Separate Compilation of Causalized Equations - Work in Progress

Christoph Höger
Technische Universität Berlin
christoph.hoeger@tu-berlin.de

Abstract

Separate Compilation is currently considered impossible for Modelica in practice, because several features of state-of-the-art Modelica compilers currently rely on global information. One prominent example for those features is causalization. This particular feature is very important for the generation of fast simulation code. In this work we show how a post-compilation causalization can fit into the operational semantics of a language like Modelica. We present the semantics of a causalizing language system are together with a prototype. This prototype shows that separately compiled models can form a causalized, thus fast, simulation program.

1. Introduction

Separate Compilation as defined in [5] is the process of creating code fragments that can be used in *any* (type-valid) context. This method is a standard approach in software engineering and allows both code re-usage and modularization. Those features are prominent design goals for Modelica [1], yet no current implementation currently implements a separate compilation model.

As we have already shown in [9], there is no principal reason, why Modelica cannot be separately compiled as a language. The only necessary difference is to interpret the process of flattening as the *operational semantics* of the language. Naturally this means that the process of flattening must then happen at runtime.

Although Modelica thus supports separate compilation, there are several obstacles when it comes to *efficient* code generation: Symbolic methods for the handling of systems of equations seem to naturally demand the presence of those equations in a uncompiled form.

One of those methods is *causalization* as described e.g. in [6]. Causalization can drastically enhance the simulation performance of models with sparse systems of equations. Therefore we consider it a must-have for a decent Modelica implementation. Since we also consider separate compilation a necessity, the question arises, how that particular symbolic method can be implemented *after* code generation. That question shall be answered.

The rest of the paper is organized as follows: We will first give a formal definition of causalization in the context of a tiny modeling language, TinyModelica. We will also present the operational semantics of that language. Subsequently we will discuss (on the basis of linear equations) how models in TinyModelica can be compiled separately and causalized afterwards. Finally we will present our current prototype implementation and demonstrate its performance compared to OpenModelica.

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zrich, Switzerland.

2. TinyModelica Syntax

For the purpose of this document, we will describe a small modeling language, called TinyModelica. We use this language to give a formal overview of the instantiation process of a model and its causalization.

TinyModelica is a very basic language for modeling continuous systems. It does not allow inheritance nor modifications. In fact, it does not even contain parameters or initial value definitions. Its only feature is the definition of models that may contain variables, equations and sub-models.

2.1 Syntax

The syntax of TinyModelica is defined below in EBNF form. We use the $*$ operator to denote repetitions of any (including zero) length and the $+$ operator for repetitions of length ≥ 1.

$$
\begin{array}{lcl}
\mathit{Statement} & ::= & \texttt{causalize} \\
 & | & \texttt{solve (}\mathit{Block}^{+}\texttt{)} \\
 & | & \mathit{Var} \\
 & | & \mathit{Eq} \\
 & | & \mathit{ModelDef} \\
 & | & \mathit{Statement}\ ;\ \mathit{Statement} \\
\mathit{ModelDef} & ::= & \texttt{model id}\ \mathit{Var}^{*}\ \mathit{Eq}^{*}\ \texttt{end} \\
\mathit{Eq} & ::= & \texttt{equation id}\ \mathit{Block}^{*} \\
\mathit{Var} & ::= & \texttt{var id :}\mathit{Name} \\
\mathit{Name} & ::= & \texttt{id}\ [.\ \mathit{Name}] \\
\mathit{Block} & ::= & \mathit{Name} := \mathit{Expr}
\end{array}
$$

A program contains a list of statements. Every model definition contains typed variables (including sub-models) and equations. Additionally, TinyModelica contains two kinds of special statements: `causalize` and `solve`. Their formal semantics will be discussed below.

One important difference between Modelica and TinyModelica is the syntax of equations. While the former one supports nearly mathematical notations (limited only by the restrictions of text editors), like $\dot{x} = f(y,t)$, TinyModelica's equations are defined by sets of *blocks*.

Every block is a variable name together with an expression. Informally, such a block represents *one solution* to an equation. This difference in presentation makes it possible to compile models separately.

Also note that our definition of equations contains a *name*. This is not only necessary for some of the semantics given below. It also seems natural to allow a modeler to at least name some equations. This ease the process of analyzing models as well as simplifying error reporting.

As a special restriction, TinyModelica does not contain a definition of its expression language. This is very important: We want to show, that separate compilation of TinyModelica models is possible. Since we do not give a definition of expressions, there is no way to use any symbolic information in the operational semantics.

This restriction also allows us to ignore important modeling aspects like integration or time events. We simply focus on algebraic equations and assume those features to be part of a decent runtime implementation (in fact, our prototype supports both).

3. TinyModelica Operational Semantics

The operational semantics of TinyModelica are rather simple. Since we do not cover features like inheritance of types, the rules given below should be easy to understand.

In the following, we will use a simple environment structure Γ. This environment can be seen as a partial function mapping names to real values.

$$\Gamma : Name \hookrightarrow \mathbb{R}$$

We also introduce another partial function $\mathcal{M}$, mapping names to model definitions. And $\mathcal{E}$ for the mapping of equations.

$$\mathcal{M} : Name \hookrightarrow Model$$

$$\mathcal{E} : Name \hookrightarrow Equation$$

The *states* of a TinyModelica program are tuples of three such functions:

$$State \subseteq \{(\mathcal{M}, \mathcal{E}, \Gamma) | \mathcal{M} : Name \hookrightarrow Model \ldots\}$$

Updates of the environment are written with the $\oplus$ operator:

$$(f \oplus (x \mapsto y))(z) = \begin{cases} y & z = x \\ f(z) & z \neq x \end{cases}$$

The operational semantics are given in the form of structural operational semantics. One simple example for such a rule would be the statement composition rule:

COMP

$$\frac{\langle (\mathcal{M}, \mathcal{E}, \Gamma)\ \ s\ \ \rightarrow (\mathcal{M}', \mathcal{E}', \Gamma') \qquad \langle (\mathcal{M}', \mathcal{E}', \Gamma')\ \ S\ \ \rightarrow (\mathcal{M}'', \mathcal{E}'', \Gamma'')}{\langle (\mathcal{M}, \mathcal{E}, \Gamma)\ \ s\ ;\ S\ \rangle \rightarrow (\mathcal{M}'', \mathcal{E}'', \Gamma'')}$$

This rule reads as follows: To evaluate a compositional statement $s\ ;\ S$, first s is evaluated. Afterwards S is evaluated in the resulting state.

3.1 Model Collection

The first rule of the operational semantics handles model collection. Essentially, every model in a program is loaded into the environment.

MODEL

$$\frac{}{(\langle \mathcal{M}, \mathcal{E}, \Gamma \rangle,\ \texttt{model}\ x \ldots \texttt{end}\ \ ) \rightarrow \langle \mathcal{M} \oplus (x \mapsto s), \mathcal{E}', \Gamma' \rangle}$$

Note that this just creates an environment $\mathcal{M}$, that can easily be calculated statically. This is an important property for separate compilation. In fact, any decent compiler will probably generate $\mathcal{M}$ as part of it's type-checking process.

3.2 Model Instantiation

Model instantiation is a recursive procedure. Every submodel (including variables) and equations are handled. During this procedure. The model-name is passed through (and attached as a prefix to subsequent sub-models).

INSTMODEL

$$\frac{\begin{array}{c} \mathcal{M}(t) = \ \texttt{model}\ N\ V\ E\ \texttt{end} \\ V = \ \texttt{var}\ x_1\texttt{:}t_1\ \ \ldots\ \ \texttt{var}\ x_n\texttt{:}t_n \\ E = \ \texttt{equation}\ e_1 B_1\ \ \ldots\ \ \texttt{equation}\ e_m B_m \\ V_{new} = \{\ \texttt{var}\ x.\ x_j\texttt{:}\ t_j\ \ | j = 1 \ldots n\} \\ E_{new} = \{\ \texttt{equation}\ x.\ e_i\ app(B_i, x)\ \ | i = 1 \ldots m\} \\ \langle (\mathcal{M}, \mathcal{E}, \Gamma), V_{new}\ E_{new} \rangle \rightarrow \langle \mathcal{M}', \mathcal{E}', \Gamma' \rangle \end{array}}{\langle (\mathcal{M}, \mathcal{E}, \Gamma),\ \ \texttt{var}\ x\texttt{:}t\ \rangle \rightarrow (\mathcal{M}', \mathcal{E}', \Gamma')}$$

This rule uses the *app* function, which basically renames all occurring variables in a set of blocks according to the given instance name:

$$app(\{\ \ n_i := e_i\ \ \}, x) = \{\ \ x.\ n_i := \ \ prefix(e_i, x)\}$$

We assume that *prefix* renames all variables in an expression accordingly. Thus the expression of every block can be seen as a function over the unknowns occurring in it.

The recursive instantiation process stops at `real` variables (which is simply a built-in type to denote unknowns in the model). Those variables are stored in the environment. They are identified with the (arbitrarily chosen) initial value 0.

LOADVAR

$$\frac{}{\langle (\mathcal{M}, \mathcal{E}, \Gamma),\ \ \texttt{var}\ x\texttt{:real}\ \ \rangle \rightarrow (\mathcal{M}, \mathcal{E}, \Gamma \oplus (x \mapsto 0)}$$

Equations are also loaded into the environment.

$$\textsc{LoadEq}\quad \frac{E = \texttt{ equation}\, x\ B}{\langle(\mathcal{M},\mathcal{E},\Gamma),E\rangle \rightarrow (\mathcal{M},\mathcal{E}\oplus(x\mapsto E),\Gamma)}$$

3.3 Causalization

With those simple rules in place, we can now define the process of causalization. Causalization is the process of converting acausal (algebraic) equations into an assignment that solves the equation.

The basic idea of that process is well known from school books. For instance if one would want to solve the equation

$$x + 3 = y - 5$$

for the variable x, knowing y, the corresponding assignment would be

$$x := y - 8$$

Computing such an expression is naturally much faster than solving the system of equations with an iterative method. The hierarchic structure of object oriented languages like Modelica makes it very likely that for many of the variables of a system, such an assignment can be found. So causalization is a very important aspect of any Compiler for such a language.

In the following we will formally define the process of causalization. Note that the general idea is not new (see, e.g. [6]). The most famous algorithm used in practice is probably that presented by Tarjan [13]. The novelty in our approach is that we do not need any symbolic information for the process (recap: we did not even define a language for such information). Therefore we can easily define causalization as part of the operational semantics being executed *after* compilation.

The first element of the process is the *occurrence* relation occ established between equations and variables: Informally, if a variable is used inside an equation both are in the occurrence relation.

$$(\ \texttt{equation id}\ B\ , x) \in occ \Leftrightarrow\ \ x := E\quad \in B$$

That relation is then used to build the *occurrence Graph* $G_{occ} = (E_{occ}, V_{occ})$ for a state $(\mathcal{M},\mathcal{E},\Gamma)$:

$$V_{occ}(\mathcal{M},\mathcal{E},\Gamma) = \text{dom}(\mathcal{E}) \cup \text{dom}(\Gamma)$$
$$E_{occ}(\mathcal{M},\mathcal{E},\Gamma) = \{(v,e) | \mathcal{E}(e)\ occ\ v\}$$

Note, that G_{occ} is bipartite (if $\text{dom}(\mathcal{E}) \cap \text{dom}(\Gamma) = \emptyset$, which we assume). G_{occ} can be used to check some properties for solvability, e.g. the existence of at least one equation for each variable. But since we do not deal with error handling for now, we assume only solvable systems of equations as input.

In a next step, the occurrence Graph is transformed into the *dependency graph*. For this task, we search a perfect match $M(\mathcal{M},\mathcal{E},\Gamma)$ in the *transposed* set of edges of G_{occ}. (In the following we will omit the parameter $(\mathcal{M},\mathcal{E},\Gamma)$ for brevity)

$$M \subseteq E_{occ}^T\ s.t. \qquad \forall v \in \text{dom}(\Gamma)\ \exists! e\ s.t.\ (e,v) \in M$$
$$\wedge \forall e \in \text{dom}(\mathcal{E})\ \exists! v\ s.t.\ (e,v) \in M$$

The existence of such a perfect match is again considered a property of the given system of equations (if it does not exist, there is a under- or over-determined subsystem).

The (directed) dependency graph G_{dep} is defined as follows:

$$G_{dep} = (V_{dep}, E_{dep}) = (V_{occ}, E_{occ} \cup M)$$

Because the edges in M are drawn from equations to unknowns, we can state an important property: Every name in V_{occ} is part of a strongly connected component with exactly 2 nodes.

Lemma 1. *Every Node in V_{occ} is part of a 2-element strongly connected component.*

$$x \in V_{occ} \Rightarrow \exists! y \in V_{occ} s.t. \{(x,y),(y,x)\} \subseteq E_{dep}$$

Proof.

$$\begin{aligned} x \in V_{occ} \Rightarrow \quad & x \in \text{dom}(\Gamma) \vee x \in \text{dom}(\mathcal{E}) \\ \Rightarrow \quad & \exists!(e,x) \in M \vee \exists!(x,v) \in M \\ \Rightarrow \quad & \exists(x,e) \in E_{occ} \vee \exists(v,x) \in E_{occ} \\ \Rightarrow \quad & \exists! y \in V_{occ} : \{(x,y),(y,x)\} \subseteq E_{dep} \end{aligned}$$

□

Such a strongly connected component (consisting of the name of an equation and the name of an unknown) can directly be identified with a block:

$$\begin{aligned} \mathcal{E}(e) = \quad & \texttt{equation}\, e \ldots v := Ex \ldots \\ \wedge \quad & \{(v,e),(e,v)\} \in E_{dep} \\ \Rightarrow \quad & block(e) = block(v) = \ \ v := Ex \end{aligned}$$

Note that $block$ yields exactly one result for every name, if the variable names of all blocks of an equation are pairwise unequal (another static property of the input program, that we (yet) do not care about). The fact that there is *at least one* block for every such pair in the graph follows directly from the construction of E_{occ}. Since we are interested in actually solving a system of equations, we need another kind of graph, the *calculation order graph*:

$$G_{calc} = (\{block(v) \mid v \in V_{occ}\}, \leq_{calc} \subseteq (Block \times Block))$$

Here $\leq_{calc}$ can be seen informally as a dependency between two blocks: $b_1 \leq_{calc} b_2$ means that b_2 depends on the variable calculated by b_1. :

$$\frac{(v,e) \in E_{occ} \qquad b_1 = block(v) \qquad b_2 = block(e) \qquad b_1 \neq b_2}{b_1 \leq_{calc} b_2}$$

For obvious reasons, such a dependency relation needs to be transitive:

$$\frac{b_1 \leq_{calc} b_2 \qquad b_2 \leq_{calc} b_3}{b_1 \leq_{calc} b_2}$$

It might occurr that $b_1 \leq_{calc} b_2 \wedge b_2 \leq_{calc} b_1$. For that reason we introduce $=_{calc}$ to express such a cyclic dependency:

$$\frac{b_1 \leq_{calc} b_2 \qquad b_2 \leq_{calc} b_1}{b_1 =_{calc} b_2}$$

Because $<_{calc}$ is transitive, we can easily lift it to a relation over sets of cyclic dependent blocks $\prec_{calc}$:

$$\begin{gathered} Q_{calc} = V_{calc}/_{=_{calc}} \\ \prec_{calc} \subseteq Q_{calc} \times Q_{calc} \\ B_1 \prec_{calc} B_2 \Leftrightarrow (b_1, b_2) \in B_1 \times B_2 \Rightarrow b_1 <_{calc} b_2 \end{gathered}$$

Lemma 2. *$T_{calc} = (Q_{calc}, \prec_{calc})$ forms a forest.*

Proof. Assume that $\exists B_1, B_2 \in Q_{calc}\ s.t.\ B_1 \prec_{calc} B_2 \wedge B_2 \prec_{calc} B_1 \wedge B_1 \neq B_2$

From the definition, it follows, that $(b_1, b_2) \in B_1 \times B_2 \Rightarrow b_1 \leq_{calc} b_2 \wedge b_2 \leq_{calc} b_1 \Rightarrow b_1 =_{calc} b_2$

But since $B_1, B_2 \in Q_{calc} = V_{calc}/_{=_{calc}}$, by the definition of the quotient set, it follows that $b_1 =_{calc} b_2 \Rightarrow b_2 \in B_1 \Rightarrow B_1 = B_2$. ↯

At that point we can now define the behavior of the `causalize` statement.

CAUSALIZE

$$\frac{\begin{gathered} T_{calc}(\mathcal{M}, \mathcal{E}, \Gamma) = (\{B_1 \dots B_n\}, \prec_{calc}) \\ B_1 \prec_{calc} \dots \prec_{calc} B_n \\ \langle (\mathcal{M}, \mathcal{E}, \Gamma),\ \texttt{solve}\ B_1;\ \dots \texttt{solve}\ B_n\ \rangle \rightarrow (\mathcal{M}', \mathcal{E}', \Gamma') \end{gathered}}{\langle (\mathcal{M}, \mathcal{E}, \Gamma),\ \texttt{causalize}\ \rangle \rightarrow (\mathcal{M}', \mathcal{E}', \Gamma')}$$

Causalization is now built into the operational semantics: First create the calculation tree and then simply walk it in order. More prominently, no symbolic information about expressions was used during this process. Yet we still have to discuss the properties of the `solve` statement.

3.4 Block evaluation

The only semantics left to discuss concern the `solve` statement. Up until now everything in TinyModelica's semantics did not yield actual calculation of simulation data. To fill this gap, we assume that the semantics of the (still undefined) expression language is given via two evaluation functions:

$$\begin{aligned} \mathcal{A}: &\quad State \times Expr \rightarrow \mathbb{R} \\ \mathcal{J}: &\quad State \times Name \times Expr \rightarrow \mathbb{R} \end{aligned}$$

The first rule for solving an equation is rather simple: If the set of blocks to solve has only one element, the block is interpreted as an assignment.

SOLVE SINGLE

$$\frac{B = \{\ x := E\ \} \qquad \Gamma' = \Gamma \oplus (x \mapsto \mathcal{A}((\mathcal{M}, \mathcal{E}, \Gamma), E)}{\langle (\mathcal{M}, \mathcal{E}, \Gamma),\ \texttt{solve}\ (B)\ \rangle \rightarrow (\mathcal{M}, \mathcal{E}, \Gamma')}$$

The actual new value for the variable x depends on the evaluation function. Complete DAEs can be solved by using builtin constructs for numerical integration and time.

SOLVE MULTI

$$\frac{\begin{gathered} B = \{\ x_1 := E_1\ , \dots,\ x_n := E_n\ \} \\ F \in \mathbb{R}^n\ F_i = \mathcal{A}((\mathcal{M}, \mathcal{E}, \Gamma), E_i) - \Gamma(x_i) \\ J \in \mathbb{R}^{n \times n}\ J_{(i,j)} = \mathcal{J}((\mathcal{M}, \mathcal{E}, \Gamma), x_i, E_j), i \neq j \\ J_{(i,i)} = -1 \qquad \Delta x \in \mathbb{R}^n : J\Delta x = -F \\ \Gamma' = \Gamma \oplus (x_1 \mapsto \Gamma(x_1) + \Delta x_1) \dots \oplus (x_n \mapsto \Gamma(x_n) + \Delta x_n) \end{gathered}}{\langle (\mathcal{M}, \mathcal{E}, \Gamma),\ \texttt{solve}\ (B)\ \rangle \rightarrow (\mathcal{M}, \mathcal{E}, \Gamma')}$$

This (last) rule of the operational semantics demands the solution of a linear systems of equations for Δx. Informally, by using Newtons method (see, e.g. [10]), we compute a fixed point for the function $F(\bar{x}) = \mathcal{A}(\bar{E}) - \Gamma(\bar{x})$.

Note, that this interpretation assumes a unique order on the variables in the system. Such an order can easily be computed (we did not include it in the semantics to keep the rules as small as possible). The numerical properties of this evaluation rule completely depend on $\mathcal{A}$ and $\mathcal{J}$.

4. Compiling systems of linear equations

In the following section we will discuss, how TinyModelica can be used to solve systems of linear equations. Especially we will show, how the block expressions must evaluate to yield a valid solution.

4.1 Block triangular form

Consider a linear system of equations: $A\bar{x} = 0$ with $\bar{x} \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n \times n}$. In the following, we assume that x_i is likewise the expression yielding the value of an unknown in the compiled model as a component of the solution vector $\bar{x}$.

For every equation $\bar{a}_i\bar{x} = 0$, we can reduce the sum to the non-zero components $\bar{a}_i\bar{x} = \bar{\alpha}_i\bar{z} = 0$. Now we can create a solution vector $b(i,j)$ for every equation $\bar{a}_i$ and every variable x_j:

$$b(i,j) = (\frac{\alpha_{i,1}}{-\alpha_{i,j}} \dots \frac{\alpha_{i,j-1}}{-\alpha_{i,j}}, 0, \frac{\alpha_{i,j+1}}{-\alpha_{i,j}} \dots)$$

Naturally, we compile every equation into all such blocks. If we evaluate such a block according to SOLVE SINGLE, we have a solution for $\bar{a}_i$. Note, that such a vector directly delivers the partial derivatives. Therefore for any set of blocks $\{(b(i_1, j_1), \dots b(i_n, j_n)\}$, we can calculate F and J as follows:

$$J = F = \begin{bmatrix} b(i_1, j_1) - e_{j_1} \\ \vdots \\ b(i_n, j_n) - e_{j_n} \end{bmatrix}$$

We know that, after evaluating SOLVE MULTI, $J\Delta\bar{x} = -F\bar{x}$. It follows directly, that $F(\Delta\bar{x} + \bar{x}) = 0$. Therefore, after one evaluation, Γ contains a valid solution to $\{\bar{a}_{i_1}, \ldots, \bar{a}_{i_n}\}$.

In the case of multiple sets of blocks to be solved (which should be considered the usual case), from [6] it is known that the calculation order returned by the causalization $B_1 \prec_{calc} \ldots \prec_{calc} B_n$ can be represented as a block matrix in lower triangular form:

$$\left[\begin{array}{cccc|c} J_1 & 0 & \cdots & 0 & 0 \\ A_{21} & J_2 & \cdots & 0 & 0 \\ \vdots & & \ddots & \vdots & \\ A_{n1} & A_{n2} & \cdots & J_n & 0 \end{array}\right]$$

Notably, here $J_i \neq F_i$, since the F_i also take the A_{nm} as inputs:

$$F_i = \left[\begin{array}{cccc} A_{i1} & \cdots A_{i(i-1)} & J_i \end{array}\right]$$

Therefore after every evaluation of SOLVE MULTI, the following equation holds:

$$J\Delta\bar{x} = -F\left[\begin{array}{c} \bar{y} \\ \bar{x} \end{array}\right] = -\left[\begin{array}{cccc} A_{i1} & \cdots A_{i(i-1)} & J_i \end{array}\right]\left[\begin{array}{c} \bar{y} \\ \bar{x} \end{array}\right]$$

This (again), leads to the fact that:

$$\left[\begin{array}{ccc} A_{i1} & \cdots & A_{i(i-1)} \end{array}\right]\bar{y} + J(\Delta\bar{x} + \bar{x}) = 0$$

In other words: applying SOLVE MULTI to systems of linear equations is not much different from applying Gaussian elimination to a block matrix. This shows that this semantic rule can be applied efficiently to systems of linear as well as nonlinear equations.

5. Nonlinear equations

Although we only described systems of linear equations, the proposed method naturally works in the nonlinear case: Instead of directly calculating the block expressions one has to use a decent computer algebra system to generate inverse functions, to generate the block representation of a single equation. It may of course occur that even a computer algebra system does not find a direct solution (or multiple solutions exist). In such a case a numerical method could be used to emulate a direct solution, as long as $\mathcal{J}$ delivers the partial derivatives (which in turn could be derived e.g. by automatic differentiation).

For the operational semantics, the existence of nonlinear equations is no big problem: After a single step of a Newton iteration, in SOLVE MULTI, we need to check the validity of the solution:

SOLVE MULTI RECURSIVE

$$\frac{\begin{array}{c} B = \{\ x_1 := E_1\ , \ldots,\ x_n := E_n\ \} \\ F \in \mathbb{R}^n\ F_i = \mathcal{A}((\mathcal{M}, \mathcal{E}, \Gamma), E_i) - \Gamma(x_i) \\ J \in \mathbb{R}^{n \times n}\ J_{(i,j)} = \mathcal{J}((\mathcal{M}, \mathcal{E}, \Gamma), x_i, E_j), i \neq j \\ J_{(i,i)} = -1 \qquad \Delta x \in \mathbb{R}^n : J\Delta x = -F \qquad \mathbf{F} \approx \mathbf{0} \\ \Gamma' = \Gamma \oplus (x_1 \mapsto \Gamma(x_1) + \Delta x_1) \ldots \oplus (x_n \mapsto \Gamma(x_n) + \Delta x_n) \end{array}}{\langle (\mathcal{M}, \mathcal{E}, \Gamma),\ \texttt{solve (}B\texttt{)}\ \rangle \rightarrow (\mathcal{M}, \mathcal{E}, \Gamma)}$$

SOLVE MULTI STOP

$$\frac{\begin{array}{c} B = \{\ x_1 := E_1\ , \ldots,\ x_n := E_n\ \} \\ F \in \mathbb{R}^n\ F_i = \mathcal{A}((\mathcal{M}, \mathcal{E}, \Gamma), E_i) - \Gamma(x_i) \\ \mathbf{F} \sim \mathbf{0} \end{array}}{\langle (\mathcal{M}, \mathcal{E}, \Gamma),\ \texttt{solve (}B\texttt{)}\ \rangle \rightarrow (\mathcal{M}, \mathcal{E}, \Gamma)}$$

A more problematic case is the distance of the initial values from a solution: The method we use, might diverge, if the initial solution is too far from the next one.

Since this paper is not about numerics, we ignore those problems for now and focus on systems of linear equations, where an exact solution can be computed.

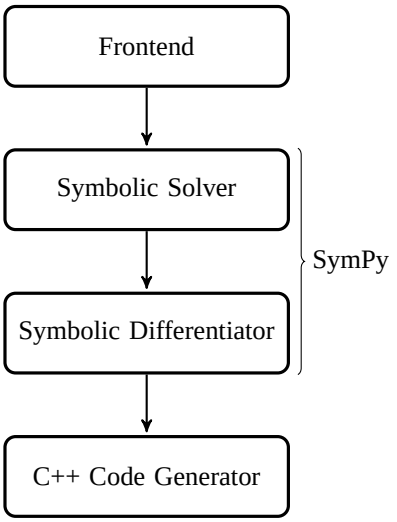

Figure 1. Prototype Compiler Architecture

6. Prototype Implementation

We implemented a TinyModelica runtime system and a Modelica to TinyModelica compiler as a prototype. The runtime is based on the C++ boost libraries [1]. Currently it features a simple explicit Euler integration, Newton's method and an implementation of TinyModelica's causalization and solve statements.

The Compiler is written in Java and translates Modelica source (actually a subset of Modelica) into C++ source files. Those output files contain classes and functions, that implement the model and equation instantiation part of the TinyModelica semantics. Models can be compiled into linkable objects and (linked against the runtime) be used in different contexts.

For the computer algebra part of the compiler we chose SymPy [2], a computer algebra system written in Python

[1] www.boost.org

[2] www.sympy.org

(Figure 1). This construction allows also nonlinear systems (although we did not yet implement the iteration semantics in the runtime by now).

6.1 Example

We tested our implementation with a very simple example for a linear system of equations:

```
model Test1
      Real a,b,c;
      equation
      a + 3*b - 2*c = time;
      2 * a + 2*b + 6*c = 4;
end Test1;

model Test2
      Test1 test1;
      equation
      test1.a + test1.b - test1.c = 0;
end Test2;

model MainTest
    Test2[1000] test2;
end MainTest;
```

We used Modelica as a concrete syntax for our TinyModelica test implementation. The translation of the above shown models into TinyModelica's abstract syntax should be pretty obvious (with the notable exception of the array definition used, our implementation was extended at that point). Below is the definition of `Test2` in TinyModelica syntax as an example:

```
model Test2
      var test1 : Test1
      equation e1
      test1.a := test1.c - test1.b
      test1.b := test1.c - test1.a
      test1.c := test1.a + test1.b
end
```

All three models were compiled separately into one C++ header and source file. By varying the size of the `test2` array, we can demonstrate the value of separate compilation. As a reference, we compiled the model also with OpenModelica 1.7 [7].

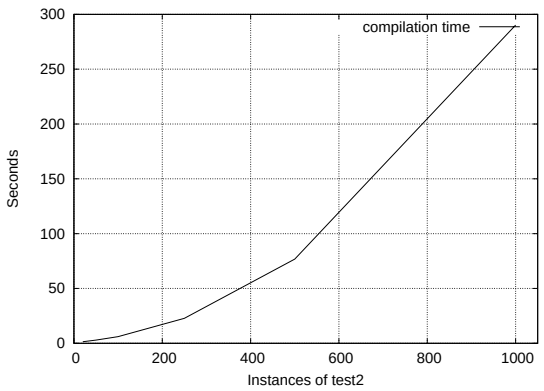

Figure 2. Compilation times in OpenModelica

The process of compiling a model with OpenModelica can be described by two phases: A Modelica front-end, which parses the source code, flattens the model and does all kind of symbolic manipulations. The output of this front-end is some C++ program that is able to solve the generated system of equations. That C++ program is then compiled and linked by a appropriate compiler. We measured the time of the back-end (the GNU C++ compiler in our case), since the front-end was faster by some orders of magnitude.

Figure 2 shows, how the compilation time for the example model grows byond unbearable limits. As mentioned this does *not* imply some performance problem with the omc, since we measured the time that is spent by the compilation of the generated C++ source code. The only problem is the sheer *size* of the generated code (the source file grew to ca. 3.2MB). The time of re-compilation with our prototype remained constantly around one second (which is mainly caused by the heavy use of C++ templates, using pure C could lead to even faster compilation times). In turn our front-end also took around a second which was caused by general slow startup of Java applications, the loading of SymPy from an embedded python interpreter and the prototype style implementation. More important than the actual numbers of our implementation is the fact that they remained *constant* regardless how many instances of `test2` models were instantiated.

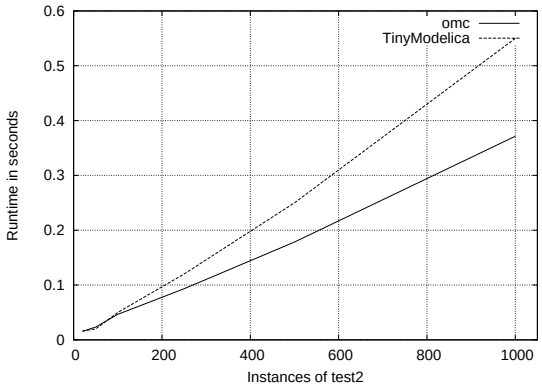

Figure 3. Performance comparison, OpenModelica - TinyModelica

To demonstrate the scalability of our approach, we measured the runtime of a one-second simulation of the test model. Note, that due to the absence of differential equations, the implemented integration method could not affect the runtime behavior. Figure 3 shows our performance comparison. Unsurprisingly, OpenModelica simulated this trivial model very fast, even for many instances. As one could expect the simulation time grows lineary. Although our prototype is not nearly as advanced as the OpenModelica implementation, it shows roughly the same scaling.

7. Conclusion

Separate Compilation of acausal equations is not only possible, but can also be implemented in an efficient equation solver. Although the worst case space consumption grows

from $\mathcal{O}(n)$ to $\mathcal{O}(n^2)$, in practice this compilation scheme saves space.

Obviously compilation time is also drastically improved: Generating all possible variations of a single equation turns out to much more efficient as soon as several instances of a model are active during simulation.

Although it might seem to be a rather technical issue, the development of efficient separate compilation techniques might have a sever impact on upcoming modeling languages: The operational semantics of languages like Modelica are currently built into the tools as interpreters. This does not only limit the size of the compiled models. It also impacts the runtime features a simulation can offer: Features like model structural dynamics [15, 12] are hard if not impossible to implement with current compiler techniques. Additionally, with our method, model libraries can be distributed in a *compiled* form, without any need for encryption methods.

8. Related Work

A different approach to separate compilation of Modelica-like languages is mentioned in [11]. Here it is proposed to extend the language itself to allow for separate compilation. As we have shown, it is not necessary to change anything at the language level as long as a different approach to model instantiation is chosen.

Defining flattening as operational semantics of a modeling language is not a new idea. It has been developed i.e. in [4] and [3]. Although to our knowledge there has been no further research into causalization or index reduction in that context.

The negative effects of a global compilation model for Modelica have been addressed in [14]. Instead of compiling every module separately, another compilation stage is proposed to detect good candidates of code re-usage. This approach clearly has the advantage of generating very good (i.e. fast) code, but seems rather complex to implement. In contrast to our proposal it is an optimization of the state-of-the-art compilation model.

Instead of relying on a classical compilation scheme [8] presents an approach to *completely* move the handling of a system of equations (i.e. symbolic processing, compilation etc.) into the runtime of a host system (Haskell in that case). This idea goes much further than our proposal. The idea of just-in-time compilation for hybrid systems is very appealing (discrete values can become constants and the code can be even more efficient than with ahead-of-time compilation). In form of specialization, a just-in-time compiler like LLVM might as well be useful in our case. Yet there is no reason to postpone the actual symbolic manipulation and code generation into the runtime, as we have shown.

9. Future Work

With TinyModelica we have shown that it is possible to implement a separate compilation scheme with efficient code generation for algebraic equations. Though, some points are still open research questions:

- Index reduction is very important for the solution of most practical models. Currently it is unclear how a compiler might generate code prior to knowing the index of the global system. Usually a index reduction demands a symbolic differentiation up to to the index of the system. Automatic differentiation (see e.g. [2]) might be a solution here.
- The performance difference observed between our prototype and OpenModelica may be caused by the runtime causalization as well as some inefficiencies (e.g. additional function calls in the code). More research here might yield a faster simulation.
- Causalization might be moved into a separate linking phase. This could as well speed up the simulation start as allow earlier error detection (e.g. singular or underdetermined systems of equations).

References

[1] The Modelica Association. Modelica - a unified object-oriented language for physical systems modeling, 2010.

[2] Willi Braun, Lennart Ochel, and Bernhard Bachmann. Symbolically derived jacobians using automatic differentiation - enhancement of the openmodelica compiler.

[3] David Broman. Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments. Licentiate thesis. Thesis No 1337. Department of Computer and Information Science, Linköping University, December 2007.

[4] David Broman and Peter Fritzson. Higher-order acausal models. In *2nd International Workshop on Equation-Based Object-Oriented Languages and Tools,2008*, pages 59–. Linköping University Electronic Press, 2008.

[5] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 266–277, New York, NY, USA, 1997. ACM.

[6] François E. Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer, 1 edition, March 2006.

[7] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The openmodelica modeling, simulation, and development environment. In *Proceedings of the 46th Conference on Simulation and Modeling*, pages 83–90, 2005.

[8] George Giorgidze and Henrik Nilsson. Mixed-level embedding and jit compilation for an iteratively staged dsl. In *Proceedings of the 19th international conference on Functional and constraint logic programming*, WFLP'10, pages 48–65, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] Christoph Höger, Florian Lorenzen, and Peter Pepper. Notes on the separate compilation of modelica. In Peter Fritzson, Edward Lee, François E. Cellier, and David Broman, editors, *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pages 43–51. Linköping University Electronic Press, 2010.

[10] C. T. Kelley. *Solving Nonlinear Equations with Newton's Method*. SIAM, Philadelphia, 2003.

[11] Ramine Nikoukhah. Extensions to modelica for efficient code generation and separate compilation. In *Proceed-*

ings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools, Linköping Electronic Conference Proceedings, page 49â€“59. Linköping University Electronic Press, Linköpings universitet, 2007.

[12] Christoph Nytsch-Geusen and Thilo Ernst. Mosilab: Development of a modelica based generic simulation tool supporting model structural dynamics. In Gerhard Schmitz, editor, *Proceedings of the 4th International Modelica Conference, Hamburg, March 7-8, 2005*, pages 527–535. TU Hamburg-Harburg, 2005.

[13] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *Siam Journal on Computing*, 1:146–160.

[14] Dirk Zimmer. Module-preserving compilation of modelica models. In *Proceedings of the 7th International Modelica Conference, Como, Italy, 20-22 September 2009*, Linköping Electronic Conference Proceedings, pages 880–889. Linköping University Electronic Press, Linköpings universitet, 2009.

[15] Dirk Zimmer. *Equation-based Modeling of Variable-structure Systems*. PhD thesis, ETH Zürich, 2010.