/
3. LI, J. and LETT, G.S.: Using MathML to Describe Numerical Computations, http://www.mathmlconference.org/2000/Talks/li/
4. CellML: mozCellML. http://www.cellml.org/tools/mozCellML/mozCellMLHelp/technical
5. W3C: Mathematical Markup Language Version 2.0, http://www.w3.org/TR/MathML2/
6. Keener,J., Sneyd, J.: Mathematical Physiology. Springer, 1 edition, 792p., (1998).
7. W3C, Document Object Model (DOM): http://www.w3.org/DOM/
8. Aho, A.V., Seit, R. and Ullman, J.D.: Compilers Addison Wesley, 500p., (1986).