Function Inlining in Modelica Models

Alessandro V. Papadopoulos [*] **Martina Maggio** [**]
Francesco Casella [*] **Johan Åkesson** [**,***]

[*] *Politecnico di Milano, Dipartimento di Elettronica e Informazione, Via Ponzio 34/5, 20133 Milano, Italy (e-mail: {papadopoulos,maggio,casella}@elet.polimi.it).*
[**] *Lund University, Department of Automatic Control, Ole Römers väg 1, SE 223 63 Lund, Sweden (email: johan.akesson@control.lth.se).*
[***] *Modelon AB, Lund, Sweden (e-mail: johan.akesson@modelon.com).*

Abstract: The equation-based Modelica language allows the modeller to specify custom *functions*. The body of a function is an algorithm that contains procedural code to be executed when the function is called. This language feature is useful for many applications; however, the insertion of a function often prevent model optimizations that require the model to be formulated in purely declarative form by equations only. This paper discusses several non-trivial cases in which the function call and the corresponding algorithmic code can be transformed into an equivalent purely equation-based model, thus allowing further optimization. The inlining algorithms presented in the paper go well beyond the state of the art in commercial and open-source Modelica tools.

Keywords: Function inlining, Symbolic manipulation, Object-Oriented Modelling.

1. INTRODUCTION

Equation-based, Object-Oriented Modelling Languages (EOOMLs) are increasingly being used for the modelling of complex dynamical systems. The key idea of EOOMLs is to describe systems declaratively in terms of differential-algebraic equations, which can then be symbolically processed to bring them in a form suitable for efficient solution. Among these languages, Modelica (Fritzson, 2003; Mattsson et al., 1998) has received much attention, both from an academic and from an industrial perspective.

A notable feature of Modelica, which is lacking in most other EOOMLs, is the possibility to define custom *functions*, beyond the built-in mathematical functions such as *sin*() or *exp*(). Informally (see The Modelica Association (2010) for details), Modelica functions are defined by declaring the input variables, the output variables, optional local protected variables, and an algorithm to compute the protected and output variables from the inputs. The algorithm is written using statements typical of procedural programming languages: assignments, conditional statements, loops. A Modelica model using such functions has therefore a mixed semantics, in part declarative and in part procedural.

This approach is convenient in many applications, where parts of the model are better described in a procedural way; for example, consider the model of a vehicle running on a test track, whose shape in 3D space is described by a suitable algorithm. On the other hand, many symbolic analysis and optimization techniques which are commonly used in EOOLMs require the model to be formulated in terms of equations only: symbolic differentiation, symbolic index reduction, symbolic solution of implicit equations, handling of overconstrained connection equations, and so forth. Furthermore, there might be cases (e.g., optimization applications) in which the Modelica model needs to be translated into some other intermediate modelling language which is purely equation-based and does not allow to define custom functions, e.g., AMPL. It is then worth investigating how and to which extent a Modelica model using custom functions can be transformed into an equivalent one using only equations.

In the context of programming languages, a technique named *inlining* is often used by optimizing compilers, which substitute the call to a function in the code with a suitably adapted copy of the function body; this eliminates the overhead of a function call at the expense of an increased memory usage. Similar techniques can be devised in the context of EOOML, though the task is made harder by the fact that the body of a function algorithm is procedural, while the rest of the code is declarative.

In fact, the concept of *inlining* is well-known in the Modelica community, to the point that some parts of the Modelica Standard Library (e.g., the Multibody and Media libraries) can only be dealt with efficiently if some functions are inlined. Surprisingly, to the authors' knowledge, there is no published paper or technical document that clearly explains how inlining should be performed. Experimental evidence demonstrates that the Dymola tool inlines functions only in the special case when the function has no protected variables and the algorithm is given by a single assignment computing the output as an expression using the inputs. In this case inlining is trivial: it is sufficient to replace the function call with the right-hand-side of the

assignment, changing the formal input variable names into the actual names. All functions in the Modelica Standard Library that need to be inlined belong to this special class, even though this requirement is not explicitly stated.

The goal of this paper is then to outline algorithms to transform Modelica models using custom-defined functions into purely equation-based models, going beyond the current state of the art. The proposed techniques will be demonstrated by a prototype implementation in the JModelica.org [1] platform.

2. OPTIMIZATIONS

It is worth stressing the aim of the proposed translation. The starting point is a model that contains equations and one or more function calls. The aim of this procedure is to obtain another model, the semantic of which is equivalent to the initial one. This second model contains only equations, if possible.

2.1 Assignments

Algorithm 1 Equation translation algorithm

Require: Ordered list of assignments A
Ensure: Set of equations E
  // Initialisation of the sets of translated equations, visited assigned variables and used labels
  $E \leftarrow \emptyset$
  $V \leftarrow \emptyset$
  $L \leftarrow \emptyset$
  for all $i = A(last)$ **to** $A(first)$ **do**
    // LHS(i) and RHS(i) return respectively the left- and right-hand-side of the i-th assignment
    $V \leftarrow V + \{LHS(i)\}$
    if $LHS(i) \subseteq RHS(i)$ **then**
      // The variable on the LHS is present on the RHS
      $L \leftarrow L + \{available(E, LHS(i))\}$ // Append the LHS(i) marked with the first available label
    else
      $L \leftarrow L + \{LHS(i)\}$
    end if
    // Substitutes in the RHS all the elements of V with the corresponding labelled element in L
    substitute(RHS(i),V,L);
    // Substitutes in the LHS of the assignments before the current one the labelled variables
    for all $j \in A \wedge A(j) < A(i)$ **do**
      substitute(A(j),V,L);
    end for
    $E \leftarrow E + \{LHS(i) = RHS(i)\}$
  end for
  return E

First, we consider the special case of a model containing a single function, composed from assignments only. This may seem a trivial case and it is in fact the easiest one to be translated. However, other cases can be reduced to this one, and therefore this procedure is at the heart of the proposed optimisation. As an example, consider the following code.

[1] www.jmodelica.org

```
function f
    input Real x;
    output Real z;
protected
    Real y;
algorithm
    z := y;
    y := x;
    y := y^2;
    z := z+y+x;
end f;
```

Inlining this function into a model that contains a function call to it is straightforward. Whenever the function call occurs, the value of the uninitialized variables (in the example `y`) is undefined and a warning should be shown to the user, since the Modelica language specification does not specify how to handle this case. In the following, we conventionally assign the value zero to all the uninitialised variables.

Furthermore, input and output arguments (in the example `x` and `z` respectively) could be assigned multiple times within the function code, i.e., they can appear in the left-hand-side of assignment statements more than once in an algorithm. As a consequence, dummy variables need to be introduced. A procedure to translate a set of assignments into a correspondent set of equations is proposed in Algorithm 1. Starting from the end of the function, a new label is introduced every time a variable that was already assigned is found in the right-hand-side of an assignment and every statement of the algorithm is translated into the corresponding equation. Note also that dummy variables need to be given names that do not conflict with existing variable names in the model containing the function call. Translating the function code provides the following set of equations

$$\begin{aligned} y_2 = 0; \qquad z_1 = y_2; \qquad y_1 = x; \\ y = y_1^2; \qquad z = z_1 + y + x. \end{aligned} \tag{1}$$

2.2 Loops

In non-trivial cases, a function is not only composed from assignments, but also contains control flow statements. This introduces additional complexity in the translation and in some cases makes the complete inlining into an equation model impossible. A procedure to handle some special cases of loops is outlined in this section.

Loop unrolling is a well-known compiler optimisation technique (Petersen and Arbenz, 2004; Sarkar, 2001), which is used at compile time to reduce the program's execution speed at the expense of the its binary size. In the context of modelling languages, however, this technique can be exploited to translate a loop into a set of assignments, and then into equations with Algorithm 1. Some limitations apply, however. We assume, for the purpose of this work, that the values of the loop iterators can be computed at compile time. More complex solutions may be considered but they would stray from the scope of this work. Suppose, therefore, that a `for` loop contains only assignments and assume that the iterator range is known at compile time. Then, `for` loops like

```
for i in 1:10 loop
```

```
for i in 1.0:1.5:5.5 loop
for i in {1,3,6,7} loop
```

can be *unrolled*, and thereby translating them into a set of assignments. This is done by writing every instruction of the loop explicitly for each iteration. Consider, for example, the case of an array initialisation

```
for i in 1:4 loop
    a[i] := i^2;
end for;
```

The resulting set of assignments is

```
a_1 := 1; a_2 := 4; a_3 := 9; a_4 := 16;
```

where all the elements of the array involved in the loop are treated as scalars and the loop itself is split into a set of assignments, which can then be handled by Algorithm 1 to produce the corresponding set of equations. In the case of nested loops, the procedure must be applied starting from the innermost loop towards the outermost one.

The case of implicit iteration ranges can be treated in the same way. For example, the loop

```
Real x[4];
Real xsquared[:] := {x[i]*x[i] for i};
```

can be rewritten in an equivalent `for` loop like

```
Real x[4];
Real xsquared[4];
for i in 1:4 loop
    xsquared[i] := x[i]*x[i];
end for;
```

and then loop unrolling (as previously described) can be applied. The concept of dynamic loop unrolling also exists, but it requires a Just In Time compiler which can compute at run-time the values of the iterators.

Addressing `while` loops is more difficult. In fact, the condition to exit the loop usually depends on some variables assigned inside the loop code. This is for example the case whenever accuracy is a target (continue the loop execution until the error becomes smaller than a threshold value).

```
while err>errMax loop
    ...
end while
```

It is not possible to *a priori* know how many times the loop will be executed. Nevertheless, the `while` statement can always be translated in terms of `for` loops and `if` statements. Hence, solving the problem of the `if` statement (treated in the next section) and limiting to the condition proposed in the `for` loop case, will lead to the solution of the `while` statement case.

2.3 If statements

The `if` statement is the most difficult statement to be treated and imposes limitations on which functions can be inlined. These limitations depend on the possibility of translating the corresponding guard condition into a "sign" expression. To explain the concepts more clearly, we present some examples. The main assumption is that the expressions used as guards can be used inside the assignment as operands, and assume boolean values. In a C-like way, the boolean values `true` and `false` are mapped respectively into 1 and 0. Consider, for instance, the following program

```
Real w, y, z;
...
// Last assignments of the variables involved in the
    if statement
w := 3;
y := 10;
z := 15;
if a>b then
    y := 3;
    z := 5;
else
    z := 6;
    w := 4;
end if
```

The run-time if branch is necessarily not known at compile time. However, we can write it in the form of assignments as follows.

```
w := 3 + if a > b then 0 else 1;
y := 10 + if a > b then -7 else 0;
z := 15 + if a > b then -10 else -9;
```

In other words, we could consider what happens to the variables in the diverse branches and write that in form of variation assignments, where each variation is multiplied by a zero or a one, depending on the branch condition to be false or true. For each single variable contained in the if statement an assignment is produced in the form:

```
var := initial_value +
       (if_condition)*(if_transformations) +
       (else_condition)*(else_transformations);
```

Generalizing, an if statement containing only assignments

```
x := x0;
...
if cond1 then
    x := x1;
    ...
elseif cond2 then
    x := x2;
    ...
elseif ... then
...
else
    x := xn1;
    ...
end if;
```

supposing we are able to translate the conditions as proposed, can be translated into the following assignment

```
x := x0 + (x1-x0)(cond1) +
     (x2-x0)(not(cond1)(cond2)) + ...
     +(xn1-x0)(not(cond1) not(cond2)... not(condn))
```

2.4 Nested functions

Functions that contains calls to other functions can be handled within our framework, assuming that the conditions for loop unrolling stated in Section 2.2 are satisfied whenever a loop is encountered in the callee and called functions. In such cases, the translation proceeds a two step process. First, using the "classical" inlining approach (von Hagen and Wall, 2006), all the called functions are inlined into the callee(s). To this extent, every variable name is modified by adding a prefix to identify the function it was belonging to, in order to avoid duplicates. When only the single callee function remains, it could be treated using Algorithm 1.

3. CONCLUSIONS AND FUTURE WORK

In this work we proposed the rules for translating a function call into a set of equations used to complement the existing equation model that executed the function call. We overcame some of the previous limitations and evidenced some restriction to the mentioned approach at the same time. The translation could be implemented as a compile function to be called before the optimization of the equation set.

An implementation of the work is under development for the JModelica compiler. The implementation has been tested for a subset of the presented functionalities and preserve the model correctness. Optimal control problems where Modelica code is exported to XML format and then in turn imported into the CasADi (Anderson et al., 2010) package for efficient solution will be used in the future research.

REFERENCES

Anderson, J., Houska, B., , and Diehl, M. (2010). Towards a computer algebra system with automatic differentiation for use with object-oriented modelling. In *Third International Workshop on Equation-based Object-oriented Modeling Languages and Tools - EOOLT 2010*.

Fritzson, P. (2003). *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley.

Mattsson, S.E., Elmqvist, H., and Otter, M. (1998). Physical system modeling with Modelica. *Control Engineering Practice*, 6(4), 501–510.

Petersen, W. and Arbenz, P. (2004). *Introduction to parallel computing*. Oxford University Press, USA.

Sarkar, V. (2001). Optimized unrolling of nested loops. *International Journal of Parallel Programming*, 29, 545–581. URL `http://dx.doi.org/10.1023/A:1012246031671`.

The Modelica Association (2010). Modelica - A unified object-oriented language for physical systems modeling - Language specification version 3.2. Online. URL `https://www.modelica.org/news_items/documents/ModelicaSpec32.pdf`. URL: `https://www.modelica.org/news_items/documents/ModelicaSpec32.pdf`.

von Hagen, W. and Wall, K. (2006). *The Definitive Guide to GCC*. Apress, Berkeley, CA, USA, second edition.