Modelica on the Java Virtual Machine

Christoph Höger

Technische Universität Berlin, Germany, christoph.hoeger@tu-berlin.de

Abstract

Modelica has seen a steady growth of adaption in industry and research. Yet, most of the currently available tools follow the same technological path: A Modelica model is usually interpreted into a system of equations which is then compiled into e.g. C.

In this work, we demonstrate how a compiler can translate Modelica models into Java classes. Those Java classes can be evaluated into a system of equations which can be solved directly on the JVM.

Implementing this tool yields some interesting problems. Among these are the representation of polymorphic data, runtime-causalisation and equation optimization and Modelica's modification system. All those problems can be solved efficiently on the JVM.

Keywords Modelica, separate compilation, Java

1. Introduction

Modelica [1] is an open, standardized language for the description of hybrid systems of differential and algebraic equations. It brings well-known features from object-oriented languages (like hierarchic composition, inheritance and encapsulation) into the domain of multi-physics modeling. Our focus is the *implementation* of Modelica. In this work, we present a prototypical extension of modim [10] that allows us to compile Modelica to the Java Virtual Machine.

The rest of the paper is organized as follows: First, we will motivate the need for a Modelica Compiler and the JVM as its target platform. Afterwards, we show some details of the translation and the runtime system. Finally we will demonstrate how our prototype performs in model instantiation and simulation.

2. Compiling Modelica

As Modelica's adoption in Industry and Science grows, three aspects of the language gain more and more attention:

First, Modelica allows for the easy exchange of models in forms of libraries. Those libraries are not developed by a tool vendor and tested and shipped with a specific implementation of the language. They rather depend (ideally) only on the language's specification and are platform-independent. When a Modelica model shall be simulated, it is first *instantiated* (or elaborated). During this process, all the equations that make up the final mathematical model are generated.

A Modelica library is thus basically a collection of more or less sophisticated methods to create a mathematical model. In that sense it is comparable to any other library of software. And as for any other computer-language it becomes important for Modelica's library-developers that their models behave *correctly* when being instantiated by a client (i.e. that they do not cause the model instantiation to fail in some situations).

Since Modelica is a statically-typed language, this question can, in theory, be answered by checking that any model conforms to its interface. Unfortunately, there is no formal algorithmic specification of the model instantiation process for Modelica. Instead, every tool provides its own, slightly different interpretation of the specification. Therefore a actual useful typecheck would have to assume a certain execution model. Hence, any safety assumption is only valid for a given interpretation of the Modelica specification.

The second problem follows directly from the usage of libraries. Naturally, libraries tend to offer more features than actually required for a single application. From the modelers perspective this may lead to unexpected large systems of equations. A simple pendulum, for example might be modeled by using the Modelica Multibody library as well as by a few equations from the textbook. Naturally, the Multibody approach is much more generic, since it might easily be adapted to (and used in) more complex mechanical systems. Such systems require the a fast model instantiation since otherwise instantiation time might exceed the actual simulation. In fact it may happen that model engineers are slowed down in their development process just because of the model instantiation overhead.

Finally, Modelica offers a lot of means to actually *compute* models. Currently, those computations range from arrays and loops to redeclarations. Yet it is not hard to predict, that more higher-order modeling features (recursive models, models as parameters etc.) will find their way into the language. All those features share one common pattern: The relation between models and their instances becomes

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. 19 April, 2013, University of Nottingham, UK.

 Proceedings available at: http://www.ep.liu.se/ecp/TBD/

EOOLT 2013 website: http://www.eoolt.org/2013/

$1 : n$ with very large n. In such a case it may become impossible to actually generate code for each instance due to practical reasons: As we have shown in [9], such an attempt yields C-code with a size proportional to the number of model-instances. Therefore for very large n, at least the C-Compiler will usually be unable to generate an executable model.

2.1 Interpreting vs. Compiling Modelica

To all of those problems exists a common solution: Modelica models should be *compiled separately*. When we think of a Modelica model as a method to describe a system of equations, it becomes quite obvious that most current tools (and even the specification) insist on *interpreting* Modelica. There are a few indicators for this:

- Most current tools can be steered into an endless loop during instantiation. A compiler should guarantee termination (for finite input).
- Some implementations will even try to call external functions during instantiation. This may lead to severe problems e.g. in case of cross-compilation.
- The language specification defines array-types to include the actual size. This kind of dependent typing naturally requires an interpreter for a type-check. Since Modelica does not place any syntactical bounds on the expressions used for array-access, this means that a fully compliant type-checker must contain a full Modelica interpreter.

In contrast, a compiler does not need to evaluate any Modelica expressions at all [1]. Instead, a model or a group of models (called the *compilation unit*) is translated into a target language. The *effect* of the model instantiation, i.e. the creation of a system of equations is done by evaluating the compilation result according to the semantics of the target language.

This way, we gain multiple benefits:

- A compiler can be implemented in a way that guarantees termination.
- The compiled fragments can be reused in different contexts.
- The instantiated model is memory-efficient: Only the parts that actually differ between different instances need to be allocated dynamically.

It is important to note that to achieve all those benefits, it is not necessary that the compilation targets are portable (i.e. compatible between different tools). It suffices that there is a known way to compile models for a known tool.

2.2 Separate Compilation & Variable Structure

Besides better language scalability and safety an additional point motivates the compilation of models: In case of a variable-structure system, every mode of the system needs to be processed into a computable form (causalisation, index reduction etc.). A way to achieve this with current implementations is to generate a model for each mode and process it *before* simulation [15].

This method has two drawbacks:

- Every possible mode needs to be known before the simulation starts. This excludes sophisticated models, where some mode depends on the simulation results of another as well as models that contain a large number of possible modes.
- It is impossible to decide which mode will become active during a simulation. Therefore it is quite likely that computational effort is wasted in the translation of unneeded modes.

On the other hand it is quite obvious that a compiler has to postpone all this processing until the instantiation was successful. Naturally, this means that variable-structure processing can be solved by the same machinery as a compiled model: Once a mode-change occurs, instantiate the new model the same way as the old one and continue simulation. Of course, the efficiency of this transition is quite important, but technically, compilation and variable-structure systems share a lot of problems.

2.3 JVM

Our compiler requires a much more sophisticated runtime system than an interpreter: We need to be able to describe and instantiate data-structures and higher-order functions (part of Modelica since version 3.2). Additionally, our runtime needs to support sophisticated graph-algorithms and some numerical methods. For reasons that are explained later, we need an accessible intermediate representation of code. Finally our runtime system needs to be fast enough for real-world applications.

At least two platforms fulfill those requirements: The Java Virtual Machine (JVM, [12]) and the Low Level Virtual Machine (LLVM, [11]). Although both are available for practically every platform we preferred the JVM for the following reasons:

- The JVM is tightly integrated with (but not bound to) Java, which is far more productive than C. This makes it easier to implement the necessary runtime system.
- The JVM is the foundation of a large, active ecosystem. Currently there exist hundreds of thousands of free Java libraries[2]. Currently this is mostly beneficial for the implementation of the runtime system. But integrating Modelica into this ecosystem might yield additional benefits in the future.
- The mainstream JVM implementation, Hotspot is fast. In fact, a modern Java program may beat an equivalent C program in certain areas [3].
- In contrast to LLVM, the JVM already contains a notion of classes and objects which makes it a natural goal for the compilation of Modelica.

[1] It might choose to do so for certain optimizations

[2] available e.g. on http://search.maven.org/

3. Modelica on the JVM

In this section we will explain the principles of the compilation from Modelica to Java. To do so, we will answer a few questions:

- What kind of Java code shall be generated? This is, more or less, a pragmatic design decision. Yet the general layout of the target code influences the runtime system and vice-versa, thus it shall be explained.
- How do we implement Modelica's data structures and the operations defined on them? Especially the translation of Modelica-models and the "`.`"-operator for projection are of interest here.
- How do we translate multiple inheritance?
- How can equations be structured as first class citizens? This is important for efficient simulation: Since we do not want to interpret equations during simulation (rather evaluate their compiled bytecode), it is an interesting question how we can package them into Java-classes.
- How can relations be structured as first class citizens? The availability of relations as first class citizens would enable to define the relation resolution outside of the runtime system and possibly by the user.

3.1 Generated Java Code

As already mentioned, separate compilation means to transform *compilation units* from one language into another. What exactly makes up a compilation unit remains a design decision. The default Java compilation unit is a class. Java-classes are not one-to-one mapped to java-source files. Instead, a `.java` file might contain several nested classes.

Since Java and Modelica share a notion of classes, it seems natural to establish a one-to-one relationship between them. Thus, our compiler will not translate Modelica source files into Java source files[3] but classes into classes.

This decision determines two other aspects of the compilation: Since Java allows aggregation, we can directly translate Modelica's aggregation into it. The same holds for model instantiation. While Modelica has no *explicit* constructors except for records, we are still forced to use them on the Java side.

Other important requirements for our target language are:

- Meaningful names. We will try to keep the names of compiled classes as close as possible to the ones used in the Modelica source. This should make the generated code more readable.
- No compilation into Generics. Java's version of polymorphic types is completely erased at object level. Therefore, there would be no gain in compiling Modelica's polymorphism into Java's.
- Creation of immutable objects. As explained e.g. in [2], immutable objects provide greater safety as well as better performance in many cases. Since Modelica is a declarative language, mutation of objects should be considered an error. Thus every class and all of its fields are declared **`final`**.

3.2 Modification by Reference

Choosing the JVM as our compilation target has one important consequence for model modification: We can directly reuse the JVM's object reference passing for modifications as proposed by Zimmer in [21]. Consider the following Modelica snippet:

```
model A
  model B
    Real x;
  end B;
  replaceable B b;
end A;

A a(b = c);
```

Traditional tools usually would interpret the modification on `a` as an additional set of equality constraints ranging over the fields that `b` and `c` have in common. Notably, most of these tools would also remove equalities in a later optimization step by storing them outside of the system of equations.

On the JVM we can instead directly pass `c` into the instance of `a`. This way, we do not need to create any additional equations or even instantiate `a.b`.

The downside of this approach is, that our runtime system needs to deal with the fact that `c` might have a different data layout than `b`. Even worse, this data layout is in general unknown at the compile time of `A`. We will deal with this issue in section 3.4.

3.3 Inheritance

Modelica allows a class to inherit from multiple distinct classes [4]. Such a class may look like this:

```
model M
  model C
    Real z;
  end C;
  extends A.B;
  extends C;
  A a;
end M;
```

The result of flattening M according to the Modelica specification would yield a representation like this:

```
model M_Flat
  Real a•b•x;
  Real x;
  Real z;
end M_Flat;
```

[3] As said, this is just a design decision. But since Modelica files tend to be rather large in practice, we think it is a well-founded one.

[4] See section 6.2 for a discussion of the "same" source rule in Modelica

Where • is filled in to make a name unique (Some tools use the same character as for projection, while others might prefer underscores etc.). While the generation of composite names is a technical detail, the semantic requirement is not: `M_Flat` contains three real-valued unknowns. This is exactly what any Modelica tool (be it an interpreter or a compiler) has to deliver.

To fulfill this requirement, we introduce special fields into the compiled classes, to represent super-objects (instances of super-classes). That way we can compile the inheritance of arbitrary many classes into a simple form of aggregation:

```
class M {
  public final MBase a;
  public final B superclass1;
  public final C superclass2;
  ...
}
```

Note that the superclasses are declared with the same (compiled) type as in the original model, while the child-object is of type `MBase`. We will motivate this in the next section.

3.4 Uniform Data Representation

Modelica has a structural type system. This means that for any compiled class the compiler cannot know all classes that are compatible (as there infinitely many of such compatible classes). When instantiating the model `A` from our example above, we could legally choose the type of `b` arbitrarily, as long as it contains a real-valued unknown named `x`.

For our implementation this means that there is no way to know where to actually look for e.g. the field `x` in `b`. It may be the first, second or 42nd field in whatever object gets passed in from the outside. Java, as our target language does not allow structural compatible expressions, but requires *nominal* compatibility. That is, every class has to denote the set of its subtypes at its definition site.

Note, that both kinds of systems are equivalent in case all types are known (by simply enumerating all existing types). Thus, the problem is tightly coupled to the proposed scheme of separate compilation.

In addition to structural subtyping, Modelica is a polymorphic language, capable of a restricted form of type-abstraction and type-application. This means we have to deal with the well known problem of compiling universally quantified types: A model may contain a so called `replaceable` type which may be substituted by any (yet unknown) concrete type at the instantiation site.

To both problems, there exists a common solution: Uniform data representation. This kind of compilation strategy aims at compiling every object-level data into the same form. This form then needs to support[5] any operation that is mapped from the source into the target language.

[5] Since Java is statically typed, this means we have to carefully design the runtime system to not cause any Java compile-time-errors.

For Modelica, the most important of these operations is the projection ("`.`"). This operation cannot be directly encoded into the Java-projection (using the same character), because of the difficulties noted above. Instead, every object in our target language has to support this operation via a Java-method `get`.

This method implements the dispatch of a projection among the local fields of a Java class (as already mentioned, aggregation is translated directly into Java). As every object in our runtime system implements this method, a Modelica projection `a.b` can be directly compiled into a Java expression `a.get("b")`

Yet, the important question remains how the compiled object itself can implement `get`. In the presence of polymorphic classes, does a compiled class know its own data layout?

Fortunately, Modelica contains two important restriction to type abstraction:

- It is only applicable in *aggregation*, not inheritance: Thus any class always knows all of it's concrete super-classes (not to confuse with it's super-types as every super-class is also a super-type but not vice-versa) and by conclusion the name of all of its fields. This rule is called the "transitively non replaceable rule" in the Modelica specification.
- Type application is *bounded*. That is, every applicable type has to be compatible with the type, it replaces. Thus, a compiled class might not know the concrete type of its fields, but it does know that they *exist*.

These two restrictions allow a compiler to compute the set of legal right hand sides for a projection: Either it is a local field, or a field inherited by some super-class. In case of a local field, the `get`-method simply returns the child-object. In case of an inherited field, the projection can be forwarded to Java's built-in projection operator, since the concrete type of all super-objects is known at compile-time.

Putting it all together our `get`-method looks like this:

```
public MObj get(String name) {
  switch(name) {
    case "a" : return a;
    case "y" : return superclass1.y;
    case "z" : return superclass1.z;
    case default: throw new
      RuntimeException("");
  }
}
```

The possibility to use a **`switch`**-statement on a `String` is a relatively new feature to Java. As it has to handle only constant input in our case (the field names are always translated into String literals), we can expect a significant performance improvement over techniques like reflection or dynamic method invocation.

3.5 Modification

One important feature of Modelica is the ability to modify certain values of an instantiated object. Although some

fields may be unmodifiable by the keyword `final`, we consider *all* fields modifiable for simplicity.

Such a modification works like a selective record update:

```
model X
  record N
    parameter Real x = 4;
    parameter Real y = 2;
  end N;
  N n(y = 1);
end X;
```

It is important to distinguish this kind of value modification from the type application mentioned above: The type application always implies a value modification, but not vice-versa.

Since we insist on creating immutable objects, value-modification cannot be implemented by assignments. Instead, we have to provide some kind of factory to the instantiated object which takes care of creating the necessary child-objects. The instantiated object is responsible to invoke the factory methods in the correct order.

Additionally the provider of the factory object does not know about the default modifications defined in the instantiated class (due to separate compilation). Therefore, every class provides a default factory which can be extended by any modifying factory at the instantiation site.

```
final class N {
  final MBase x;
  final MBase y;

  public N(NModification factory) {
    x = factory.newX();
    y = factory.newY();
  }

  public class NModification {
    public final MBase newX() {
     return new MParameter(4.0);
    }
    public final MBase newY() {
      return new MParameter(2.0);
    }
  }
}
```

3.6 Equations

The most important parts of a Modelica model are certainly its equations. In this section we will describe the compilation of such an equation on the basis of the Modelica equation below:

```
der(x) = 2*y;
```

As we have already proposed in [9], equations can be compiled into *solution functions* that can be used to compute a root for a given variable. Compiling these functions into Java-classes is then straightforward. In best object-oriented manner, an equation might access the surrounding object via a `self`-reference (comparable to Java's **`this`**). Additionally, an equation must be able to report the set of unknowns it depends on. Now the runtime system can ask an equation what value a certain unknown has as at a certain time (if the runtime system provides this equation with the values of all other unknowns).

```
final MObj self;

...

public final double solveFor(
  final MVar v, final SolvableDAE sys) {

  final MVar v1 = system.der.apply(
    self.get("x"));
  final MVar v2 = (MVar)self.get("y");

  if (v == v1) {
    final Double tmp1 = 2.0;
    final Double tmp2 = sys.valueOf(v2);
    final Double tmp3 = tmp1 * tmp2;
    return tmp3;
  } else if (v == v2) {
    final Double tmp4 = 0.5;
    final Double tmp5 = sys.valueOf(v1);
    final Double tmp6 = tmp4 * tmp5;
    return tmp6;
  }
  throw new RuntimeException();
}
```

This general scheme can of course be easily specialized: Our runtime system contains special classes for linear equations and direct equalities as these can be solved more directly (or even be removed before simulation).

Higher-order equations (e.g. `if`- or `for`- equations) can now be translated into corresponding Java code that instantiates the appropriate equation-objects. It is the task of the runtime system to collect all those objects, sort them and invoke their solution functions in a correct order to compile the global system result. See section 4.2 for a discussion on how this can be achieved.

3.7 Relation Semantics

Modelica does not only allow equations to be generated by model instantiation, but also by *relations*. Relations can take different forms, but the most widely known one is probably the `connect`-statement:

```
model C
  ...
  connect(x, y);
end C;
```

Modelica contains several different kinds of those relations that are all currently specified in the language standard. This increases the size and complexity of the specification significantly and makes it hard to implement Modelica from scratch. Although our favored solution to this

issue (userdefined connection semantics) is not the topic of this paper, our runtime system and compiler are naturally prepared for it. So we give a short overview:

Unlike equations, relations only have a meaning in a global context. Only the complete set of all relation instances (usually called the set of connections, but not limited to the `connect`-relation) can be used to generate a set of equations.

Fortunately, these generated equations always have a certain form. Only some parts of the equations change (e.g. flow sets always generate sum-to-zero equations etc.). Thus, those equations can be seen as some kind of higher order model (parameterized over the relation sets).

To implement this, our runtime contains relations as first-class citizens. As well as equations, they are computed by the model-instances (again allowing loops, conditionals etc.). For this purpose, every compiled model-instance contains a method `relations()` to compute all locally defined relation-objects.

Since relations are often defined on *ports*, which in turn often distribute a relation over their fields, the relation creation is invoked on the first argument of the relation definition. In case of our example model `C` above, the relation method would look like this:

```
public final Iterable<MRelation>
  relations() {

  final MObj tmp1 = self.get("x");
  final MFunction tmp2 =
    tmp1.get("connect");
  final MObj tmp3 = self.get("y");
  return tmp2.apply(tmp1, tmp3);
}
```

The runtime system collects all relations and hands them over to specialized relation-handlers, which are basically models[6] with a certain signature. In a future version, they should be implementable in pure Modelica.

4. The Runtime System

As already mentioned, the runtime system's responsibilities contain the collection of relations and equations, the symbolic manipulation of systems of equations and the control of the simulation.

4.1 Classes

To achieve this, it comes with a set of classes that are to be implemented or generated by the compiled models.

MBase Since we need to distinguish our object system from Java's, `MBase` was introduced. It is the base class of all other classes and serves as the same purpose in our system as `java.lang.Object` on the JVM.

MObj Every model instance may contain equations and relations. This is what distinguishes an instance of this class from the other runtime objects.

[6] This means, it is quite natural, that they create equations.

MEquation As already mentioned, equations have a special interface to enable causalisation.

MFunction As in any functional languages, functions are first-class citizens in our runtime environment. Once Modelica gets (full) support of closures, we will implement this here.

MRelation This class is basically a tag to distinguish a relation from the other runtime objects.

MArray Objects of type `MArray` implement Java's `List` interface. Additionally they provide a specialized implementation of the `get`-method, which is necessary for Modelica's vectorized field access.

MVar Variables basically serve two purposes: They decide what kind of information need to be written into a result file and they work as a placeholder for actual double-values in equations. In a future version, they will probably hold all the additional attributes (e.g. start, min, max) that Modelica supports.

MParameter Objects of type `MParameter` need to be instantiated from user provided values.

MPotentialVar Since Modelica contains a builtin definition for potential variables, our runtime contains a builtin class for them.

MFlowVar As for potentials, flows are builtin to the language and thus mirrored in the runtime system.

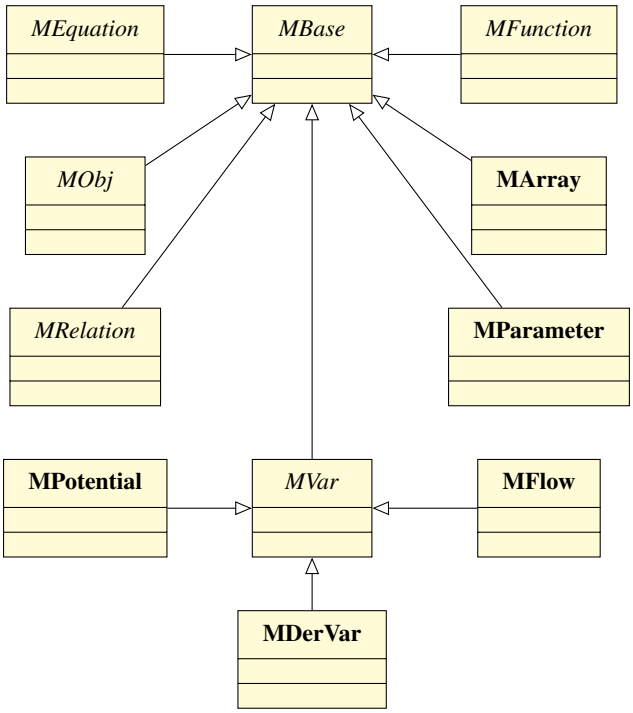

Figure 1. Modelica Runtime Classes

4.2 Causalisation & Index Reduction

Bringing Modelica models into an efficiently computable form usually requires two process-steps: Causalisation and Index-reduction. Informally, causalisation tries to use

Edges	Vertices	Hopcroft-Karp
80	60	5ms
800	600	28ms
8000	6000	135ms
80000	60000	278ms
800000	600000	1.131ms

Table 1. Causalisation runtime on the Automaton model

as much forward-substitution as possible while index-reduction aims at transforming a DAE into an ODE, which can be handled by numerical integration methods.

As we have shown in [9], runtime causalisation can be implemented as a graph algorithm without any symbolic processing if the solution functions are precompiled. We implemented this technique in modim for our compiler. Yet it remained an open question, whether the causalisation would be fast enough to be invoked on every startup. To answer this question we measured the the performance of the Hopcroft-Karp algorithm implemented in the JGraphT library [17] for different sizes of our Automaton model (see Appendix).

The results can be seen in table 4.2. In the given case the algorithm behaved much better than the theoretical limit of $O(|E|\sqrt{|V|})$. But more important is the fact that even for large models (400000 non-equality equations) the causalisation overhead would still be bearable (as $\approx 1s$ of additional instantiation time would be quite small compared to the expected simulation time).

With causalisation settled, index-reduction remains an open problem for our runtime system. Although the actual process is a solved problem with the Dummy Derivatives Method [14] or even more advanced methods ([13]), from our perspective, a more fundamental problem has to be solved first:

The purpose of an index-reduction algorithm is to identify equations that need to be differentiated at most n times to reduce a system of index n to an ODE. The actual value of n can only be computed, once the whole system of equations is known. And as we have stated in the beginning, our system will not do any symbolic processing at this point in time. So the question is not, which algorithm to choose for index-reduction, but how to resolve this conflict.

This is precisely the reason why we demanded an accessible just-in-time compilation format for our runtime: Automatic Differentiation [16] allows for precise computation of derivatives. It has been used with some success for Modelica already [4]. Yet, our approach goes further: With AD, we can compute the derivative of *virtually any* function that can be compiled to the JVM. Since we have access to the Bytecode, we do not need any symbolic information about the equations to do so.

Currently, work is ongoing to implement automatic differentiation for modim. This work aims to leverage the existing AD library nabla [7]. Once the implementation is mature, we can investigate, which index-reduction algorithm is best suited for our approach.

4.3 Numerical Methods

Naturally, numerical simulation requires a set of numerical tools. The numerical algorithms required for a Modelica simulation can roughly be divided in three categories:

- DAE solvers
- ODE solvers
- Algebraic solvers

Unfortunately, there is no implementation of a full-featured DAE solver like DASSL for Java. In theory, we could interface with an existing C/C++ implementation but the data transport overhead would probably cause a too severe performance penalty. In fact there seems to be a general lack of implicit integration methods for the JVM. Therefore future versions of our runtime will probably need to carry their own implementation.

The situation for ODE integrators is slightly better: The apache commons math library [8] contains a comprehensive collection of explicit ODE solvers. There we also find a comprehensive list of root-finding algorithms and some basic linear algebra support. For now modim's numeric runtime depends on those algorithms for simulation.

5. Performance

Benchmarking is a delicate business at best. This is especially true on the JVM: The Java Virtual Machine is a *garbage collecting* environment and issues a just-in-time compiler. For that reasons, the runtime behavior of a program might vary drastically depending on the state of the virtual machine. This is the reason why our performance measurements sometimes seem to run *faster* for larger inputs. It is thus not our goal to give precise information about runtime but to show the general time-frame of a certain problem size.

5.1 Compilation

Compilation performance is negligible in our approach, as long as it does not exceed the instantiation time for even large models. In fact, our current implementation is quite fast: Compiling the Automaton and Cell model below took less than 100ms on a warmed-up JVM.

5.2 Instantiation

There is no standardized or widely adopted Modelica-benchmark (as it exists for other multi-implementation languages like JavaScript). Even if there was one, our compiler would not be able to execute it, due to its prototypical state.

So instead of presenting some semi-accurate numbers comparing our compiled models to another implementation, we will present synthetic benchmarks for instantiation without comparing the results to another tool. The focus here is more on the general *ability* to instantiate such models at all in a timely manner.

[7] http://http://commons.apache.org/sandbox/nabla/

[8] http://http://commons.apache.org/math/

n	# variables	instances	time
1	1	1	15ms
2	2	3	20ms
3	6	10	1ms
4	24	41	2ms
5	120	206	10ms
6	720	1237	60ms
7	5.040	8660	308ms
8	40.320	69281	476ms
9	362.880	623530	873ms

Table 2. Instantiation time of `Faculty`

```
model Faculty "recurses n! times"
  constant Integer n;
  Faculty[n] faculties(each n = n - 1)
    if (n > 1);
  Real root if n == 1;
end Faculty;
```

The model `Faculty` above is instantiated recursively. As one might see at a glance it will create $n!$ unknowns for any parameter $n > 0$. But to do so, it also needs to instantiate a lot of models (actually $n! \sum_{k=1}^{n} \frac{1}{k!}$). Therefore although this model does not have any usage for simulation, we consider it a valuable stress-test for our runtime system. Table 2 shows the instantiation time results for up until $n = 9$.

width	# equations	instance	equations
10	60	28ms	145ms
100	600	3ms	18ms
1000	6000	41ms	202ms
10000	60000	237ms	682ms
100000	600000	41ms	1.175ms

Table 3. Instantiation time for `Automaton`

As a second, more practical test for instantiation performance, we created a small model of a cellular automaton. This kind of models is found in practice in e.g. in thermodynamic models, where space-discretization leads to higher accuracy. Thus we consider the instantiation performance of this model of practical relevance.

```
model Cell
  Real center;
  Real target;
  Real n,e,s,w;

  equation
  target = (n + e + s + w) / 4;
  der(center) = target - center;
end Cell;
```

To improve readability, the whole automaton model can be found at the end of this document. We instantiated the automaton model for different values of `width`. The `height` parameter was set to 1. In contrast to the `Faculty` model above, the Automaton actually creates equations. Thus, the time it takes to calculate them is also of interest in this benchmark. The results are shown in table 3.

5.3 Simulation

When it comes to simulation, a high performance is crucial. While it might seem acceptable to loose some percent against another tool for the benefit of faster compilation and instantiation, already a doubled simulation time would probably rule out our implementation in any relevant applications.

Thus unlike for instantiation, we have to demonstrate that a Java-based solution can be as fast as a C-based one. To show this, we need a reference implementation and a simple model (simple enough for our prototype to compile it as well as simple enough to rule out any relevance of the choice of integration algorithms).

Integration Steps	Java Runtime	OMC Runtime
10000	993ms	1090ms
25000	2464ms	2277ms
50000	5103ms	5304ms
100000	10034ms	11393ms
150000	14810ms	15904ms

Table 4. Simulation times of JVM prototype compared to OMC

For the reference computation, we favored OpenModelica due to its openness and availability. For the model we choose the cellular automaton described above. This model instantiates into an ODE and we do not want to compare sophisticated numerical algorithms but sheer floating point computation. Therefore we choose a simple forward Euler for the integration method.

Table 5.3 contains the results of this comparison for different integration periods. As can be seen our compiled model can indeed be as fast as the omc output. Yet, this does not mean that modim is already faster. Since omc is a much more mature implementation it is highly likely that the small difference between the two is due to additional tasks computed by the omc simulation.

6. Conclusion

Our compiler is still rather prototypical and does not cover even a small part of Modelica. Yet, we can already make some conclusions about the general ability to compile Modelica into Java:

- As we have proposed in [8], Modelica *can* be compiled separately.
- The Java Virtual Machine is a viable platform not only to host an object-oriented modeling language like Modelica, but also to simulate its output.
- The high-level compilation-scheme moves several parts of current tool's "Frontend" into a runtime environment or even a core-library. This decouples language aspects from simulation aspects and hopefully simplifies both further development and maintenance.

6.1 Benefits

The proposed compilation shows several benefits over the classic interpretation:

- Compiled fragments can be *safe*. Once a Modelica model has been typechecked compiled, it is guaranteed[9] to instantiate safely in any other context.
- Instantiation is *fast*. Even without special optimization, structurally changing a model does only involve insignificant cost. This is especially useful during the development cycle of a model, when changes and tests are iterated fast.
- A compiled model is *accessible*. Since Java is practically ubiquitous, a compiled model can be used in many different contexts at ease.
- There seems to be no *general* performance penalty at simulation time.
- Since our compiler naturally provides a foreign function interface (ffi) to Java, the whole JVM-ecosystem can be used easily from within the model.

6.2 Drawbacks

Obviously, our approach also comes with some caveats, that should be mentioned:

- The compiled model always *instantiates* itself before it simulates. If there are no structural changes, this effort is wasted. Of course the model could provide some caching-mechanisms to circumvent this problem, but this would increase both the complexity of implementation and usage.
- The Java-ffi is unlikely to be standardized and adopted by other tools soon. While it is easy to provide from within Java, it is equally hard to do so from within a C-implementation.
- It may conflict with some parts of the language specification. While this case should be rare, we cannot always avoid it. Consider the "same-element" rule in the Modelica specification: It basically states that multiple inheritance of the same element name is fine as long as the definition of that element is the same on all definition sites. Even if we ignore the fact, that “same-ness” is at least quite hard if not impossible to decide in the general case, we could still not fulfill this rule. Simply put, it demands, that a tool *knows* the right-hand side of the definition. This is a violation of the separate compilation principle. So either we could be too negative and simply forbid double definitions completely or be too positive and allow them as long as every right-hand-side evaluates to the same value. Since the last option would raise other questions (what context to evaluate in etc.), we simply forbid double definitions, thus leaving the Modelica Specification.

Although these are valid arguments, we think that the benefits outweigh the drawbacks significantly.

[9] Of course only if the compiler and the type-checker are correct.

6.3 Related Work

Reusing multiple instances of a Modelica model has already been proposed by Zimmer in [19]. This method is applied after instantiation and can only detect some cases of shared instances.

Sol [20] is a language with unbound structural dynamism. It contains an incremental mechanism for index-reduction and causalisation. In contrast to our approach, Sol is completely interpreted and does full symbolic analysis of equations. Yet, the proposed incremental index reduction might be of great value for the further development of modim.

First-class models have been proposed by Broman for the MKL [5] and Giorgidze for Hydra [7]. Hydra contains an implementation of just-in-time compilation and supports structural variable models. Thus, it is closely related to our approach. Yet it does not deal with the Modelica specific aspects of compilation and embeds equations as data-structures into the host language.

Using the JVM as simulation backend has been proposed for JModelica in [18]. Here Modelica is still interpreted. Another JVM simulation backend is implemented in openmodelica [6].

6.4 Future work

As it was mentioned multiple times, our system is far from being complete. Thus the implementation of larger parts of the Modelica specification remains an important, but academically mostly uninteresting task for the future.

Yet, some more interesting tasks remain:

- While a formal description of Modelica as a whole seems unfeasible because of the complexity of the language, our runtime system could be small enough to formally defined. If this was the case, a correctness proof of modim’s typechecker would become feasible.
- Optimization. The JVM gives raise to a large set of interesting features. One could research runtime bytecode generation or solver parallelization. As already mentioned, current work is focusing on the implementation of automatic differentiation. Since this method involves bytecode manipulation, it seems logical to also investigate the application of *bytecode specialisation* on Modelica models.
- Variable structure systems. As mentioned in the beginning, this work now gives the framework for a true variable structure Modelica implementation. Yet it remains a research topic, how efficient e.g. incremental index reduction can be implemented.

Acknowledgments

The author wants to thank the anonymous reviewers for their in-depth reviews and valuable suggestions to improve the quality of this work.

References

[1] Modelica - a unified object-oriented language for physical systems modeling, 2012.

[2] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[3] R.F. Boisvert, J. Moreira, M. Philippsen, and R. Pozo. Java and numerical computing. *Computing in Science Engineering*, 3(2):18 –24, mar/apr 2001.

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

[5] David Broman. *Meta-Languages and Semantics for Equation- Based Modeling and Simulation*. PhD thesis, Department of Computer and Information Science, Linköping University, Sweden, 2010.

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

[7] G. Giorgidze. *First-class models: On a noncausal language for higher-order and structurally dynamic modelling and simulation*. PhD thesis, The University of Nottingham, 2012.

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

[9] Christoph Höger. Separate compilation of causalized equations -work in progress. In François E. Cellier, David Broman, Peter Fritzson, and Edward A. Lee, editors, *EOOLT*, volume 56 of *Linköping Electronic Conference Proceedings*, pages 113–120. Linköping University Electronic Press, 2011.

[10] Christoph Höger. Modim - a modelica frontend with static analysis. In *Vienna International Conference on Mathematical Modelling 2012*, Vienna, Austria, February 2012.

[11] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[12] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[13] Sven Erik Mattsson, Hans Olsson, and Hilding Elmqvist. Dynamic selection of states in dymola. In *Modelica Workshop 2000 Proceedings*, pages 61–67, 2000.

[14] Sven Erik Mattsson and Gustaf Söderlind. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM Journal on Scientific Computing*, 14(3):677–692, 1993.

[15] A. Mehlhase. A Python Package for Simulating Variable-Structure Models with Dymola. In Inge Troch, editor, *Proceedings of MATHMOD 2012*, Vienna, Austria, feb 2012. IFAC. submitted.

[16] Uwe Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. SIAM, 2012.

[17] Barak Naveh et al. Jgrapht. *Internet: http://jgrapht. sourceforge. net*, 2008.

[18] Franck Verdiere, Abir Rezgui, Sana Gaaloul, Benoit Delinchant, Laurent Gerbaud, Frédéric Wurtz, and Xavier Brunotte. Modelica models translation into java components for optimization and dae solving using automatic differentiation. In David Al-Dabass, Alessandra Orsoni, and Richard Cant, editors, *UKSim*, pages 340–344. IEEE, 2012.

[19] 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.

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

[21] Dirk Zimmer. A reference-based parameterization scheme for equation-based object-oriented modeling languages-modim - a modelica frontend with static analysis. In *Vienna International Conference on Mathematical Modelling 2012*, Vienna, Austria, February 2012.

Appendix

```
model Automaton
 parameter Integer width=20, height=20;
 Cell[width, height] cells;

 initial equation
 cells[1,1].center = 1.0;

 equation

 for i in 1:width loop
  for j in 1:height loop
   if i > 1 then
    cells[i,j].w = cells[i-1, j].center;
    cells[i-1, j].e = cells[i, j].center;
   else
    cells[i,j].w = cells[width, j].center;
    cells[width, j].e = cells[i,j].center;
   end if;

   if j > 1 then
    cells[i,j].n = cells[i, j-1].center;
    cells[i, j-1].s = cells[i,j].center;
   else
    cells[i,j].n = cells[i, height].center;
    cells[i, height].s = cells[i,j].center;
   end if;
  end for;
 end for;
end Automaton;
```