A New OpenModelica Compiler High Performance Frontend

Adrian Pop[1] Per Östlund[1] Francesco Casella[2] Martin Sjölund[1] Rüdiger Franke[3]

[1]PELAB - Programming Environments Lab, Dept. of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden, {adrian.pop,per.ostlund,martin.sjolund}@liu.se
[2]Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy, francesco.casella@polimi.it
[3]ABB, IAPG-A26, Kallstadter Str. 1, 68309 Mannheim, Germany, ruediger.franke@de.abb.com

Abstract

The equation-based object-oriented Modelica language allows easy composition of models from components. It is very easy to create very large parametrized models using component arrays of models. Current open-source and commercial Modelica tools can with ease handle models with a hundred thousand equations and a thousand states. However, when the system size goes above half a million (or more) equations the tools begin to have problems with scalability. This paper presents the new frontend of the OpenModelica compiler, designed with scalability in mind. The new OpenModelica frontend can handle much larger systems than the current one with better time and memory performance. The new frontend was validated against large models from the ScalableTestSuite library and Modelica Standard Library, with good results.

Keywords: OpenModelica, compiler, flattening, frontend, modelling, simulation, equation-based, scalability

1 Introduction and Motivation

System-level dynamic modelling and simulation is a key activity in modern system engineering design. In parallel to the detailed component design, which is performed using advanced 3D CAD, CFD and FEM software tools, system-level modelling, usually including systems of systems and large numbers of interacting components, allows predicting the dynamic performance of complex systems, which emerges from the interaction of its components.

The Modelica language (Modelica Association, 2017; Fritzson, 2015) is a standardized tool-independent non-proprietary equation-based object-oriented modeling language, which was introduced 20 years ago by the non-profit Modelica Association, with strong links to industry and academia. This language, and the related eco-system of tools, model libraries and the FMI standard (Blochwitz et al., 2011), is ideally suited to system-level modeling of complex, heterogenous and multi-domain cyber-physical systems. It has become a de-facto standard in many industries, most notably the automotive one. The Modelica language is currently supported by about 10 different modeling and simulation software tools; one of them, in particular, the open-source OpenModelica software suite (Fritzson et al., 2018), is the only Modelica tool owned and maintained by a non-profit organization – the Open Source Modelica Consortium (OSMC).

The main applications of Modelica tools so far have been the study of individual systems, such as a car's drivetrain and active suspension and steering control system, a single industrial robot, a single power plant, a single HVDC power link, the air conditioning system of a car, etc. Existing Modelica tools employ strategies and algorithms that are optimized for such system models, whose typical complexity lies in the range of 1000-50000 equations and up to a few thousand state variables. The advent of the internet-of-things paradigm is now fostering the development of innovative very large-scale cyber-physical systems, for example smart grids, or fleets of autonomous vehicles. It is also sparking a renewed interest at the modernization of traditional large-scale systems. A first example is continental-size high-voltage power generation and transmission, which is facing increasing challenges due to the introduction of power electronics equipment and to the increased penetration of intermittent renewable energy sources. A second example is district heating, possibly integrated with heat pumps and distributed power generation in an integrated electrical and thermal smart grid. See (Casella, 2015) for further examples and motivation.

Unfortunately, when Modelica is used to tackle the modelling of large-scale systems with sizes exceeding the ones mentioned above, currently available simulation software that support Modelica fall short at providing adequate performance. The time required to compile the models vastly exceeds what end users typically expect for system level studies, i.e., a few minutes at most. The size of the generated code and the memory requirements for compilers vastly exceed what is normally available on laptops and workstations used for daily work (8-16 GB).

In the last couple of years there have been some pioneering attempts at pushing the boundary of the size of Modelica models that can be handled with reasonable time and effort. In particular, some of our published papers have demonstrated the feasibility of Modelica models of high-voltage power generation and transmission systems (Braun et al., 2017; Casella et al., 2017) and of detailed models of key system components of future nuclear fusion reactors, see (Froio et al., 2017). The size of the largest models handled so far is about 750000 equations, which is about one order of magnitude bigger than the typical size mentioned earlier.

The results were indeed very interesting and sparked a lot of interest in the Modelica community. On the other hand, they clearly showed the limits of current Modelica tool technology, which is rather strained in terms of time and memory requirements at that scale, and that cannot in practice handle models of a size larger than one million equations. Breaking that barrier and achieving the 10 million equations model size goal requires fundamental methodological breakthroughs.

To summarize, the Modelica language has a lot of potential to support the system-level modelling of innovative engineering systems that require large-scale models. However, current Modelica tools have serious limitations as the system size grows.

1.1 Overcoming the Size Barrier with New Efficient Flattening Approach

An important goal of this work is overcoming the size barrier of current Modelica simulation tools, making it possible to efficiently generate fast simulation code for systems of up to 10 million equations, enabling new important applications such as those mentioned earlier, including large-scale networked system models. Overcoming the size barrier to 10 million equations means handling one to two orders of magnitude larger models than what is currently possible with state-of-the-art tools. To keep the total simulation time within reasonable bounds, the time needed for the model compiler to generate executable code should be in the order of minutes in the worst case, and the memory requirements should be fulfilled by the standard memory size available on laptop computers (16 GB), or possibly on engineering workstations for the largest problems (64 GB). The size of the executable code should also be much smaller than what can be achieved today, otherwise most of the simulation time risks to be spent waiting for data to be shuffled back and forth between RAM and CPU cache.

The availability of such a tool will allow to use the Modelica language, its high-level declarative modelling paradigm, to support a wide range of large-scale system design activities, as discussed in the previous sections. To realize these goals, a large new tool development within the OpenModelica tool suite was initiated about two years ago, in particular the development of a new highly performing compiler frontend, reported in this paper. More than half of the OpenModelica model compiler has been re-written and extended, and the software tool architecture significantly enhanced.

Traditionally, when a Modelica compiler is generating the simulation code, the system model is first flattened (expanded), i.e., reduced to a large system of scalar equations, before performing structural analysis and code generation. Although this process allows to combine components belonging to different domains in a straightforward way, this approach is obviously inefficient when there are many similar components in a system model, that only differ by their parameters, because the structural analysis and the generated code will be highly redundant.

Arrays of models, or even multiple instances of the same model, which only differ by the values of their parameters, should not be flattened to their scalar equations, but rather handled in an efficient way throughout the code generation process. Structural analysis and symbolic simplification of models which are instantiated multiple times should be performed only once instead of many times. At the system level, structural analysis of the overall system of equations should use algorithms and methods that consider arrays as symbolic entities instead of breaking them down to individual components. The efficiency of the final code generation process should also be improved so that ideally, if there are 1000 instances of the same component in a model, code should be generated for the equations of one of them only, and then called 1000 times, so as to drastically reduce the code generation time and memory consumption.

Achieving this goal requires fundamental changes to the structure of the Modelica tool with respect to the current state-of-the-art, which is to perform flattening to scalar equations before starting the code generation phase.

2 Related Work

Instantiation and flattening of Modelica is quite complex. Even as of the time of this writing there are open discussion on the Modelica issue tracker about unclear parts of the Modelica specification with regards to flattening. Furthermore, there is no available information on the instantiation and flattening process in the commercial Modelica tools – this is only available for the two open-source Modelica tools available: OpenModelica and JModelica.org.

JModelica.org is based on JastAdd (Hedin and Magnusson, 2003), a Java based meta-compilation system that supports Reference Attribute Grammars (RAGs). The instantiation and flattening in JModelica.org is detailed in (Åkesson et al., 2010). The process is similar to the one in OpenModelica. The instance ASTs (abstract syntax trees) are created from source ASTs and data is referenced using inter-AST references. From the instance ASTs trees the Flat ASTs are generated. The difference between OpenModelica and JModelica.org comes down to the fundamental differences between JastAdd and MetaModelica (Pop and Fritzson, 2006). The JastAdd framework computes attributes in the ASTs based on user-defined equations that relate to existing or circular attributes. In MetaModelica we use functional programming via functions and pattern matching to compute these attributes. JastAdd translates to Java, MetaModelica translates to C code. Both frameworks have automatic garbage collectors.

Interested readers can read more about compilers in (Aho et al., 1986). More on functional programming is available in (Hudak, 2000; Milner et al., 1997). Our previous work on boostrapping the OpenModelica compiler can be found in (Sjölund et al., 2014).

3 OpenModelica Compiler New Frontend Architecture

This section details the architecture and design of the new frontend. The new frontend is implemented in modern MetaModelica 3.0 which combines Modelica features with functional languages features. The implementation consists of 65 MetaModelica packages or uniontypes defining encapsulated data structures and functions that operate on the defined data.

3.1 New Frontend Typical File Structure

The new frontend uses the full capabilities of MetaModelica 3.0 which simplifies the code, control flow and architecture.

Data structures are defined using uniontypes and records. For example the flat model obtained after instantiation and flattening was performed is defined as below.

```
encapsulated uniontype NFFlatModel
  import Equation = NFEquation;
  import Algorithm = NFAlgorithm;
  import Variable = NFVariable;

  record FLAT_MODEL
    list<Variable> variables;
    list<Equation> equations;
    list<Equation> initialEquations;
    list<Algorithm> algorithms;
    list<Algorithm> initialAlgorithms;
    Option<SCode.Comment> comment;
  end FLAT_MODEL;

end NFFlatModel;
```

Encapsulation of data definition and functions that work on the defined data is similar to Modelica. Below is a partial definition of a binding in the new frontend together with functions to access or query it.

```
encapsulated package NFBinding
  public
    import Expression = NFExpression;
    import NFInstNode.InstNode;
    import SCode;
    import Type = NFType;
    import NFPrefixes.Variability;
    import Error;
  protected
    import Dump;
  public
    constant Binding EMPTY_BINDING
      = Binding.UNBOUND();

uniontype Binding
  record UNBOUND
  end UNBOUND;

  record UNTYPED_BINDING
    Expression bindingExp;
    // ...
  end UNTYPED_BINDING;

  record TYPED_BINDING
    Expression bindingExp;
    // ...
  end TYPED_BINDING;

public
  function isBound
    input Binding binding;
    output Boolean isBound;
  algorithm
    isBound := match binding
      case UNBOUND() then false;
      else true;
    end match;
  end isBound;

  function untypedExp
    input Binding binding;
    output Option<Expression> exp;
  algorithm
    exp := match binding
      case UNTYPED_BINDING()
        then SOME(binding.bindingExp);
      else NONE();
    end match;
  end untypedExp;

  function typedExp
    input Binding binding;
    output Option<Expression> exp;
  algorithm
    exp := match binding
      case TYPED_BINDING()
        then SOME(binding.bindingExp);
      else NONE();
    end match;
  end typedExp;

end Binding;
end NFBinding;
```

One can note some of the new features in MetaModelica 3.0:

- does not require verbose listing of all components (or named component access) of the record in the pattern matching (`UNTYPED_BINDING()`)
- accesses record components via the dot notation inside the case (`binding.bindingExp`).
- allows definitions of functions inside uniontypes
- allows definitions and the use of generic datatypes such as trees using redeclare/replaceable types

3.2 Features Relevant to High Performance

The new frontend was carefully designed with performance and scalability in mind.

References (pointers) are used to link component references to their definition scope via lookup and usage scope via application.

Constant evaluation and expression simplification are more restricted compared to the old frontend.

Both arrays of basic types and arrays of models are not expanded until the Scalarization phase (see next section).

Expansion of arrays is currently needed because the backend cannot handle all the cases of non-expanded arrays. See Section 4 on preliminary handling of non-expanded arrays of models in the backend and runtime.

3.3 New Frontend Design

The old OpenModelica frontend builds a DAE structure (flattened Modelica code) directly from the SCode structure (simplified parsed abstract syntax tree) for a model.

This means that it takes one component and flattens it to list of variables and equations for that single component before continuing with the next component. This flattening process involves doing instantiation, scalarization of arrays, typing, and so on.

Components in Modelica models often have dependencies on other components though, and the approach taken by the old frontend means that components sometimes need to be partially or fully flattened out of order. This has made it hard to implement certain features, such as redeclares, and has also led to a lot of superfluous flattening where parts of the model are flattened multiple times.

One of the driving forces in the design of the new frontend has therefore been to find ways to break dependencies between the various frontend phases. Instead of being component-focused like the old frontend it has instead been designed to be model-focused, meaning that each frontend phase processes the whole model before the model is passed on to the next phase. The result is the design seen in Figure 1, which shows the flow of the model through the different phases of the new frontend. The following sections will describe each phase in more detail.

3.3.1 Instantiation

The instantiation phase takes all the libraries and models that have been loaded by the compiler in the form of an SCode structure as well as the name of the model that should be instantiated, and builds an instance tree for that model. The instance tree consists of the class instance corresponding to the model as the root node, with the component instances of the class as child nodes that themselves have component instances (as seen in Figure 2).

Because the SCode structure is not suitable for name lookup, as it only contains lists of elements, the instance tree is instead used for this purpose by the new frontend since each node contains a lookup tree. The first task of the instantiation is thus to partially instantiate the conceptual root class that contains all top-level classes, which mainly involves constructing a lookup tree. The first part of the model name can then be looked up in the root class, and the rest of the name is looked up recursively using the same process.

Once the SCode element of the model's class has been found, it will then be instantiated, which involves three stages: partial instantiation, expansion, and full instantiation. Partial instantiation will, as mentioned, construct a lookup tree, but only local classes and imported names are added in this stage. This is needed to be able to look up the names of base classes, since Modelica allows classes to inherit from local and imported classes but not from inherited classes.

The next stage, expansion, uses this partial lookup tree to resolve any base classes of the class. All the inherited names as well as the names of local components are then added to the lookup tree. The reason why local components are not added until this stage is because the order in which the local and inherited components are declared needs to be preserved, since this is important for e.g. functions where the order of the function parameters matter. The class elements are therefore stored in declaration order in arrays, with the lookup tree only referencing elements in those arrays, and the inherited components need to be known before all components can be added in the correct order.

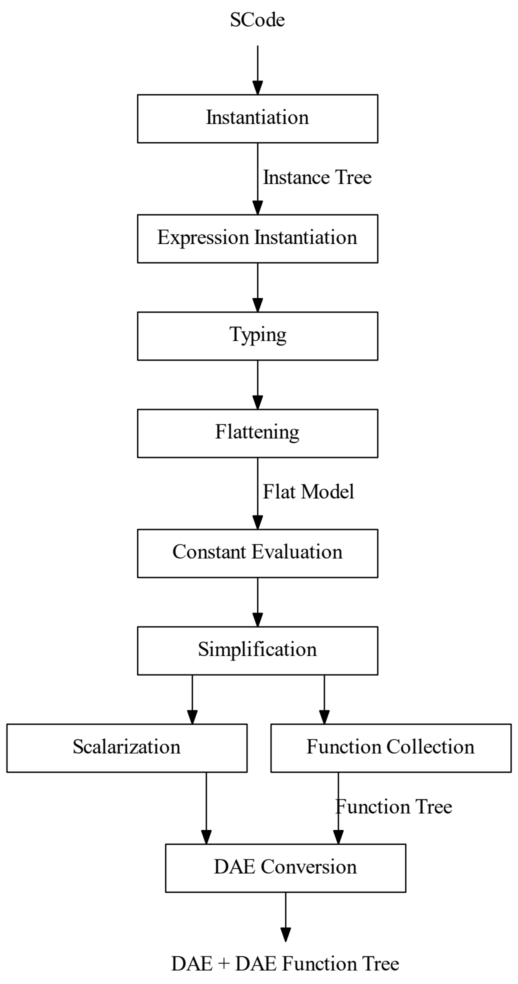

Figure 1. Frontend phases.

The final stage is full instantiation in which the components of the class are instantiated, which involves looking up the type of each component and instantiating it. In this stage modifiers are also associated with the elements they modify, and redeclares are applied. The names of any base classes are also looked up again in this stage, to make sure that the same classes are found as in the earlier expansion stage since inheriting from an inherited class is illegal in Modelica (see Figure 3). This conveniently also allows the frontend to also check that no extends is referencing a component, since those are, as mentioned earlier, added to the lookup tree after resolving base class names.

Because modifiers are only applied in the full instantiation stage, it is possible for the new frontend to cache the

```
model A
  Real x;
  Real y;
end A;

model B
  A a;
  Real z;
end B;

model M
  B b;
end M;
```

$\Rightarrow$

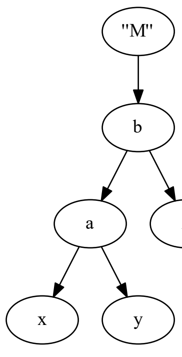

Figure 2. Example of a model and its instance tree.

```
model A
  model B
    ...
  end B;
end A;

model M
  extends A;
  // Illegal, B is inherited from A.
  extends B;
end M;
```

Figure 3. Example of illegal inheritance in a Modelica model.

work done during partial instantiation and expansion for each class. This means that e.g. the lookup tree for a class is only constructed once and then reused for all instances of that particular class, unlike the old frontend where a new lookup tree is constructed for each instance.

3.3.2 Expression Instantiation

Expressions in the compiler are things such as numbers, strings, unary and binary operator expressions, and named references to elements such as `a.b[2].c[4]`. They are used to represent things such as equations and algorithms, modifiers, and array dimensions.

The old frontend represents names used in expressions as a nested structure where each node contains the referenced element's name and type, the subscripts used, and for qualified names a reference to the next part of the name. The new frontend uses a similar representation, but instead of storing the element's name it stores a reference to the element's instance tree node (which itself contains the name).

This small difference in representation has a large impact on the design of the new frontend, because unlike the old frontend it always has direct access to the referenced elements. The old frontend is instead forced to look up names whenever it requires additional information about a name used in an expression, which due to how the old frontend is designed might require additional instantiation and other performance issues. Rewriting the old frontend to use a similar representation would have required a complete redesign, since it has no instance tree in the same way that the new frontend has.

For the new frontend this means that it needs to find the correct instance node during name lookup, which can be tricky to do during the instantiation since names can refer to elements that have not yet been instantiated. Expressions are not needed to build the instance tree though, so the instantiation is therefore separated into two phases: the instantiation phase described in the previous section and the expression instantiation phase described in this section.

The instantiation phase builds the instance tree and constructs all the nodes, and the expression instantiation phase instantiates all expressions in that instance tree. This involves looking up the names used in expressions and associating them with the correct nodes in the instance tree. In the case of function calls this also triggers instantiation of the called functions, which mostly involves instantiating the function as a normal class.

3.3.3 Typing

The typing phase traverses the instance tree and determines the type of all components and expressions. Similar to the instantiation this is again done in two stages: typing of components and typing of expressions.

The typing of components involves determining the type of each component in the instance tree. For components of basic types, such as `Real` or `Integer`, this is trivial by virtue of them being instances of said types. For composite types, such as instances of models, blocks or records, it means typing each child of the instance tree node and constructing a type from them.

The most complex part of the typing of components is typing dimensions for array components and classes. This is partly because dimensions can be expressions that need to be evaluated, and partly because they can be declared as `:` which means that the dimension size must be deduced from the component's binding equation.

This can require typing components that have not yet been typed, and must be done with care to avoid introducing unnecessary dependencies between dimensions. Having dimensions whose size depend on each other could result in a typing loop that would cause the compiler to hang or crash, but the new frontend detects such loops and gives an appropriate error message instead. In many cases such loops can be avoided by typing as little as possible when determining the size of a dimension, such as only typing the first dimension of `a` when typing a dimension defined as `size(a, 1)`.

The next stage of the typing phase is typing of expressions, which involves typing binding equations, equations, and algorithms. This also includes checking that expressions are type compatible, for example checking that binding equations are type compatible with the components they belong to. Having a separate stage for typing of expressions is not strictly necessary in the same way as during the instantiation, but it means that expressions can be typed with the assumption that all components are already typed and acyclic. The typing of expressions therefore becomes less complicated and more optimized than it would be if all the typing were to be done in a single combined stage.

Most of the typing of expressions is fairly straightforward, but the typing of binding equations becomes somewhat non-trivial in the new frontend due to the way array components are handled. Take for example this model:

```
model A
  Real x;
end A;

model M
  A a[3](x = {1, 2, 3});
end M;
```

The old frontend would instantiate and type each element of the array component `a` and the modifier on `a` would be split, so the component would thus be instantiated and typed as `a[1].x = 1`, `a[2].x = 2`, and `a[3].x = 3` (where `[1]`, `[2]`, and `[3]` are subscripts).

The new frontend instead treats this as one component, `a[3].x = {1, 2, 3}` (where `[3]` is a dimension), which is achieved by keeping track of where a modifier comes from and adding the appropriate dimensions to the component's type when type checking the binding equations. In this particular case it would thus add the dimension `[3]` to the type of `x` when checking that the binding equation is type compatible, in other words type checking one `Real[3] == Real[3]` relation whereas the old frontend would type check three separate `Real == Real` relations. The superior efficiency of this approach in the case arrays with thousands of elements need to be instantiated is pretty obvious.

3.3.4 Flattening

The flattening phase of the new frontend traverses the instance tree and flattens the tree into a flat model that consists of a list of variables, a list of equations and a list of algorithms:

```
model A
  Real x;
  Real y;
equation
  y = der(x);
end A;                  model M
                          Real b.a.x;
model B                   Real b.a.y;
  A a;           ⇒      equation
equation                  b.a.y = der(b.a.x);
  a.x = time;             b.a.x = time;
end B;                  end M;

model M
  B b;
end M;
```

The flattening involves prefixing component names and element name references in expressions with the names of their parents in the instance tree, to make sure all variables in the flat model have unique names. It also collects all the connect-equations in the model and inserts the required equations generated from the connections into the flat model.

Another task done by the flattening phase is unrolling for-equations into scalar equations:

```
for i in 1:3 loop                x[1] = 1;
  x[i] = i;             ⇒        x[2] = 2;
end for;                         x[3] = 3;
```

This might be considered more appropriately done by the later scalarization phase, or preferably not done at all, even though the connection handling requires for-equations containing connect-equations to be unrolled. The current backend additionally requires all for-equations to be unrolled, so at the moment the flattening unrolls all for-equations by default, regardless of whether they contain connect-equations or not. However, it is possible to disable the loop unrolling (as well as other scalarization features discussed later on in the paper) with the `-d=-nfScalarize` debug flag, which allows to experiment with extensions of the backend, code generation, and runtime phases that can handle arrays directly.

3.3.5 Constant Evaluation

Some parts of the frontend evaluate expressions when needed, for example when typing dimensions consisting of arbitrary expressions where the actual size needs to be known in a model context (unlike in a function context). Constants that are not used in such places should still be evaluated though, which is done in the constant evaluation phase. This phase traverses the flat model and replaces references to constants with the values bound to those constants:

```
model M
  constant Real x = 1.0;         model M
  Real y;                          Real y;
equation                 ⇒       equation
  y = x;                           y = 1.0;
end M;                           end M;
```

Models can also contain so called structural parameters, which are parameters used in places where they affect the structure of the model. One example is array dimensions which must as mentioned be known in a model context, but are allowed to be defined by parameters. Once such a parameter has been evaluated it should no longer be considered changeable, since changing its value after the model has been compiled could result in parts of the model using the old value and other parts the new value. The earlier parts of the new frontend therefore mark such parameters as structural, and the constant evaluation phase makes sure all occurrences in the model are replaced with the parameter's value.

3.3.6 Simplification

The simplification phase traverses the flat model and simplifies expressions, equations and algorithms. This includes doing trivial simplifications such as evaluating unary and binary operations involving numerical literals, e.g. `1 + 1` ⇒ `2`, but also structural changes such as removing for-loops with zero-sized iteration ranges.

3.3.7 Scalarization

The scalarization phase expands array variables and equations into separate scalar variables and equations:

```
                                      model M
                                        Real x_1;
model M                                 Real x_2;
  Real x[3];                            Real x_3;
equation                    ⇒         equation
  x = {1, 2, 3};                        x_1 = 1;
end M;                                  x_2 = 2;
                                        x_3 = 3;
                                      end M;
```

This is not necessary for the operation of the frontend itself, but is done because the old frontend does it and the backend expects it to be done. A long term goal is to improve the handling of arrays in the backend though, partially or completely removing the need for this phase (see Section 4).

Since the scalarization is a separate phase in the new frontend it can also easily be disabled, unlike in the old frontend where the scalarization is an integral part of the flattening process that is hard to isolate. This is currently possible by means of the already mentioned `-d=-nfScalarize` debug flag.

3.3.8 Function Collection

The flat model contains only the variables, equations and algorithms of the model. The functions used in the model are stored in the instance tree nodes corresponding to the functions' classes, but the backend expects to get a binary search tree containing the functions that are used in the model.

The new frontend therefore has a phase that goes through the model and collects all functions that are used in the model into a function tree. Besides explicitly called functions, this also includes, e.g., record constructors for all record instances, which might be needed by the backend even if they are not explicitly called in the model.

3.3.9 DAE Conversion

The old frontend produces a DAE structure that's used as an immediate representation of the flat model, and the OpenModelica backend expects the model to be given in this format.

The flattening phase of the new frontend uses its own representation of a flat model though, since using the old DAE structure would cause many of the advantages of the new frontend to be lost (such as name references in expressions having direct access to the instance tree nodes).

The new frontend therefore contains a final phase that converts the flat model and the function tree to the DAE structure expected by the backend. This serves as an interface between the new frontend and the backend, and is relatively straightforward since the DAE structure is mostly a subset of the data structures used by the new frontend.

4 Compilation of Vectorized Models for New Digital Applications

Many emerging applications require the individual control of vast numbers of similar devices that share a common system infrastructure. Such applications include distributed renewable power generation and charging of electric vehicles that share the same power grid. Other applications arise from autonomously driving cars that share the same roads. The Internet of Things opens the possibility to connect such devices to digital twins and to implement supervisory control applications in the cloud.

Unfortunately today's Modelica tools typically suffer from bad performance for the translation of resulting large models. This is caused by the Modelica DAE representation defined for a flat model. Even though the Modelica syntax supports arrays of model objects to express repeated structures, the expansion of arrays during flattening results in large numbers of scalar variables and equations that slow down the translation. This is particularly bad if the computational effort of the used algorithms grows more than linearly with increasing model size. Moreover, the resulting executable model code becomes unnecessarily large.

The new frontend offers the feature to convert arrays of component models to array equations and to keep arrays during flattening. Additionally, the current backend has been extended prototypically, by exploiting previous work by (Schuchart et al., 2015) to treat for-equations during model translation and by (Franke et al., 2015) to treat unexpanded arrays and array slices in the generated code.

Consider the following example. It instantiates a large number of solar plants as array of component models and connects them to a collector grid.

```
package Vectorized
  import SI = Modelica.SIunits;

  connector Terminal
    SI.Voltage v;
    flow SI.Current i;
  end Terminal;

  model SolarPlant
    input Boolean on "Plant status";
    input SI.Power P_solar "Solar power";
    parameter Real eta = 0.9 "Efficiency";
    Terminal term;
  equation
    term.v * term.i =
      if on then eta * P_solar else 0;
  end SolarPlant;

  model Collector
    parameter Integer n;
    parameter SI.Voltage V = 1000;
    output SI.Power P_grid;
    Terminal terms[n];
  equation
    for i in 1:n loop
      terms[i].v = V;
    end for;
    0 = P_grid + terms.v * terms.i;
  end Collector;
  model SolarSystem
    parameter Integer n = 1000;
    SolarPlant plant[n](
    each on = true,
    P_solar = 100:100:n*100);
    Collector grid(n = n);
  equation
    connect(plant.term, grid.terms);
  end SolarSystem;

end Vectorized;
```

With a number of n=1000 solar plants, the flat model would have 6001 variables and 6001 equations. The number of variables reduces to 7 when preserving arrays. Then the whole dependency analysis, equation sorting and code generation only treat 7 equations. This is possible as each array variable is defined with one array equation, resulting in a balanced array model.

The current implementation in OpenModelica still considers the dimension parameter *n* as structural and fixes its value during model instantiation. This is not needed though. The actual value of *n* could be left undefined until model execution.

The flat array model reads:

```
class SolarSystem
  parameter Integer n = 1000;
  Real[1000] plant.term.i;
  Real[1000] plant.term.v;
  parameter Real[1000] plant.eta = 0.9;
  Real[1000] plant.P_solar =
    (100:100:100000);
  Boolean[1000] plant.on = true;
  parameter Integer grid.n = 1000;
  parameter Real grid.V = 1000.0;
  Real grid.P_grid;
  Real[1000] grid.terms.i;
  Real[1000] grid.terms.v;
equation
  plant.term.v = grid.terms.v;
  plant.term.i + grid.terms.i = 0.0;
  for $i in 1:1000 loop
    plant[$i].term.v * plant[$i].term.i =
      if plant[$i].on then
        plant[$i].eta * plant[$i].P_solar
      else
        0.0;
  end for;
  for i in 1:1000 loop
    grid.terms[i].v = grid.V;
  end for;
  0.0 = grid.P_grid +
    grid.terms.v * grid.terms.i;
end SolarSystem;
```

The array of connectors between plant and grid results in array equations. The flow equation, as well as some variable bindings, relate arrays to scalars. This simplifies the further treatment up to code generation, implicitly assuming an "each" qualifier. Note that this only happens after the check of types and dimensions. It is crucial to avoid unnecessary expansions of large literal arrays.

Many Modelica expressions cannot be vectorized easily. For instance the condition of an if-expression must be a scalar boolean in an array equation as well. This is why the vectorization of component models converts non-trivial equations to for-equations. See the equation with P_solar as an example.

The early prototype implementation of vectorized models presented here already proved useful in first production uses (see next section for benchmarks). The model given in this section can be compiled and simulated without array expansion via flags `-d=newInst,-nfScalarize --simCodeTarget=Cpp`. Future work needs to focus on enhanced preservation of arrays during symbolic transformations in the backend. The frontend might expand arrays selectively, e.g. expand two or three dimensional arrays in electrical multi-phase models, while preserving large arrays of component models along with dimension parameters.

5 Status and Benchmarks

The new frontend is still work in progress at the time of this writing (January 2019). It is currently able to process about 75% of the 7884 models with an `experiment(StopTime)` annotation in the set of 55 tested open-source Modelica libraries that are included in the extended test-suite of the OpenModelica continuous integration system. The development effort so far has been focused towards achieving full coverage of the Modelica Standard Library (MSL) 3.2.3, for which the fraction of successfully simulating models is currently 92%, including non-trivial models such as the 6 d.o.f. robot model of `Modelica.Mechanics.Multibody` and models using the IF97 water model of `Modelica.Media`.

The updated status of the coverage is available online (New FrontEnd - Modelica Library Coverage). The development of the new frontend can be followed on (New FrontEnd - Ticket 4138); in particular, the progress of the coverage of the development version of MSL 3.2.3 is shown in Fig. 4. The new frontend is currently able to process all models except one, though there are still some issues that are revealed later in the code generation process, either because of incorrectly flattened models, or because the model is flattened in a different way than the old frontend, which the back-end cannot handle correctly. Note that the verification indicator is not reliable, due to many false negatives and to the lack of reference results for all the models introduced in version 3.2.3.

During the last six months, the number of successfully simulating MSL models has steadily increased at a rate of about 8%/month, so it is expected that full coverage for the MSL will be achieved by the end of Q2 2019 at the latest; the coverage of the 55 open-source library testsuit should approach 100% before the end of 2019.

All the benchmarks have been realized on a portable computer: HP ZBook Studio G3 I7 QuadCore 6820HQ @ 2.7Ghz with 16Gb of RAM.

To validate performance and scalability we have benchmarked the new OpenModelica frontend against the state-of-the-art commercial tool Dymola 2019 (2018-04-11) (Dassault Systèmes) on some large models from the ScalableTestSuite library (Casella, 2015).

An example of the Dymola and OpenModelica scripts

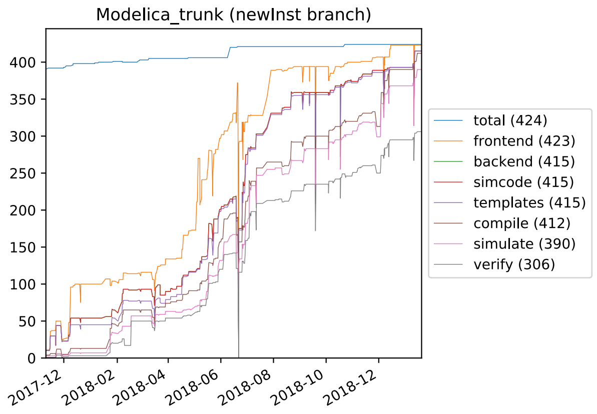

Figure 4. Modelica Standard Library version 3.2.3 coverage

used are given below. Note that this include parsing of the Modelica Standard Library, the ScalableTestSuite and running `checkModel` on the given model where ModelPath is the full path to the model in the library.

Dymola script:

```
openModel("ScalableTestSuite/package.mo");
checkModel("ModelPath");
exit();
```

OpenModelica script:

```
loadFile("ScalableTestSuite/package.mo");
getErrorString();
checkModel(ModelPath);
getErrorString();
```

The benchmarking was performed from command line using running adaptations of the scripts above. The results for selected ScalableTestSuite (STS) models and the Vectorized.SolarSystem from section 4 are given below in Table 1.

One can see that the new OpenModelica frontend performs very well in comparison to Dymola, in some cases faster, in some cases slower. The comparision between the current frontend (CF) and the new frontend (NF) is also included where possible. From these benchmarks one can also see that investigation is needed to find out why parameter arrays are scaling poorly in the new frontend (models 6, 7, 8). For models 10 and 11 the number in the parentheses is for the new frontend not expanding arrays at all during the flattening. The performance improvement in this case is extreme.

In Table 2 we compare the current frontend (CF) with the new frontend (NF) when instantiating and flattening models from `Modelica.Mechanics.MultiBody` and evaluating their graphical annotation. The OpenModelica compiler API function that is called to evaluate the graphical annoations is `getComponentAnnotations()`. The new frontend performs 20 to 200 times better than the current OpenModelica frontend, allowing to obtain a nearly immediate response time of the OMEdit GUI, which relies on this API.

6 Conclusions and Future Work

In this paper, the new high-performance frontend of the OpenModelica compiler is presented. The frontend has been completely redesigned, with the main objective of achieving dramatically improved performance on large models, as well as of resolving many corner-cases that the old frontend could not handle without the need of excessive ad-hoc work.

The architecture of the new design is presented in detail, particularly concerning the new approach that avoids the full expansion and scalarization of components one at a time, thus allowing significant optimizations when large numbers of instances of the same class, and/or large arrays, are present in the model. Many of these optimization would also require a substantial redesign of the compiler backend, code generation, and runtime system. For the time being the new non-scalarization approach has been experimented with a prototype implementation in the backend, which works in some specific cases, for which very promising results are reported.

Future developments involve first and foremost the finalization of the new frontend, with the aim of achieving 100% coverage of most open-source Modelica libraries, particularly the MSL. This goal is planned to be achieved during the first half of 2019. In the long term, the plan is to use the new frontend to achieve full support of non-expanded arrays of equations and models in the entire compiler toolchain, including also the backend, code generation, and runtime system.

Another research direction is to improve the compilation speed by using the LLVM framework to perform function evaluation in the new frontend.

7 Acknowledgements

This work has been supported by Vinnova in the ITEA OPENCPS and EMPHYSIS projects and in the Vinnova RTISIM and EMISYS projects. This work was supported in parts by the German Federal Ministry of Education and Research (BMBF) in the PARADOM project. Support from the Swedish Government has also been received from the ELLIIT project. The OpenModelica development is supported by the Open Source Modelica Consortium. Many students, researchers, engineers have contributed to the OpenModelica system. There is not room here to mention all these people, but we gratefully acknowledge their contributions.

References

Alfred V. Aho, Ravi Sethi, and Jeff D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1986.

Torsten Blochwitz et al. The Functional Mockup Interface for tool independent exchange of simulation models. In Christoph Clauß, editor, *Proceedings of the 8th International Modelica Conference*. Linköping University Electronic Press, March 2011. doi:10.3384/ecp11063105.

Willi Braun, Francesco Casella, and Bernhard Bachmann. Solving large-scale Modelica models: new approaches and experimental results us-

No	Model	Equations	Dym (s)	OMC NF/CF (s)
1	Electrical.DSystemAC.SE.DistributionSystemLinear_N_40_M_40	99776	15.53	06.32 / 91.33
2	Electrical.DSystemAC.SE.DistributionSystemLinear_N_80_M_80	397936	40.50	17.76 / 435.32
3	Electrical.DSystemAC.SE.DistributionSystemLinear_N_112_M_112	779312	74.21	32.31 / 1076.54
4	Electrical.DSystemDC.SE.DistributionSystemModelicaActiveLoads_N_80_M_80	129929	18.04	08.33 / 159.28
5	Electrical.TransmissionLine.SE.TransmissionLineModelica_N_1280	26915	09.84	04.45 / 47.77
6	Elementary.ParameterArrays.SE.Table_N_100_M_100	0	06.59	05.09 / 06.21
7	Elementary.ParameterArrays.SE.Table_N_400_M_400	0	10.25	12.19 / 18.03
8	Elementary.ParameterArrays.SE.Table_N_1600_M_100	0	09.77	19.04 / 28.17
9	Power.ConceptualPowerSystem.SE.PowerSystemStepLoad_N_64_M_16	11907	17.29	03.99 / 28.57
10	Vectorized.SolarSystem(n=10000) from section 4	60001	146.30	34.12 / 314.8 (**02.95**)
11	Vectorized.SolarSystem(n=100000) from section 4	600001	14458.68	2450.57 / 19760.42 (**02.95**)

Table 1. Flattening performance comparison Dymola vs. OpenModelica (NF vs CF included). Bold numbers in parentheses are with Scalarization disabled `-d=-nfScalarize`. Shortened names: SE=ScaledExperiments, DSystem=DistributionSystem.

Model	CF (s)	NF (s)	Factor
World	9.53	0.28	33.9
Joints.FreeMotionScalarInit	28.90	0.14	199.4
Joints.Planar	3.56	0.13	25.6
Joints.UniversalSpherical	6.99	0.22	30.5
Joints.SphericalSpherical	4.64	0.11	39.5
Joints.Universal	2.31	0.12	18.4

Table 2. Flattening performance comparison of the current (old) vs the new frontend in OpenModelica (OMEdit GUI impact).

ing OpenModelica. In *Proc. 12th International Modelica Conference*, pages 557–563, Prague, Czech Republic, May 15–17 2017. doi:10.3384/ecp17132557.

Francesco Casella. Simulation of large-scale models in Modelica: State of the art and future perspectives. In Peter Fritzson and Hilding Elmqvist, editors, *Proceedings 11th International Modelica Conference*, pages 459–468, Versailles, France, Sep 21–23 2015. The Modelica Association. ISBN 978-91-7685-955-1. doi:10.3384/ecp15118459.

Francesco Casella, Alberto Leva, and Andrea Bartolini. Simulation of large grids in OpenModelica: reflections and perspectives. In *Proc. 12th International Modelica Conference*, pages 227–233, Prague, Czech Republic, 2017. doi:10.3384/ecp17132227.

Dassault Systèmes. Dymola version 2019, 2018. URL `http://dymola.com`.

Rüdiger Franke, Marcus Walther, Niklas Worschech, Willi Braun, and Bernhard Bachmann. Model-based control with FMI and a C++ runtime for Modelica. In *Proceedings of the 11th International Modelica Conference*. Modelica Association, Paris, France, 2015.

Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley-IEEE Press, 2 edition, April 2015. ISBN 978-1-118-85912-4.

Peter Fritzson, Adrian Pop, Adeel Asghar, Bernhard Bachmann, Willi Braun, Robert Braun, Lena Buffoni, Francesco Casella, Rodrigo Castro, Alejandro Danós, Rüdiger Franke, Mahder Gebremedhin, Bernt Lie, Alachew Mengist, Kannan Moudgalya, Lennart Ochel, Arunkumar Palanisamy, Wladimir Schamai, Martin Sjölund, Bernhard Thiele, Waurich Volker, and Per Östlund. The OpenModelica Integrated Modeling, Simulation and Optimization Environment. In Michael Tiller and Luigi Vanfretti, editors, *Proceedings of the 1st American Modelica Conference*. Linköping University Electronic Press, October 2018. URL `http://www.ep.liu.se/`.

Antonio Froio, Francesco Casella, Fabio Cismondi, Alessandro Del Nevo, Laura Savoldi, and Roberto Zanino. Dynamic thermal-hydraulic modelling of the EU DEMO WCLL breeding blanket cooling loops. *Fusion Engineering and Design*, 124:887–891, 2017. doi:10.1016/j.fusengdes.2017.01.062.

Görel Hedin and Eva Magnusson. JastAdd: An aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58, April 2003. ISSN 0167-6423. doi:10.1016/S0167-6423(02)00109-0. URL `http://dx.doi.org/10.1016/S0167-6423(02)00109-0`.

Paul Hudak. *The Haskell School of Expression: Learning Functional Programming Through Multimedia*. Cambridge University Press, New York, NY, USA, 2000. ISBN 0-521-64408-9.

Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814.

Modelica Association. Modelica: A unified object-oriented language for physical systems modeling, language specification version 3.4, 2017. URL `http://www.modelica.org/`.

New FrontEnd - Modelica Library Coverage. New FrontEnd - Modelica Library Coverage, 2018. URL `https://libraries.openmodelica.org/branches/overview-newinst.html`.

New FrontEnd - Ticket 4138. New FrontEnd - Ticket 4138, 2018. URL `https://trac.openmodelica.org/OpenModelica/ticket/4138`.

OSMC. Open Source Modelica Consortium, 2007. URL `https://openmodelica.org/home/consortium`.

Adrian Pop and Peter Fritzson. MetaModelica: A unified equation-based semantical and mathematical modeling language. In *7th Joint Modular Languages Conference, JMLC 2006 Oxford, UK, September 13-15, 2006 Proceedings*, pages 211–229. Springer Berlin Heidelberg, 2006. doi:10.1007/11860990_14.

Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Implementation of a Modelica compiler using JastAdd attribute grammars. *Sci. Comput. Program.*, 75(1-2):21–38, January 2010. ISSN 0167-6423. doi:10.1016/j.scico.2009.07.003. URL `http://dx.doi.org/10.1016/j.scico.2009.07.003`.

Joseph Schuchart, Volker Waurich, Martin Flehmig, Marcus Walther, Wolfgang E. Nagel, and Ines Gubsch. Exploiting repeated structures and vectorization in Modelica. In *Proceedings of 11th International Modelica Conference*. Modelica Association, Paris, France, 2015.

Martin Sjölund, Peter Fritzson, and Adrian Pop. Bootstrapping a Compiler for an Equation-Based Object-Oriented Language. *Modeling, Identification and Control*, 35(1):1–19, 2014. doi:10.4173/mic.2014.1.1.