Modeling, Identification and Control, Vol. 31, No. 1, 2010, pp. 1–19, ISSN 1890–1328

Bootstrapping a Compiler for an Equation-Based Object-Oriented Language

M. Sjölund P. Fritzson A. Pop

PELAB, Linköping University, SE-581 83 Linköping, Sweden.
E-mail: {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 of a full-scale EOO (Equation-based Object-Oriented) modeling language such as Modelica. Bootstrapping means that the compiler of a language can compile itself. However, the usual application area for the Modelica is modeling and simulation of complex physical systems. Fortunately it turns out that with some minor extensions, the Modelica language is well suited for the modeling of language semantics. We use the name MetaModelica for this slightly extended Modelica. This is a prerequisite for bootstrapping which requires that the language can be used to model and/or implement itself. The OpenModelica Compiler (OMC) has been written in this MetaModelica language. It originally supported only the standard Modelica language but has been gradually extended to also cover the MetaModelica language extensions. After substantial work, OMC is able to quickly compile itself and produces an executable with good performance. The benefits include a more extensible and maintainable compiler by introducing improved language constructs and a more powerful runtime that makes it easy to add functionality such as parser generators, debuggers, and profiling tools. Future work includes extracting and restructuring parts of OMC, making the compiler smaller and more modular and extensible. This will also make it easier to interface with OMC, making it possible to create more powerful and user-friendly OpenModelica-based tools. The compiler and its bootstrapping is a major effort – it is currently about 330 000 lines of code, and the MetaModelica extensions are used routinely by approximately ten developers on a daily basis.

Keywords: compilation,equation-based,object-oriented,meta-programming,modeling

1 Introduction

The phenomenon of programming languages and compiler bootstrapping is not uncommon. Early examples include Lisp (Steele and Gabriel, 1993) and Pascal (Wirth, 1971). One of the advantages is assumed to be higher quality since the designers and developers of a language and its compiler will be major users, and therefore will be highly motivated to correct possible design flaws and errors. Another advantage is portability – the bootstrapped compiler is primarily dependent on itself, not on other languages, once it has been bootstrapped.

Bootstrapping means that the language and its compiler is defined and implemented using itself. On a first impression this sounds impossible, especially for a new language, since the language must exist before it is used. The term bootstrapping comes from lifting yourself in your own boot straps, which of course is impossible. However, bootstrapping of compilers is possible. A common approach is to write a subset language compiler in another available language, then rewrite the subset compiler into its own language, extend the compiler to handle the remaining constructs, and finally update the compiler to use the full language.

This paper describes the first bootstrapping of a full-

doi:10.4173/mic.2010.1.1

scale EOO (Equation-based Object-Oriented) modeling language, in this case the Modelica language. Modelica is rather unusual since it supports mathematical modeling with equations, that is, differential, algebraic, and discrete equations. However, such equations are not very suitable for modeling language semantics and symbolic transformations. Therefore Modelica was extended with pattern equations, pattern matching, and tree and list data structures. The extended language is called MetaModelica.

The bootstrapping of Modelica is a large-scale practical effort since the compiler is currently about 330 000 lines of code, and approximately ten full-time developers are using MetaModelica on a daily basis. The standard Modelica language is used by several thousand developers including people developing large industrial applications, many of which are described in Modelica conference proceedings, for example Otter and Zimmer (2012), Clauß (2011), Casella (2009), and Bachmann (2008).

In the case of Modelica, we have at least two motivations for bootstrapping in addition to those mentioned previously. One motivation is a scientific experiment in language design with the goal of a generalized equation-based language suitable for both physical system modeling and language modeling (Pop and Fritzson, 2006).

A second motivation comes from the growth of the Modelica language itself, and the corresponding increase of complexity and size of its compilers. This is caused by steadily increasing user application requirements on the language including widening of its application domains.

One approach to manage such an increasing language and compiler complexity is the language core library approach used by several functional languages (Section 7). A number of language features are defined in core libraries rather than in the compiler itself.

After several years of discussion and a number of language prototypes (Broman, 2010; Fritzson et al., 2005b; Fritzson, 2005; Nilsson et al., 2007; Pop and Fritzson, 2006; Pop, 2008; Zimmer, 2010), this approach finally became accepted as a design goal for future Modelica versions during the 67$^{\text{th}}$ Modelica design meeting Modelica Association (2010).

This paper gives a rather complete account of the Modelica bootstrapping effort in the OpenModelica compiler project of which an early preliminary description is available in Sjölund et al. (2011).

1.1 Modelica – An Equation-Based Object-Oriented Language

Before the Modelica language effort started, some of us were involved in developing a language and tool called ObjectMath (Viklund et al., 1992), an equation-based object-oriented (EOO) specification language for mathematical modeling. Several other groups developed related languages and tools, for example Dymola (Elmqvist, 1978), NMF (Sahlin and Sowell, 1989), Smile (Kloas et al., 1995), and gPROMS (Barton and Pantelides, 1994). In 1996, some of these groups joined forces to create an internationally viable declarative mathematical modeling language. The result is the Modelica language (Elmqvist et al., 1999; Fritzson, 2004; Fritzson and Engelson, 1998; Tiller, 2001; Modelica Association, a). It is an equation-based object-oriented (EOO) modeling language for declarative mathematical modeling of large and heterogeneous (multi-domain) physical systems. For modeling with Modelica, commercial software products such as System Modeler (Fritzson et al., 2002; Wolfram Mathcore, 2012), Dymola (Brück et al., 2002), SimulationX (ITI GmbH, 2012), MapleSim (Maplesoft, 2012), IDA Simulation Environment (Equa AB, 2002), etc are available. There are also open source implementations like our own tool, OpenModelica (Fritzson et al., 2005a), and the more recent JModelica.org (Åkesson et al., 2010)[1].

The Modelica language has been designed to allow tools to generate efficient simulation code automatically with the main objective of facilitating exchange of models, model libraries and simulation specifications. The typical result of using a Modelica tool on a model is a plot window with the results of a simulation.

The language is statically strongly typed and provides object-orientation[2] with multiple inheritance and generics templates within a single class construct. This facilitates reuse of components and evolution of models.

While this overview is sufficient for the purpose of reading the rest of this text, Modelica has many more features from a simulation practitioner's point of view, including acausal modeling, multi-domain modeling, hybrid system modeling, graphical modeling, index reduction of differential-algebraic equations (DAE's), and more). To learn more about Modelica or the motivations and design goals that led to it, see books (Fritzson, 2004, 2011; Tiller, 2001; Fritzson, 2014) as well as shorter overviews (Elmqvist et al., 1999; Fritzson and Engelson, 1998; Fritzson and Bunus, 2002), and the language specification (Modelica Association, a).

1.2 Specification of Language Constructs

The Modelica specification and modeling language was originally developed as an object-oriented declarative

[1] `http://modelica.org` has a complete list of Modelica tools.

[2] Object-orientation in the sense of hierarchical modeling, not object-oriented programming like Java, which tends to focus on methods.

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 also well suited for another modeling task, namely modeling of the semantics, that is, the meaning, of programming language constructs. Since modeling of programming languages is often known as meta-modeling, we use the name MetaModelica (Fritzson et al., 2005b; Pop and Fritzson, 2006; Fritzson and Pop, 2011a; Fritzson et al., 2011) for this slightly extended Modelica.

Well-known language specification formalisms such as Structured Operational Semantics and Natural Semantics (Pettersson, 1995a, 1999) also have the property of being declarative equation-based formalisms. These formalisms fit well into the style of the Modelica 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. 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, that is, a kind of compilation.

Figure 1 shows typical translation stages in a Modelica compiler.

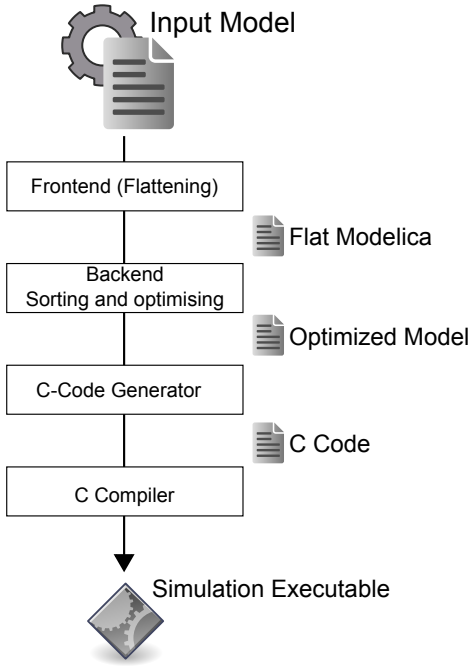

Figure 1: The typical stages of translating and executing a Modelica model.

2 Vision – Extensible Tools

Traditionally, a model compiler performs the task of translating a model into executable code, which subsequently 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 (partial differential equation) 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 Motivation for Compiler Bootstrapping

As mentioned in the introduction, bootstrapping means that a compiler can compile itself. Why is this relevant in the context of language specification and language semantics modeling? The most important factor is probably that the language itself can be used as a large test case for language specification using the developed specification language. This has mostly advantages, but also a disadvantage:

- 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 them-

selves.

- 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, more effort is needed to create it since the implementation must be good enough to be usable. Moreover, initially a new language and environment may lack some qualities and tools available already in existing languages. For example, during the first years of using MetaModelica there was no good debugger available.

2.2 The Stages of Bootstrapping OMC

The bootstrapping of the OpenModelica Compiler (OMC) has been a 7-year effort, consisting of the following stages:

1. Design of an early MetaModelica language version (Fritzson et al., 2005b) 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 (Pettersson, 1995a; Fritzson et al., 2009a), 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 (Pop et al., 2006, 2008), including both browsing, debugging, semantic contextsensitive 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, at that time 3-4 full-time developers. This version 1.0 of MetaModelica is described in (Fritzson, 2007; Fritzson and Pop, 2011b). Fall 2006.
6. Preliminary implementation of pattern-matching by Stavåker et al. (2008) and exception handling by Pop et al. (2008) in the OpenModelica 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. This was part of the metamodeling support in the OMC Java interface that was implemented by Sjölund (2009) in spring-fall 2009.
8. Implementation of function arguments to functions (used in MetaModelica), also in OpenModelica (Brus, 2009). This also became part of the Modelica language specification (Modelica Association, a). 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 becoming future Modelica extensions. Fall 2010, spring 2011.
10. Further adding, enhancing, and redesigning MetaModelica language features in Fritzson et al. (2011) was based on usage experience, the Modelica design effort, and inspiration from functional languages and languages such as Scala (Odersky et al., 2008). Refactoring parts of the compiler to use the enhanced features.
11. Adding garbage collection. Fall 2012.
12. Improving the build system, parallel builds. Reaching full testsuite coverage, good performance, and running the tests nightly. 2013.

3 MetaModelica

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

3.1 Pattern Matching

MetaModelica pattern matching expressions (Stavåker et al., 2008) 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;
```

In the MetaModelica language extension only local, time-independent equations may occur inside a pattern matching expression which 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, see Section 3.1.2. The `matchcontinue` variant is a `match` with backtracking and continuation of the next case in the `match` at failure. The match variant has no backtracking.

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 $C(pat_1, ..., pat_n)$, where C is a record identifier and $pat_1, ..., pat_n$ are patterns. The arguments of C may be named (for instance $field_1 = pat_1$) 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 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 variable, x, will be bound to the value of v just as x `as _` would.
- A constant literal of built-in type will be matched against v.
- A constructor pattern of the form $C(pat_1, ..., pat_n)$: 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 in this paper primarily deal with `matchcontinue` expressions.

3.2 Data Types

`List`, `Tuple` and `Option` are algebraic data types which are common in many languages used for meta-programming and symbolic programming. The `uniontype` is a recursive type required to represent trees and directed acyclic graphs, which is similar to algebraic data types in functional languages such as SML (Milner et al., 1997) and Haskell (Peyton Jones et al., 2003), and case classes in Scala (Odersky et al., 2008).

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 reference to an empty list.
- the call `cons(element, lst)` adds an element in front of the list and returns the resulting list. Also available as a new built-in operator `::` (colon-colon), for example 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 the extended Modelica (that is, MetaModelica) 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 and Java-like languages this is often represented by `NULL` pointers, which are not type-safe and may cause program crashes even when the data has been initialized by the programmer. Examples include double-free, or usage of a copy of a now freed pointer.

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

The option type is declared analogous 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 can be used to represent ragged arrays, that is, arrays of arrays that may have unequal size. More importantly for the point of view of performance is Modelica arrays need a deep copy when assigning one array to another. This means it allocates new memory and copies all the data content from the old array to the new, a linear-time operation. In MetaModelica copying of arrays is by reference, which is a constant-time operation. New arrays have to be explicitly created. The main use of arrays is currently for side effects that increase performance, for example caching partial results even if a function fails.

3.2.5 Union Types

The `uniontype` declaration in MetaModelica is used to introduce union types. This is similar to the algebraic data types represented by the `datatype` construct in the ML family of languages (Milner et al., 1997). Consider for example the `Number` type below, 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 for 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 some information 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 compiler front-end currently comprises almost half of the source code. Here the front-end is defined as the compiler phases that convert the source model into a flattened intermediate form consisting of equations, variables and functions. It elaborates Modelica and MetaModelica code into so-called flattened intermediate code, which is passed on to the backend equation optimizer (for simulation models) or directly to the code generator (for functions).

The number of lines of MetaModelica code used to implement OpenModelica are in Table 1. Empty lines and C-style comments have been removed, but Modelica-style string comments are included. The OpenModelica text generation template language Susan described in Fritzson et al. (2009b) is used in the compiler to generate most of the files in the code generation module. Susan generates MetaModelica code for appending large strings since appending strings without a string builder is very inefficient if done incorrectly. It is also easier to understand and edit the Susan source code than optimized MetaModelica source code.

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 numerical solvers and also performs event handling.
- The compiler runtime system handles runtime options, settings, CORBA communication between OMC and clients, as well as system calls.
- The parser sources are currently generated by the ANTLRv3 (Parr, 2010) parser generator. This tool produces code that can be called through the foreign function interface of MetaModelica (Section 4.4.2). When run, the parser produces a MetaModelica abstract syntax tree that the compiler can use directly. An alternative approach would be to use a parser generator that produces MetaModelica code instead, analogous to MLLex (Appel et al., 1994) and MLYacc (Tarditi and Appel, 2000) for the SML language. A recent prototype of such a parser generator is OMCCp (Lopez-Rojas, 2011; Palanisamy et al., 2014).
- The built-in environment of predefined functions, types, and constants is specified by 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.

The MMC foreign function interfaces is completely different from the OMC interface which means they cannot use the same code directly. Most of the runtime is written as an implementation conforming to the OMC interface and an MMC wrapper around this, as shown below:

```
RML_BEGIN_LABEL(Lapack__dorgqr)
{
  void *A, *WORK;
  int INFO;
  LapackImpl__dorgqr(RML_UNTAGFIXNUM
      (rmlA0), ...);
  rmlA0 = A;
  rmlA1 = WORK;
  rmlA2 = (void*)RML_TAGFIXNUM((long
      )INFO);
  RML_TAILCALLK(rmlSC); /* Success
      continuation */
}
RML_END_LABEL;
```

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

Compiler phase	**Files**	**Lines**
Constants	1	27
Entry-point	1	902
Scripting environment	6	26788
Utility functions	48	24202
FrontEnd (up to flat Modelica)	106	165569
BackEnd (from flat Modelica to sorted eq.syst.)	46	96372
Code generation (hand-written Susan runtime)	5	13816
Code generation (Susan template source code)	21	36624
Code generation (generated MetaModelica code)	21	211088
Total size (excl. generated code)	213	327676

The lines of code listed in Table 2 exclude these MMC wrapper functions. Header files are included in these measurements, but comments and blank lines are not counted in either headers or source code.

4.2 Platform Availability

The OpenModelica compiler runs on all the major platforms, including Windows, Mac OSX, and a number of Linux variants. With the bootstrapped compiler it is now also possible to compile on platforms without a Standard ML compiler, which used to be a limiting factor in porting OpenModelica to new operating systems.

4.3 Language Feature Implementations

4.3.1 Pattern Matching Implementation

In the old MMC compiler the `matchcontinue` construct was implemented using the continuation passing style. This is the same as in the RML language by Pettersson (1995b) on which it was based. It provides exception handling without additional cost. The main drawback is the lack of a regular C stack. The lack of a regular C stack precludes some optimizations typically performed by C compilers, and also makes it impossible to use standard performance analyzers and debuggers.

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`.

The first approach (Stavåker et al., 2008) introduced C-like constructs (like `goto`) 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. But the DFA was unable to handle all of the aspects of the `matchcontinue` semantics. The code that generated the DFA was too hard to maintain, and did not produce faster code, only slightly smaller code size in some cases. Therefore this part was removed in the second approach.

Removing the DFA solved the issues of `match` and `matchcontinue` giving unexpected results. It also made it possible to nest `match`-expressions. But there still existed flaws in the implementation. Each `match`-expression needed to be a statement, and the inputs of the `match` were required to be variable names. While this is the same restriction as in the MMC implementation, the language design requires expressions to be more general.

One of the problems with the implementation was that it elaborated the expression twice (from a typed expression back to untyped, and then typed again so it could add temporary variables with the correct type name). Trying to add special code to allow for empty lists proved futile as its internal type (list of Any) cannot and should not be possible to express in the MetaModelica abstract syntax.

The start of the final attempt was the implementation of pattern-matching statements/assignments, for example:

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

Previously, pattern-matching assignments were handled by being converted to nested `matchcontinue` blocks. This did not always work as types of temporary variables could not always be expressed in MetaModelica abstract syntax. The new implementation has the concept of *patterns* in addition to *expressions*. They are treated differently and elaboration is now performed in a single step without converting an elaborated expression back to abstract syntax. Generation of temporary variables is performed by the code generator. This allows a complete implementation of `matchcontinue` without any major issues, and also allows good error

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 maintained in other projects; for ex. 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

messages when elaborating invalid patterns.

Since `match` expressions are now implemented as expressions instead of as a specific kind of assignment, they can now be used as regular expressions. Since they have result types they may even be the input of another match expression, for example:

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

This approach is much simpler to maintain because it works at the correct level of abstraction, that is, elaborated expressions. The compiler has access to the full type information while doing the translation.

While this works correctly for all cases, it might be a bit slower than the first implementation in some cases. 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 enabled switching from a C++ runtime to C (see Section 5). For example, if the target language knows what a `match`-expression is, it is possible that it knows how to optimize such constructs very well – this would be true if a functional language was the target language. We have implemented our own optimizations based on patterns. For example, dead code eliminiation (unreachable patterns) and detection of expressions where a case can be directly selected instead of doing a linear search of all patterns (similar to `switch` in C, but for more data types). We could also generate more low-level code (like a DFA) based on this structure. However, experience has shown that the OpenModelica compiler currently generates sufficiently fast code directly from the intermediate code without allocating registers, creating temporary variables and so on.

4.3.2 Type Implementations

Adding support for the `Array`, `Option`, `List` and `Tuple` types to a Modelica compiler is not an easy task due to subtle incompatibilities in the type systems of MetaModelica and Modelica.

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}`, which can then be cast to a list. The MetaModelica 2.0 design (Fritzson et al., 2011) solves these issues.

The `Option` type was easier to implement than the other types introduced in MetaModelica. The reason is that its syntax does not overlap with arrays (like lists), or multiple outputs of functions (like tuples).

Tuples are treated differently in the compiler and the runtime depending on whether they are multiple outputs of a function or a tuple. A function that returns `Tuple<Integer,Integer>` has a different type than one that returns two integers. This is different from most functional programming languages, but is consistent with the Modelica design as well as the design of the RML specification language.

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 that 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 that accessing fields from `uniontype` records currently is limited to pattern-matching.

4.3.3 Polymorphism

There could be several different ways to implement polymorphic functions in a Modelica compiler. Hindley-Milner-style type inference (Hindley, 1969; Milner, 1978) is used to infer the type of a function based on its interface.

That is, only inputs and outputs of functions are considered when doing type inference. From the inputs of a call expression, a set of constraints are created and subsequently unified. If unification succeeds, 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 general `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.

Even though implementing polymorphic functions in a Modelica compiler is rather straight forward, there are some pitfalls. 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 initially 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 with some enhancements for faster debugging (see Section 6). 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 was needed for the garbage collection implementation as used in MMC. It is now instead needed in order to provide type information in the debugger. The headers have 32 bits or 64 bits depending on the platform (32/64 bit platforms). Inside the header the bits are split into:

- slots – represent the size of data in words
- constructor – represents the type of the data (that is, 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 (that is, integer N is represented as N << 1, N shifted left by 1).

Bit zero (0) is set if the box (node) contains pointers and unset otherwise, or the other way around depending on what garbage collection characteristics are desired.

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.

There is a header that contains macros to access the desired fields without requiring to know which bits to read, taking care of cross-platform issues.

4.4.2 Foreign Function Interface

Modelica has support for calling `external "C"` functions with a mapping of Modelica types to C types. The types introduced in MetaModelica (`List`, `Tuple`, `Option`, `Array`, and `uniontype`) all map to `void*`. The macros used to access and create these types are different for MMC and OMC, but a compatibility header exists so code can be shared between the two implementations. One major difference between the MMC and OMC implementation is that within an external function, variables may have been moved by the garbage collector after calling another function in the MMC implementation. In OMC, calling other functions is considered safe.

4.4.3 Builtin MetaModelica Functions

New built-in functions are needed to perform operations on the MetaModelica boxed values. The functions 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. These are some of the available built-in functions:

- **Booleans**: `boolEq`, `boolString`
- **Integers**: `intEq`, `intLt`, `intLte`, `intGt`, `intGe`, `intNe`, `intString`, `intAdd`, `intSub`, `intMul`, `intDiv`, `intMod`, `intMin`, `intMax`

- **Reals**: `realEq`, `realLt`, `realLe`, `realGt`, `realGe`, `realNe`, `realString`, `realAdd`, `realSub`, `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`

4.4.4 Garbage collection

Garbage collection (GC), that is, 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 (Pettersson, 1995a; Wilson, 1992). 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 directly reuse this old MMC garbage collector for the OMC runtime because *it moves pointers* and makes it hard to write external functions.

The interface to the garbage collector in OMC is currently simple in order to make it easier to test different strategies. The garbage collector (GC) used for the remainder of the paper uses the Boehm-Demers-Weiser conservative mark-and-sweep garbage collector (Boehm and Weiser, 1988; Boehm et al., 1991). It is simply plugged into our code and collects garbage by analyzing the stack, searching for live objects to free.

4.5 Issues

We have identified several 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 regarding the syntax it accepts.

In order to find some common errors, the whole abstract syntax of a function is traversed to verify that all `matchcontinue` expressions have the same input and output as the function. 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 type check expressions of the following kind:

```
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 produces code that is easier to read and maintain.

Other issues include allowing code like:

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

In the standard 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. Since the same restrictions apply for OMC, these language constructs were refactored away.

4.5.2 OpenModelica Issues

Some parts of the compiler have been rewritten so that it became 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 also enhanced OMC to propagate file, line and column information to more error messages because when there is a large application that needs to be corrected, 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 is now giving better error messages both for Modelica and MetaModelica code.

OpenModelica has problems with shadowing of 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 Problems

The Modelica Specification (Modelica Association, a) 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 (some) 64-bit platforms, so we had to move functions from external C functions in the compiler into the MetaModelica language itself (as external builtin functions).

5 Performance

While we are not fully satisfied with the performance of the compiler at the time of writing this text, it has about the same performance as the old compiler (MMC) despite the garbage collector using up a large part of the total execution time.

The performance of the working examples varied widely in the first versions of the bootstrapped compiler. Some tests could be heavily improved by doing small changes.

In the first executable version 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. It is also possible to use our MetaModelica debugger on the code (Section 6).

To improve performance, we used a standard tool, `valgrind --tool=callgrind`, to profile a simple example. The `PEXPipe` model was chosen because it took 1-2 seconds to run it, making it of suitable size (profiling adds 10-100 times overhead). The model spent around 80% of its time doing exception handling. The C++ try-throw-catch feature had been used to implement the exception handling (Pop et al., 2008) of `matchcontinue` expressions. C++ exceptions are slow because they are not supposed to be used often. However, in MetaModelica 1.0 `matchcontinue` (that is, match with possible backtracking and continuation of the next case in the match) is used as the basic general control flow construct instead of special purpose constructs such as `if`-statements (which MMC does not support). The exception handling was rewritten using the cheaper `setjmp/longjmp` operations. Since the OpenModelica runtime does not use reference counting[3], C++ classes or C++-heap-allocated data, this level of exception handling is sufficient.

The new exception handling resulted in a 10 times speed-up for certain examples (on the average 3 times). 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 removed some of the `setjmp` overhead.

5.1 Benchmarks

All benchmarks were performed using a desktop running 64-bit Ubuntu 13.10 . The machine was equipped with a 6-core Intel Core i7-3930K CPU @ 3.20GHz and 32GB of memory. Both compilers (OMC and MMC) were taken from the standard OpenModelica package repository, compiled with default settings[4].

The two implementations use virtually the same parser, the only difference is in the garbage collector. The Modelica Standard Library (MSL) (Modelica Association, b) is distributed in multiple files that are parsed and merged into a single tree by the OpenModelica Compiler. Version 3.2.1 of the MSL was used. MSL 3.2.1 is 10.9MB of Modelica text when stored as a single file.

Unparsing the tree is the process of going from the internal abstract syntax tree structure back to Modelica concrete syntax (source code). In Table 3, you can see that the bootstrapped implementation is slightly faster at parsing. This is due to the MMC implementation having slower memory management for external C-code – the data is allocated in a special buffer which is then copied to the normal heap in an extra phase. The OMC implementation uses the Boehm GC (Boehm and Weiser, 1988; Boehm et al., 1991) as its garbage collector, which allows for using a parallel mark phase. The only disadvantage is parallel marking increases the total work of the thread due to thread synchronization, which means you should only enable it if the CPU would be idle without it.

Having the option to use parallel marking is a good benefit for most users of the compiler since they will typically work on only one model at a time. Compiler developers on the other hand will run enough tests at a time to fully utilize over 40 threads without parallel marking.

A few known large models were chosen to test the scalability of the compiler for large tests run as a single process. The tests that take the longest to execute are simulation tests, code is generated, compiled, ran, and the results are verified. However, most of the time is spent running a C compiler on generated code that is the same in both compilers. Thus, the chosen tests are large models that are not simulated.

The HumMod model (Kofrakek et al., 2011) has more than 28083 equations and the V6 engine from MSL (Modelica Association, b) contains 9016 equations. What the results in Table 4 tells us is that large models have a higher overhead than small models com-

[3] `longjmp` does not unwind the stack so using reference counting becomes hard to use.

[4] http://build.openmodelica.org/apt/ saucy/nightly

Table 3: Compiler performance, parsing.

Task	MMC		OMC			
			serial mark		parallel mark	
Parse MSL 3.2.1, single file	1.8s	0.96GB	1.74s	1GB	1.7s	1GB
Parse MSL 3.2.1, multiple files	1.71s	0.22GB	1.52s	0.23GB	1.43s	0.23GB
Parse and unparse	2.89s	0.37GB	3.32s	0.53GB	2.55s	0.54GB

pared to MMC. While OMC performs well overall, the parallel mark phase speeds up performance a lot which suggests the memory management can be further improved and that it will suffer on throughput of parallel tests where parallel mark is not a good option.

Running the entire OpenModelica testsuite is another important application for the developers. The results in Table 5 are rather good. For users who run only a single model at a time, parallel marking will improve performance roughly to the average speed of MMC. While the results show that MMC is faster on throughput, this is because a few models are using algorithms that scale poorly with the bootstrapped compiler implementation. These issues will be resolved once the compiler sources are changed to be incompatible with MMC, using the new language features that the bootstrapped OMC is capable of handling. It is expected that the single-process performance of OMC with parallel marking would surpass MMC at that time.

There is a difference in how the two tool chains create the executable compiler, but it is currently quite similar. Both tool chains analyze the MetaModelica source code to build the dependencies. OMC is more conservative and always succeeds while MMC uses a more optimistic approach and does not always recompile required files. However, the OMC analysis is more coarse-grained and translates more files than MMC does. If a translated file is the same as the previous one, the C-compiler will not try to re-compile this file, making the OMC approach managable since translating .mo-files is fast.

After dependency analysis, MMC will compile each mo-file into a C-file using one process per file. OMC will create one script per core and translate many mo-files into many C-files using one process per batch. The reason for this is OMC needs to parse the entire compiler sources in order to function, which takes longer than generating the source code for a single package. Both methods are run in parallel since it is important for developers to re-compile source code fast.

Table 6 shows the performance of the OMC approach to compiling the compiler. Not surprisingly, the performance figures of MMC and OMC are very close since profilers were run on the bootstrapped compiler to maximize performance on this benchmark.

Comparing the MMC and OMC build systems in Table 7, OMC shows better performance. GCC versions since 4.5 have problems handling the huge C-code generated by MMC. GCC has no problems with code generated by OMC however, giving OMC an advantage for compiler developers preferring GCC.

5.1.1 Comparison with old benchmarks

Compared to preliminary work performed in (Sjölund et al., 2011), the numbers reported here reflect a number of updates made in OpenModelica since 2011. Most important is the addition of garbage collection. The old implementation simply allocated a huge chunk of memory and never deallocated it. As such, a number of larger test cases could not run and were removed from the comparison. Now all test cases run and produce identical results. For some test cases, garbage collection takes up close to 50% of total run-time, which means performance has also improved since 2011 despite the numbers at that time saying MMC and OMC were closer in 2011.

Further, the tests can now be run in parallel since they use up much less memory. The 2011 build system for the bootstrapped compiler was always run in serial, utilizing a single CPU. In this version, parallel builds are compared for both MMC and OMC.

There is also a comparison on parallel garbage collection using parallel marker threads.

Judging by these changes, the current version of the bootstrapped compiler is much more mature than the 2011 version.

6 Debugger

The MMC compiler has a debugger developed by Pop (2008) in the Eclipse environment, including a data inspector for the MetaModelica types. It uses code instrumentation and tracing to perform debugging, which produces very large executables and slow execution. Moreover, the debugger is required to fetch the complete data of every single variable on a breakpoint. It also parses the source code of the compiler to find out the names and fields of records in uniontypes. Needless to say, the debugger is very slow. There exist

Table 4: Compiler performance, checkModel() on large models, time spent and memory consumption.

Task	**MMC**		**OMC**			
			serial mark		**parallel mark**	
HumMod	92.21s	0.76GB	134.6s	1.1GB	104.08s	1.1GB
EngineV6 (analytic)	6.85s	0.43GB	11.27s	0.76GB	8.54s	0.77GB

Table 5: Compiler performance, OpenModelica testsuite, run in parallel.

Task	**MMC**		**OMC**			
			serial mark		**parallel mark**	
1622 fast tests	126.64s	0.49GB	142.66s	0.83GB	145.89s	0.82GB
all 2499 tests	1076.7s	2.5GB	1291.5s	4.5GB	1317.5s	4.2GB

some ways of making it faster, for example by limiting the maximum depth of data that it fetches from a variable. It also lacks the possibility to show the stack frames to view a call chain, since the MMC compiler does not use the regular C stack. The same user interface for our new debugger, but the implementation is different Pop et al. (2012).

One aspect of the design of the bootstrapped compiler was to make sure that the debugged compiler code would have low overhead compared to the released version, to only query data that the user asks for, and to make sure the stack is preserved (except for tail-recursive calls). The changes made to the representation of data ensures that there is no need to parse the source code of the compiler to find out if a variable has the value `NONE()` or the empty list `{}`, and that the name of the record and its fields are stored as members in records (see Figure 2). Since the bootstrapped compiler uses the regular C stack and is compiled into C, it is possible to generate debug symbols for it and debug it using gdb. Compiling it takes no longer than compiling it in the regular manner. Disabling compiler optimizations gives access to more variables at the cost of significantly more stack space being used.

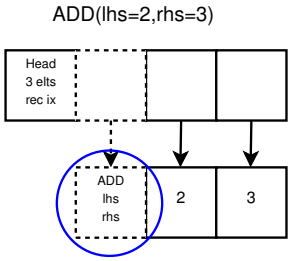

Figure 2: Adding record information to the data structure to simplify the work of the debugger at a small memory overhead (marked with a circle).

Table 6: Running BuildOMC.mos (the new build script for compiling OpenModelica).

Task	**MMC**		**OMC**			
			serial mark		**parallel mark**	
BuildOMC.mos (generate C sources)	44.58s	1.1GB	44s	1.3GB	43.05s	1.3GB

Table 7: Compiling the OpenModelica source code with the old build system (MMC) and the new one (OMC).

Task	**MMC**	**OMC**	**Factor**
Translating MetaModelica to C	135.2s	43.05s	0.32x
Compiling C-files using gcc 4.8 -O2	423.8s	44.55s	0.11x
Compiling C-files using clang 3.3 -O2	36.13s	36.04s	1x

Our Eclipse frontend uses the gdb machine interface to query runtime functions in order to get the types and values of variables whenever a breakpoint is reached. When the user inspects a certain variable, only the required data is fetched and displayed. When the debugger steps in or out, special care is taken for `setjmp/longjmp` calls to make sure that the debugger works the way you expect it to do. Stepping over a function call will either end up right after the call or at the last point caught by a `matchcontinue` or `failure` statement.

7 Related Work

Functional programming language compilers often bootstrap themselves during the build process. Some also include integrated lexer and parser generators specific to their own programming language. The OpenModelica project has recently developed the OMCCp parser generator (Lopez-Rojas, 2011) integrated with MetaModelica using scanning and parsing tables generated by `flex` and `bison` (Levine, 2009). It uses features in the OMC compiler, such as `for`-loops, that were never intended to be supported by MMC. Therefore OMCCp cannot be used as the regular OpenModelica parser generator until MMC is phased out completely at the end of the bootstrapping process.

In the Lisp (Steele and Gabriel, 1993) family of languages, bootstrapped compilers and interpreters have been available since the 1960's. These languages are dynamically strongly typed and defer the check to runtime instead of at compile-time. Thus, type checking is only performed on code that is executed and type errors may still be present in un-executed code.

Objective Caml (Chailloux et al., 2000), SML/NJ (Blume, 2001) and MLton (MLton, 2011) are examples of bootstrapped compilers in the ML family of programming languages. These languages are similar to MetaModelica in that they both use very similar language constructs, statically strong typing and type inference. This is not surprising 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, this design choice 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 variable declarations).

The concept of the `matchcontinue` expression, that is, matching with backtracking, 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, but is planned to be introduced.

To summarize: the `matchcontinue` expression is more general than the `match` expression, which is common in functional programming. As mentioned, it is related to clauses in logic programming since it provides backtracking on failure. However, in Prolog (Nilsson and Maluszynski, 1995) 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.

8 Conclusions

We have demonstrated that is it possible for a compiler of an equation-based object-oriented (EOO) language such as Modelica to compile itself. Thus, all of the static and translational semantics of Modelica is expressed in a slightly extended version of Modelica. We have also shown that the performance of the implementation is sufficient and is expected to improve in the future. 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 future versions of Modelica that allow modeling of language features in libraries. We believe that this work will be an important input and proof-of-concept to the design effort.

The benefits for users of OpenModelica are many. It is possible to use OMC as a shared library and directly call functions asynchronously in threads. This is much faster than the current synchronous interface which operates over the network and does not allow calls to be interrupted. For users of the graphical user interface, this will enable progress bars in long operations, in addition to the ability to cancel them.

The bootstrapped compiler is also able to use threads, utilising multiple cores that are now present in almost all consumer CPUs.

It is expected that we will be able to develop features to the compiler much faster since we will now be able to use concepts like loops and if-expressions without

using function pointers (a limitation of MMC).

Many requested features, mostly regarding interfacing with OpenModelica, have been put on hold since they are much simpler to implement in the bootstrapped compiler.

8.1 Future Work

The garbage collection needs to be improved by generating code that gives better hints to the garbage collector like allocation of data that does not contain pointers. This has been implemented for strings, but not for vectors of numbers and similar structures used in the back-end.

Introduction of register allocation (optimized intermediate code) in the compiler will help reduce the burden of the garbage collection algorithms. The current version tries to limit the number of registers, but it is not optimal. It is an optmization that greatly helps for using less stack space even when generating code for debugging.

Parts of the compiler can be rewritten/refactored using certain more powerful and concise language constructs in MetaModelica 2.0 (Fritzson et al., 2011). This will be done once MMC support is discontinued.

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.

Introduce language constructs for threading since the garbage collector is capable on handling threads. A prototype using external C functions was tested on memory-intensive tasks like parsing compiler sources. However boehm gc will stop all threads when collecting garabage, severely limiting performance. Speedup was limited to around 2x due to Amdahl's law (Amdahl, 1967). For this reason, these extensions would be more useful for computation-intensive tasks rather than memory-intensive.

We will improve on the build system by generating interface files in order to perform a more fine-grained dependency analysis. This will enable recompiling dependent packages only if the interface changes. For example, recompiling a package if an input to a dependent function changes, but not if only a comment in that package changes.

Acknowledgments

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 project.

References

Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring). ACM, New York, NY, USA, pages 483–485, 1967. doi:10.1145/1465482.1465560.

Appel, A. W., Mattson, J. S., and Tarditi, D. R. A lexical analyzer generator for Standard ML. 1994. URL http://www.smlnj.org/doc/ML-Lex/manual.html.

Bachmann, B., editor. *Proceedings of the 6th International Modelica Conference*. Modelica Association, 2008.

Barton, P. I. and Pantelides, C. C. Modeling of combined discrete/continuous processes. *AIChE Journal*, 1994. 40:966–979. doi:10.1002/aic.690400608.

Blume, M. CMB - The SML/NJ Bootstrap Compiler - User Manual. 2001. URL http://www.smlnj.org/doc/CM/btcomp/.

Boehm, H.-J., Demers, A. J., and Shenker, S. Mostly parallel garbage collection. *SIGPLAN Not.*, 1991. 26(6):157–164. doi:10.1145/113446.113459.

Boehm, H.-J. and Weiser, M. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 1988. 18(9):807–820. doi:10.1002/spe.4380180902.

Brück, D., Elmqvist, H., Olsson, H., and Mattsson, S. E. Dymola for multi-engineering modeling and simulation. In Otter (2002), 2002.

Broman, D. *Meta-Languages and Semantics for Equation-Based Modeling and Simulation*. Doctoral thesis No 1333, Department of Computer and Information Science, Linköping University, Sweden, 2010. URL http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-58743.

Brus, S. *Bootstrapping the OpenModelica Compiler: Implementing Functions as Arguments*. Bachelor's thesis draft, Linköping University, 2009. Not published.

Casella, F., editor. *Proceedings of the 7th International Modelica Conference*. Linköping University Electronic Press, 2009.

Chailloux, E., Manoury, P., and Pagano, B. *Developing applications with Objective Caml*. O'Reilly and Associates, 2000. URL http://caml.inria.fr/pub/docs/oreilly-book/ocaml-ora-book.pdf.

Clauß, C., editor. *Proceedings of the 8th International Modelica Conference*. Linköping University Electronic Press, 2011.

Elmqvist, H. *A Structured Model Language for Large Continuous Systems*. Ph.D. thesis, Department of Automatic Control, Lund University, Sweden, 1978.

Elmqvist, H., Mattsson, S. E., and Otter, M. Modelica - a language for physical system modeling, visualization and interaction. In *Proceedings of the 1999 IEEE International Symposium on Computer-Aided Control System Design*. pages 630 –639, 1999. doi:10.1109/CACSD.1999.808720.

Equa AB. IDA Simulation Environment. 2002. URL http://www.equa.se/eng.se.html.

Fritzson, P. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, 2004.

Fritzson, P. Language modeling and symbolic transformations with Meta-Modelica. 2005. URL http://openmodelica.org. Later versions Fritzson (2007); Fritzson and Pop (2011a).

Fritzson, P. Language modeling and symbolic transformations with Meta-Modelica. 2007. URL http://openmodelica.org. Slightly updated version of Fritzson (2005); later update Fritzson and Pop (2011a).

Fritzson, P. *Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica*. Wiley-IEEE Press, 2011.

Fritzson, P. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley-IEEE Press, 2014.

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

Fritzson, P. and Bunus, P. Modelica - a general object-oriented language for continuous and discrete-event system modeling and simulation. In *Proceedings 35th Annual Simulation Symposium*. pages 365–380, 2002. doi:10.1109/SIMSYM.2002.1000174.

Fritzson, P. and Engelson, V. Modelica - a unified object-oriented language for systems modeling. In E. Jul, editor, *ECOOP'98 - Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 67–90. Springer Berlin / Heidelberg, 1998. doi:10.1007/BFb0054087.

Fritzson, P., Gunnarsson, J., and Jirstrand, M. Mathmodelica – an extensible modeling and simulation environment with integrated graphics and literate programming. In Otter (2002), pages 41–54, 2002.

Fritzson, P. and Pop, A. Meta-programming and language modeling with MetaModelica 1.0. Technical Report 9, Linköping University, PELAB - Programming Environment Laboratory, 2011a. URL http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-66440. Almost identical to Fritzson (2007), but tech. report.

Fritzson, P. and Pop, A. Meta-programming and language modeling with MetaModelica 1.0. Technical Report 9, Linköping University, PELAB - Programming Environment Laboratory, 2011b. URL http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-66440.

Fritzson, P., Pop, A., and Aronsson, P. Comprehensive meta-modeling and meta-programming capabilities in Modelica. In G. Schmitz, editor, *Proceedings of the 4th International Modelica Conference*. 2005b.

Fritzson, P., Pop, A., Broman, D., and Aronsson, P. Formal semantics based translator generation and tool development in practice. In C. Fidge, editor, *Proceedings of the 2009 Australian Software Engineering Conference*. IEEE Computer Society, Washington, DC, USA, pages 256–266, 2009a. doi:10.1109/ASWEC.2009.46.

Fritzson, P., Pop, A., and Sjölund, M. Towards Modelica 4 meta-programming and language modeling with MetaModelica 2.0. Technical Report 2011:10, Linköping University, PELAB - Programming Environment Laboratory, 2011. URL http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-68361.

Fritzson, P., Privitzer, P., Sjölund, M., and Pop, A. Towards a text generation template language for Modelica. In Casella (2009), pages 193–207, 2009b. doi:10.3384/ecp09430124.

Hindley, J. R. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 1969. 146:29–60. URL http://www.jstor.org/stable/1995158.

ITI GmbH. SimulationX. 2012. URL http://www.itisim.com/simulationx.html.

Åkesson, J., Årzén, K.-E., Gäfvert, M., Bergdahl, T., and Tummescheit, H. Modeling and optimization with Optimica and JModelica.org - languages and tools for solving large-scale dynamic

optimization problems. *Computers & Chemical Engineering*, 2010. 34(11):1737 – 1749. doi:10.1016/j.compchemeng.2009.11.011.

Kloas, M., Friesen, V., and Simons, M. Smile - a simulation environment for energy systems. In *Proceedings of the 5th International IMACS - Symposium on Systems Analysis and Simulation (SAS'95*. Gordon and Breach Publishers, pages 503–506, 1995.

Kofrakek, J., Matejak, M., and Privitzer, P. HumMod – large scale physiological models in Modelica. In Clauß (2011), pages 713–724, 2011. doi:10.3384/ecp11063713.

Levine, J. *flex & bison*. O'Reilly Media, 2009.

Lopez-Rojas, E. A. *OMCCp: A MetaModelica Based Parser Generator Applied to Modelica*. Master's thesis, Linköping University, Department of Computer and Information Science, 2011. URL http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-68863.

Maplesoft. MapleSim. 2012. URL http://www.maplesoft.com/.

Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 1978. 17:348–375.

Milner, R., Tofte, M., Harper, R., and MacQueen, D. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

MLton. Installation instructions (bootstrapping). 2011. URL http://mlton.org/PortingMLton.

Modelica Association. Minutes of the Modelica Design Meeting 67. 2010. URL http://modelica.org.

Modelica Association. Modelica: A unified object-oriented language for physical systems modeling, language specification version 3.3. 2012a. URL http://www.modelica.org/.

Modelica Association. Modelica Standard Library version 3.2.1. 2013b. URL https://modelica.org/libraries.

Nilsson, H., Peterson, J., and Hudak, P. Functional hybrid modeling from an object-oriented perspective. In P. Fritzson, F. Cellier, and C. Nytsch-Geusen, editors, *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Linköping University Electronic Press, 2007. URL http://www.ep.liu.se/ecp/024/.

Nilsson, U. and Maluszynski, J. *Logic, Programming and Prolog (2ed)*. John Wiley & Sons Ltd, 1995. URL http://www.ida.liu.se/~ulfni/lpp/.

Odersky, M., Spoon, L., and Venners, B. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

Otter, M., editor. *Proceedings of the 2nd International Modelica Conference*. Modelica Association, 2002.

Otter, M. and Zimmer, D., editors. *Proceedings of the 9th International Modelica Conference*. Linköping University Electronic Press, 2012.

Palanisamy, A., Pop, A., Sjölund, M., and Fritzson, P. Modelica based parser generator with good error handling. In H. Tummescheit and K.-E. Årzén, editors, *Proceedings of the 10th International Modelica Conference*. Modelica Association and Linköping University Electronic Press, 2014. doi:10.3384/ecp14096567.

Parr, T. ANTLR parser generator 3.3. 2010. URL http://www.antlr.org/.

Pettersson, M. *Compiling Natural Semantics*. Doctoral thesis No 413, Department of Computer and Information Science, Linköping University, Sweden, 1995a. Also published in Pettersson (1999).

Pettersson, M. *Compiling Natural Semantics*. Doctoral thesis No 413, Department of Computer and Information Science, Linköping University, Sweden, 1995b.

Pettersson, M. *Compiling Natural Semantics*, volume 1549 of *Lecture Notes in Computer Science*. Springer, 1999. doi:10.1007/b71652.

Peyton Jones, S. et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 2003. 13(1). doi:10.1017/S0956796803000315.

Pop, A. *Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages*. Doctoral thesis No 1183, Department of Computer and Information Science, Linköping University, Sweden, 2008.

Pop, A. and Fritzson, P. Metamodelica: A unified equation-based semantical and mathematical modeling language. In D. Lightfoot and C. Szyperski, editors, *Modular Programming Languages*, volume 4228 of *Lecture Notes in Computer Science*, pages 211–229. Springer Berlin / Heidelberg, 2006. doi:10.1007/11860990_14.

Pop, A., Fritzson, P., Remar, A., Jagudin, E., and Akhvlediani, D. OpenModelica development environment with Eclipse integration for browsing, modeling, and debugging. In C. Kral and A. Haumer, editors, *Proceedings of the 5th International Modelica Conference*. 2006.

Pop, A., Sjölund, M., Asghar, A., Fritzson, P., and Casella, F. Static and dynamic debugging of Modelica models. In Otter and Zimmer (2012), 2012. doi:10.3384/ecp12076443.

Pop, A., Stavåker, K., and Fritzson, P. Exception handling for Modelica. In Bachmann (2008), pages 409–418, 2008.

Sahlin, P. and Sowell, E. F. A neutral format for building simulation models. In *Proceedings of the Conference on Building Simulation*. pages 147–154, 1989.

Sjölund, M. *Bidirectional External Function Interface Between Modelica/MetaModelica and Java*. Master's thesis, Linköping University, Department of Computer and Information Science, 2009. URL http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-20386.

Sjölund, M., Fritzson, P., and Pop, A. Bootstrapping a Modelica compiler aiming at Modelica 4. In Clauß (2011), 2011. doi:10.3384/ecp11063510.

Stavåker, K., Pop, A., and Fritzson, P. Compiling and using pattern matching in Modelica. In Bachmann (2008), pages 637–646, 2008.

Steele, G. L., Jr. and Gabriel, R. P. The evolution of lisp. In *The second ACM SIGPLAN conference on History of programming languages*, HOPL-II. ACM, New York, NY, USA, pages 231–270, 1993. doi:10.1145/154766.155373.

Tarditi, D. R. and Appel, A. W. ML-Yacc User's Manual. 2000. URL http://www.smlnj.org/doc/ML-Yacc/.

Tiller, M. *Introduction to Physical Modeling with Modelica*. Springer, 2001.

Viklund, L., Herber, J., and Fritzson, P. The implementation of ObjectMath - a high-level programming environment for scientific computing. In U. Kastens and P. Pfahler, editors, *Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 312–318. Springer Berlin / Heidelberg, 1992. doi:10.1007/3-540-55984-1_28.

Wilson, P. R. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92. Springer-Verlag, London, UK, pages 1–42, 1992.

Wirth, N. The programming language Pascal. *Acta Informatica*, 1971. 1:35–63. doi:10.1007/BF00264291.

Wolfram Mathcore. System Modeler. 2012. URL http://mathcore.com/.

Zimmer, D. *Equation-based modeling of variable-structure systems*. Ph.D. thesis, Eidgenössische Technische Hochschule ETH Zürich, Switzerland, 2010. doi:10.3929/ethz-a-006053740.