Bootstrapping a Modelica Compiler aiming at Modelica 4

Martin Sjölund, Peter Fritzson, Adrian Pop

PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, SE-581 83 Linköping, Sweden
{ martin.sjolund, peter.fritzson, adrian.pop}@liu.se

Abstract

What does it mean to bootstrap a compiler, and why do it? This paper reports on the first bootstrapping (i.e., a compiler can compile itself) of a full-scale EOO (Equation-based Object-Oriented) modeling language such as Modelica. The Modelica language has been modeled/implemented in the OpenModelica compiler (OMC) using an extended version of Modelica called MetaModelica. OMC models the MetaModelica language and is now compiling itself with good performance. Benefits include a more extensible maintainable compiler, also making it easier to add functionality such as debugging support.

This work is in line with the recently started Modelica 4 design effort which includes moving implementation of language features from the compiler to a Modelica Core library, allowing compilers to become smaller while increasing correctness and portability.

A number of language constructs discussed for Modelica 4 are already supported in some form by the bootstrapped compiler. Future work includes adapting language constructs according to the Modelica 4 design effort and extracting and restructuring parts of the Modelica implementation from the OMC compiler to instead reside in a Modelica Core library, making the compiler smaller and more extensible.

Keywords: Compilation, Modelica, MetaModelica, meta-programming, metamodeling, modeling, simulation

1 Introduction

Since user requirements on the usage of models grow over time, and the scope of modeling domains increase, the demands on the Modelica modeling language and corresponding tools increase. This has caused the Modelica language and model compilers to become increasingly large and complex.

One approach to manage this increasing complexity used by several functional languages (Section 6) is to define a number of language features in libraries rather than in the compiler itself. After several years of discussion and a number of language prototypes, e.g., [3][8][9][23][30][32][37], this approach finally became part of the new Modelica 4 effort started during the 67th Modelica design meeting [22].

1.1 Modeling Language Constructs

The Modelica specification and modeling language was originally developed as an object-oriented declarative equation-based specification formalism for mathematical modeling of complex systems, in particular physical systems.

However, it turns out that with some minor extensions, the Modelica language is well suited for another modeling task, namely modeling of the semantics, i.e., the meaning, of programming language constructs. Since modeling of programming languages is often known as meta-modeling, we use the name MetaModelica [8] [11][12] [30] for this slightly extended Modelica.

The semantics of a language construct can usually be modeled in terms of combinations of more primitive builtin constructs. One example of primitive builtin operations are the integer arithmetic operators. These primitives are combined using inference and pattern-matching mechanisms in the specification language.

Well-known language specification formalisms such as Structured Operational Semantics/Natural Semantics [28] are also declarative equation-based formalisms. These fit well into the style of the MetaModelica specification language, which explains why Modelica with some minor extensions is well-suited as a language specification formalism. However, only an extended subset of Modelica here called MetaModelica is needed for language specification since many parts of the language designed for physical system modeling are not used at all, or very little, for language specification.

Another great benefit of using and extending Modelica in this direction is that the language becomes suitable for meta-programming and meta-modeling. This means that Modelica can be used for specification and

transformation of models and programs, including transforming and combining Modelica models into other (lower-level) Modelica models, i.e., a kind of compilation.

Figure 1 shows typical translation stages in a Modelica compiler.

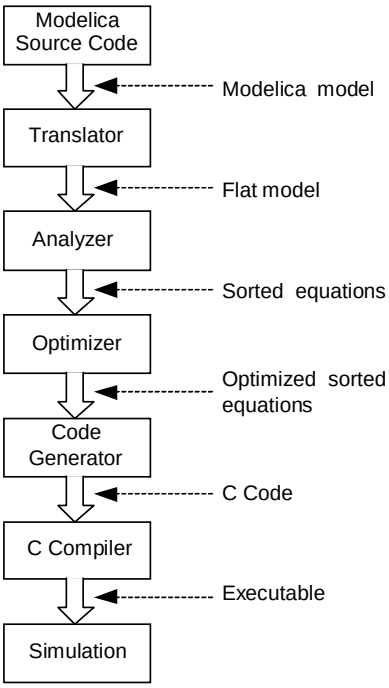

Figure 1. The typical stages of translating and executing a Modelica model.

2 Vision – Extensible Tools

Traditionally, a model compiler performs the task of translating a model into executable code, which then is executed during simulation of the model. Thus, the symbolic translation step is followed by an execution step, a simulation, which often involves large-scale numeric computations.

However, as requirements on the usage of models grow, and the scope of modeling domains increases, the demands on the modeling language and corresponding tools increase. This causes the model compiler to become large and complex.

Moreover, the modeling community needs not only tools for simulation but also languages and tools to create, query, manipulate, and compose equation-based models. Additional examples are optimization of models, parallelization of models, checking and configuration of models.

If all this functionality is added to the model compiler, it tends to become large and complex.

An alternative idea already mentioned in Section 1 is to add features to the modeling language defined in library packages that can contain model analysis and translation features that therefore are not required in the model compiler. An example is a PDE discretization scheme that could be expressed in the modeling language itself as part of a PDE package instead of being added internally to the model compiler.

2.1 Why Bootstrapping?

As mentioned, bootstrapping means that a compiler can compile itself. What are the pros and cons of bootstrapping?

- The implemented language becomes well tested, since the developers are using it on a large application (the compiler).
- The developers are motivated to make a high quality implementation, since they are using it themselves.
- The developers are motivated to create a good development environment, since they are using it themselves.
- There is also a negative factor: since the tool must be able to build itself, there is more work to do significant changes to it – the implementation must be good enough to be useable.

2.2 The Stages of Bootstrapping OMC

The bootstrapping of the OpenModelica Compiler (OMC) has been a 5-year effort, consisting of the following stages:

1. Design of an early MetaModelica language version [8] as an extended subset of Modelica, spring 2005.
2. Implementation of a MetaModelica Compiler (MMC) by adding a new compiler frontend to the old RML compiler [13] [28], translating MetaModelica into RML intermediate form, spring-fall 2005.
3. Automatically translating the whole OpenModelica compiler, 60 000 lines, from RML to MetaModelica.
4. In parallel, developing a new Eclipse plugin, MDT (Modelica Development Tooling), for Modelica and MetaModelica [29][31], including both browsing, debugging, semantic context-sensitive information, etc., 2005-2006.
5. Switching to using this MetaModelica 1.0 (an extended subset of Modelica), the MMC compiler, and the new MDT Eclipse plugin for the OpenModelica compiler development, 3-4 full-time developers. This version 1.0 of MetaModelica is described in [10][11]. Fall 2006.
6. Preliminary implementation of pattern-matching [34] and exception handling [31] in the OpenMo-

delica compiler, to enable future bootstrapping. Spring-fall 2008.

7. Continuation of the work on better support for pattern-matching compilation, support for lists, tuples, records, etc. in OpenModelica, as part of metamodeling support in the OMC Java interface [33] Spring-fall 2009.
8. Implementation of function arguments to functions (used in MetaModelica), also in OpenModelica [1]. This also became part of the Modelica 3.2 standard [20]. Fall 2009, spring 2010.
9. The current work on finalizing the bootstrapping reported in this paper. The bootstrapped compiler supports MetaModelica 2.0, which includes both standard Modelica as well as further improved MetaModelica extensions aiming at Modelica 4 support. Fall 2010, spring 2011.
10. Further Adding, enhancing, and redesigning MetaModelica language features [12] based on usage experience, the Modelica 4 design effort, and inspiration from functional languages and languages such as Scala [26]. Refactoring parts of the compiler to use the enhanced features.

3 MetaModelica

As already mentioned, MetaModelica provides language extensions for language modeling and model transformations. The basic language extensions are briefly described here. Some features are defined in libraries.

3.1 Pattern Matching

Pattern matching expressions [34] may occur where expressions can be used in Modelica code. There are two kinds of match-expressions in MetaModelica, using the `match` or `matchcontinue` keywords. The syntax can be described (approximately) as follows.

```
matchcontinue (<var-list>)
  local
    <var-decls>
    ...
   case (<pat-expr>)
     equation
       <equations>
     then <expr>;
   ...
end matchcontinue;
```

Only local, time-independent equations may occur inside a pattern matching expression and this must be checked by the semantic phase of the compiler. The difference between a pattern matching expression with the keyword `match` and a pattern matching expression with the keyword `matchcontinue` is in the fail semantics.

The `<pat-expr>` expression is a sequence of patterns. A pattern may be:

- A wildcard pattern, denoted _.
- A variable, such as *x*.
- A constant literal of built-in type such as *7* or *true*.
- A variable binding pattern of the form `x as` *pat*.
- A constructor pattern of the form *C(pat1,…,patN)*, where *C* is a record identifier and *pat1,…,patN* are patterns. The arguments of C may be named (for instance *field1=pat1*) or positional but a mixture is not allowed. We may also have constructor patterns with zero arguments (constants).

3.1.1 Semantics

The semantics of a pattern matching expression is as follows: If the input variables match the pattern-expression in a case-clause, then the equations in this case-clause will be executed and the matchcontinue expression will return the value of the corresponding then-expression. The variables declared in the uppermost variable declaration section can be used (as local instantiations) in all case-clauses. The local variables declared in a case-clause may be used in the corresponding pattern and in the rest of the case-clause. The matching of patterns works as follows given a variable *v*.

- A wildcard pattern, _, will succeed matching anything.
- A variable, *x*, will be bound to the value of *v*.
- A constant literal of built-in type will be matched against *v*.
- A variable binding pattern of the form `x as` *pat*: If the match of *pat* succeeds then x will be bound to the value of *v*.
- A constructor pattern of the form *C(pat1,…,patN): v* will be matched against C and the subpatterns will be matched (recursively) against parts of *v*.

3.1.2 Pattern Matching Fail Semantics

If a case-clause fails in an expression with the keyword `matchcontinue` then an attempt to match the subsequent case-clause will take place. If we have an expression with the keyword `match`, however, then the whole expression will fail if there is a failure in one of the case-clauses. We will henceforth only deal with `matchcontinue` expressions.

3.2 Data Types

`List`, `Tuple` and `Option` are algebraic data types that are common in many languages used for meta-programming and symbolic programming. The `union-`

`type` is a recursive type required to represent trees and directed acyclic graphs.

3.2.1 Lists

The following operations allow creation of lists and addition of new elements in front of lists in a declarative way. Extracting elements is done through pattern-matching in match-expressions shown earlier.

- **`List`**`(el1,el2,el3, ...)` creates a list of elements of identical type. Examples: **`List`**`()`– the empty list, **`List`**`(2,3,4)` – a list of integers.
- **`{}`** – denotes an empty reference to a list.
- cons – the call `cons(element, lst)` adds an element in front of the list `lst` and returns the resulting list. Also available as a new built-in operator `::` (coloncolon), e.g. used as in: `element::lst`.

Types of lists and list variables can be specified as follows:

```
type RealList = List<Real>;
```

or directly in a declaration of a variable `rlist` that denotes a list of real numbers:

```
List<Real> rlist;
```

3.2.2 Tuples

Tuples can be viewed as instances of anonymous records. The syntax is a parenthesized list. The same syntax is used in extended Modelica presented here, and is in fact already present in standard Modelica as a receiver of values for functions returning multiple results.

- An example of a tuple literal:
  `(a, b, "cc")`
- A tuple with a single element has a comma in order to have different syntax compared to a parenthesized expression: `(a,)`
- A tuple can be seen as being returned from a function with multiple results in standard Modelica:
  `(a,b,c) := foo(x, 2, 3, 5);`
- Access of field values in tuples is achieved via pattern-matching or dot-notation, `tupval.fieldnr`, analogous to `recval.fieldname` for ordinary record values. For example, accessing the second value in `tup`:
  `tup.2`

The main reason to introduce tuples is for convenience of notation. You can use them directly without explicit declaration. Tuples using this syntax are already present in the major functional programming languages.

A tuple will of course also have a type. When tuple variable types are needed, they can for example be declared using the following notation:

```
type VarBND = Tuple<Ident, Integer>;
```

or directly in a declaration of a variable `bnd`:

```
Tuple<Ident, Integer> bnd;
```

3.2.3 Option Types

Option types have been introduced in MetaModelica to provide a type-safe way of representing the common situation where a data item is optionally present in a data structure. In C-like languages this is often represented by `NULL` pointers, which are not type-safe and may cause program crashes.

- `NONE()` represents no data present
- `SOME(e)` represents e present

The option type is declared similar to the list type.

```
type MaybeResult = Option<Result>;
```

3.2.4 MetaModelica Array Types

There is also an array type in MetaModelica, which is different from the Modelica array type. A MetaModelica array may be updated by a function other than the function it was created in. It is mainly used to store side effects, for example caching results of function calls or efficient hash tables.

3.2.5 Union Types

The `uniontype` declaration in MetaModelica is used to introduce union types. This is similar to the `data-type` concept in the ML family of languages [18]. Consider for example a `Number` type, which can be used to represent several kinds of number types such as integers, rational numbers, real, and complex within the same type.

```
uniontype Number
  record INT
    Integer int;
  end INT;
  record RATIONAL
    Integer dividend, divisor;
  end RATIONAL;
  record REAL
    Real real;
  end REAL;
  record COMPLEX
    Real re,im;
  end COMPLEX;
end Number;
```

The most frequent use of the union type is representation of trees or directed acyclic graphs. A tree is a recursive data type, and representing these is as simple as

using the name of the union type as the type of a field in a record that is part of the union type. There are no restrictions or special syntax required to defined mutually dependent union types as shown by `Expression.VAR` and `Subscript.SUBSCRIPT`.

```
uniontype Expression
  record RCONST "A real constant"
    Real r;
  end RCONST;
  record ADD "lhs + rhs"
    Expression lhs, rhs;
  end ADD;
  record SUB "lhs - rhs"
    Expression lhs, rhs;
  end SUB;
  record MUL "lhs * rhs"
    Expression lhs, rhs;
  end MUL;
  record DIV "lhs / rhs"
    Expression lhs, rhs;
  end DIV;
  record VAR "name[sub1, …, subn]"
    String name;
    List<Subscript> subscripts;
  end Var;
end Expression;

uniontype Subscript
  record NOSUB end NOSUB;
  record SUBSCRIPT
    Expression subscript;
  end SUBSCRIPT;
end Subscript;
```

4 Implementation

This section provides the details about the implementation of the compiler and the runtime system for the MetaModelica extensions.

4.1 Compiler

The OpenModelica compiler is implemented mostly in MetaModelica. The front-end, which is where most of OpenModelica development is focused, takes up nearly half of the source code. It elaborates Modelica and MetaModelica code, which is passed on to the backend (for simulations) or directly to code generation (for functions).

The numbers in Table 1 were generated by loading the appropriate sources in the compiler and using the `list()` command to pretty-print the sources. This removes C-style comments, but includes the Modelica-style string comments. The OpenModelica text generation template language Susan [14] is used in the compiler to generate the code generation module.

Table 1. Sizes of OMC Compiler Phases, Lines of Code.

Compiler Phase	Lines
BackEnd (from flat Modelica to sorted eq.syst.)	**29190**
Code generation (generated code)	**35971**
Code generation (template source code)	**8957**
FrontEnd (up to flat Modelica)	**92192**
OpenModelica scripting environment	**21883**
Template language Susan compiler	**12119**
Unparsing modules (pretty-printing data structures)	**16984**
Utility modules	**12983**
Total size (excl. generated code)	***194218***

OpenModelica also requires a runtime system. It is divided into five parts:

- The basic runtime system contains all Modelica builtin function definitions, such as `String`() or `div`() and handles array operations. This runtime needs to be linked to the compiler itself and any source code it produces.
- The simulation runtime system contains the solvers and does event handling.
- The compiler runtime system handles runtime options, settings, CORBA communication and system calls.
- The parser sources are generated by ANTLRv3 [27]. They produce a MetaModelica abstract syntax tree that the compiler can use without further transformations.
- The builtin environment is a Modelica file that contains a list of all builtin functions in the Modelica language as well as the scripting functions that are available in OpenModelica.

Since MMC and OMC external C functions are named and defined differently, they call the same runtime base functions which perform all the work. The lines of code listed in Table 2 exclude the MMC wrapper functions. Header files are included in these measurements, but comments and blank lines are not counted in either headers or source code.

Table 2. Sizes of OMC Parser and Runtime.

Supporting functionality	Lines
Parser (ANTLR sources)	**1968**
Parser (generated C sources)	**49256**
Parser (wrapper code)	**339**
Compiler runtime	**6159**
Simulation runtime	**3405**
Basic runtime (excl. MetaModelica and External)	**9675**
Basic runtime (External code that is maintained in other projects; e.g. Fortran code from LAPACK)	**13268**
Basic runtime (MetaModelica)	**1930**
Modelica builtin environment	**744**
MetaModelica builtin environment	**825**
Total size (excl. generated and external code)	***25045***

4.2 Platform Availability

The OpenModelica compiler runs on all the major platforms, including Windows, Mac, and a number of Linux variants. It also runs under .net using its C# code generator. A code generator emitting Java code is under development.

4.3 Language Feature Implementations

4.3.1 Pattern Matching Implementation

There have been three main attempts to add this language feature to the MetaModelica compiler. These attempts all used a regular C stack and exception handling to model `matchcontinue`. In MMC `matchcontinue` is implemented using the continuation passing style [22], which essentially makes exception handling free, at the cost of not having a regular C stack. The main reason for using a regular stack is debugging, which then becomes much easier.

The first approach [34] introduced C-like constructs to the intermediate representation so that it could be mapped to the runtime at an early stage. It created a DFA (Definite Finite Automaton, a deterministic state machine) so that results of pattern-matching in earlier cases could be remembered when matching later cases. The implementation suffered on several points.

First, it worked directly on the abstract syntax, creating a new expression to be elaborated (basically converting the `match`-expression into an internal representation of Modelica that contained `goto`, `label`, `try`, `throw` and `catch` instead of `match`-expressions). It is hard to change such an implementation or to optimize the code further.

Second, its semantics did not cover the exception handling aspects of `matchcontinue`, so the whole OpenModelica compiler could not be handled by it.

Third, the ways in which it could be used was very specific (only as an assignment statement with variable names as input and output; it could not be nested). The third limitation is minor, since the MMC implementation of MetaModelica did the same thing.

The second attempt to implement match expressions was based on the first one, but removing the DFA part. This made the semantics of `match`/ `matchcontinue` work more as expected. However, while match expressions could now be nested, it still required variable names as input/output and worked directly on abstract syntax.

Maintaining this code and adding special cases for language constructs proved futile. Again, it was not possible to completely cover the semantics of match-expressions.

For example, it is impossible to translate the type of the empty list back to abstract syntax (you cannot write `list<Any>` in a program, even though the type system uses this type for the empty list internally).

The start of the final attempt was the implementation of pattern-matching assignments, e.g.:

```
(a,1.0) := fnCall(...)
```

Previously, this was handled by converting the statement to nested `matchcontinue` blocks, which did not always work. This implementation of pattern-matching was then used to express match expressions, which are now treated as expressions instead of statements.

They have result types like any other expression and may even be the input of another match expression match, e.g.:

```
match (match str
  case "Modelica" then true; else false;
end match)
  case true then 3.0;   else 2.0;
  end match;
```

This approach is much simpler to maintain because it works on the correct level of abstraction. It has access to full type information while doing the translation.

While it does work correctly for all cases, it is a bit slower than the first implementation. It no longer has a DFA to avoid pattern-matching the same thing several times. However, because the code for pattern-matching is essentially created during code-generation instead of during elaboration, it enables us to switch from a C++ runtime to something else if C/C++ is not powerful enough. It also enables us to do optimizations based on patterns. For example, we can detect dead code (un-

reachable patterns) or detect if we can select a case directly instead of doing a linear search of all patterns.

4.3.2 Union Types Implementation

To add support for the `Array`, `Option`, `List` and `Tuple` types to a Modelica compiler is not an easy task. Array expressions sometimes need to be type converted to lists, but there are more expressions than the array data constructor that elaborate to the same expression. For example, the `fill(3,2)` operator call produces an array expression `{3,3}`. This expression can be converted to a list.

The `Option` type was easier to implement than the other types introduced in MetaModelica. The reason for this is that its syntax does not overlap with arrays (like lists), or multiple outputs of functions (like tuples).
Tuples are treated differently in the runtime depending on if they are multiple outputs of a function or a tuple.

The union type syntax looks like it would be accessed by `Package.Uniontype.RECORD`, but in MMC the record is implicitly added to the package scope so it is accessed as `Package.RECORD`. In order to bootstrap the compiler, the same principle is used in OpenModelica. The type of a call to a record constructor is the union type, not the record type itself, which means accessing fields is limited to pattern-matching.

4.3.3 Polymorphism

There could be several different ways to implement polymorphic functions in a Modelica compiler. We use Hindley-Milner-style type inference [15][17] to infer the type of a function based on its interface.

That is, we only consider inputs and outputs of functions when doing type inference. From the inputs of a call expression, we create constraints that we unify. If we succeed, we have determined the actual type of each type variable, which is used to calculate the result type of the call. All other variables have a type and no type inference is done for local variables.

Note that neither Modelica nor MetaModelica has the concept of a `Number` type (some Modelica builtin operators take either `Integer` or `Real` as input, but this cannot be represented in Modelica code). Because there is no `Number` type, a call to `valueEq(1,1.5)` would not pass type checking because the types are different.

While implementing polymorphic functions in a Modelica compiler is rather straight forward, there are a lot of things to think about when solving the type constraints. The names of type variables can be from the current scope, from the called function or from a function pointer used as an input argument. Keeping track of the different sources is the key to getting the correct semantics.

4.4 Runtime System

Because the OpenModelica compiler runtime only handled the static Modelica structures we had to extend the runtime system to handle the new functional constructs.

4.4.1 Data layout

The same layout of values (objects) in memory as for the MMC compiler runtime was used. Basically all values besides integers are boxes that contain a header followed by the actual data. The pointers are tagged (the small number 3 is added to them) to be able to differentiate between pointers and other values such as integers. The differentiation is needed for garbage collection. The headers have 32 bits or 64 bits depending on the platform (32/64 bit platforms). Inside the header the bits are split intro:

- slots – represents the size of data in words
- constructor – represents the type of the data (i.e. index in an uniontype or string, real, etc)

On 32 bit platforms the slots are 22 bits and constructor (the tag) is 10 bits. On 64 bit platforms the slots are 54 bits and 10 bits are devoted to the constructor (the tag).

Integers are either 31 or 63 bits depending on platforms and are represented as even values (i.e. integer N is represented as N << 1, N shifted left by 1).

Bit zero (0) is zero if the box (node) contains pointers and 1 otherwise.

A list value is represented as a box containing a `CONS` header, a pointer to the element and a pointer to next. The end of the list is represented by a box containing a `NIL` header with zero slots.

An Option value is represented as a box containing a `SOME` header and a pointer to the element or a `NONE` header with zero slots.

4.4.2 Builtin MetaModelica functions

New built-in functions are needed to perform operations on the MetaModelica boxed values. The functions are can be used directly in the MetaModelica code or they are generated in the C code from operators. We can classify these functions based on the types they operate on.

- **Booleans**: `boolEq`, `boolString`
- **Integers**: `intEq`, `intLt`, `intLte`, `intGt`, `intGte`, `intNe`, `intString`, `intAdd`, `intSub`, `intMul`, `intDiv`, `intMod`, `intMin`, `intMax`
- **Reals**: `realEq`, `realLt`, `realLte`, `realGt`, `realGte`, `realNe`, `realString`, `realAdd`, `real-`

`Sub`, `realMul`, `realDiv`, `realMod`, `realMin`, `realMax`

- **Strings**: `stringEq`, `stringCompare`, `stringHash`, `stringGet`, `stringAppend`
- **Lists**: `listAppend`, `listMember`, `listGet`, `listReverse`, `cons` (:: operator), `stringAppendList`
- **MetaArrays**: `arrayCreate`, `arrayGet` ([x] index operator), `arrayCopy`, `arrayUpdate`

4.4.3 Garbage collection

Garbage collection (GC), i.e., automatic memory reclamation, of un-used heap-allocated data is a must for a functional language to be able to collect unused values and reuse memory. The garbage collector used in MMC is a simple 2-generational copying compacting garbage collector [28][36]. It has two main memory areas: a young generation where the objects where initially allocated and an older region to which the objects are promoted if they survive a minor collection. We could not reuse this old MMC garbage collector for the OMC runtime because *it moves pointers* and makes it hard to write external functions.

A new garbage collector (GC) has been implemented in OMC. It is a mark-and-not-sweep GC which *does not move* pointers to new addresses. Instead, it marks the unused values (objects) and these can be used for the new allocations. This kind of collector has a better memory usage than the old MMC GC which could only use half of available memory (the other half was needed for the reserve region).

Currently GC can only happen immediately after a function is entered. Basically, the GC starts from a list of roots and marks all values that are reachable from these roots (transitive closure). Then all regions that are not marked become part of a free list (and is garbage) that can be reused for new allocations.

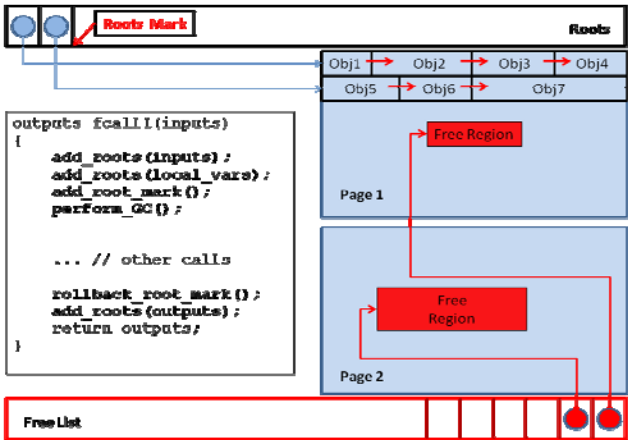

Figure 2. Roots and memory regions during garbage collection.

When we enter a function we add all inputs to the root list and we mark the position in this list. If the function fails we rollback the roots mark to the previous mark and everything allocated above that mark becomes garbage. If the function succeeds the roots mark is rollback and the outputs are added to the roots.

As a future improvement we can allow local allocations in functions as part of roots and then GC could run at any allocation. This way loops that allocate data can perform GC and not exhaust memory. Also we could remove from roots the pointers allocated in the loops because when a loop is finished these pointers should be reachable from variables in the enclosing scope.

4.5 Issues

We identified problems with the tools we use, MMC and OMC. However, there are also design issues in the Modelica language that we needed to work around in order to bootstrap OMC.

4.5.1 MMC Problems

The old MMC compiler has several problems. While it has very good performance, maintaining the code is cumbersome. We would like to have only one tool to maintain, the OpenModelica Compiler. Still, before we can switch to using the OpenModelica Compiler as the default MetaModelica Compiler, we need to be able to compile the same code as MMC during a transition period. The problem is that the MetaModelica to RML translator is very relaxed when it comes to what syntax it accepts.

In order to find some common errors, we traverse the whole abstract syntax and verify that all `matchcontinue` expressions have the same input and output as the function has. This is a limitation of MMC, but it does not actually check that this assertion holds. Moreover, while MMC does some type checking, it does not do it for expressions of the type:

```
x = fnCall(x);
```

For these expressions, the `lhs` and `rhs` `x` may have different types and MMC may allow some code that is not valid MetaModelica code. Finding such code and rewriting it takes a lot of time, but does produce code that is easier to maintain.

Other issues include allowing code like:

```
import Env;
type Env = Env.Env;
```

In the Modelica language the import will essentially be ignored and the tool will probably get a stack overflow because the type `Env` is recursive on itself.

Code that looked like this had to be refactored.

4.5.2 OpenModelica Issues

Some parts of the compiler were rewritten so that it was easier to detect common errors that MMC does not handle. As a result, the OpenModelica Compiler now has vastly better error messages for algorithmic code. We have also started propagating file, line and column information to more error messages because when you have a large application that you want to fix, it is essential that you find the correct line quickly. Improving the error messages probably saved more time than it cost to implement and as a result, the compiler should be giving better error messages both for Modelica and MetaModelica code.

OpenModelica has issues with shadowing certain builtin operators. For example, creating a function `ndims` and calling it in the same package will result in the Modelica builtin ndims operator being used instead of this function. A few of these functions existed in the compiler and had to be renamed (rewriting Lookup in OpenModelica is an alternative, but would take more time).

4.5.3 Modelica Issues

The Modelica Specification [17] says that external functions map `Integer` to `int`, with no way to change their behavior. If external functions in the compiler runtime need to access large integers, this is a problem. Examples of such a function are `referenceEqual`, which checks if two pointers are equal and the `stringHash` family of functions. Values were being truncated because of the limited precision of `int` on 64-bit platforms, so we had to move functions from external C functions in the compiler into the MetaModelica language itself.

5 Performance

While we are not fully satisfied with the performance of the compiler at the time of writing this text, OMC compiled by OpenModelica has only been running for a few weeks. This initial version lacks a garbage collector; we simply allocate memory until we run out. Even without memory management, the majority of the OpenModelica regression and unit tests run.

The performance of the working examples varied widely and some tests could be heavily improved by doing small changes.

In the first executable version (a few weeks ago) of the bootstrapped compiler even simple test cases were nine times slower than in the MMC version. Since the new version of OMC uses a real C stack, it is possible to use general-purpose debugging and profiling tools, such as gdb, gprof or valgrind on it.

Using `valgrind --tool=callgrind`, we could see that the `PEXPipe` example spent around 80% of its time doing exception handling. The C++ try-throw-catch feature had been used to implement the exception handling [31] of `matchcontinue` expressions. C++ exceptions are slow because they are not supposed to be used often. We rewrote the exception handling using `setjmp/longjmp`. Since we do not use C++ classes or C++-heap-allocated data, this level of exception handling is sufficient for us.

The new exception handling resulted in a 10x speed-up for certain examples (on the average 3x). Exception handling still uses approximately ~25% of the compiler execution time. We also implemented optimizations that rewrite `matchcontinue` to `match` automatically as a compiler optimization. This reduced some of the setjmp overhead.

5.1 Benchmarks

All benchmarks were performed using a laptop running 64-bit Ubuntu 10.10. The machine was equipped with a dual-core Intel Core i7 M620 (2.67GHz) and 8GB of memory. All file IO was performed on RAM-disk in order to reduce disk overhead. Both compilers were compiled using GCC 4.4.5 with optimization level set to –O3.

The two implementations use virtually the same parser, so speed here is identical. The Modelica Standard Library (MSL) is distributed in multiple files that are parsed and merged into a single tree by the OpenModelica Compiler. The version of MSL used was 3.1. MSL 3.1 is 7.8MB stored as a single file contains 922 models and 615 functions.

Unparsing the tree is the process of going from the internal tree structure back to Modelica concrete syntax (source code). Here MMC suffers from an additional performance penalty because the tree was created in an external C function and not directly on its heap.

Table 3 Compiler performance, parsing

Task	MMC	OMC	Factor
Parse MSL3.1, single file	1.577s	1.577s	1x
Parse MSL3.1, multiple files	1.297s	1.253s	0.97x
Parse MSL3.1+Unparse	3.674s	2.172s	0.59x
Unparse only	2.377s	0.600s	0.25x

Table 4 shows the performance of OMC for simple tests, ~1x the speed of MMC.

Table 4 Compiler performance, simple tests

Task	MMC	OMC	Factor
drmodelica, 104 flattening tests	2.136s	2.049s	0.96x
mofiles, 548 flattening tests	12.770s	13.409s	1.05x
msl 1.5, 60 flattening tests	1.726s	1.578s	0.91x
mosfiles, 120 simulation test	110.67s	107.96s	1.03x

The tests are limited to 5GB of virtual memory through the use of `ulimit`. This rather low limit was chosen because then swap memory would never be accessed. Because of this limit, a few tests fail. The bootstrapped compiler has issues running large models. For example, running `checkModel` command on the `EngineV6Analytic` model, which has 9491 equations, uses roughly 7GB of memory. The overall performance shown in Table 5 is acceptable.

Table 5 Compiler performance, big tests

Task	MMC	OMC	Factor
EngineV6Analytic, checkModel, time	55.908s	55.553s	0.99x
EngineV6Analytic, checkModel, memory	400MB	7GB	17.5x
All 1379 tests, single thread, time	24m8s	20m55s	-
All 1379 tests, number of failed tests	13	30	-
Excluding failing tests, 1349 total	17m25s	18m	1.03x

There is a difference in how the two tool chains create the executable compiler. MMC is created by translating each package into a C-file, compiling them separately, and linking everything together. OMC is created by translating all packages into a single 31MB C-file. While this may produce a faster executable, the whole compiler needs to be recompiled even if only one package changes. We are working on supporting both methods in the future, including fine-grained separate compilation for faster turn-around.

As Table 6 shows, compiling OMC using OMC is significantly faster than using MMC. The size of a non-debug executable was calculated by stripping it of all debug symbols and tracing functionality. Since the bootstrapped compiler compiles quite fast, we can spend more time on optimization algorithms. MMC uses code instrumentation when compiling a debug executable. The bootstrapped compiler has a C stack and simply adding debug symbols is sufficient to run a debugger or profiler on it.

Table 6 Compilation speed

Task	MMC	OMC	Factor
Translate sources to C	98.8s	49.7s	0.50x
Compile C, gcc –O3	233.3s	152.0s	0.65x
Executable size	18.7MB	5.43MB	0.29x
Compile C, gcc –Os	218.9s	102.8s	0.47x
Executable size	10.3MB	5.10MB	0.50x
Debug version, gcc –g –O3	1405s	152.5s	0.11x
Executable size	125MB	25MB	0.20x

6 Related Work

Functional programming languages often bootstrap themselves during the build process. Some also include integrated lexer and parser generators specific to their own programming language, which is something MetaModelica is currently missing, but under development.

In the Lisp family [35] bootstrapped compilers have been available since the 1960's. These languages are strongly typed, like MetaModelica, but defer the check to runtime instead of compile-time. This often leads to error messages that are hard to understand because you don't always know where a piece of data originated if you get a type error.

Objective Caml [25], SML/NJ [1] and MLton [19] are examples of bootstrapped compilers in the ML family. These languages are similar to MetaModelica in that they both use very similar language constructs and type inference. This should come as no surprise since RML/MMC is written in Standard ML and compiles using either SML/NJ or MLton. One major difference is that all variables in MetaModelica have a specific type while in ML each expression has a most general type. MetaModelica can generate error messages that are easier to understand because type inference only has to be performed when calling a polymorphic function. However, it also results in more local variable declarations since all temporary variables need to be declared. This is both positive (you document what type you expect a variable should have) and negative (you end up with a lot of local variables).

The concept of the `matchcontinue` expression is something that the ML family is missing. It is instead possible to use explicit exception handling or use guard expressions to prevent a case from actually matching a pattern, which is often sufficient. The ML family of languages also has lambda functions, which is currently missing in MetaModelica.

The `matchcontinue` expression is more general than the `match` expression, which is common in functional programming. It is related to clauses in logic programming since it provides backtracking on failure. However, in Prolog [24] there are usually many possible answers to a given logic program since it evaluates combinations of clauses that satisfy the program. In MetaModelica only the first valid answer is returned and no subsequent case is evaluated. Thus, MetaModelica is more efficient and consistently statically strongly typed, whereas logic programs sometimes can be expressed more concisely.

7 Conclusions

We have demonstrated a Modelica compiler that is capable of compiling itself. Thus, it is possible to describe all of the semantics of Modelica in a slightly extended version of Modelica. We have also shown that the performance of the implementation is sufficient. The effort to achieve these goals was higher than initially expected, and included not only developing the compiler but also a development environment including an Eclipse plugin and a debugger.

Many of the MetaModelica language extensions that allow language modeling are in line with the design goals for Modelica 4 to allow modeling of language features in libraries. We believe that this work will be an important input and proof-of-concept to the Modelica 4 design effort.

7.1 Future Work

The garbage collection implementation needs to be finalized and tested before we can switch to using the bootstrapped compiler for development work.

Once that is done, parts of the compiler can be rewritten/refactored using certain more powerful and concise language constructs in MetaModelica 2.0 [12].

Furthermore, to realize important design goals, a number of language features should be moved into libraries and an enhanced API for accessing compiler functionality from such libraries need to be developed.

The debugger [32] in the Eclipse environment, including a data inspector for the MetaModelica types, needs to be ported to the bootstrapped OMC. Furthermore, the generated code needs to be annotated with line numbers to support the debugger and performance analyzer in mapping to Modelica source file positions.

8 Acknowledgements

This work has been supported by Vinnova in the ITEA2 OPENPROD project, and by SSF in the Proviking HIPo project. The Open Source Modelica Consortium supports the OpenModelica work.

References

[1] Matthias Blume. The SML/NJ Bootstrap Compiler User Manual. July, 2001.

[2] David Broman and Peter Fritzson. Higher-Order Acausal Models. In *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools, (EOOLT'2008*), Pathos, Cyprus, July 8, 2008. Published by Linköping University Electronic Press, http://www.ep.liu.se/ecp/024/, July 2008.

[3] David Broman. *Meta-Languages and Semantics for Equation-Based Modeling and Simulation*. Dissertation No 1333, www.ep.liu.se, Linköping University, October 1, 2010.

[4] Stefan Brus. Bootstrapping The OpenModelica Compiler: Implementing Functions As Arguments. Master thesis draft, 2009. www.ep.liu.se. Finalized Spring 2010.

[5] Emil Carlsson. Translating Natural Semantics to Meta-Modelica. Master Thesis, LITH-IDA-Ex--05/073—SE, Linköping University, October 2005.

[6] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, 940 pages, Wiley-IEEE Press, 2004.

[7] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. *Simulation News Europe*, 44/45, Dec. 2005. http://www.openmodelica.org

[8] Peter Fritzson, Adrian Pop, and Peter Aronsson. Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica. In *Proc. of the 4th International Modelica Conference*, Hamburg, Germany, March 7-8, 2005.

[9] Peter Fritzson. *Language Modeling and Symbolic Transformations with Meta-Modelica*. (Later versions[10], [11]), www.ida.liu.se/~pelab/Modelica, and www.openmodelica.org Version 0.5, June 2005.

[10] Peter Fritzson. *Modelica Meta-Programming and Symbolic Transformations - MetaModelica Programming Guide*. (Slightly updated version of [9], later update: [11]) http://www.openmodelica.org/index.php/developer/devdocumentation. June 2007.

[11] Peter Fritzson and Adrian Pop. *Meta-Programming and Language Modeling with MetaModelica 1.0*. (Almost identical to [10], but

tech. report). Technical reports in Computer and Information Science, No 9, Linköping University Electronic Press, http://www.ep.liu.se/PubList/Default.aspx?SeriesID=2550, January 2011.

[12] Peter Fritzson, Adrian Pop, and Martin Sjölund. *Towards Modelica 4 Meta-Programming and Language Modeling with MetaModelica 2.0*. Technical reports in Computer and Information Science, No 10, Linköping University Electronic Press, http://www.ep.liu.se/PubList/Default.aspx?SeriesID=2550, February 2011.

[13] Peter Fritzson, Adrian Pop, David Broman, Peter Aronsson. Formal Semantics Based Translator Generation and Tool Development in Practice. In *Proceedings of ASWEC 2009 Australian Software Engineering Conference*, Gold Coast, Australia, 2009.

[14] Peter Fritzson, Pavol Privitzer, Martin Sjölund, and Adrian Pop. Towards a text generation template language for Modelica. In *Proceedings of the 7th International Modelica Conference*, Como, Italy, Sept. 20-22, 2009

[15] J. Roger Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29-60, Dec. 1969.

[16] Kenneth C. Louden. *Programming Languages, Principles and Practice*. ISBN 0-534-95341-7, Thomson Brooks/Cole, 2003.

[17] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:248-375, 1978.

[18] Robin Milner, Mads Tofte, Robert Harper and David MacQueen. *The Definition of Standard ML*, 128 pages, MIT Press, 1997.

[19] MLton. Installation Instructions. Jan. 2011. http://mlton.org/PortingMLton.

[20] Modelica Association. *The Modelica Language Specification Version 3.2*, March 2010. http://www.modelica.org.

[21] Modelica Association. *Modelica Standard Library 3.1*. Aug. 2009. http://www.modelica.org.

[22] Modelica Association. Minutes of the Modelica Design Meeting 67, www.modelica.org, Atlanta, Georgia, September 2010.

[23] Henrik Nilsson, John Peterson, and Paul Hudak. Functional Hybrid Modeling from an Object-Oriented Perspective. In *Proc. of EOOLT'2007*, LIU Electronic Press, www.ep.liu.se, Berlin, Germany, August, 2007.

[24] Ulf Nilsson and Jan Maluszynski. *Logic, Programming and Prolog*. Wiley, 1995.

[25] Objective Caml 3.12.0 installation notes. Aug, 2010. http://caml.inria.fr

[26] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, 2008.

[27] Terence Parr. ANTLR Parser Generator 3.3, Accessed November 2010. http://antlr.org/

[28] Mikael Pettersson. *Compiling Natural Semantics*, Department of Computer and Information Science, Linköping University, PhD Thesis No. 413, 1995. Published in *Lecture Notes in Computer Science* No 1549, Springer Verlag, 1999.

[29] Adrian Pop, Peter Fritzson, Andreas Remar, Elmir Jagudin, and David Akhvlediani. OpenModelica Development Environment with Eclipse Integration for Browsing, Modeling, and Debugging. In *Proc 5th International Modelica Conf. (Modelica'2006)*, Vienna, Austria, Sept. 4-5, 2006.

[30] Adrian Pop and Peter Fritzson. MetaModelica: A Unified Equation-Based Semantical and Mathematical Modeling Language. In *Proceedings of Joint Modular Languages Conference 200*6 (JMLC2006) Published in Lecture Notes in Computer Science No 4228, ISSN 0302-9743, Springer Verlag. Jesus College, Oxford, England, Sept 13-15, 2006.

[31] Adrian Pop, Kristian Stavåker, and Peter Fritzson. Exception Handling for Modelica. In *Proceedings of the 6th International Modelica Conference* (Modelica'2008), Bielefeld, Germany, March.3-4, 2008.

[32] Adrian Pop. *Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages*. www.ep.liu.se, PhD Thesis No. 1183, Linköping University, June 5, 2008.

[33] Martin Sjölund. Bidirectional External Function Interface Between Modelica/MetaModelica and Java. Master thesis, IDA/LITHEXA09/041SE, Aug 2009.

[34] Kristian Stavåker, Adrian Pop, and Peter Fritzson. Compiling and Using Pattern Matching in Modelica. In *Proceedings of the 6th International Modelica Conference (Modelica'2008)*, Bielefeld, Germany, March.3-4, 2008.

[35] Guy L. Steele Jr. and Richard P. Gabriel. The Evolution of Lisp. In *Proceedings of the Conference on History of Programming Languages*. New York, April, 1993.

[36] Paul R. Wilson, Uniprocessor garbage collection techniques, Lecture Notes in Computer Science, Volume 637/1992, page 1-42. Springer Verlag. 1992.

[37] Dirk Zimmer. *Equation-Based Modeling of Variable Structure Systems*. PhD Dissertation, ETH Zürich, 219 pages, 2010.