Notes on the Separate Compilation of Modelica

Ch. Höger F. Lorenzen P. Pepper

Fakultät für Elektrotechnik und Informatik, Technische Universität Berlin
{choeger, florenz, pepper}@cs.tu-berlin.de

Abstract

Separate compilation is a must-have in software engineering. The fact that Modelica models are compiled from the global sources at once results from the language design as well as from the way compiled physical models are finally simulated. We show that the language in fact can be compiled separately when certain runtime conditions are met. We demonstrate this by transforming some specific Modelica language features (structural subtyping and dynamic binding) into a much simpler form that is closer to current OO languages like C++ or Java.

Keywords Modelica, Separate Compilation

1. Introduction

Separate compilation is the state of the art in today's software engineering and compilation tools. Therefore it is also a natural request for Modelica tools. Large models (having hundreds or even thousands of equations) evolve over time, undergo small incremental changes and are often developed by whole teams. So it is unacceptable in practice, when a small modification in one class causes the recompilation of hundreds of unchanged classes. This could be avoided by the creation of smaller compilation units which are finally combined to create the desired model.

Unfortunately, it is not a priori clear that this approach is feasible for Modelica, since some features of the language are quite complex to handle with separate compilation. Potential problems could in particular arise in connection with the process of flattening, structural subtyping, inner/outer declarations, redeclarations and expandable connectors. We will sketch in the present paper solutions for all of these issues with the exception of expandable connectors, which would require a more in-depth discussion of the operational semantics of runtime instantiation.

The flattening of the whole model can actually be avoided by using object-oriented features of the target language: Instead of creating the set of equations, a compiler can directly translate the object tree into the target language. The equations can then be simply collected at runtime. This principle also leads to a elegant solution for the compilation of expandable connectors, as we will show in Section 2.

While Modelica's type system gives great flexibility in code re-usage, its translation into an object-oriented language with nominal subtyping like C++ is not straightforward. We will show how a compiler can handle this with the usage of coercions in Section 3.

Rumor has it that *separate compilation* is made very complex through the presence of Modelica's **inner** and **outer** pairing. We will show (in Section 4) that this rumor is unsubstantiated: By combining some kind of parameterization with standard typing principles we can not only solve the separate-compilation issue in a straightforward manner but also obtain a very clean and well comprehensible semantic definition of the inner/outer principles.

Finally we will summarize the costs and drawbacks of the separate compilation of Modelica models in Section 5.

1.1 Related work

Although separate compilation of Modelica models seems to be an important topic, only little research has yet been done in this field. In [8] and [9] some extensions to the language are proposed that would allow users to write models which can be compiled separately with nearly no impact on the quality of the generated code (as mentioned in Section 5). The work presented in [12] aims to decide wether or not separate compilation is possible (with respect to causalization) and worth the price for a given model. While both approaches do not allow the separate compilation of arbitrary compilation units, they could be combined easily with our method to raise the quality of generated code.

2. Principles of Separate Code Generation

As stated earlier the main reason for separate code generation is to reduce the overall compilation time by re-using output units that have not been affected by source level changes after the last compiler run. This at least requires a compiler to create output files for each input file given. Furthermore the content of those generated files should not depend on any context of their usage but only on their interfaces: If content from a source file is used two or more times the output file(s) should be usable in all those cases.

While the above requirements would generally suffice for separate compilation, a good design additionally enforces the separation of the compiler from the build sys-

3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. October, 2010, Oslo, Norway.

tem (so that the build system of choice can be used). This requires the compiler to produce a predictable set of output files for a given input file and the language to allow the detection of dependencies with simplistic tools. A good example for the usage of this principle is the compilation of C++ programs with GNU Make: Since the C++ compiler will create one object file per given input file and dependencies are clearly marked as file-wise includes, the build system can decide which source files need to be rebuilt given only modification times and regular pattern rules. Although imports in Modelica are only allowed from packages (and packages have a clear mapping into a file system layout) a Modelica type name cannot be mapped to the file it is defined in by such pattern rules. Therefore a complete dependency analysis must fully implement Modelica's (rather complex) lookup mechanisms. For smaller projects this dependency analyis might still be handled manually, but for complex uses one would have to generate e.g. makefiles automatically, to get all the benefits from incremental builds.

While the possibility to have a more fine grained build system is already extremely useful, there is another advantage of separate compilation that should be considered for Modelica: Currently Modelica library developers have to hide their expert knowledge from their customers by obscure methods like the Digital Rights Management that is part of the current Modelica specification[1]. If a library would be compiled separately, only the source code of the interface (comparable to a C/C++ header file) would have to be shipped. Also a clear separation between interface and code would be possible without any additional cost.

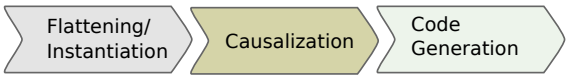

Figure 1. Common compilation of Modelica.

Unfortunately the compilation target of a Modelica model is a single matrix consisting of both differential and algebraic equations (short DAE). The usual processing scheme can be seen in Figure 1: After parsing (and correctness-checking) a Modelica model is flattened, meaning that both the inheritance and instance trees are resolved into sets of variables and (acasual) equations. Those equations can then be causalized and finally translated into the target language.

As can be seen in Figure 1, causalization of the model's equations depends on the model being completely instantiated. Obviously, there is no method to work around this dependency. Therefore, the only way to achieve real separate compilation is to move the code generation stage **in front** of the instantiation. This means that we are not going to generate code from the DAE but actually code that itself can generate the DAE with some help from the runtime system. Although this design has been in use with the Mosilab [10] compiler, it has neither been used for separate compilation nor formally specified. The closest thing to a specification of this method (although not implemented for Modelica) would be the Modeling Kernel Language semantics by David Broman [2].

The runtime instantiation implies that there is no need to translate flattened models (since the generation of the DAE can be done in the correct order). Also there is no need to generate all equations directly at compile-time. Instead the compiler might generate special functions that can create equations at runtime, which allows a quite natural translation of advanced language features like expandable connectors.

With this method, the output of our compilation process can easily satisfy the above requirements by directly mapping every Modelica source file to a compilation unit of the target language. The compiler of course has to use a naming scheme that allows the generated files to reference each other, but this is a trivial task.

3. Subtypes

In Modelica's type system [3] subtype relations are defined implicitly by the set of fields of a class. For separate compilation this means that (contrary to e.g. Java or C++) there is actually no need for a class B to know anything about a class A to be a subtype of that class. Therefore the compiler output of B must be able to be used with A and vice versa even if both were compiled completely separately.

```
class A                                  1
  output Real x;                         2
 equation                                3
  x = sin(time);                         4
end A;                                   5
class B                                  6
  Real y;                                7
  output Real x;                         8
 equation                                9
  y = 23.0;                              10
  x = 42.0;                              11
end B;                                   12
class Foo                                13
  replaceable A a;                       14
  output Real x;                         15
 equation                                16
  connect(a.x, x);                       17
  ...                                    18
end Foo;                                 19
class Bar                                20
  Foo foo1;                              21
  Foo foo2(redeclare B a);               22
end Bar;                                 23
```

Listing 1: Example for redeclaration.

In the listing of Listing 1 the instantiation of `foo2` is modified with a so called *redeclaration*, a construct that allows the enclosing class to replace an element of its child with an element of any compatible type (in other words: a subtype). Since both `Foo` and `B` may already have been compiled earlier, there is no way to modify one of the compiled classes to be type compatible to the redeclaration.

The same problems can arise from the use of the **extends** declaration:

```
class Baz                                    1
  extends Foo(redeclare B a);                2
end Baz;                                     3
```

In the above example we assume that code generation does not compile code from `Foo` into `Baz` (which would violate the first requirement of separate compilation, since compiling `Foo` would become useless). Therefore, if `Foo` is already compiled, all code generated from its equations has to be made type compatible with the new type of the field `a` *prior* to knowing how that new type actually looks like. If the target language has a nominal type system [1] (a rather common case), this is actually impossible without compiling the complete source code at once [5] — which is clearly no option in our case.

While we cannot change the code itself due to separate compilation requirements, we can build a bridge between the use and declaration site by means of coercion semantics.

3.1 Coercion semantics

Coercion semantics [11] is an implementation technique for languages with subtyping that translates a program with subtyping into a simpler one without subtyping. The general idea is as follows:

Whenever a given program context Γ requires an object a of class A any object b of a subclass B of A may also be used (context criterion). Since the runtime representation of b and a are different (usually apparent by a different type in the target language), e.g. b might have additional fields, the compiled form of Γ must be able to cope with many different object representations. If we demand that the compiled form of Γ is independent of the object given, i.e. a or b, it must inspect the object at runtime to access its fields correctly. This inspection, unfortunately, has a non-negligible runtime-overhead. We can circumvent this overhead by always passing an object of class a. Therefore, we have to coerce, hence the name coercion semantics, b to a before handing it to Γ. Since B is a subclass of A this coercion is always possible. Experience in other systems [6, 7] shows that the coercion is much cheaper, in terms of performance, than runtime inspection of objects. Applying coercions semantics, the compiled form of Γ has to operate on objects of class A, exclusively.

We illustrate this technique by an example. Class `Bar` in Listing 1 creates an instance of Class `Foo` while redeclaring `Foo`'s field `a` from `A` to `B`. To enable reuse of the code of `Foo` — `Foo` now acting as the context Γ — without expanding `B` into `Foo` and recompiling we coerce an object of class `B` to class `A`. We can describe the effect of this coercion by introducing a temporary class `B'` that is structurally identical to `A` but has `B`'s fields:

```
class B                                  class B'
  Real y;                                  output Real x;
  output Real x;        coerce           equation
equation                  ⟹               x = 42.0;
  x = 42.0;                              end B';
  y = 23.0;
end B;
```

We now redeclare `a` to `B'` instead of `B` and replace line 21 of Listing 1 by

```
Foo foo2(redeclare B' a);
```

This is possible since all information of `B` that is relevant to `Foo` is contained in `B'` and objects of `B'` can be represented in the same way as objects of `A`.

3.2 xModelica

To make use of this technique in Modelica with as little overhead as possible we have to make use of the fact that the object tree of a Modelica model is static. Thus coercions should not contain values (and be updated every time the original value changes), but rather express equivalence between two names. Usually in Modelica this can be expressed by equations, but since those are part of the actual model (which we do not want to change) and would have to be evaluated at runtime, we prefer to introduce the notion of references.

Technically we achieve this by adding some expressive power to Modelica. Inside a compiler this is simply done by providing additional constructs and fields in the abstract syntax tree, which do not have counterparts in the external syntax. But for discussing these issues it is preferrable to present the concepts in concrete syntax. Therefore we augment Modelica by some notations that allow us to express the additional concepts, thus obtaining an extended language *xModelica*. (Keep in mind, xModelica is not available to users, it is just a "prettyprinting" of the internal abstract syntax trees.) The following changes are made to transform a Modelica program into its xModelica representation:

1. Introduce type modifiers **{indirect, direct}**. Modifiers of this kind are well known in languages; for example, Java has a modifier set `{public, protected, standard, private}` (of which `standard` is invisible, that is, represented by writing nothing) or the modifier set `{final, nonfinal}` (of which the second one is again invisible). Modelica itself does the same with **inner** or **outer**.
2. A function **indirect** is added that creates an instance of an indirect type from a direct variable. This function is defined for every type but does not really need an implementation. In fact, this function could be seen as an explicit coercion itself (explicit because direct types are not subtypes of indirect types).
3. This modifier is an actual part of the type, that is, the type **direct** `Real` is different from the type **indirect** `Real`. But to simplify the usage of indirect types, we consider them being subtypes of their respective direct counterpart.

[1] Even if the generated code has no type system at all, at least the memory layout of objects would have to be made compatible.

4. The default type declaration is **direct**. Indirect types will only be introduced by some transformations. The distinction between indirect and direct types allows the clear usage of object references (which are not part of Modelica): Every instance of an indirect type can be seen as the reference to a direct type. This interpretation gives a natural meaning of the function **indirect** which simply returns a new reference to an already existing direct typed object. Note, that instead of explicitly defining a function **direct** that would do the opposite, we kept the design simple by using the subtype relation mentioned above (which has the same natural interpretation). Also we intentionally did not introduce something like references of references (we do not need that level of complexity until now).
5. The instantiation of a class can now have indirect parameters (declared after the name of the class). Since they are a special kind of parameter, we denote them by the special brackets $\langle \ldots \rangle$. The usual scoping rules apply to those parameters as well as the ability to use default parameters. The actual indirect parameters must be given at object instantiation.
   Note: In a clean language design, one could argue that the indirect arguments should be added to the class, not to objects. But we only want an external representation of the internal data structures. Hence, we properly reflect the fact that the additional field is indeed added to the object, not to the class.

```
record A* ⟨indirect output Real x⟩                                  1
end A*;                                                             2
class A                                                             3
  direct output Real x;                                             4
  ...                                                               5
end A;                                                              6
class Foo⟨indirect A a = indirect(A())⟩                             7
  direct output Real x;                                             8
  ...                                                               9
end Foo;                                                            10
function C_{B →A}                                                   11
  input indirect B b;                                               12
  output direct A* a⟨x = indirect(b.x)⟩();                          13
end C_{B →A};                                                       14
class Bar                                                           15
  Foo foo1;                                                         16
  B a;                                                              17
  A* a' = C_{B →A}(indirect(a));                                    18
  Foo foo2⟨indirect(a')⟩();                                         19
end Bar;                                                            20
```

Listing 2: Example of Listing 1 converted to xModelica

In our example of Listing 1 a coercion can then be expressed as shown in Listing 2. As can be seen the coercion transformation consists of several parts:

1. A class `A*` is introduced that contains an **indirect** field for each field of `A`. This class is by definition always a subtype of the original class (and can thus safely be used instead of the original). Note that it can be created while compiling the definition of `A`.
2. Every replaceable field of `Foo` is moved up to the indirect parameters list as the default definition of a indirect parameter of the same name. Again the resulting class is a subtype of the original class.
3. A coercion function $C_{B \to A}$ `: B` $\to$ `A*` is created. This function lifts every field of `B` into an indirect field of the same name in a instance of `A*`.
4. Finally the coercion function is applied to a local instance of `B` and its output is fed into the instatiation of `Foo`.

With this method the redeclaration statements can be transformed into a much more common concept. Therefore the translation into object oriented (or even functional) code is now much simpler. The new interface of instantiation ensures that no dependencies from redeclaration statements are left in the generated code and thereby make separate compilation possible.

3.3 Subtyping in functions

Subtyping in Modelica is not restricted to models and records but also covers functions. Unfortunately, the Modelica Specification (version 3.2 [1]) is unclear about when one function is a subtype of another: Since Functions are special classes in Modelica (and their parameters special fields), the subtyping rule for functions just references the rule for classes which makes no difference between **output** and **input** fields. Thus the principle of contravariance in function parameters [4] is violated. Since we want to focus on separate compilation, we will assume that this problem has been solved, by redefining the rules for subtypes of functions in the Modelica Specification.

```
function foo                                                        1
  input Bar bar;                                                    2
  output Baz bar;                                                   3
  ...                                                               4
end foo;                                                            5
...                                                                 6
MyBaz baz; //subtype of Baz                                         7
MyBar bar; //subtype of Bar                                         8
algorithm                                                           9
...                                                                 10
baz := foo(bar);                                                    11
```

Listing 3: Example for function application.

Function application with subtypes is pretty straightforward due to our coercion routine: The Modelica fragment in Listing 3 can easily be transformed into its corresponding xModelica fragment:

$C_{Baz \to MyBaz}$`(foo(`$C_{MyBar \to Bar}$`(bar)))`

The application of (subtype) functions is a little bit more complicated but still no big problem, as long as the above mentioned error is corrected. With the principle of contravariance a function $f : A \to B$ is a subtype of $g : C \to D$ if, for the types A, B, C, D, $A \subseteq C$ and $D \subseteq B$ holds true.

```
class C;                                          1
class A extends C;                                2
class B;                                          3
class D extends B;                                4
function F                                        5
  input A a;                                      6
  output B b;                                     7
end F;                                            8
function G                                        9
  input C a;                                     10
  input D b;                                     11
end G;                                           12
function H                                       13
  input function F f;                            14
  input  A a;                                    15
  output B b = f(bar);                           16
  ...                                            17
end H;                                           18
...                                              19
function G g //subtype of F;...                  20
B b := H(g, bar);                                21
```

Listing 4: Example for function contravariance.

The problem here is that while parameter coercion can take place just at the location of the function call, this is not possible for function parameters (simply because the possible type of the function is again unknown). Therefore the function parameter itself must be subject to coercion but not the function application. Since we did not introduce a xModelica notation for function parameters, we will use a mathematical notation here. The parameter g of `foo` in line 21 will be replaced by $C_{D\to B} \circ g \circ C_{A\to C}$. We'll leave it open on how to implement function composition but since function parameters are allowed in Modelica some kind of functional object will be needed in the runtime system anyway. The step to a higher order function is not too big from there.

4. Dynamic Binding (inner/outer)

One of the Modelica language features that causes trouble among users is the inner/outer pairing. Some people consider it as being mandatory for reasons of practical usability in certain application scenarios, others are deterred by the incomprehensibility and error-proneness, which the feature exhibits in particular in slightly more intricate constellations.

What are the reasons for these contradictory viewpoints? As usual a clarification of such seeming ad-hoc phenomena can be obtained by mapping them to classical concepts of programming language theory. Then it is immediately seen that we simply encounter the standard dichotomy between *static and dynamic binding* of variables. And – as usual – problems arise, whenever two concepts (even though each of them may be clean and clear in isolation) are mixed in some odd fashion.

As usual, the *scope* of a name `x` is the textual region of the program text, where it is known; let us denote that

```
class A                                           1
  Real x;                                         2
  class A1                                        3
    ... use x ...                                 4
  end A1;                                         5
  class A2                                        6
    Real x;                                       7
    ... use x ...                                 8
  end A1;                                         9
  ...                                            10
end A;                                           11
```

Listing 5: Simple scoping

region as $\texttt{scope}_{\texttt{x}}$. And it is also standard knowledge that such scopes can contain *holes* due to declarations of the same name inside the scope. As a consequence, any point in the program has for each name `x` a unique scope $\texttt{scope}_{\texttt{x}}$. In Listing 5, the scope of the variable `x` in line 2 is the whole region of the class `A` with the exception of class `A2`; hence, $\texttt{scope}_{\texttt{x}_2} = \texttt{[1..5]} \cup \texttt{[10..11]}$. Analogously, the scope of the variable `x` on line 7 is the region of class `A2`, that is, $\texttt{scope}_{\texttt{x}_7} = \texttt{[6..9]}$.

```
{ int a=0;                                        1
 fun f(int x) = a * x;                            2
 { int a = 2;                                     3
  f(3)                                            4
  ...                                             5
 }                                                6
}                                                 7
```

Listing 6: Different binding example.

Static binding states that for an applied occurrence of a name `x` at some program point p the corresponding declaration is directly given by the scope $\texttt{scope}_{\texttt{x}}$, in which p lies. In the above example, the application of `x` in line 4 refers to the declaration of `x` in line 2, since $4 \in \texttt{scope}_{\texttt{x}_2}$ and analogously `x` in line 8 refers to the declaration in line 7, since $8 \in \texttt{scope}_{\texttt{x}_7}$.

Dynamic binding uses a more complex principle for the association between an applied occurrence of a name `x` and its corresponding declaration – at least for local applications in functions, classes etc. Now we don't use the scope of the point, where `x` is applied, but the scope of the point, where the function, the class etc. is applied.

Listing 6 illustrates dynamic binding in some fictitious λ-style language, thus demonstrating that the concept is long known in many languages.

Under static binding the non-local name `a` in line 2 would refer to the declaration in line 1 such that the call `f(3)` would yield the value 0. But under *dynamic binding* the application of the non-local name `a` would refer to the declaration of `a` that is valid in line 4, that is, to the declaration in line 3. Hence the call `f(3)` yields the value 6.

```
class A                                   1
  outer Real x;                           2
end A;                                    3
class E                                   4
  inner Real x;                           5
  class F                                 6
    inner Real x;                         7
    class G                               8
      Real x;                             9
      class H                            10
        A a;                             11
      end H;                             12
      H h;                               13
    end G;                               14
    G g;                                 15
  end F;                                 16
  F f;                                   17
end E;                                   18
class I                                  19
  inner Real x;                          20
  E e;                                   21
  A a;                                   22
end I;                                   23
```

```
class A⟨indirect Real x⟩                  1
                                          2
end A;                                    3
class E                                   4
  indirect Real x;                        5
  class F                                 6
    indirect Real x;                      7
    class G⟨indirect Real dx⟩             8
      direct Real x;                      9
      class H⟨indirect Real dx⟩          10
        direct A a⟨dx⟩;                  11
      end H;                             12
      H h⟨dx⟩;                           13
    end G;                               14
    G g⟨x⟩;                              15
  end F;                                 16
  F f;                                   17
end E;                                   18
class I                                  19
  indirect Real x;                       20
  E e;                                   21
  A a⟨x⟩;                                22
end I;                                   23
```

Listing 7: Complex example from section 5.4 of the Modelica Specification.

```
class A                                   1
  outer Real x;                           2
  ... use x ...                           3
end A;                                    4
class B                                   5
  inner Real x;                           6
  A a1, a2;                               7
  ... use a1.x ... a2.x ...               8
end B;                                    9
```

Listing 8: Simple example for inner/outer.

```
class A⟨indirect Real x⟩                  1
  ... use x ...                           2
end A;                                    3
class B                                   4
  indirect Real x;                        5
  direct A a1⟨x⟩, a2⟨x⟩;                  6
  ... use a1.x ... a2.x ...               7
end B;                                    8
```

Listing 9: inner/outer removed.

This kind of binding is well known from a number of functional languages, from object-oriented languages such as Java (in the form of method binding with superclasses), but also from typesetting languages such as TEX.

In Modelica it takes the syntactic form of **inner**/**outer** pairs. The simplest instance is illustrated by the example given in Listing 8 (adapted from Section 5.4 of the Modelica 3.1 reference).

Here `B.x` ≡ `B.a.x` ≡ `A.x` holds, that is, all three names are the same. Except for the possibly aberrant syntax, this looks fairly simple. However – as we will see in a moment – there are much more intricate scenarios, where the correct associations are not so easily seen.

In the class `I` in the complex example from Listing 7 (section 5.4, due to space concerns only parts of the example are shown) we have, among others the following equivalences: `e.f.x` ≡ `e.f.g.h.a.x` (lines 22+8 and lines 22+8+12) or `a.x` ≡ `x` (lines 23+1 and line 21). By contrast, other applications are different, for example `e.x` ≢ `e.f.x` (lines 22+6 and 22+8) or `e.f.g.x` ≢ `e.f.g.h.x` (lines 9+10+14 and line 11).

The root of the problem is that we encounter an isolated occurrence of dynamic binding in an otherwise statically binding language. It is this clash of paradigms that makes things both hard to digest and hard to implement. Therefore the obvious solution is to transform the dynamic bindings into static bindings. This will be sketched in the following.

We will use the code from Listing 9 to illustrate the augmented concepts. As is illustrated by this example, our transformation consists of four parts:

1. Both **outer** and **inner** declarations are converted to indirect types.
2. An outer declaration is converted into a parameter of the corresponding class. This new parameter has (in contrast to replaceable fields) no default value. Therefore this class becomes what in other languages is called abstract, since it cannot be instantiated without that parameter.
3. Finally we add to each instantiation of the abstract class a corresponding argument. If the current class has no indirect field, that class is made abstract too and the parameter is pulled up.

```
model CI                                          1
  outer Boolean b;                                2
  Real x(start=1);                                3
equation                                          4
  der(x) = if b then ?x else 0;                   5
end CI;                                           6
model Sub                                         7
  Boolean c = time<=1;                            8
  inner outer Boolean b = b and c;                9
  CI i1;                                         10
  CI i2;                                         11
end SubSystem;                                   12
model System                                     13
  Sub s;                                         14
  inner Boolean b = time>=0.5;                   15
  // s.i1.b will be b and s.c                    16
end System;                                      17
```

```
model CI⟨indirect Boolean b⟩                      1
                                                  2
  Real x(start=1);                                3
equation                                          4
  der(x) = if b then x else 0;                    5
end CI;                                           6
model Sub⟨indirect Boolean db⟩                    7
  Boolean c = time =< 1;                          8
  indirect Boolean b = db and c;                  9
  CI i1⟨b⟩;                                      10
  CI i2⟨b⟩;                                      11
end SubSystem;                                   12
model System                                     13
  Sub s⟨b⟩;                                      14
  indirect Boolean b = time>=0.5;                15
  // s.i1.b will be b and s.c                    16
end System;                                      17
```

Listing 10: Simultaneous inner/outer.

4. If necessary (e.g. because an element with the same name already exists) the newly introduced parameter is renamed.

Let us briefly look at the typing issue. We again take an example from Section 5.4 of the Modelica 3.1 specification (Listing 11). By the standard rules of typing under static binding it is clear that the argument `x` that would be given to the instantiation in line 7 has type **indirect** `Real`, whereas the requested type in line 3 is **indirect** `Integer`. This shows that the transformation naturally preserves the requirement that a inner declaration shall be a subtype of all corresponding outer declarations.

```
class A                                           1
 inner Real x; //error                            2
 class B                                          3
  outer Integer x;                                4
  ...                                             5
 end B;                                           6
 B b;                                             7
end A;                                            8
```

Listing 11: Type error in inner/outer usage.

```
partial function A                                1
  input Real u;                                   2
  output Real y;                                  3
end A;                                            4
function B                                        5
  extends A;                                      6
algorithm                                         7
  ...                                             8
end B;                                            9
class D                                          10
  outer function fc = A;                         11
  ...                                            12
equation                                         13
  y = fc(u);                                     14
end D;                                           15
class C                                          16
  inner function fc = B;                         17
  D d;                    // equation now y = B(u) 18
end C;                                           19
```

Listing 12: inner/outer with functions.

Finally, there is the odd case of allowing both inner and outer modifiers simultaneously. According to the Modelica 3.1 specification this "conceptually introduces two declarations with the same name: one that follow the above rules for inner and another that follow the rules for outer." With this advise our transformation works straightforward for that case too: When a direct field of the same name already exists, the indirect parameter is renamed. The example from Listing 10 shows how the informal interpretation of simultaneous inner/outer declarations fits naturally in our transformation.

Also functions are an interesting issue (taking the example from the specification shown in Listing 12). Listing 13 shows that this case can be handled easily too, if we also allow functions as indirect parameters. Since (direct) function parameters are allowed in the latest Modelica specification and there is no conceptual difference between direct and indirect function parameters, doing so does not introduce any complexity. The only inconvenience results from the fact that functions are not first class citizens of Modelica. Therefore a (useless) instantiation has to be made. But since our target language is object oriented, this would probably have to happen anyway (in a functional language the instantiation would just be a renaming).

What does this achieve? The answer is simple: We eliminated dynamic binding! In other words, we now have a uniform system of static bindings (and thus a uniform compiler design without complex exceptions) but still retain all effects of Modelica's dynamic inner/outer binding in a semantically correct fashion.

```
partial function A                                          1
  input Real u;                                             2
  output Real y;                                            3
end A;                                                      4
function B                                                  5
  extends A;                                                6
algorithm                                                   7
  ...                                                       8
end B;                                                      9
class D⟨indirect function A fc⟩                             10
                                                            11
  ...                                                       12
equation                                                    13
  y = fc(u);                                                14
end D;                                                      15
class C                                                     16
  indirect function B fc;                                   17
  D d⟨fc⟩;                    // equation now y = B(u)      18
end C;                                                      19
```

Listing 13: Listing 12 translated to xModelica.

This also applies to the important issue of *separate compilation*. We now only implement the classical static-binding variant of separate compilation and obtain a correct treatment of Modelica's inner/outer binding for free.

5. Drawbacks and Costs of Separate Compilation

Although separate compilation is a worthy goal, there are of course some drawbacks that occur from our design:

1. By creating code that does not contain a DAE but only tries to create one, we lose the ability to check for certain model properties. The most important of those is probably the requirement of the model being balanced. Since we do not know how compiled models are actually used, it is at least hard, if not impossible, to know if a model is balanced (if, e. g. only an interface is shipped with pre-compiled code). This is a general problem with separate compilation and usually solved by having the linker checking the global assertions. Since there is no Modelica linker, currently the only solution left is to move these checks into the runtime as well.
2. A more subtle effect of separate compilation is the compatibility of generated code. If module `A` is compiled with a more recent compiler than module `B`, there is no guaranty that both work together as expected, since small changes in the compiler may render both modules incompatible. The only way to workaround this issue is either to not change the output format at all (rather unrealistic) or to have some kind of versioning that hinders the user from plugging together incompatible compilation units.
3. The probably most far-reaching restriction that results from our approach is the impossibility to run optimizations at compile-time. For Modelica this means that there can be no causalization or index reduction done by the compiler. Again this problem depends on the absence of a dedicated Modelica linker. Thus both these operations have to be run before the actual simulation starts. Luckily this is generally possible and even necessary if one adds model structural dynamics to Modelica [13] (which is e. g. already present in the Mosilab compiler [10]).

6. Conclusion

With the usage of separate compilation a Modelica compiler comes closer to what state-of-the art tools can offer a software engineer. This applies to the overall compilation effort as well as to the opportunity to ship pre-compiled libraries with a clean interface.

We have shown that the most complex features of the Modelica language can in fact be transformed into an object oriented language by only using features that are much more common. This enables the creation of Modelica compilers for many target languages. The used translation scheme gives a new way of handling constructs with special semantics like stream connectors by simply moving those semantics into the runtime system.

While the language is thereby ready for separate compilation, some problems still remain open for future work. The most important is the conciliation of separate compilation with causalization. The same applies for symbolic manipulations. We will investigate both topics in the future.

References

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

[2] David Broman. Flow lambda calculus for declarative physical connection semantics. Technical Report 1, Linköping University, PELAB - Programming Environment Laboratory, The Institute of Technology, 2007.

[3] David Broman, Peter Fritzson, and Sébastien Furic. Types in the modelica language. In *Proceedings of the Fifth International Modelica Conference*, 2006.

[4] Luca Cardelli. A semantics of multiple inheritance. In *Proc. of the international symposium on Semantics of data types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[5] Gilles Dubochet and Martin Odersky. Compiling structural types on the jvm: a comparison of reflective and generative techniques from scala's perspective. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 34–41, New York, NY, USA, 2009. ACM.

[6] Christopher League, Zhong Shao, and Valery Trifonov. Representing java classes in a typed intermediate language. *SIGPLAN Not.*, 34(9):183–196, 1999.

[7] Christopher League, Zhong Shao, and Valery Trifonov. Type-preserving compilation of featherweight java. *ACM Trans. Program. Lang. Syst.*, 24(2):112–152, 2002.

[8] Ramine Nikoukhah. Extensions to modelica for efficient code generation and separate compilation. In *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools*, Linköping Electronic Conference Proceedings, pages 49–59. Linköping University Electronic Press, Linköpings universitet, 2007.

[9] Ramine Nikoukhah and Sébastien Furic. Towards a full integration of modelica models in the scicos environment. *Proceedings of the 7th International Modelica Conference*, 43(74):631–645, 2009.

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

[11] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

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

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