A Transformational Approach to Code Generation for Numerical Simulation: the SimGen System

Jean-Luc Dormoy
EDF-R&D Division MTI/NTIC
1, avenue du Général de Gaulle
92141 Clamart Cedex France
Jean-Luc.Dormoy@edf.fr
http://dormoy.org

Sébastien Furic
TNI - Techniques Nouvelles Informatiques
Sebastien.Furic@tni.fr
http://www.tni.fr

Abstract:
In the last decade, researchers in numerical simulation have worked on defining standards for specifying physical models at the mathematical level of abstraction of *Differential Algebraic Equations*, or DAEs - one can in particular mention the *Modelica* initiative. Today, tools for performing simulation, i.e. checking, interpreting or compiling mathematical models into simulation programs, are being proposed.
It turns out that, despite some peculiarities, general methods from advanced software engineering are applicable to this field. In particular, code generation can be done using frameworks such as the transformational approach.
We describe in this paper such a system, called *SimGen*, that generates simulation code using a set of small freely applicable transformations. Indeed, transformations start from an initial set of differential algebraic equations - DAEs - and yield another set that is suitable for being used by a solver. Solvers are numerical programs that embed generic methods for solving DAEs, provided they satisfy some syntactic as well as semantic constraints. Moreover, in the course of code generation - i.e. equation transformation - some semantic checking can be done, and errors reported, e.g. equation set singularity.
SimGen is being used on real-world large-scale physical models, having more than one thousand equations, and we give experimental results. We also compare SimGen to other similar systems.

Keywords: Numerical Simulation, Differential Algebraic Equations, Physical Modeling, Modelica, Code Generation, Program Transformation

Introduction

Physical models are normally described by sets of differential equations. They are categorized into Ordinary Differential Equations (ODEs), Differential Algebraic Equations (DAEs), and Partial Differential Equations (PDEs), the solution of which is of increasing difficulty. We shall be concerned here by DAEs, which can in general be written in the form:

$$F(x',x,y) = 0$$
$$G(x,y) = 0 \qquad (1)$$

x and y are vectors (of variables), the x's are called state variables, and the y's algebraic. They all depend on time.
DAEs are suitable for « 0D » or pre-discretized « 1D » models (higher dimensions would be theoretically possible, though not practical, one must then consider PDEs).

In early computer simulation, building a computable model meant writing down a specific numerical program for simulating a given physical model. Quickly, it became clear that numerical simulation could be performed by a generic numerical program, called a *solver*, which would take as data a customized model description. Moreover the remaining code, which is specific to some physical model, could be split into components more or less corresponding to technical components, when this notion was clear.

Initially, solvers were very demanding on how to pass it physical models. Beside more or less cumbersome syntactic details - things had very often to be expressed as some FORTRAN code, plus more or less cryptic chunks of data - implicit equations were not accepted. This means that equations have to be symbolically « solved » in some way prior to simulation, e.g. put into the form

$$x' = f(x,y)$$
$$G(x,y) = 0$$

In this case, we say that the system is semi-implicit, or explicit on state variables. Most solvers today accept semi-implicit systems, though some older ones do not accept implicit algebraic variables either. Indeed, this means that some f and G must be given under some form, for example as a Fortran function that computes f(x,y), and for G as a set of assignments resembling fix-point equations and « equivalent » to G - other forms exist as well.
Moreover, it is often necessary to provide the solver with extra symbolic information - still under some imposed form - such as the system jacobian matrix.

Fully implicit solvers exist now, that accept the general system form (1), with some additional syntactic constraints, and sometimes also semantic ones. So, it could be thought that this story has come to an end: it is now possible to directly simulate a physical process from its equational model.

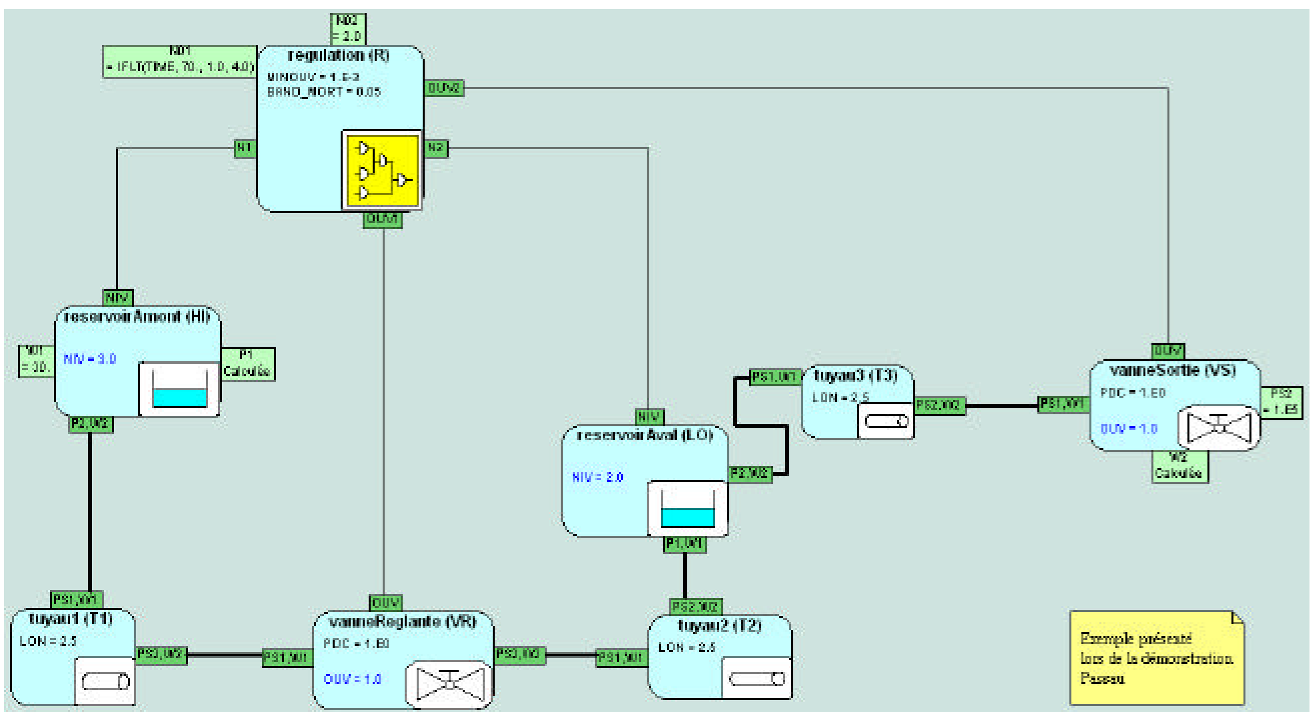
Indeed, this is not true, yet, for several reasons. First of all, the number of equations in realistic models can add up to several thousands, and a user - scientist or engineer - cannot be asked to directly provide this. Indeed, the equation set can often be built by using a *graphical interface* for assembling « components ». Each component contains a small set of equations, and exhibits ports for connecting variables between components – Figure 1 gives an example, including the graphical representation, and some component equation sets[1]. Components can either be abstract or correspond to some technological component such as a pump, a resistor. Graphs of connected components can be organized into a hierarchy of « subsystems ». This graphical interface is called a *modeler*, and can also be in charge of various verifications and checking. When the user has finished assembling his model, the modeler generates the appropriate set of equations, together with some connection information.

[1] Readers familiar with Simulink-like block diagrams should notice that connections are *not* oriented here, they represent equations, not computation data flow.

Second, even if implicit solvers accept equation sets that are *syntactically* implicit, they often impose some *semantic* constraints. For example, the jacobian matrix structure must have some given form (e.g. having non zero diagonal, or being sparse), or must be given if possible, or systems must have low index[2]. There is no reason why a given physical model should be naturally respectful of these constraints.

Third, even when the equation set is available and falls under all the constraints imposed by the solver, writing the equations in another way could improve the computations performed by the solver. So, a model should be *optimized* for numerical simulation.

To sum up, if all is done to help the user to express her model at the level of physics or technology, she cannot be required at the same time to implement it for numerical simulation.



Pipe: W2 = W1
W1' = (PS1-PS2)*SEC/LEN-W1*ABS(W1)*AMORT/(RO*SEC*LEN)
Valve: XINT = W1*(ONE+ABS(W1))
YINT = W2*(ONE+ABS(W2))
W1 = (XINT/ONE)-(PS1-PS2)*(OPEN/LL)
W2 = (YINT/ONE)-(PS1-PS2)*(OPEN/LL)
Tank: PR2 = PRESS+RO*G*LEVEL
PR1 = PRESS+RO*G* LEVEL
XINT = W1*(ONE+ABS(W1))
YINT = W2*(ONE+ABS(W2))
W1*ONE = XINT-(P1-PR1)/KEPS
W2*ONE = YINT+(P2-PR2)/KEPS
RO*SECTION* LEVEL'= IFGT(LEVEL,N0,W1-W2,IFGT(W1,W2,W1-W2,W0))
Here, ONE=1.0

Figure 1: Two tanks, two valves and a regulation in OpenModeler
Some generic component equations

So, for all these reasons, even for implicit solvers, « equations » they are provided with should not be considered as mathematical or physical knowledge, but as *code*. And it is natural to develop a *code generator* which would start from physical models as defined by the user through a modeler,

[2] We shall define index of a DAE later on.

to produce « equations » proper to numerical simulation by a solver.

We have built such a code generator, named SimGen. And to do this, we have followed a transformational approach: initial equations are step by step transformed using small transformations, for finally yielding an equivalent set that is suitable both syntactically and semantically to a given solver. Moreover, we defined an architecture that makes it relatively easy to produce « code » for a new solver. We have actually experimented this by adapting the code generator to various existing solvers.

As we shall see, our « transformational » approach might look in our case very similar to conventional computer algebra. Indeed, we do transform and « simplify » algebraic expressions. However, the goal is different: we do not seek any « syntactically simple » form, but an equivalent form that is suitable for numerical simulation. And it turns out that transformations for generating « equational code » can be opposite to generating simplified expressions. Secondly, our transformations also resemble regular rewriting rules, and in particular we can ask whether our transformations are convergent. For doing this in rewriting rule theory, one has to seek a well-founded order on expressions, such that all rules yield a « smaller » expression. Indeed, we do something similar, except that the order is not based on simple syntactic criteria, but on an assessment of numerical simulation efficiency[3]. And « efficiency » in numerical simulation does not mean only efficiency in time and memory, but also in convergence radius, numerical precision, etc.

An example

Let's consider a simple example, consisting of a current source and two condensers in parallel, as in Figure 2.

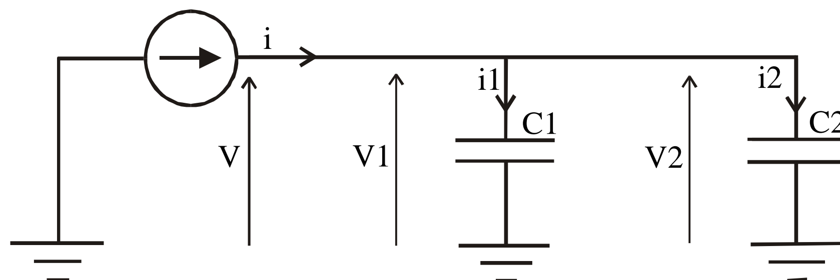


Figure 2: Two condensers in parallel

This system is characterized by equations S1 (Figure 3).
For a physicist, this model could be made simpler, as the two condensers obviously behave like a single one. Nevertheless, an engineer might wish for any reason to introduce two condensers. On the other hand, many systems dealing with DAEs would be in trouble: some would find some causality problem, others would be disturbed by the high index of this model.

Nevertheless, equations S1 are very easy to reduce. First, one should delete variables V1 and V, and replace them by V2, to obtain S2 – indeed, relations V1=V2 and V=V2 are kept as assignments for presenting simulation results in terms of the initial variables. Then, derivative variables are formally computed, if possible. Here, this means equation V2' = i1/C1 is kept, for example, and then i1/C1 is substituted for V2' in other equations, to get S3.

[3] Well such criteria *always* come down to syntax, we are discussing here the way such criteria can be *found*.

$$\begin{cases}(Eq1): i = i(t)\\(Eq2): i = i1 + i2\\(Eq3): V1' = \dfrac{i1}{C1}\\(Eq4): V2' = \dfrac{i2}{C2}\\(Eq5): V = V1\\(Eq6): V = V2\end{cases}\qquad\begin{cases}(Eq1): i = i(t)\\(Eq2): i = i1 + i2\\(Eq8): V2' = \dfrac{i1}{C1}\\(Eq4): V2' = \dfrac{i2}{C2}\end{cases}\qquad\begin{cases}(Eq1): i = i(t)\\(Eq2): i = i1 + i2\\(Eq9): 0 = \dfrac{i2}{C2} - \dfrac{i1}{C1}\\(Eq8): V2' = \dfrac{i1}{C1}\end{cases}$$

Figure 3: Equation sets S1, S2 and S3

At this point, the system is ready for most implicit or semi-implicit solvers. Nevertheless, it can still be transformed, by eliminating i1 and i2, to get

$$\begin{cases}(Eq1): i = i(t)\\(Eq10): i1 = \dfrac{C1}{C1 + C2} i(t)\\(Eq11): i2 = \dfrac{C2}{C1 + C2} i(t)\\(Eq12): V2' = \dfrac{1}{C1 + C2} i(t)\end{cases}$$

However, except if extra simplification is possible, and as far as numerical simulation is concerned, this may be of little interest. We do not play exactly the same rules as computer algebra. So, even though the transformations involved are simple, we must determine and express the right criteria corresponding to solver requirements, in order to get the right tactics.

The initial language

Indeed, we do not start from a raw set of equations, but from a model expressed in a model language. Such a language can be seen as the equivalent of a specification language specific to the field of numerical simulation.
Several languages have been proposed in the past, the more complete one being Modelica [Elqvist et al.]. Modelica encapsulates subsets of equations into classes, and adopts an object-oriented approach to component assembling. In particular, it is possible to represent a hierarchy of graphs stemming from a graphical modeler directly into the language.
Though our system can import and export Modelica files, we do not use exactly this language for the purpose of code equation transformation. Instead, we use a functional language that we have defined, and which looks much more like VDM than like Java. We give here a sketch of this language, we called SimL, [Dormoy] gives a detailed description.

Fundamentally, SimL is a typed lambda-calculus. Types include "mathematical" features to describe vector, matrix, function structures, and also "dimensional" features to represent physical dimensions. For example, an n-vector of velocities will have type (m*s^-1)[^]n, with m the dimension for length and s for time. Units have to be represented within a different framework – not a type system. We distinguish type constructor for "mathematical" functions a -> b and for "physical" time functions that represent physical time dependent "variables" s ~>b – we could as well use other "favored dimensions", but this would get us away from the DAE framework. This makes it possible to consider two distinct polymorphic definitions of multiplication, where f*g = λ(x,y) f(x)*g(y) if f and g are of "->" types, and x*y = λ (t) x(t)*y(t) if x and y are (physical) time variables.
A model simply is a function having boolean as codomain, i.e. a *predicate*. To define models, various operators (arithmetical, mathematical, logical, and *derivative*) are available. For example,

```
LET Resistor(u:Voltage,i:Intensity,R:Resistance) : boolean =
        U = r*i
```

is a model for a resistor – we skip here a precise description of syntax, and other definitions such as

```
LET Intensity = s ~> A.
```

A limited form of comprehension is also accepted, for example

```
for i:=1 to n exp(i)
```

where exp(i) must be boolean. Indeed, this notation is very inaccurate in such a formal context, as it has nothing to do with fortran loops, but people within the simulation community use it. It is equivalent to a "for all" quantifier. One can also use notations for "sigmas" and products, etc. Comprehensions here are limited to an integral variable ranging over an interval. This makes it possible to write n equations in a single line, where "n" is not known when the model is considered – the system imposes that it be known at some time. This makes limited multidimensional modeling possible.

In order to unify all the "models", we add a record type constructor, and we assume that models have a single parameter that must be a record. Fields are first-class objects, we call them *symbols*. They embed into the language the physical ontology that consider physical quantities, components such as pumps, etc, as objects.
Models can be assembled by second order functions. For example, connecting two components "named" c1 and c2 behaving according to models m1 and m2 by making their respective quantities x1 and x2 equal is performed by connector

```
LET Connect(c1,c2,m1,m2,x1,x2) =
        λ(X)   m1(X.c1) and m2(X.c2) and X.c1.x1 = X.c2.x2
```

where X.c1 means "the value of field c1 of record X".
This connector is just an example, it is not very convenient. More complex connectors can be defined, that take into account Kirchoff's laws, for example, or that connect a whole graph of components in a single "step".
Connectors make it possible to represent the graphical and hierarchical structures entered by a user in a modeler directly into the language.

Type inference can be implemented on this language, and type errors – either for mathematical or physical reasons – can be issued.
It is also possible to add *guards* to models, something like preconditions, that define some model physical validity conditions. Any expression is suitable as a precondition. It must be stressed that proving things within this framework is not an easy task, as we deal with time functions, differential equations, and so on. And crucial properties or assumptions, notably continuity, derivability, are nor represented. This would draw us into the whole field of mathematical analysis. Nevertheless, guards are useful at least for specifying physical limitations, and also – if the solver allows it – to generate defensive programs, i.e. programs that raise an exception if guards turn out to be violated during simulation.

As said above, if unknown vector sizes or unknown number of equations are authorized in local models, they must all be fixed when the model in completed, before code generation. So, at least theoretically, it is always possible to come down to a set of differential algebraic equations, the variables of which being scalar, by means of unfolding and "type flattening". However, this is not always a clever attitude, as we shall mention it in the next section.

A last and important point: we do not take into account here *hybrid simulation*. Hybrid systems mix up continuous and discrete aspects; for example, this is necessary for modeling both an industrial process and its control, which often involves discrete behavior. This topic is difficult, and research is going on in the simulation community. Anyway, discussing this would be out of the scope of this paper, at least to date.

General transformations

In the following, we shall assume that we start from a set of equations on scalar variables.

Transformations are divided into two categories. First of all, we have some syntactic transformations, for example for dealing with arithmetical simplifications. They are systematically applied, as they are suppose to make the equation set both simpler for numerical computations and more informative for the ongoing transformational process. Indeed, these simplifications trigger as a consequence of higher level transformations.

A second group of transformations has a deeper effect: they invert equations, substitute expressions, create variables, and derive equations, see Figure 4.

Invert equation*: Given equation $a(e) = b$, where e is a subexpression of a, if a is invertible, replace by equation $e = a^{-1}(b)$.*

Substitute expression*: Given equations $a(e) = b$ and $e = f$, substitute f for e in $a(e)$.*

Create variable*: Given equation $a(e) = b$, create variable x, replace equation by $a(x) = b$ and add equation $x = e$.*

Derive equation*: Given equation $a = b$, add equation $a' = b'$*

Figure 4: Higher-level transforms

Transformation "invert equation" requires some comment. We suppose here that "invertible" means that equation $e = a^{-1}(b)$ is equivalent to $a(e) = b$, and so the old equation can be replaced by the new one. It might be the case that the initial equation implies the new one, without being equivalent. In this case, the new equation should be added to the set without removing the old one.

It must also be pointed out that e can represent any expression, not only variables or their derivatives.

Implementation features

Clearly, these transformations require tactics for focusing on the code generation goals. Roughly speaking, they aim at making the equation set syntactically acceptable by the targeted solver, and the best possible according to some criteria. The strategy for dealing with the first point does not deserve the name "tactics", as it relies upon a mere algorithm. The second point – optimization – is more subtle. As for any code, the optimization criteria include efficiency both in time and space. However, this is not enough, and we must also consider other goals, such as improving, or at least not degrading the convergence radius, or in general "helping" the solver algorithm to converge. Numerical precision and accuracy is another important issue. Let's make this more precise.

Efficiency in time and space first depends on equation set size n. Numerical algorithms take in general time proportional to n^2 for solving the system at each (physical) time step. So, a simple idea for improving programs is to decrease the system size, when possible. For example, this could be done through gaussian elimination, which is a combination of inversions and substitutions. However, as we shall see next, this is not always a good idea. Indeed, it can turn out to be more efficient to increase the system size.
A more sophisticated idea is to decompose if possible the system into square diagonal blocks of size $n_1, \ldots n_p$ like in Figure 5. Then, *provided quantities under the diagonal blocks are easy and reasonable to compute*, the implicit resolution algorithm can be applied to each block separately – provided some obvious order is not violated – and the overall computing time is proportional to

$n_1^2+\ldots+n_p^2$, which is less that $(n_1+\ldots+n_p)^2$. The "easy to compute" condition is very important, as for example derivatives are not reasonable to compute numerically: they are too sensitive to numerical noise. Indeed, diagonal block decomposition improves performance whenever the implicit resolution algorithm time complexity is a convex function of size.

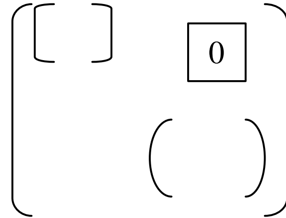


Figure 5: Diagonal Block Decomposition

In general, numerical algorithms also depend on other important features. For example, for large systems, they take advantage of the fact that the structural matrix is sparse[4]. Algorithms for sparse matrices can be much worse than straightforward ones when the matrix is not sparse. So, the system matrix should be left at least as sparse as it was initially[5].

Let's turn now to non optimizing features. At some time during numerical computations, the system Jacobian matrix has to be inverted, and so should be kept regular – though some algorithms can sometimes cope with some local singularities. It is certainly regular if the diagonal dominates[6]. A good heuristic for achieving this is to bring "large" terms, i.e. large in magnitude, to the diagonal. As we are reasoning on a symbolic level, this is not easy to do in an exact way, but we can consider approximations by, for example, bringing non zero terms close to the diagonal, and avoiding zero terms on the diagonal. One could also think of considering numerical coefficients.

Another important property is to "remain around the unit sphere", which means that quantities to be computed should vary in an interval containing 1, and possibly balanced. Indeed, it often happens in physical problems that some variables are very small or large in magnitude: for example, a pressure expressed in Pascals is usually large, while a condenser capacity is small. It is well known that floating point arithmetics is not exact, and that it becomes less accurate if one approaches to the smallest or largest possible magnitudes. A good solution is to normalize the problem, i.e. to substitute x-normalized = x/typical-x for variables x, where typical-x is a typical value of x – usually given by the user.
So, the initial equation may be provided already normalized, and it should be kept so.

Economic functions and transformation instance generation

We have chosen to translate these criteria into *economic functions*. Then, the code generation process will work like a gradient algorithm, i.e. transformations will be applied if they decrease, or at least do not increase, the economic objectives. The economic functions may depend on the solver, though the examples we give here are general.

Moreover, not all possible transformation applications are tried, this would lead to combinatorial explosion. Instead, we have implemented specific, more constrained versions of these transformations. At each step of the transformation process, some transformation application is generated and proposed, and checked against economic functions: if they are improved, or at least not degraded, then the transformation is applied.

---

[4] The structural matrix i-j component is 0 when variable j does not occur in equation i, and 1 otherwise. A matrix is sparse when it has "many" zeros, and "few" ones.
[5] It should be noted that diagonal block decomposition is also a relevant feature for efficiently representing sparse matrices.
[6] I.e. if diagonal element absolute values are sufficiently large with respect to non diagonal ones.

Let's consider each of the previously mentioned implementation features, and see how they can be implemented within this framework. We shall show some economic functions, and see how they can accept or reject some simple transformations. For this, we shall consider a – very tempting – sequence of transformations, that inverts an equation a(e) = b on e, and then substitute $a^{-1}(b)$ for e in all the other equations. We shall do this in two particular cases: when e is a variable x, and when it is a derivative of x, with the extra assumption in both cases that $a^{-1}(b)$ does not mention x. We shall also examine an even more particular case, when equation $a^{-1}(b)$ mentions a single other variable.

Consider first sparse matrices. A very simple version of the economic function can be defined for example as an increasing function of N, the number of non zero terms, and decreasing in n, the system size, such as N/n, which is in average the number of variable occurrences per equation. A more realistic one would take the average and the variance, or the maximum number of variable occurrences in the equations, if the variance is low enough. We shall consider this last version in the following. This means that the number of variables in any equation must not be greater than a given threshold.

Consider first substituting $a^{-1}(b)$ for variable x, and an equation eq where this substitution is performed. This is legal if the number of variables occurring in eq plus the number of variables of $a^{-1}(b)$ not already in eq, minus 1 – since x is removed - is no greater than the threshold. In particular, if $a^{-1}(b)$ mentions just one variable, this is always true. So, removing equations having the form x = f(y) is always legal. This is what we did in the two-condenser example: in this example, the threshold can even be set at 2.

Consider now the case where an equation could be inverted to x' = $a^{-1}(b)$. Here, we have another constraint, which is that x' is not well computed from x. So, we are tempted to substitute $a^{-1}(b)$ for x, but this can "unsparse" the matrix. So, the solution here is first to create a new variable Xprime, to install equation x' = Xprime, and to subtitute Xprime for Xprime in the whole equation set. Then, when inverting equation a(Xprime) = b, we are in previous situation, and Xprime will be eliminated if conditions on sparse matrices are not fulfilled.

Let's come now to decomposition. Matrix decomposition size $(n_1,\ldots,n_p)$, seen as a bag, clearly defines an order on decompositions. One could even refine this order by considering zero and non zero terms in blocks under the diagonal. It can be noticed that this objective function computation can be differentiated with respect to program transformation consequences: at most, only the blocks concerned by the transformation should be decomposed again.
As a consequence of this objective function decreasing, merging blocks should be avoided. In particular, eliminating variable x from equation eq should be performed only if (eq,x) is in a diagonal block. It is also possible in some cases to improve decomposition, notably by creating new variables. For example, if a new variable x is created, and a new equation x = e is introduced, while all occurrences of expression e are replaced by x, then an existing block can be split. For example, consider the system

F1(x,y,f(z,u)) = 0
F2(x,y,f(z,u)) = 0
F3(x,y) = 0
F4(x,y,z,u) = 0

This system cannot be decomposed. However, if we introduce the new variable v and equation v = f(z,u), we get a first block including x,y,v

F1(x,y,v) = 0
F2(x,y,v) = 0
F3(x,y) = 0

And a second block on z,u with equations

$F4(x,y,z,u) = 0$
$V = f(z,u)$

To formalize this, one must extend the order on decompositions of different sizes for example by considering the order on the sum of the decomposition squares. So, in this case, we passed from decomposition (4) whose evaluation is 16 to decomposition (3,2) whose evaluation is 13, and so is better.

In a practical way, looking for syntactically identical subexpressions is combinatorially explosive if all combinations of commutative-associative operator subterms are considered, for example subterms of addition. So, only those subexpressions, the operator of which are of fixed arity, are identified, and a new variable is tentatively created and an expression "folded" only when the expression has multiple occurrences .

Causality analysis

To tackle other features of numerical programs, such as convergence and numerical precision, we need to introduce a new technique, called causality analysis. Causality analysis is an important topic discussed within the community of numerical simulation, and more specifically in the bond-graph community. Here, causality analysis will be defined as *deciding which variable should be solved in which equation*. If the system is regular, which we hope, and if we had a sufficient stock of functions for making any inversion, then we could always formally "solve" the equation set to transform it to *diagonal* form (with diagonal blocks having size 1). Would it be possible, causality analysis comes down to choosing in which equation a variable should be inverted. The result is called a causality assignment.

If the system is square, a causality assignment induces a one-to-one map between variables and equations. Or, if we consider the bipartite graph variables/equations where a variable is linked to an equation iff it occurs in the equation – no variable can be computed in an equation were it does not occur - then a causality assignment comes down to a maximum coupling.

Algorithms for computing maximum are well-known, and polynomial. They are based on the Ford and Fulkerson algorithm for maximizing flows.

One a causality assignment is found, one can consider still the bipartite variable/equation graph, and orientate links in one direction for the assignment, and in the opposite directions for the other links. Then, it can be proved that the strong connected components of this graph are in one-to-one correspondance with a block diagonal decomposition. Indeed, this is what we use for computing the initial decomposition.

Causality analysis can be refined. We have already mentioned that it is not reasonable to numerically compute a derivative from the variable. So, it is preferable not to compute variable x in an equation, and then to use x' in other equations. To do this, we can define a scale, where 0 would correspond to "very easy", ∞ to "impossible", and quantities behind to "not desirable". For example, we could assign 1 to derivatives. We could also extend the scale to consider properties such as "not well invertible", for example for functions such as sin x or x^2. Finally, we have a matrix not only with zero and ones, but with scale values. Let's call it the causality matrix.

Causality analysis goal changes: it has now to optimize the assignment. If we consider the sum of assignment values in the causality matrix as the quantity to optimize, then other polynomial algorithms exist for finding an optimal solutions, for example the hungarian algorithm. We shall not enter here in the details of the algorithm. In a practical way, we use it with at least taking derivatives into account in the scale, to avoid having derivatives under the diagonal blocks.

The causality matrix, in fact the associated objective function, provides another objective function

for our general framework. Transformations should not decrease the causality objective function.

Now, we have described one possible causality scale. One could consider others, for example, for characterizing variable coefficient magnitudes. Then, optimizing the objective function of this matrix would come down to moving "large" components to the diagonal, which is a desirable feature for ensuring convergence.

Experimental results

SimGen has been experimented on various examples, from simple ones to realistic. The largest example was a model of a primary circuit in a power plant, which mentioned, as built by the engineer who did it, 1556 variables and equations. Only 579 equations remain after optimizing the problem. Code generation on an INTEL 300 MHz computer takes less than 300 seconds.

SimGen code generator can generate for two solvers, ESCAP, an implicit solver which is a commercial product of the company StanSim, and LEGO, which is a proprietary semi-explicit solver at EDF.

Comparison to previous work

The closest pieces of work are around Omola [Andersson 94], [Mattsson, Andersson 92] and perhaps also Dymola, which is a commercial product around the same ideas, and possibly an offspring. Besides issues related to hybrid modeling, Andersson describes in [Andersson 94] Omola code generation. It is also based as in SimGen on equation inversion and variable substitution. What's new in our approach is the tactical control on transformations, preventing possible variable elimination from being performed when this goes against numerical efficiency constraints. Also, code generation in Omola and Dymola has a specific solver as target, adapting it to other existing tools do not seem to be attempted. This is an important feature in our case, as we can mix up generated code together with existing legacy simulation code. Recent publications [Mattsson, Otter, Elmqvist 99] report experiments with systems containing hundreds of variables, and so of a similar order of magnitude as those presented here.
In a different community, Ellman and Murata show how Numerical Simulation Programs can be generated from algebraic-differential equations. However, if the goal is eventually the same, the spirit is different. Ellman and Murata's work attempts to find the "first principles" behind numerical simulation software strategies. In particular, subcomponents of what is considered here a black box, the solver, are described and assembled by the synthesizer. So, the numerical code is synthesized as well in this work. We, in the contrary, suppose that numerical programs are given, and we do not use any specification of the underlying programs nor of components, but, embedded in objective functions, some of its properties. We did not seek to seek the fundamentals of code generation, even if restricting to the field of numerical simulation. We target more short-term results, and we clearly intend to tackle real-size engineering systems.
Finally, there is nothing in our work close to a piece of work like Meta-Amphion. Meta-Amphion is the generalization at the meta-level of Amphion, which was a code synthesizer in astronomy. Meta-Amphion succeeded at capturing the main features in Amphion, to represent them as data of a higher-level system, which could re-generate Amphion and also other code synthesizers. Meta-Amphion was a code synthesizer synthesizer.
In our case, we could also think of "going meta", for example to automatically synthesize specific transformation generators or economic functions from a description of the general transformations and, in some way to make more precise, of the goals of code generation. This is clearly outside the scope of the work presented here, and, we must confess that we feel it as a difficult undertaking.

Conclusion

We have shown how a transformational approach could be adapted to numerical simulation code generation. Code generation starts from a set of differential-algebraic equations, to yield a form

suitable to existing numerical solvers. It requires specific transformation generators, and economic objective functions. Then, possible transformation instances are selected to decrease the objective function values.
This work has been experimented, and is used on real-size engineering cases, having more than one thousand equations.
One of our main future objectives is to extend code generation capabilities in order to take into account non scalar variables, such as vectors or matrices, and models having a structure depending on its physical state, for example where equations mention (many) local "if" expressions.

References

M. Andersson, *Object-Oriented Modeling and Simulation of Hybrid Systems*, PhD thesis ISRN LUTFD2/TFRT—1043—SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, December 1994.
R. Brachmann, R. Brüll, Th. Mrziglod, U. Pallaske, *On Methods for Reducing the Index of Differential Algebraic Equations*, Computers chem. Engng, Vol. 14 No 11, pp. 1271-1273, 1990.
J. Dawes, *the VDM-SL Reference Guide*, Pitman Publishing, 1991.
F. Derrough Darche, J-L Dormoy, Set Differentiation: a Method for the Automatic Generation of Filtering Algorithms, in Proceedings of the Eleventh Knowledge-based Software Engineering Conference, Syracuse, NY, 1996.
J-L Dormoy, S. Furic, *SimL, a Specification Language for Physical Modeling*, to appear.
A. Duff, A. Erisman, J. Reid, *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford, 1986.
Dymola http://www.dynasim.com .
Fabian G., *A Language and Simulator for Hybrid Systems*, PhD Thesis, Eindhoven University of Technology, 1999.
T. Ellman, J. Keane, T. Murata, M. Schwabacher, *A Transformation System for Interactive Reformulation of Design Optimization Strategies*, in Proceedings of the Tenth Knowledge-based Software Engineering Conference, Boston, MA., 1995.
T. Ellman, T. Murata, *Deductive Synthesis of Numerical Simulation Programs from Networks of Algebraic and Ordinary Differential Equations*, in Proceedings of the Eleventh Knowledge-based Software Engineering Conference, Syracuse, NY, 1996.
Elmqvist et al., *Modelica™ - A Unified Object-Oriented Language for Physical Systems Modeling*, http://www.modelica.org .
C.W. Gear, Differential-algebraic Equation Index Transformations, SIAM J.Sci.Stat.Comp.,9(1988),39-47.
Kant E., *Synthesis of Mathematical Modeling Software*, IEEE Software, pp 30-41, May 1993.
Y. Ledru, *Specification and Animation of a Bank Transfer*, in Proceedings of the Tenth Knowledge-based Software Engineering Conference, Boston, MA., 1995.
M. Lowry, J. van Baalen, *META-AMPHION: Synthesis of Efficient Domain-Specific Program Synthesis Systems*, in Proceedings of the Tenth Knowledge-based Software Engineering Conference, Boston, MA., 1995.
S.E. Mattsson, M. Andersson, *The Ideas behind Omola*, in Proceedings of the 1992 IEEE Symposium on Computer-aided Control System Design, CADCS'92, Napa, California, March 1992.
S.E. Mattsson and G. Söderlind, *Index Reduction in Differential_algebraic Equations using Dummy Derivatives*, SIAM J.Sci.Comput., 14(1993), 677-692.
S.E. Mattsson, M. Otter, H. Elmqvist, *Modelica Hybrid Modeling and Efficient Simulation*, The 38th IEEE Conference on Decision and Control, CDC'99, Phoenix, Arizona, USA, Dec 7-10, 1999
Bernt Nilsson and Jonas Eborn, *An Object-Oriented Model Database for Thermal Power Plants*.
C.C. Pantelides, *The Consistent Initialization of Differential-algebraic Systems*, SIAM J.Sci.Stat.Comput. Vol.9, No 2, March 1998.
C.C. Pantelides, *gPROMS, and Advanced Tool for Process Modelling, Simulation and Optimization*, ChemPuters Europe Conference, Frankfurt, 29-30 October, 1996.

H.A. Partsch, *Specification and Transformation of Programs, a Formal Approach to Software Development*, Springer Verlag, 1990.
D. Smith, KIDS, *a Semiautomatic Program Development System*, IEEE Trans. On Software Engineering, vol. 16, n°9, September 1990.