Module-Preserving Compilation of Modelica Models

Dirk Zimmer
Department of Computer Science, ETH Zurich
CH-8092 Zurich, Switzerland
dzimmer@inf.ethz.ch

Abstract

Large Modelica models pose serious problems for compilation and simulation. The standard process for the compilation of Modelica models is insufficient since it requires the flattening of the system and generates thereby overly large executables. In this paper we elaborate the concept of module-preserving compilation. This technique aims to generate more compact executables and thereby shall enable the simulation of very large systems in the future. To this end, we introduce an appropriate terminology and design a set of data structures and algorithms that enable the embedment of module preservation into the translation of Modelica models. This paper represents theoretical work only and aims to open up a fruitful discussion on this topic. *Keywords: Flattening; Translation; Causalization.*

1 Motivation

The object-oriented modeling paradigm of Modelica promotes a modular design of systems. Simple Modelica models are thereby composed in order to form a complex, hierarchically structured top-model. The individual submodels are mostly stated in declarative non-causal form. This is a prerequisite for their general applicability. Whereas the declarative form benefits the usability, it prevents the models from being directly "executed." Hence, the models must be translated into a computationally feasible form (e.g. an executable program), mostly for the purpose of time integration.

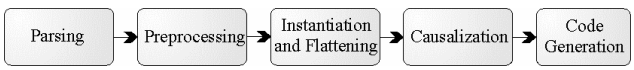


Figure 1: Compilation stages of Modelica code

Figure 1 represents a common compilation scheme that is shared by typical Modelica translators like Dymola [3] or OpenModelica [4]. We see that Modelica models are getting instantiated in the middle stage of the compilation process. The instantiation is carried out in a flattened form. This means that the hierarchic structure is destroyed and that the resulting system represents one large system of equations.

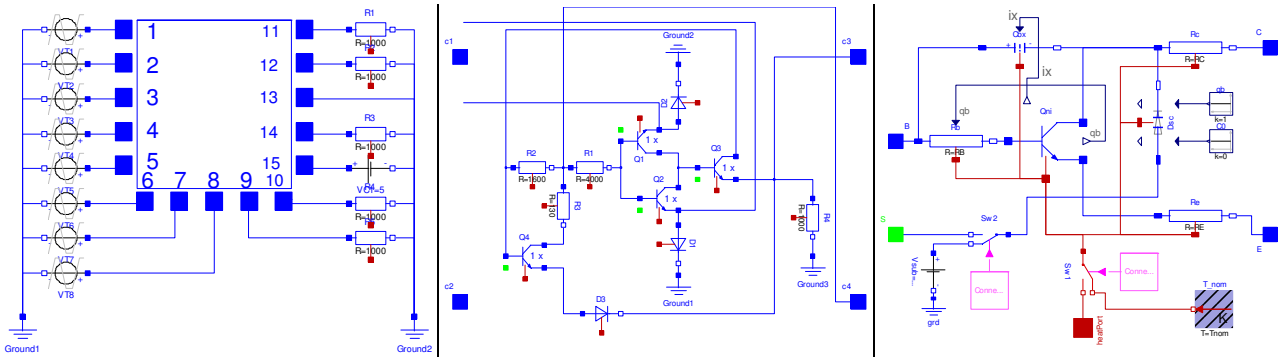
The process of flattening benefits further tasks of the compilation process. First of all, it enables the removal of alias variables (that mostly result from the objects' interfaces) and thereby reduces the system size. The process of causalization is able to handle algebraic loops that extend over many different submodels. State selection and index reduction reduce the dimension for the numerical ODE solver. By these and other means the overall system can be significantly simplified and the resulting code is competitive to the best manually coded simulations.

Unfortunately, the process of flattening also has its deficiencies. It gets problematic for very large systems. Since the model is always processed as a whole, it does hardly scale and gets increasingly inefficient for large models. Also the generated code starts to lack in quality. It gets overly large and contains many redundant parts.

To get a better understanding of the problem, let us look at an example. The *Verification Package for Modelica Spice 2.1* [2] includes the model of a four bit adder (c.f. figure 2). Because it is modeled down to the layer of single bipolar junction transistors, the model is very detailed and indeed very large: it contains in total 481'915 scalar equations. Dymola (v7.2) fails in the attempt to simulate this model. The translation needed almost 1 GB of RAM and finally generated an 88MB executable. Its simulation in Dymola failed in the simulation-environment.

Whereas the final reason of breakdown is most probably a minor bug that could be corrected, we shall not overlook that there is a serious issue with very large models. This is by no means specific to Dymola, it is a principal problem concerning the general processing of languages like Modelica.

Large models contain often a high degree of redundancy, and the larger they get, the more redundant they typically are. This rule of thumb also applies to our example model. The four-bit adder contains 2 two-bit adders. The two-bit adder contains 2 one-bit adders and each one-bit adder circuit contains 9



(a) four-bit adder (b) NAND (c) BJT

Figure 2: Model diagram of the four bit adder and two of its components

NANDs. The NAND gate itself consists in 4 bipolar junction transistors (BJTs) plus 3 diodes. Consequently, the four-bit adder contains 36 NANDs, i.e., 144 BJTs plus 108 diodes.

On the modeling layer, this redundancy is not a problem, since the number of individual modules is far lower than the number of their corresponding instances. On the computational level, all the hundreds of instances get flattened and corresponding code is generated for each of them. In direct consequence, the code gets large, bulky, and redundant.

It is an evident question to ask: If hundreds of sub-model instances share the same equations, can they not share (at least partly) the same code? And can this code be modularized in form of a function? It is the aim of this paper to examine how and when a given modularization on the modeling layer can be preserved and mapped to code modules in the final executable. This shall provide future benefits for both the speed of the translation and the size of the executable.

2 Modules and their Representation

Most readers will be very familiar with the typical process of module creation. It is mostly applied to a structure that occurs several times in a system. It can be described by 5 steps:

1. Extract all the elements you want to put into your module from one occurrence of your structure.
2. Determine all the variables that are part of your module, and separate this set of variables into two distinct sets: The set of local variables that occur in your module only and the set of interface variables that are also being used elsewhere.
3. Form an interface for your model given the corresponding set of interface variables.
4. Replace all occurrences of your structure by instances of your module (e.g. sub-model declarations or function calls, respectively).
5. Connect the interface of your module with the corresponding variables.

This way of modularization can be applied to transform code segments into functions but also to group clusters of equations into a Modelica model. Hence modules are a common concept for both the source and target of a Modelica compiler.

On the modeling layer, the modular design is given by the modeler. A module is represented by a Modelica model and it consists essentially in an unordered set of equations. In order to form meaningful modules, the modeler aspires to create sub-models that form a semantic entity and offer a preferably small interface that wraps a more complex inner part.

The target of the compilation is program code. That is an (ordered) list of statements. These statements are mostly computations of operators and value assignments. If several pieces of code share equivalent sub-lists of statements (again, disregarding the naming of variables), these statements can be modularized. Such a code module is typically represented as a function.

Module-preserving compilation aspires a mapping between the modules on the model level and the potential modules on the code level. In concrete terms: how and when can a Modelica model or a part of it be translated into a function of an imperative programming language?

3 Entities of Modularization

In principle, any arbitrary code segment can be modularized, but to gain any advantage, the subpart needs to occur frequently in the main code and it needs a feasible interface. One might attempt to find such suitable subparts in the flattened code by pattern-finding algorithms, but this approach is hardly

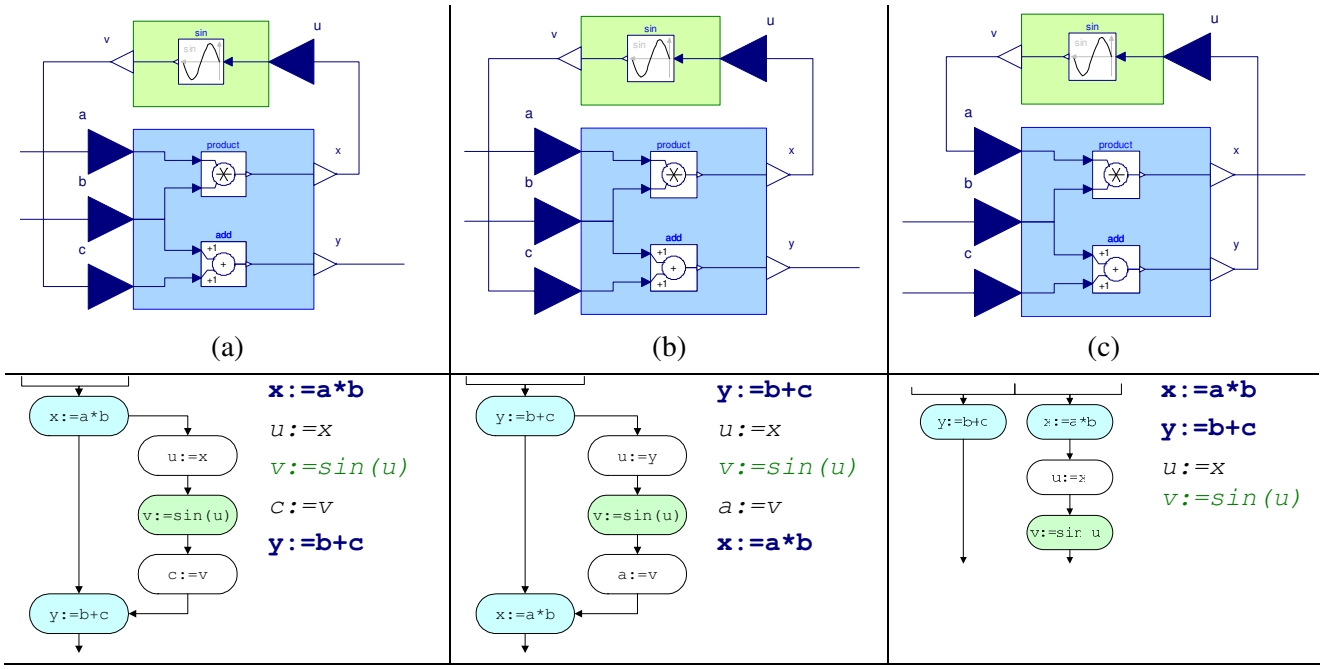


Figure 3: A causal block in different contexts with the corresponding code.

promising since this is a computationally demanding task. It can be expected to fail for very large systems when modularization is needed the most.

The hierarchic structure of the equation-based model gives us a priori information about those patterns that may occur frequently in the resulting code. In order to use this information, we have to know about the requirements for the translation of a sub-model into a function.

A typical model in the Modelica library has a non-causal interface with non-causal equations. Hence, many models (like the model of a mechanical rod) can be causalized in many different forms that all require a different code for the computation. Thus, code can only be shared for model instances of the same causality.

However, thinking that causal models can be directly transformed into code, is misleading. Figure 3 (a) and (b) present a simple counterexample. The corresponding code for the presented models is placed underneath the modeling diagrams. We see that the same causal block (blue) not only yields different code, also its code separates into two parts. Hence it cannot be expressed by a single function.

The problem is that causality only gives rise to a partial order, but the transformation into code requires an absolute order. The fact that the variable x is determined by a and b and that y is determined by b and c does not say anything about the order between x and y. This might be stipulated by the remaining system as in (a) and (b), but it might be left for choosing as well as in (c).

The causal relations between assignments are best expressed by a causality graph $G(E,V)$. This is a directed acyclic graph where the vertices V_G correspond to the assignments. The edges E_G are formed by those pairs of assignments (s_1, s_2) where v is a variable of s_2 and determined by s_1. Examples of such causality graphs are placed beside the code segment in figure 3.

Those assignments belonging to a certain model M induce a sub-graph G_M of G. We are interested in a very specific form of sub-graphs:

Definition 1:

- A vertex-induced sub-graph G' of G is called *path-complete* iff all paths in G between any vertex pair (s_1', s_2') in G' are also included in G'.

Path-complete sub-graphs are of high interest to us because each one of them can be translated into a separate cohesive program segment and thereby can be modularized into a function.

Any vertex-induced sub-graph G_M can be decomposed into a set of path-complete sub-graphs $\{G_{E1}, G_{E2}, ...\}$, but since there are many such decompositions, we need to specify further restrictions to derive a unique decomposition. First of all, we demand the decomposition to be minimal in the sense that the decomposition contains no pair (G_{Ea}, G_{Eb}) that can be merged to another path-complete sub-graph.

Since G is a directed acyclic graph, any *minimal* decomposition into path-complete sub-graphs is given an *absolute* order by G. There may now be vertices of G_M that cannot be uniquely assigned to one of the decomposition's sub-graphs. If we define these ver-

tices to be assigned to the sub-graph of the lowest order, we get a unique decomposition. We denote this as the busy, minimal decomposition into path-complete sub-graphs. Fortunately, this decomposition can be derived incrementally, as will be described in section 5.2.

We recognize that the code for any model M may be split into several entities E_1, E_2, ... that represent cohesive code segments and that such a decomposition into program segments can be uniquely determined. Thus, we define:

Definitions 2 and 3:

- A *causal entity* E of a model M represents a list of vertices of a sub-graph G_E that results out of the busy, minimal decomposition of G_M into path-complete sub-graphs. The order of the list E is partially determined by the underlying directed graph G_E.
- A *causal interface* I_E of a causal entity E represents a pair of variable sets. The first set contains the input variables that are formed by the ingoing edges from G to the sub-graph G_E. Correspondingly, the second set is formed by the outgoing edges and represents the output variables.

The causalization of a model can now be precisely defined by the causal signature:

Definition 4:

- The *causal signature* S_M is a complete list of causal interfaces I_E for all causal entities E belonging to a given model M. The list determines the order of the corresponding causal entities.

For illustration, let us look at the causal signatures from Figure 3. Each model has a different one:

(a) [$(\{a, b\},\{x\})$,$(\{c\},\{y\})$]

(b) [$(\{b, c\},\{y\})$,$(\{a\},\{x\})$]

(c) [$(\{a, b, c\}, \{x, y\})$]

We have seen that (a) and (b) require different code. They also have different causal signatures with different causal entities. We further recognize that each pair corresponds to a block of code, hence to a potential function. Code that is generated for (c) shall not be used for (a) and (b), but not necessarily vice versa. Code for (a) and code for (b) would be usable also for (c). Thus, we define the terms sub- and super-signature:

Defintion 5:

- A causal signature S_M is *sub-signature* of another causal signature S_M' over the same model if S_M can be transformed into S_M' by merging[1] subsequent pairs of the list. S_M' is then defined as a *super-signature* of S_M.

Example: The signature $[(\{a, b, c\}, \{x, y\})]$ from example (c) is a super-signature for both (a) and (b).

Sub-models that share the same causal entities can share the same code. The number of causal entities corresponds thereby to the number of separate code blocks that could be turned into functions. In order to reuse code efficiently, one may decide to replace the code of one causal entity by the code of one or several entities that originate from a model instance that has a causal sub-signature.

Example: If the causal signature of 2(a) occurs frequently, the compiler may decide to create two code modules in form of the functions f_1 and f_2:

```
function f1(a,b)
       x := a*b;
       return (x);
end
```

```
function f2(b,c)
       y := b+c;
       return (y);
end
```

If the causal signature of 2(c) occurs only once, the corresponding code may now be formulated using these modules by the segment:

```
x := f1(a,b)
y := f2(b,c)
```

4 Which entities shall be preserved?

Let us look at the academic example of figure 4. It contains a lot of addition blocks and hence a compiler might be tempted to create a code module (function) for the block. However, this is obviously not a good idea. Modularization of the correspondent causal entity will decrease the performance and quality of the code in this case.

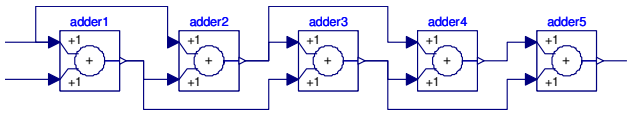


Figure 4: A bad example for remodularization

The reason for this is that the modularization of a causal entity needs to provide its interface (in concrete terms: function parameters and return value). Once implemented, it is hardly possible to optimize across the interface, and hence the systems cannot be simplified. Auxiliary or alias variables cannot be

[1] Please regard: The merging of two causal entities corresponds to the merging of its causal entities, and hence variables can get removed from the interfaces.

removed out of the system. Furthermore, the simple additions are replaced by more costly function calls. We see that preserving modules per se does not improve the code. It is a tool that demands proper application.

Modularization is not for free, it incurs additional cost. Memory is needed to define the interface of the function. Computational time is needed for the assignment of the interface values and the corresponding function call.

Thus, the modularization of causal entities is only meaningful, if the additional computational cost is marginal to the cost of the function and if the memory cost of the interface is compensated for by the memory savings that are attained by replacement of multiple instances through function calls. Let us therefore make a distinction between the inner and outer complexity of a model.

4.1 Inner complexity of a module

Definitions 6 and 7:

- The *inner computational complexity* $C_{i,E}$ of a causal entity E is the total amount of all memory assignments and basic computations from code that corresponds to E.
- The *inner data complexity* $D_{i,E}$ is the total amount of local data that is required for those computation.

Since both definitions for the inner complexity refer to the actual code, their estimates are dependent on the simplification mechanisms of the preceding compilation stages.

Attaining a fair estimate for $D_{i,E}$ is actually unproblematic. However, its complexity may depend on the modularization of potential sub-entities. Estimates for $C_{i,E}$, can be difficult to obtain when the number of computations is unsure, for instance, an iterative solver has to be applied in order to solve a non-linear equation system. Fortunately, it turns out that a determination of $C_{i,E}$ is not necessary.

4.2 Outer complexity of a module

Definitions 8 and 9:

- The *outer computational complexity* $C_{o,E}$ of a module is the amount of all memory assignments and basic computations that refer to data of its interface and to data outside the module.
- The *outer data complexity* $D_{o,E}$ of a module is the total amount of data in its interface, defined by I_E.

Knowing the interface of a potential code module means knowing about its outer complexity. However, the interface may contain more than intuition suggests. The interface variables of the corresponding equation-based model M are not the only members of the interface. If the causal entity E represents only a part of the model M, auxiliary variables will be added to the interface. Furthermore, if the causal entity defines integrators and hence possesses state variables, these state variables have to be part of the interface as well, since they are determined by the global algorithm for synchronous time integration. The same is true for variables that trigger events. They are also part of the interface, since their values must be accessible to the event finding algorithms.

Variables that form the simulation output are not necessarily part of the interface. The tracking of the correspondent data can be done within a code module.

The distinction between inner and outer complexity is however dependent on the computational framework that will embed the resulting code. Here, we assumed a typical environment for synchronous time integration, but in a different computational framework like QSS [5], the integrators and event triggers are local and the corresponding variables do not belong to the outer complexity.

4.3 Frequency of entities

There is no incentive to turn any causal entity E into a code module, if there is only one instance of it. The number of occurrences N_E is therefore a crucial criterion. It influences cost and benefit of the modularization.

The **cost** of modularization is:

- Additional computational cost: $N_E\, C_{o,\,E}$
- Additional data complexity required: $D_{o,\,E}$

The **benefit** of modularization is:

- Saved computational cost: 0
- Saved data complexity: $(N_E - 1)\, D_{i,E}$

Let μ be a coefficient that translates the computational complexity into data complexity. It needs to be determined by experience, but mostly it will be chosen in such a way that $\mu \cdot C_{o,E}$ is close to $D_{o,E}$. Now we can compare the overall cost with the total benefit:

$$N_E\, D_{i,E} > N_E\, \mu\, C_{o,E} + D_{o,E} + D_{i,E}$$

Consequently, let R be the fraction:

$$R = (\mu \cdot C_{o,M} + D_E / N_E) \;/\; D_{i,E}$$

A modularization becomes profitable if $R < 1$. This implies that $\mu \cdot C_{o,E}$ must be smaller than $D_{i,E}$. Furthermore we see that the inner computational complexity is irrelevant. If we presume that $\mu \cdot C_{o,E}$ equals $D_{o,E}$, the computational complexity can be neglected entirely.

One of the difficulties of modularization is that a causal entity E for a model M may contain a sub-entity E' from a sub-model M' of M. The inner data complexity $D_{i,E}$ is then dependent on the potential modularization of E'. If E' forms a module of its own, $D_{i,E}$ obtains a lower value, and E itself is less likely to be modularized. Hence the module preservation of sub-models influences the modularization of its super-models. Fortunately, Modelica enforces a strict model hierarchy, implying that models cannot be sub-models of themselves. This will simplify the further analysis in section 5, but first let us look at an example.

4.4 Example

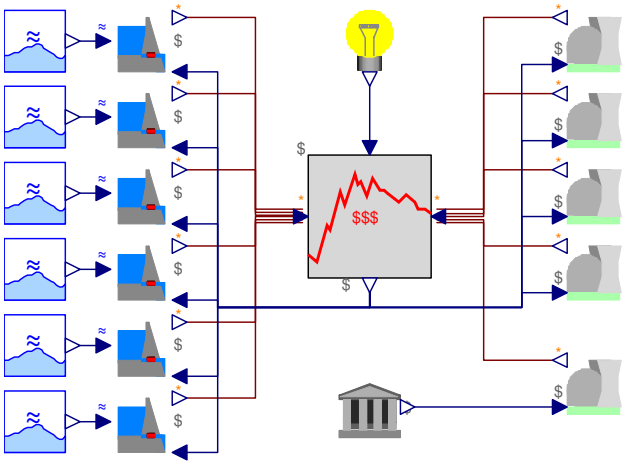


Figure 5: Simple model of an electric energy market with producer models.

Figure 5 presents a very simple model of an electric energy market. The electric power originates from 5 nuclear power plants and 6 hydropower plants. Each of the power plants has its own parameters, and the hydropower plants are dependent on the waterflow from its rivers. Furthermore, one of the nuclear power plants is state owned and works for a fixed-price scenario whereas all other power plants compete on the free market. The actual model of the market is placed in the center and determines the price from the current balance of supply and demand.

The diagram represents the top model. Its implementation is based on the system dynamics library [1]. The overall model contains about 1100 variables, whereby 800 of them represent alias variables. The multiple power plants are an obvious target for modularization. Let us therefore examine their causal signatures.

A hydropower plant produces a certain amount of power given its current state. The price is determined by the market. The current price influences the monetary profit or loss of the plant and hence drives a controller that aims at maximizing the profit. Each hydropower plant possesses two state variables: the desired outflow f and the current water level w. Its behavior is controlled by the inputs the inflow i and the current price $\$$. The power p forms the output. Without knowing much of the interior we can determine the causal signature that is shared by all six hydropower plants:

$$[(\{f, w, i\},\{p, \mathrm{d}w/\mathrm{d}t\}),(\{\$, w\},\{\mathrm{d}f/\mathrm{d}t\})]$$

We see that state and derivatives have become part of the causal interfaces. Furthermore we need two code modules to compute the model of the power plant. Also the nuclear power plants demand two code modules although their causal signature is simpler since it has only one state: the current production level l:

$$[(\{l\},\{p\}),(\{\$, l\},\{\mathrm{d}l/\mathrm{d}t\})]$$

The state owned nuclear power plant is an exception though. Its signature is a super-signature of the other plants:

$$[(\{l, \$\},\{p, \mathrm{d}l/\mathrm{d}t\})]$$

Table 1 presents the ratio R for all four causal entities. 3 of the 5 entities are suited for modularization and so the overall data complexity can be reduced to roughly 50%.

Table 1: Analysis of causal entities

Entity	$D_{o,E}$	$D_{i,E}$	N_E	R
$(\{f, w, i\},\{p, \mathrm{d}w/\mathrm{d}t\})$	5	10	6	**0.75**
$(\{\$, w\},\{\mathrm{d}f/\mathrm{d}t\})$	3	12	6	**0.46**
$(\{l\},\{p\})$	2	1	4	**2.75**
$(\{\$, l\},\{\mathrm{d}l/\mathrm{d}t\})$	3	14	4	**0.51**
$(\{l, \$\},\{p, \mathrm{d}l/\mathrm{d}t\})$	4	15	1	**1.53**

5 Revised compilation process

The proposed methods so far are feasible to apply an analysis to an already flattened model and to optimize the resulting code by modularization. But the flattening alone can represent an unaffordable task, and hence the modularization shall be integrated in all of the important stages of the compilation process.

5.1 Preparation

In a first preparatory stage, we attempt to estimate R for any causal entity E of a model M from the non-causalized Modelica model itself. The idea is to get rid of all the small models that contain just a few equations. Therefore, this analysis does not need to be pursued for large models. The inner and outer data complexity of the corresponding Modelica model enables an estimation value $\check{R} = \check{D}_{o,M} / \check{D}_{i,M}$ that mostly is a lower bound for the effective R of its causal entities. This is because causalization reduces the inner complexity mostly more than the outer complexity, and the split into causal entities mostly increases the overall interface. Furthermore N_E is assumed to be infinite.

Sub-models with $\check{R} < 1$ are not expected to contain modules that are valuable to preserve. The same is true for sub-models that occur only once. All other models are put into the set Ω, and their causal entities may form modules of the program code. Please regard that equations of models that are not in Ω can still become part of a code module if any of their super-models is in Ω. Hence the selection criterion for Ω can be chosen even stricter than suggested.

5.2 Instantiation and Causalization

In the classic scheme, all models get instantiated first and then causalized. For very large systems this procedure is not feasible anymore. Ideally, the process of module preservation shall be implemented in such a way that the full flattening of the model can be avoided. Thus we propose to instantiate and causalize in several alternating iterations.

To this end, the models are being instantiated into a buffer of fixed size. When the capacity limit is reached, the equations in the buffer are causalized as much as possible. Those equations that could be causalized are transformed into assignments and added to the causality graph G. Last, the buffer is cleared and the non-causalized equations are put aside for a latter iteration.

In order to causalize the whole system, many sweeps over the buffer may be required. During the whole process, the causality graph G is constantly growing. When an assignment s of a model M in Ω is added, G and the induced sub-graph G_M grow by one vertex. A decomposition $\{G_{E1}, G_{E2}, ..., G_{En}\}$ into path-complete sub-graphs will be affected in two possible ways:

- a causal entity is enlarged $G'_{Ek} = G_{Ek} + s$ (the entity that is of lowest order in G, in case there are several options).
- else, the new vertex forms a new causal entity $G_{E(n+1)} = s$.

This procedure will lead to a busy, minimal decomposition into path-complete sub-graphs. We further recognize that existing causal entities can just grow, but they will not be cut or merged. This is very important because this means that we can track all entities: When a common causal entity in the graph G_E exceeds a certain threshold size, we can decide to modularize G_E in the graph by storing it separately. In this way, we can avoid to store the complete causality graph in plain form. This does not mean that the corresponding causal entity will form a code module. This modularization within the causality graph is a separate mechanism that is suggested in order to save a potentially substantial amount of memory.

5.3 Model hierarchy

At the end of the causalization, we have a complete causality graph where larger common parts share the memory. The graph contains a potentially large number of causal entities that all have to be analyzed for a potential modularization. This analysis has to be executed in a certain order: A model may have instances with different causal signatures. Some of these signatures may be super-signatures of others. This will influence the modularization of the sub-signatures, and thus all causal entities belonging to a model have to be analyzed at once.

Furthermore, the modularization of an entity of a model M may influence the modularization of an entity in any of M's super-models (remember section 4.3). Therefore we have to execute the analysis according to the model hierarchy starting with the lowest models first.

The term model hierarchy might be misleading since it suggests a tree-like structure. In fact, a Modelica model hierarchy can also be represented by a directed acyclic graph that gives rise to a partial order on its models representing the vertices of the graph. The bottom-up procedure can therefore be implemented as a (breadth first) graph traversal.

5.4 Modularization

For each model M in Ω, we built up a prefix tree of its causal signatures, where each node owns a counter, and the branches denote the corresponding causal entity E. A path from the root to a leaf then represents a causal signature S_M of a model instance. Figure 6 presents an exemplary prefix tree for the block model of figure 3.

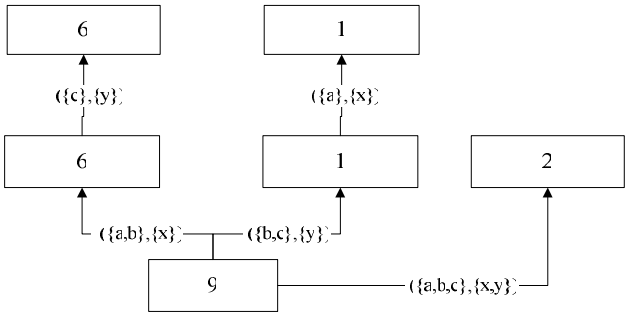


Figure 6: Prefix tree of causal entities

We want to find out, which causal entities are profitable to modularize. To this end, we have to consider the super-signatures first (right branch in figure 6). We can either create an extra causal entity for a super-signature or re-use existing ones. In general, this is a hard optimization problem. For our purposes a simple heuristic procedure shall be sufficient:

- Let E_0 be the first causal entity of the super signature S_{M0} for the signatures $S_1 \ldots S_n$ and let $E_1 \ldots E_n$ be their corresponding first causal entity.
- We find the best R: $R_{min} = \min \{ R(E_1) \ldots R(E_n) \}$
- If $R(E_0) > R_{min}$ we decide to split the super-signature and integrate it into the path that belongs to R_{min}.
- At last, we mask out the root and repeat this process recursively for all sub-trees.

If we assume the values out of column 1 from table 2, the prefix tree of figure 6 will transform into the one depicted in figure 7. Assuming the values of column 2 will cause no changes in the original tree.

	Assumption 1	Assumption 2
$R(\{a, b, c\}, \{x, y\})$	1.2	0.8
$R(\{a, b\},\{x\})$	0.6	0.85
$R(\{b, c\},\{x\})$	1.8	2.0

Table 2: Scenarios for super-signatures

Finally, we can compute R for all causal entities of the prefix tree and either chose to modularize the code or not, given the criteria from section 4.3. Please remember that in order to compute R, symbolic simplifications should take place beforehand.

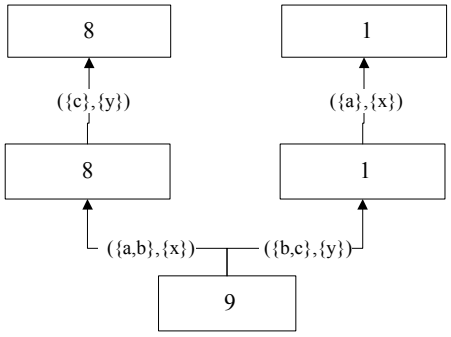


Figure 7: Prefix tree of causal entities

5.5 Summary and run-time efficiency

Step 1: Preparation

Estimate $\check{R}$ for all models and enter selected models into the set Ω.

Step 2: Instantiation and causalization

Install a buffer of limited capacity.

While there are non-causalized equations **do**

Fill buffer with non-causalized equations.

Attempt to causalize them.

Reject non-causalized equations for future iterations.

Track causal entities for the models in Ω.

Store larger entities separately.

end

Step 3: Modularization

For all models M in Ω.

in order of the model hierarchy **do**

Built up the prefix tree of causal signatures for M.

Simplify the code of the corresponding causal entities.

Manage occurring super-signatures.

Compute R for all remaining entities and decide to modularize the entity if R < 1.

end

It is important to note that none of these processes has to solve an NP-hard optimization task. The precise algorithmic efficiency depends on the concrete implementation. However, let us look at the causality graph. If we (realistically) suppose a maximum number of variables in an equation, the memory demand is linear to the size of the system and even less than linear if modularization can be applied. The fact that the code modules can be created ad-hoc helps to keep the memory demand small. The most expensive algorithm that works on this graph is the stepwise causalization. In worst case, it will lead to a quadratic run-time.

6 Further issues

The implementation of a mechanism for re-modularization has implications for other processes in the compilation. In the following, we investigate the most important points that need to be concerned:

6.1 Algebraic loops

A proper implementation of module-preserving compilation requires that the process of model instantiation and causalization is conducted in several iterations. As long as the model contains no algebraic loop and/or requires index reduction, this is a

non-issue. For instance, the complete domain of system dynamics is mostly non-critical.

However, many systems cannot be represented in lower triangular form and thus a block lower triangular (BLT) form is typically aspired. A standard algorithms for this purpose, the Dulmage-Mendelsohn permutation, [7,8] cannot always be applied since it assumes that the whole system is readily available. This is not naturally the case for very large systems.

Other algorithms for a BLT transformation are therefore required that are able to cope with local information only. It is possible in doing so by applying a tearing method directly on the whole system that identifies the corresponding blocks of equations (denoted as algebraic loops) later on. Such mechanism have already been developed (although for another purpose) in the SOL framework [9,10].

In general, the tearing will be needed for the efficient solution of the algebraic loops. A tearing method selects (using certain heuristics) a sufficient number of tearing variables and assumes them to be known. Now the algebraic loop can be causalized and an equal number of residual equations results. In order to solve the system, an iterative numerical solver is typically applied. We need to investigate how modularization can be applied for the torn system of equations.

For causal entities there are two cases that need to be considered with respect to algebraic loops:

- *Causal entities that contain a complete algebraic loop.* This is in principal unproblematic. The code can be wrapped like any other code. However, depending on the heuristics, it is not guaranteed that equivalent models will be torn in an equivalent way. This is still a serious issue.
- *Causal entities that are only part of a loop.* The modularization of such entities is in general not very meaningful. They may contain residuals or tearing variables that would enlarge the interface of these entities. Furthermore the entities may contain additional computations that are not necessary to compute the residuals. These increase the computational effort and (what is worse) may not be fail-safe with respect to the numerical solver.

6.2 Symbolic Differentiation

The mechanisms for index reduction (see [7]), but also the application of iterative solvers may require the differentiation of subparts of the equation system. In the case of index-reduction the differentiation often generates algebraic loops.

The differentiation adds new equations to the system. Whereas there are given models for the original set of equations there are no models for the differentiated equations and hence no modularization can take place on differentiated subparts of the system.

We therefore propose that differentiated equations become part of their original model can therefore also be part of causal entities. However, this topic also needs further investigation.

6.3 Pre-compilation and re-modularization

Causal entities map to an enclosed code segment (i.e. a function) that of course can be provided also in pre-compiled form. Hence an M&S-environment may decide to maintain a library of precompiled code from the most frequently used causal entities. The underlying motivation is to decrease the compile time.

There remains doubt that pre-compilation will represent an effective means. It could as well be that the reading from the disc is slower than the actual compilation process. Only for very large code segments pre-compilation will be profitable for sure. Such code segments would correspond to sub-models that are not only large but also hardly decomposable into further sub-models. Otherwise the smaller sub-models will be modularized and the large model shrinks in its inner data complexity. Ideal vehicle models in a traffic simulation could be one such example.

6.4 Modeling requirements

Module-preserving compilation requires that the provided model owns a suitable hierarchic structure and this needs to be provided by the modeler. Since the compilation by itself is not able to detect any patterns or to form feasible substructures the most principle rule of information processing applies: garbage in – garbage out. Badly structured or flat models cannot be handled efficiently. An implementation of a finite-element mesh within Modelica would represent one such example.

7 Conclusions

The concept of module-preserving compilation bases on the observation that a causalized model can be decomposed into a list of causal entities whose interfaces form the causal signature. Each causal entity corresponds thereby to a potential code module. Regarding the outer and inner complexity of a potential

code module we could derive a criterion for the selection of appropriate code modules.

The integration of module preservation in the translation process is clearly a non-trivial task that involves a whole bunch of issues. The model hierarchy needs to be taken into account. Furthermore we suggest managing the different causal entities in form of prefix trees. Methods for tearing and state selection need to be provided that do not require the complete system to be readily available.

Module-preserving compilation represents not more than one tool to optimize code and hence it cannot solve all problems. It will fail for flat or unstructured models and it may be difficult to apply if there are huge algebraic loops.

Nevertheless, there are many suitable examples like the large electric circuits of figure 2. We also presented a model based on system dynamics (figure 5). Although this example is still quite small, module preservation is expected to decrease the code complexity already substantially. Evidently, much larger models of an energy market can be envisioned with much more elaborated models. A model of the complete European grid could be one example. Such a model would contain several thousands of power plants and many different market places. For such large models, module preservation becomes a vital tool in order to enable a simulation of the system at all.

Our proposal for module-preserving compilation aims to be a general approach for a wide range of models. It is however possible to simplify the outlined procedure significantly if we can make certain assumptions about the model structure or require additional hints or help from the modeler himself. Such an approach will lose generality but might be better achievable in a practical implementation.

References

[1] Cellier, F. E.: World3 in Modelica: Creating System Dynamics Models in the Modelica Framework. In: *Proc. 6th Interna-tional Modelica Conference*, Bielefeld, Germany (2008) Vol.2 393-400.

[2] Cellier, F.E., C. Clauß, A. Urquía: Electronic Circuit Modeling and Simulation in Modelica. In: *Proc. 6th Eurosim Congress on Modelling and Simulation*, Ljubljana, Slovenia (2007) Vol.2, 1-10.

[3] Dynasim AB, *Dymola Users' Manual*, Version 6.0, Lund, Sweden, 2006.

[4] Fritzson, P., P. Aronsson, H. Lundvall, K. Nyström, A. Pop, L. Saldamli D. Broman: The OpenModelica Modeling, Simulation, and Software Development Environment. In: *Simulation News Europe* (2005) 44/45.

[5] Kofman, E., S. Junco: Quantised State Systems: A DEVS Approach for Continuous Systems Simulation. In: *Transactions of SCS*, (2001) 18(3), pp.123-132.

[6] Pantelides, C.: The Consistent Initialization of Differential-Algebraic Systems. In: *SIAM J. Sci. and Stat. Comput*. (1988) Vol 9, No. 2, 213-231.

[7] Pothen, A., Chin-Ju Fan: Computing the Block Triangular Form of a Sparse Matrix. In: *ACM Transactions on Mathematical Software* (1990) Vol 16, No. 4 303-324.

[8] Tarjan, R.: Depth-first search and linear graph algorithms. In: *SIAM Journal on Computing*. (1972) Bd. 1, No. 2, 146-160.

[9] Zimmer, D.: Introducing Sol: A General Methodology for Equation-Based Modeling of Variable-Structure Systems In: *Proc. 6th International Modelica Conference*, Bielefeld, Germany, (2008) Vol.1, 47-56

[10] Zimmer, D.: Enhancing Modelica towards variable structure systems. In: *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT)*, Berlin, Germany (2007) 61-70

Biography

Dirk Zimmer received his MS degree in computer science from the Swiss Federal Institute of Technology (ETH) Zurich in 2006. He gained additional experience in Modelica and in the field of modeling mechanical systems during an internship at the German Aerospace Center DLR 2005. Dirk Zimmer is currently pursuing a PhD degree with a dissertation related to computer simulation and modeling under the guidance of Profs. François E. Cellier and Walter Gander. His current research interests focus on the simulation and modeling of physical systems with a dynamically changing structure. To this end, the Sol-project was founded in 2007 together with F.E. Cellier and the support of the Swiss National Science Foundation.